A FIFO Spin-based Resource Control Framework for Symmetric Multiprocessing

Shuai Zhao

Doctor of Philosophy

University of York
Computer Science
July 2018

Abstract

Managing shared resources in multiprocessor real-time systems can often lead to considerable schedulability sacrifice, and currently there exist no optimal multiprocessor resource sharing solutions. In addition, the choice of task mapping and priority ordering algorithms also has a direct impact on the efficiency of multiprocessor resource sharing. This thesis argues that instead of adopting a single resource sharing protocol with the traditional task mapping (e.g., the task allocation schemes that are based on utilisation only) and priority ordering (e.g., the Deadline Monotonic Priority Ordering) algorithms, the schedulability loss for managing shared resources on multiprocessors can be effectively reduced by applying a combination of appropriately chosen resource sharing protocols with new resource-oriented task allocation schemes and a new search-based priority ordering algorithm (which are independent from multiprocessor resource sharing protocols and the corresponding schedulability tests).

In this thesis, a Flexible Multiprocessor Resource Sharing (FMRS) framework is proposed that aims to provide feasible resource sharing, task allocation and priority assignment solutions to fully-partitioned systems with shared resources, where each resource is controlled by a designated locking protocol. To achieve this, the candidate resource sharing protocols for this framework are firstly determined with a new schedulability test developed to support the analysis of systems with multiple locking protocols in use. Then, besides the existing algorithms, three new resource-orientated task allocation schemes and a search-based priority ordering algorithm are developed for the FMRS framework as the task mapping and priority ordering solutions. The choices of which locking protocols, task allocation and priority ordering algorithm should be adopted to a given system are determined off-line via a genetic algorithm. As demonstrated by evaluations, the FMRS framework can facilitate multiprocessor resource sharing and has a better performance than the traditional resource control and task scheduling techniques for fully-partitioned systems.

Contents

List of Figures

List of Tables

List of Symbols

Symbol	**Description**
τ_x	A given real-time task with index x.
C_x	Worst-case computation time of τ_x without accessing any shared resources.
T_x	Period of τ_x.
D_x	Deadline of τ_x.
R_x	Response time of τ_x.
$Pri(\tau_x)$	Priority of τ_x.
$U_x = \frac{C_x}{T_x}$	Utilisation of τ_x without shared resources.
r^k	A given shared resource with index k.
c^k	Critical section length of r^k.
N_x^k	The number of requests issued from τ_x to r^k in one release.
$F(\tau_x)$	The set of resources that are accessed by τ_x.
$G(r^k)$	The set of tasks that request r^k.
τ_i	A task that is currently been studied by the schedulability analysis.
τ_l	A task with a priority lower than that of τ_i.
$\hat{c}_i$	The arrival blocking incurred by τ_i.
$\mathbf{hp}(i)$	The set of tasks with a priority higher than that of τ_i.
$Pri(r^k)$	The resource ceiling priority of r^k.
τ_{ll}	A local task with a priority lower than that of the task that is currently being studied (i.e., τ_i).

$\widehat{C_i}$	The pure computation time of τ_i (i.e., C_i in Table 2.1) plus the time τ_i spends on waiting for and executing with each requested resource (i.e., with the potential delay for accessing shared resources).
e^k	The total accessing cost of r^k, including the pure execution cost of r^k and the potential delay for accessing this resource.
$\hat{e}_i$	The arrival blocking incurred by τ_i with the potential remote blocking included.
$\mathbf{lhp}(i)$	The set of local tasks with a priority higher than that of τ_i.
τ_j	A remote task.
τ_h	A high priority task.
$\tau(P_m)$	Tasks allocated to P_m.
$P(\tau_x)$	Partition of τ_x.
C^k	The cost for executing resource r^k with implementation overheads.
CX_1, CX_2	The cost of context switches of the operating system.
E_x	Total resource-accessing time of τ_x to all resources.
$I_{x,h}$	Indirect spin delay incurred by τ_x from a local high priority task.
$e^k_x(l, \mu)$	Total resource-accessing time of τ_x accessing r^k during the time l with a jitter μ.
$e^k_x(l)(n)$	Resource-accessing time of τ_x's n-th access to r^k during the time l.
α^k_i	A set of partitions with requests that can cause τ_i to incur arrival blocking.
$N^k_x(l, \mu)$	Number of requests of τ_x to r^k during the time l with a jitter μ.
$Nh^k_x(l)$	Number of requests of τ_x's higher priority tasks to r^k during the time l.
$Np^k_m(l)$	Number of requests issued from tasks on P_m to r^k during the time l.

$NS^k_{x,m}(l)$	The maximum number of requests on a remote processor m that could block τ_x directly for accessing r^k within the duration l.
$(f(x))_0$	Function $f(x) >= 0$.
$(f(x))^b_a$	Function $min\{max\{f(x), a\}, b\}$.
C_{retry}	The implementation overheads for cancelling a resource request and re-accessing a resource.
S_i	The additional blocking of τ_i due to the cancellation mechanism of PWLP.
NoP_i	The number of preemptions τ_i can incur during one release.
L^k_i	A list of additional blocking times incurred by τ_i for re-accessing r^k due to each preemption.
LS_i	A list of additional blocking times (ordered decreasingly) that τ_i can incur for re-accessing shared resources in $F^S(\tau_i)$.
$F^S(\tau_i)$	The set of resources that can cause τ_i to incur the additional blocking under PWLP.
$L(n)$	The n-th element in the given list L
$\{\}^{dList}$	A list with the elements ordered non-increasingly by their values.
C_{mig}	The cost of one migration.
C_{np}	The length of the NP section.
$\hat{np}_i$	The blocking that τ_i can incur due to the NP section.
$mt^k_x(l)(n)$	The migration targets of τ_x's nth access to r^k within the duration l.
$mtp(mt, r^k)$	A set of migration targets with tasks that can preempt tasks that accessing resource r^k in the given set of migration targets mt.
$Mig(mt, r^k)$	The total migration cost a task can incur for accessing r^k with the given set of migration targets mt.
$Mhp(mt, r^k)$	The migration cost of a single access to r^k bounded by the releases of high priority tasks on the given set of migration targets mt.

Mnp^k	The migration cost of a single access to r^k bounded by the length of the NP section.
$hpt(r^k, P_m)$	tasks on partition m that have a higher priority than the ceiling of r^k.
MC_i	The total migration cost incurred by τ_i in the spin delay.
$MIG_i^k(l, \mu)$	The amount of migration cost caused by τ_x for accessing r^k within the given duration l and jitter μ.
R	The shared resources in the given system.
R_{MSRP}	The resources that are controlled by MSRP.
R_{PWLP}	The resources that are managed by PWLP.
R_{MrsP}	The resources that are controlled by MrsP.
B_i^{MSRP}	The arrival blocking caused by the MSRP resources.
B_i^{PWLP}	The arrival blocking caused by the PWLP resources.
B_i^{MrsP}	The arrival blocking caused by the MrsP resources.
$F_{NP}^A(\tau_i)$	The set of resources that can cause τ_i to incur arrival blocking with the non-preemptive resource-accessing rule.
$F_{Ceiling}^A(\tau_i)$	The set of resources that can cause τ_i to incur arrival blocking with the ceiling priority resource accessing rule applied.
$c_x^k(n)$	The pure execution cost of τ_x's $\mathrm{n^{th}}$ access to r^k.
$L = \{\}^{dList}$	A given list L with a set of positive values ordered by a non-increasing fashion.
$L(n)$	The $\mathrm{n^{th}}$ element from a given list L. A value of 0 is returned if the $\mathrm{n^{th}}$ element does not exist.
$\|L\|$	The size of a given list L.
Lcs_x^k	A list of execution costs of τ_x for accessing r^k during one release.
$Lcs_x^k(l, \mu)$	A list of execution costs of τ_x for accessing r^k within a duration l and a jitter μ.
$Lcsp_m^k(l)$	A list of execution costs from tasks on P_m to r^k within a given duration l.

List of Abbreviations

Abbreviation	**Description**
FMRS	Flexible Multiprocessor Resource Sharing Framework
ABS	Anti-lock Braking System
DMPO	Deadline Monotonic Priority Ordering
SC	Success Criteria
FPS	Fixed Priority Scheduling
EDF	Earliest Deadline First Scheduling
VBS	Value-based Scheduling
LLF	The Least Laxity First Scheduling
FPPS	Fixed Priority Preemptive Scheduling
RMPO	Rate Monotonic Priority Ordering
DMPO	Deadline Monotonic Priority Ordering
OPA	Audsley's Optimal Priority Assignment
RPA	Robust Priority Assignment
RTA	Response Time Analysis
RTOS	Real-Time Operating System
NP	Non-Preemptive
SMP	Symmetric Multiprocessing
AMP	Asymmetric Multiprocessing
UMA	Uniform Memory Access Model
NUMA	Non-Uniform Memory Access Model
WF	Worst Fit Allocation
BF	Best Fit Allocation
FF	First Fit Allocation
NF	Next Fit Allocation
FCFS	First Come First Serve

PIP	Priority Inheritance Protocol
PCP	Priority Ceiling Protocol
OPCP	Original Priority Ceiling Protocol
IPCP	Immediate Priority Ceiling Protocol
SRP	Stack Resource Policy
MPCP	Multiprocessor Priority Ceiling Protocol
DPCP	Distributed Priority Ceiling Protocol
MSRP	Multiprocessor Stack Resource Protocol
FMLP	Flexible Multiprocessor Locking Protocol
PWLP	Preemptable Waiting Locking Protocol
PPCP	Parallel Priority Ceiling Protocol
OMLP	O(m) Locking Protocol
SPEPP	Spinning Processor Executes for Preempted Processor
M-BWI	Multiprocessor BandWidth Inheritance Protocol
MrsP	Multiprocessor resource sharing protocol
SPA	Synchronisation-aware Partitioning Algorithm
BPA	Blocking-aware Partitioning Algorithm
ILP	Integer Linear Programming
AUTOSAR	Automotive Open System Architecture
ANOVA	Analysis of Variance
RCF	Resource Contention Fit
RLF-L	Resource Length Fit-Long
RLF-S	Resource Length Fit-Short
SBPO	Slack-based Priority Ordering
SA	Simulated Annealing
GA	Genetic Algorithm
Litmus^{RT}	Linux Testbed for Multiprocessor Scheduling in Real-time Systems
P-FP	Preemptable-Fixed Priority Scheduler in Litmus^{RT}.

Acknowledgement

It would not be possible to compete my Ph.D study and to finish this dissertation without the help and support of the people around me.

Firstly, I would like to present my deepest appreciation to Prof. Andy Wellings for his persistent guidance, help and care for the past four years. Without him, my research work and this Ph.D thesis would not be even possible. From him, I learned not only the research skills but also the right attitude for work and life.

In addition, I would like to thank Prof. Alan Burns for his guidance and help duration my Ph.D study. He is always nice and patient to explain the questions I asked and encourages me to carry on my study.

I would also like to thank my parents and my wife for their unconditional support and continuously encouragement, which allows me to focus on my research and gives me the strength to overcome the difficulties and challenges from both my study and daily life.

I also acknowledge the Computer Science Department for the funding, which relieves me and my family from heavy financial costs.

The appreciation also goes to the members in the Real-time Systems group for their brilliant advice to my research and their encouragements.

Finally, I would like to thank my friends for their help and company during my entire Ph.D study, which makes this journey a delightful memory.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. The work present in this dissertation has not previously been presented for an award at this, or at any other, University. All sources are acknowledged as References. Parts of this thesis have previously been published in the following papers:

- S. Zhao, J. Garrido, A. Burns, and A. Wellings. *New Schedulability Analysis for MrsP.* In IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 1-10. IEEE, 2017.

- S. Zhao and A. Wellings. *Investigating the Correctness and Efficiency of MrsP in Fully Partitioned Systems.* In 10th York Doctoral Symposium on Computer Science and Electronics (YDS). The University of York, 2017.

- J. Garrido, S. Zhao, A. Burns, and A. Wellings. *Supporting Nested Resources in MrsP.* In Ada-Europe International Conference on Reliable Software Technologies (Ada Europe), pages 73-86. Springer, 2017.

  In this work, I provided the fundamental analysing techniques for MrsP systems with nested resources. These analysing techniques for nested resources are described in Appendix A.

- J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings. *Implementation and Evaluation of Multiprocessor Resource Synchronization Protocol (MrsP) on Litmus^{RT}.* In Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), 2017.

  In this work, I provided a fully functional MrsP implementation, which provides the basis of the implementations and evaluation results given in this thesis. The techniques and implementation details for realising MrsP in fully-partitioned systems in described in Appendix C.

Chapter 1

Introduction

1.1 Motivation

Nowadays, embedded systems can be found almost everywhere in life, from portable household and consumer electronics, to heavy machinery in transportation and manufacturing, and to large and complex control or communication systems in medical imaging, aircrafts and spaceships. According to Burns and Wellings [26], approximately 99% of microprocessors are produced for the use in embedded systems. A key characteristic of embedded systems is the guaranteed timing constraints, where the correctness of the system depends not only on the correctness of logical results being generated but also on the time at which the results are delivered. Such systems are referred to *real-time systems.*

As described in [26], "a real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment". Failing to deliver the required operation in time is recognised as a *deadline miss*, which could cause huge economic loss or even casualties. For instance, when performing emergency braking in a rapid moving vehicle, the anti-lock braking system (ABS) should be activated within 30 milliseconds so that the vehicle is stable and under control [5]. A typical ABS will apply and release the break pressure alternately to prevent the wheels from locking up, where each operation must be finished within a pre-defined time interval [92]. However, failing to do so within the required period (i.e., deadline miss) can result in uncontrolled slipping and prolonged braking distance, which directly jeopardise the safety of the driver and passengers in the vehicle.

Based on the sensitivity to the timing constrains, real-time systems are categorised as either *hard real-time* or *soft real-time* systems. Hard real-time systems are the ones where no deadline misses will be tolerated, that is, one single deadline miss can directly lead to a total system failure and cause a huge damage to the surrounding environment. The ABS equipped in modern vehicles described above is a typical hard-real time system and is safety critical, where system failures endanger safety and health. In comparison, soft real-time systems are able to cope with occasional deadline misses (but usually with a maximum limit of misses within a specific period) while still functioning correctly, for example multi-media systems or communication systems [25]. In such systems, deadline misses can degrade the value of the results being generated, and hence, undermine the quality of the service. However, continuously missing deadlines can still cause the failure of the system.

To satisfy such strict timing requirements, many real-time facilities and techniques have been proposed to coordinate the concurrent executions of tasks so that each task is able to meet its deadline i.e., it is schedulable. In a real-time system, each task (i.e., a single thread of control) is assigned with a priority to denote its level of urgency. During execution, a scheduling algorithm is applied to designate the task that is allowed to execute at a given time either based on the statically assigned priorities or dynamically assigned ones according to the urgency of tasks (e.g., schedule the task with the closest deadline). To obtain predictability of the system, which is a key characteristic of real-time systems, schedulability tests are supported to provide a safe upper bound of the response time for each task in the system during the worst-case execution scenario. With the presence of shared resources, resource sharing protocols must be adopted to provide mutually exclusive access as well as safely bounded blocking time of each resource-accessing task.

The real-time technology for uniprocessor systems is reasonably matured and has been well-practised for decades, and there exist multiple optimal algorithms and techniques [37]. A scheduling algorithm is said to be *optimal* if it can schedule all task sets that are schedulable by other algorithms [25]. Deadline Monotonic Priority Ordering [65] assigns static priorities (from highest to lowest) to tasks based on the non-decreasing order of deadlines and is proved to be optimal, even with the presence of blocking [16]. Audsley's Algorithm [9] proposes a sophisticated approach for priority ordering and is optimal in a wider range of application semantics, such as systems with offset release times [38]. As for the scheduling algorithm, Fixed Priority Scheduling [72] and Earliest Deadline First scheduling [71] provide static (with predictions before execution) and dynamic (by run-time decisions) scheduling approach respectively and are well-understood. For schedulability test, Response Time Analysis developed in [7] is a sufficient and necessary schedulability test for uniprocessors that provides the worst-case response time of each task in a given system, and is the fundamental work that inspires the development of many advanced schedulability tests for more complex application semantics [25]. The term *sufficient and necessary* indicates that passing the test can guarantee that all deadlines will always be met, yet failing the test can indeed result into deadline misses at certain point during execution [26]. In addition, the Priority Ceiling Protocol [93] and the Stack Resource Protocol [11] are proved to be optimal resource sharing protocols for uniprocessor systems that minimise the waiting time for resources and are supported with matured schedulability analysis [25]. A comprehensive review of the real-time technology in uniprocessor systems is presented in Section 2.1.

1.1.1 Transition from Uniprocessors to Multiprocessors

Over the last few years, the increasing demand of computation power has led to a trend of the transition from uniprocessor to multiprocessor real-time systems [23]. With more computing units i.e., processors, it is possible to construct large and complex real-time systems, where multiple tasks can execute in parallel, performing one complex computation or multiple independent operations [103]. Moving to multiprocessor platforms is a significant advance in the development of real-time systems. However, while obtaining more computation power and widening real-time applications, open issues and new challenges are raised with multiprocessor real-time systems. Due to the simple fact that tasks now can execute in parallel, such a transition directly causes the matured uniprocessor techniques to be inapplicable or breaks the optimality. Although huge progress has been made, the technology in multiprocessors real-time systems is not as matured as that of uniprocessors [37].

With multiprocessors, scheduling tasks becomes significantly more complicated than that of the uniprocessor platforms. The first problem encountered is to map tasks into processors, where two fundamental approaches are available: *global* and *partitioned* schemes [107]. The global scheduling approach dynamically assigns tasks to available processors when they become runnable during the execution of the system, and tasks are allowed to migrate between processors during run-time. In contrast, a partitioned algorithm statically allocates tasks into each processor before run-time, and migrations are not allowed so that tasks will spend their entire lifetime in their designated processors. The partitioned scheduling approach can be attractive as it divides the multiprocessor systems into multiple uniprocessor systems to a certain extent so that matured technology on uniprocessors can be applied. However, allocating tasks in partitioned systems is proved to be a typical NP-hard bin-packing problem [66], where optimal solutions are infeasible. Later, more scheduling schemes are proposed that combine the two fundamental approaches, such as the semi-partitioned scheduling [24,59] and the clustered (i.e., hybrid) scheduling [95]. Yet none of the existing algorithms can dominate others and each approach has its own advantages and drawbacks. A detailed review of the scheduling algorithms for multiprocessors is presented in Section 2.2.

Another major concern is to provide predictable mutually exclusive access to shared objects. Resource sharing technology in multiprocessors is still developing with many open issues, and there exists no optimal solutions as the agreed best practice [41]. This is due to the fact that multiple tasks can now issue requests to a resource from more than one processors (i.e., global resource) at the same time, which leads to prolonged blocking time as well as various blocking effects. Matured uniprocessor locking protocols cannot be applied directly as they can only manage local resources, which are accessed from one processor. In addition, the essential differences of the dispatching schemes for multiprocessor systems (e.g. global and partitioned schemes) increase the difficulty of the development of a general-purpose protocol. Although various multiprocessor resource sharing protocols have been proposed, each protocol has its own advantages, drawbacks and limitations [3, 22]. Meanwhile, the research towards the schedulability tests for multiprocessor locking protocols is still in progress, where some of the multiprocessor locking protocols either lack efficient schedulability analysis support or have analysis with considerable pessimism [37]. In addition, the schedulability, availability and run-time overheads of the existing protocols in practice require further investigations [19]. Section 2.3 to 2.6 provide a detailed description of real-time resource sharing technology for both uniprocessor and multiprocessor systems.

Summarising the above, this section provides a brief background of real-time systems; introduces general terms and algorithms in both uniprocessor and multiprocessor real-time systems; and briefly presents the state of art in multiprocessor systems with shared objects. The next section presents the aim of this thesis with the description of the challenges and open issues in resource control of multiprocessor real-time systems.

1.2 Thesis Aim

With the presence of shared objects, many factors can affect the performance of a multiprocessor system. From the underlying hardware platform and the processor architecture to the choice of scheduling algorithms, priority ordering algorithms and resource sharing protocols, each factor can have an impact on the schedulability of multiprocessor systems. The overall aim of this thesis is to investigate the impact of the major factors to the schedulability of multiprocessor real-time systems with shared objects, and to propose solutions that can effectively reduce the schedulability loss due to shared resources control and to improve resource sharing performance on multiprocessors.

The resource sharing protocols specify the behaviours of tasks while accessing resources and can directly affect the schedulability of the system. Optimal solutions of resource sharing for multiprocessors may not be achievable, where each protocol demonstrates varied performance with different application semantics, such as the critical section length and the degree of resource contention. For instance, the spin-based protocols can demonstrate better performance than that of the suspension-based approaches if the length of critical sections is no more than 20% of that of the total computation time [22]. Thus, for an application with both short and long critical sections, applying either approach can lead to certain degree of pessimism as some resources in the system cannot be managed by their favourable synchronisation approach. However, with both approaches adopted, where each protocol only manages the resources that it can benefit, such pessimism can be minimised with improved schedulability.

With more than one resource sharing protocols adopted into a single system, the schedulability tests must be modified to support the analysis with the presence of multiple protocols. However, the existing schedulability tests can only support the analysis of systems with one protocol adopted [37]. In addition, as mentioned in Section 1.1.1, the research of schedulability analysis with the presence of blocking is still under development. Although advanced analysis techniques have been proposed for some protocols, which can provide less pessimistic results than that of the original tests [15, 106], some of the existing protocols still either lack sufficient schedulability analysis support or have an analysis with considerable pessimism, which can make the protocols less preferable or even inapplicable in practice even with attractive characteristics. In addition, the run-time overheads (e.g., the costs of context switch from the underlying operating system and the overheads due to the protocol implementations) are often not taken into account in schedulability tests as such costs varies with different run-time environments and is difficult to bound. However, ignoring the run-time costs can lead to inaccurate schedulability analysis, where an application that passes the schedulability test can become unschedulable during execution, or a resource sharing protocol that is favourable in theory but is much less attractive in practise due to its unanalysed run-time overheads.

Besides the resource sharing protocols and the corresponding analysis, different choices of the scheduling approaches can result into various execution scenarios of a given application, and hence, lead to varied performance while managing shared resources. With the globally scheduled schemes, results derived for uniprocessors are not applicable due to frequent task migrations. Most importantly, the Response Time Analysis is difficult to apply in a global scheme due to the lack of a precisely measured critical instant for an given task set, which specifies the worst-case alignment of task releases and represents the maximum load of the system [25]. As for the partitioned approach, the major challenge is to allocate tasks into each processor, which is a bin-packing NP-hard problem so that heuristic approaches must be applied [25]. The traditional task allocation schemes assign tasks based on the task utilisation, such as the Worst-Fit and the First-Fit algorithms [12, 32]. However, such task mapping approaches can not benefit resource sharing as tasks are allocated without the knowledge of resource usage, which can lead to a large number of global resources with prolonged waiting time. Recently, several resource-aware task allocation schemes have been proposed, which attempt to reduce the number of global resources so that the impact of resource sharing can be reduced [62, 81]. Yet many of the resource-aware algorithms can only be applied to their designated protocols, which limit the range of applications. Therefore, generic resource-orientated task allocation algorithms that are independent from resource sharing protocols (i.e., can be adopted with any resource sharing protocol assumed) are more desirable in general.

Another major impact on the performance of resource control is the priority ordering. As indicated in [25], optimal priority assignment algorithms are not available for multiprocessor schedulability analysis. The Deadline Monotonic Priority Ordering (DMPO) is optimal for uniprocessors, but its optimality is not extended to the multiprocessor case. In Section 4.2.1, a formal proof is presented to prove that the DMPO is not optimal in multiprocessor systems with the presence of shared resources. On the other hand, the Optimality Priority Assignment (OPA) by Audsley [9] is available, which is able to search for a feasible solution as long as there exists one. However, this algorithm can only be applied to its compatible analysis, such as the original analysis of resource sharing protocols [11, 27]. For the schedulability tests where the response time of a given task depends on the response time of potentially all other tasks in the system (e.g., the analysis in [106]), Audsley's algorithm is not applicable due to its limitations in nature [38]. Therefore, a new search-based priority ordering that is compatible with the schedulability analysis where DMPO is not optimal and OPA is inapplicable could also improve the schedulability of multiprocessor real-time systems with shared resources.

Combining the discussions above, this thesis aims to propose a Flexible Multiprocessor Resource Sharing framework (FMRS) that provides feasible solutions (if they exist) to resource sharing, task allocating and priority ordering issues for partitioned multiprocessor systems with shared resources under the new schedulability analysis, which supports systems with the presence of multiprocessor resource sharing protocols. For a given system, this framework aims to designate an appropriately chosen resource sharing protocol for each shared resource and to assign an allocation and a priority to each task that can lead to a schedulable system (if achievable).

1.3 Thesis Hypothesis

This thesis addresses the hypothesis that:

> *With shared resources, the schedulability of a multiprocessor real-time system can be undermined due to the considerable amount of blocking time. Such schedulability penalty can be reduced by adopting (i) a combination of appropriately chosen resource sharing protocols, where each protocol only controls certain resources; (ii)*

new resource-orientated task allocation schemes with full knowledge of the usage and characteristics of each resource; and (iii) a search-based priority assignment that is compatible with schedulability tests where the Deadline Monotonic Priority Ordering is not optimal and Audsley's Optimal Algorithm cannot be applied. The decisions of which resource sharing protocols, task allocation scheme and priority ordering algorithm that can lead to a schedulable system are made off-line by a genetic algorithm.

1.4 Success Criteria and Contributions

To facilitate the assessment of the work proposed in this thesis, a set of success criteria (SC) are given. In order to support the thesis hypothesis given in Section 1.3, the following need to be developed:

SC-1 A new schedulability analysis framework that can be applied to systems with the presence of multiple resource sharing protocols, which includes a response time analysis that can provide more accurate results than that of their original analysis, and a pluggable run-time overheads analysis that takes the run-time costs from both the underlying operating system and the resource sharing protocols into account.

SC-2 Resource-oriented task allocation schemes that are independent from the resource sharing protocols, where each task allocation scheme assigns tasks to processors based on certain characteristics of the shared resources, such as the length of critical sections and the degree of resource contention.

SC-3 A new priority ordering algorithm that inherits the philosophy of the OPA algorithm i.e., search-based, but is fully compatible with the schedulability tests where DMPO is not optimal and OPA cannot be applied, such as the one in [106] and the new analysis framework in SC-1.

SC-4 A flexible multiprocessor resource sharing framework that takes a system as the input, and aims to search for a schedulable solution (with the new schedulability analysis in SC-1) of resource sharing, priority ordering and task allocating issues to the given system, which include a combination of locking protocols to control each resource in the system, a task allocation scheme that can benefit resource sharing and a feasible priority ordering decided via examining all the candidate solutions provided by this framework.

SC-5 An evaluation with evidence that the resource sharing framework proposed in SC-4 demonstrates at least equal or better schedulability than that of the typical real-time resource control approaches, where one resource sharing protocol is adopted to manage all shared resources in a system with the existing task allocation and priority ordering approaches applied.

In addition to the success criteria listed above, additional contributions have been made during the work of this thesis, listed below:

1. A genetic algorithm-based approach to search for feasible solutions among the candidate resource sharing, task allocation and priority ordering solutions of the multiprocessor resource sharing framework in SC-4.

2. A formal proof that the DMPO is not optimal in multiprocessor systems with the presence of blocking.

3. A performance comparison of the candidate multiprocessor resource sharing protocols [27, 48, 100] of the resource control framework. The candidate protocols are determined in Section 3.1 with the schedulability analysis supported in Section 3.2.

4. An extension to the new schedulability analysis framework in SC-1 with the support of the heterogeneous and nested resource accesses.

5. An investigation towards the correctness and efficiency of a helping-based multiprocessor resource sharing protocol [27] in fully partitioned systems. This protocol is one of the candidate protocols of the resource control framework with details presented in Section 2.5.9.

6. An implementation of the candidate multiprocessor resource sharing protocols in a real-time operating system named Litmus$^{\text{RT}}$ [19, 30].

7. An evaluation of the run-time overheads of the candidate resource sharing protocols under Litmus$^{\text{RT}}$.

1.5 Thesis Outline

The rest of the thesis is organised as follows:

Chapter 2 Presents a detailed review of the concepts and facilities of both uniprocessor and multiprocessor real-time systems, including the real-time system model, task scheduling approaches, resource control technology and schedulability analysis.

Chapter 3 Determines the candidate resource sharing protocols of the resource control framework and introduces a new schedulability test framework that supports systems with potentially all candidate resource sharing protocols working in collaboration. The materials provided in this chapter demonstrates that the thesis meets SC-1.

Chapter 4 Proposes three new resource-orientated task allocation schemes and a new search-based priority ordering algorithm that can benefit resource sharing in multiprocessor real-time systems and are independent from the resource sharing protocols and their schedulability tests. The materials provided in this chapter demonstrates that the thesis meets SC-2 and SC-3.

Chapter 5 Proposes the genetic algorithm-based Multiprocessor Resource Sharing Framework (i.e., the FMRS framework) that aims to provide feasible solutions of resource sharing, task allocating and priority ordering to multiprocessor real-time systems with shared resources. The materials provided in this chapter satisfies SC-4.

Chapter 6 Investigates the performance of the typical resource control approach and the new resource control framework on fully-partitioned systems with shared resources, and presents evidence that the FMRS framework developed in this thesis can outperform the typical multiprocessor resource sharing and task scheduling approaches. The materials provided in this chapter demonstrates that the thesis meets SC-5.

Chapter 7 Summarises the thesis, reviews the contributions of this work and presents the future work.

Chapter 2

Literature Review

This chapter provides a review of the basic concepts and previous works related to the research proposed in this thesis. The review firstly describes characteristics of the real-time task and system model, and explains the scheduling technology for both uniprocessor and multiprocessor systems. Then, the resource model and previous works on the resource sharing technology in real-time systems are discussed. Finally, the scope of the research proposed in this thesis is presented based on this review.

The literature related to the materials presented in this thesis can be broad. In the interest of brevity, this chapter focuses on presenting a top-level view that describes the context of our work. Detailed descriptions of a particular technique will be given later on when it is adopted in this thesis. In addition, as there exist many multiprocessor resource sharing protocols [25, 27, 106], it is not possible to review each of the existing protocols. This thesis provides descriptions of the major multiprocessor resource sharing protocols and then focuses on the FIFO spin-based ones (a major approach for managing shared resources in multiprocessor real-time systems [106]). The rationale of this decision is given later in Section 3.1 in details.

2.1 Real-time Task and System Model

This section describes the characteristics of real-time tasks and presents the system model assumed in the research of this thesis. In addition, the background materials and the fundamental concepts for real-time systems that are related to this thesis are explained.

2.1.1 Real-time Tasks

A real-time task refers to a single thread of control that is required to be finished within a pre-defined time interval in each release with predictable behaviour. To guarantee this property, a set of parameters are introduced to facilitate the release, execution and analysis of real-time tasks.

In this thesis, τ_x represents a real-time task with index x. For a given real-time task, say τ_x, it has a worst-case computation time C_x, a period T_x that indicates its release interval, a relative deadline D_x that represents the time that the task must be finished after being release, an unique priority $Pri(\tau_x)$ that can be assigned either statically before run-time or dynamically during execution, and a worst-case response time R_x that represents the time passed from the release of the task to the time that the task finishes its execution of the release. The utilisation of τ_x (denoted as U_x) is calculated by $\frac{C_x}{T_x}$. A deadline miss is identified where $R_x > D_x$ while meeting a deadline requires $R_x \leq D_x$. Table 2.1 summarised the notations of real-time tasks. Note, the above task model is presented without the presence of shared resources i.e., C_x is the pure worst-case computation time of τ_x without accessing any shared resources. The system model with shared resource is presented in Section 2.3.

Table 2.1: Notions for Real-time Tasks

τ_x	A given real-time task with index x.
C_x	Worst-case computation time of τ_x without accessing any shared resources.
T_x	Period of τ_x.
D_x	Deadline of τ_x.
R_x	Response time of τ_x.
$Pri(\tau_x)$	Priority of τ_x.
$U_x = \frac{C_x}{T_x}$	Utilisation of τ_x without shared resources.

Real-time tasks can have various activation patterns, where tasks can be released at a fixed time interval (i.e., periodically), or with a minimum interval (i.e., sporadically), or within an arbitrary interval of time (i.e., aperiodically). As the most generic and the domaint activation model [25], the general *sporadic task model* is assumed in this thesis, where tasks cannot be released within a minimum interval of time.

In addition, various constraints exist for deadlines, where the deadlines of tasks can (1) be equal to their periods (i.e., implicit deadlines), (2) be less than or equal to their periods (i.e., constrained deadlines) or (3) be arbitrary. To facilitate the schedulability analysis, deadlines are assumed to be constrained, where a task can generate a bounded set of sequential jobs during its lifetime but only one job can be executable at a time (i.e., $R_x \leq D_x \leq T_x$).

Finally, in this thesis, the index of a task represents its priority, where a higher priority value indicates a higher execution eligibility. For instance. $Pri(\tau_1) = 1 < Pri(\tau_2) = 2$ so that τ_2 will execute in preference to τ_1.

2.1.2 Scheduling in Uniprocessors

In real-time systems, the term *scheduling* represents a scheme that contains an algorithm to order the usage of the processor or processors in the system via co-ordinating the concurrent executions of tasks in the system in order to meet the temporal requirements [26]. In addition, scheduling algorithms usually work in collaboration with resource control protocols to manage tasks' behaviours when accessing shared resources to achieve bounded resource-accessing time. In this section, the scheduling policies for uniprocessor systems are summarised with those features most relevant to this thesis described in detail.

2.1.2.1 Non-Preemption, Preemptions and Deferred Preemptions

A scheduling scheme can be categorised as non-preemptive, preemptive or deferred preemption [50]. During run-time, a high priority task can be released while a lower priority task is executing. With the preemptive scheduling approach, the system will immediately switch to the high priority task, and the low priority task is preempted. However, with a non-preemptive scheduling policy adopted, the high priority task has to wait for the low priority task to finish before it can start executing. As for the deferred preemption, the low priority task can still execute for a specific period of time before it is switched by the system.

Compared to the non-preemptive and deferred preemption schemes, the preemptive approach is more responsive in the context of real-time systems. With the preemptive scheme, high priority tasks (which usually have a close deadline) are more likely to meet their deadlines as they can execute immediately in each release without the need to cope with the interference of low priority tasks. Therefore, this thesis focuses on the systems with the preemption-based scheduling schemes.

2.1.2.2 Uniprocessor Scheduling Algorithms

In real-time uniprocessor systems, various scheduling schemes exist to achieve schedulable systems (i.e., all tasks in the system are guaranteed to meet their deadlines), as summarised below:

Fixed Priority Scheduling: Fixed Priority Scheduling (FPS) is the most widely adopted scheduling policy for uniprocessor systems [63]. Under this policy, priorities of each task are statically assigned before run-time and are fixed during the entire lifetime of tasks. During execution, tasks are scheduled based on the order of their priorities and the current executing task is always the task that has the highest priority among all the executable tasks.

Earliest Deadline First: Earliest Deadline First (EDF) scheme is a dynamic scheduling approach, where the priorities of the tasks are determined based on their absolute deadlines during run-time [28]. With EDF adopted, the task with the closest deadline will be assigned with the highest priority. However, since EDF needs to compute priorities for all tasks at each scheduling point during run-time, this algorithm can lead to a complicated system with high run-time overheads. In addition, the priorities of tasks under EDF can only reflect the absolute deadlines of tasks. In contrast, the priorities in FPS can represent other task properties, such as the criticality of tasks.

Value Based Scheduling: To cope with the case where overload can occur (e.g., the total utilisation of tasks is too high so that the system cannot be schedulable), Value-based Scheduling (VBS) proposes an on-line scheduling scheme [52] that provides guaranteed execution opportunity to certain tasks. Under VBS, each task is assigned with a value that indicates the task's importance level. When overload occurs, the system will only allow the critical tasks to execute according to the assigned importance level with EDF scheduling. By adopting VBS, overloaded systems can generate a more valuable output than simply applying either FPS or EDF.

Least Laxity: The Least Laxity First (LLF) scheduling policy [64] proposes a dynamic scheduling approach where tasks are scheduled by their laxities, which denotes the workload of tasks or the slack of $D_x - C_x$ for a given task τ_x, assuming without shared resources. Under LLF, the task with the least laxity will be scheduled to execute. For instance, a system contains two tasks τ_1 and τ_2, where $C_1 = 20$ and $D_1 = 30$ while $C_2 = 5$ and $D_2 = 30$ respectively. Let L_x denote the laxity of τ_x. Accordingly to LL, τ_1 will execute first as it has less slack than that of τ_2 ($L_1 = 10$ while $L_2 = 25$ at time 0). Then the system dynamically computes the laxities of these two tasks with the passage of time. After 15 units of time, the laxities of the tasks become identical ($L_1 = L_2 = 10$ at time 15). Thus, after this point τ_2's laxity is less than that of τ_1 so that the system will switch to τ_2. This procedure repeats until all the tasks finish their executions. As with EDF, LLF can incur considerable run-time overheads as it needs to compute the laxities of all the tasks in the system during each scheduling point. In addition, such overheads can become significant in the case where there are two or more tasks with similar laxities so that system needs to switch between these tasks frequently [83].

The above briefly reviews the major scheduling approaches in uniprocessor real-time systems. Among these scheduling schemes, FPS is the most commonly adopted approach and the dominant scheduling scheme for real-time systems [26]. In addition, most of the existing resource sharing protocols can be directly applied to FPS [37]. Accordingly, this thesis aims at systems where the Fixed Priority Preemptive Scheduling (FPPS) is adopted.

2.1.3 Priority Assignments

With FPPS assumed, the priorities of tasks must be assigned prior to run-time. Based on the survey conducted in [38], this section provides a review of the priority assignment rules for various application characteristics.

2.1.3.1 Rate Monotonic Priority Ordering

The Rate Monotonic Priority Ordering (RMPO) proposed in [73] is an optimal priority ordering algorithm for FPPS with sporadic tasks and implicit deadlines (i.e., $D = T$). As defined in [38], a priority ordering P is said to be *optimal* with respect to a task model, a fixed priority scheduling algorithm, and a schedulability test, if and only if every set of tasks that is compliant with the task model and is deemed schedulable with the scheduling algorithm G by schedulability test S with some priority assignments is also deemed schedulable under algorithm G by test S using policy P.

With RMPO adopted, tasks are assigned with a priority in the inverse order of the periods, where the task with the shortest period is assigned with the highest priority. The intuition of this approach is that the tasks that have frequent demands on the processor should be regarded as more urgent and thus have higher priorities. However, this assignment is only optimal for tasks with periods that are equal to their deadlines.

2.1.3.2 Deadline Monotonic Priority Ordering

Deadline Monotonic Priority Ordering (DMPO) is proposed in [65] via generalising RMPO to provide an optimal priority assignment for sporadic tasks with constrained deadlines (i.e., $D \leq T$). The DMPO is similar with RMPO but it assigns priorities to tasks in the inverse order of deadlines rather than periods, where the task with shortest deadline has the highest priority. Notably, DMPO remains optimal with the presence of shared resources managed by either the Stack Resource Policy or the Priority Ceiling Protocol on uniprocessors [16] while the Deadline Minus Release Jitter Monotonic Priority Ordering is optimal with the presence of release jitters [112]. However, the optimality of DMPO can be undermined with minor changes to the system, such as with the presence of offset release times or arbitrary deadlines [63, 65]. In addition, whether its optimality remains for multiprocessor systems with the presence of shared resources is unproved [16].

2.1.3.3 Audsley's Optimal Priority Assignment

The Audsley's Optimal Priority Assignment (OPA) developed in [9] proposes a sophisticated priority assignment approach. This algorithm is proved to be optimal for a wider range of application semantics than that of DMPO, such as systems with offset release times [8], arbitrary deadlines [101] and non-preemptive scheduling [50]. The pseudo code cited from [38] for OPA's algorithm is described below.

By giving a set of tasks with priorities unassigned and a compatible schedulability analysis (say S) that is applicable to OPA, this algorithm guarantees that a schedulable priority ordering (if there exist one) can be found according to analysis S. The algorithm starts from the lowest priority level and tests each unassigned task (say τ_x) to check whether τ_x is schedulable by assuming all other unassigned tasks have a higher priority. If τ_x can be schedulable at a given priority level, it is assigned with this priority. The algorithm then moves on to the next priority level and tests the rest of the unassigned tasks. The algorithm returns a schedulable solutions if each task is assigned with a priority. If no tasks can be schedulable at a given priority level, the algorithm is finished with no schedulable solution being found. The order that which unassigned task should be checked first at each priority level is not specified.

```
for each priority level $Pri$, lowest first {
   for each unassigned task $\tau_x$ {
      if ( $\tau_x$ is schedulable at priority $Pri$ according to a compatible
          schedulability test S with all unassigned tasks assumed to
          have a priority higher than $Pri$ ) {
            assign $\tau_x$ with priority $Pri$;
            break (continue outer loop);
      }
   }
   return unschedulable;
}
return schedulable;
```

Compared to searching through all possible priority orderings (For n tasks, it requires $n!$ calculations to test S), OPA can significantly reduce the number of calculations required, which is $n(n+1)/2$. However, applying OPA incurs the limitation that a compatible schedulability test must be supported. In [36], three conditions that are both necessary and sufficient for OPA to deliver optimal priority ordering with a given schedulability analysis S are presented with detailed proof, while violating any of the conditions can break the optimality of OPA or even cause this algorithm to be inapplicable. The conditions are cited from [36], as shown below. The term *independent properties* here refers to the task properties that are independent from priority (i.e., properties that cannot be affected by changes of priority), such as C, T and D.

Condition 1: "The schedulability of a task τ_x may, according to test S, depend on any independent properties of tasks with priorities higher than $Pri(\tau_x)$, but not on any properties of those tasks that depend on their relative priority ordering."

Condition 2: "The schedulability of a task τ_x may, according to test S, depend on any independent properties of tasks with priorities lower than

$Pri(\tau_x)$, but not on any properties of those tasks that depend on their relative priority ordering."

Condition 3: "When the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority cannot become unschedulable according to test S, if it was previously schedulable at the lower priority. (As a corollary, the task being assigned the lower priority cannot become schedulable according to test S, if it was previously unschedulable at the higher priority)."

With the development of OPA, several modifications to this algorithm are developed to provide further optimized priority ordering with the basic rationale of OPA but from different metrics, such as minimising the number of priority levels [9], or minimising the lexicographical distance [34]. However, the discussion towards these algorithms is out of the scope of this thesis as none of them assumed the presence of multiprocessor with shared resources. In [36], a detailed description and discussion of the modifications to OPA are provided.

2.1.3.4 Robust Priority Assignment

Despite that OPA can provide an optimal priority ordering solution in a wide range of application semantics, it has a disadvantage that the algorithm does not specify which task should be assigned at a given priority level, assuming there exist more than one schedulable tasks with that priority. Such an approach can result into a system that is merely schedulable, which is fragile to minor changes of task parameters, unexpected interrupts or execution budgets overrun [36]. To address this concern, the Robust Priority Assignment (RPA) was developed in [35] with an approach to specify the exact task that should be assigned with the priority at each priority level.

In RPA, an interference function $E(\alpha, w, i)$ is introduced to model the amount of potential interference at each priority level, where α is a scaling factor to reflect the variability of interference, w indicates the time interval that the interrupts can occur and i denotes the priority level that is affected by the interference. Assuming a given system with an interrupt that can occur only once during the release of any task in the system, which causes an interrupt handler to execute for a certain amount of time, the additional interference for this system is simply $E(\alpha, w, i) = \alpha$, where α represents the indeterminate execution time of the interrupt handler. For such a system, the RPA algorithm aims to produce a priority ordering that can tolerate the maximum amount of the additional interference (i.e., the largest value of α that the system can cope with).

The RPA algorithm starts with the lowest priority level and requires $n(n+1)/2$ binary searches to find the maximum α for all priority levels. At each priority level, RPA checks the schedulability of all the unassigned tasks and calculates the maximum α value that each schedulable task can tolerate with. The starting value of the binary search is bounded by a lower limit of 0 and a higher limit of certain reasonable value based on the interference function, where the upper limit is doubled on each iteration of the binary search, if found to be schedulable.

At a given priority level and among the schedulable tasks, the task with the maximum α value will be assigned with the priority. The algorithm then iterates to the next priority level until all tasks are assigned with a priority, and then returns the robust priority assignment for the given system. If a schedulable priority ordering can be found, this priority assignment is able to cope with the amount of additional interference of the minimum α among all the assigned tasks while remains schedulable. The pseudo code of RPA's algorithm cited from [38] is described as follows:

```
for each priority level $Pri$, lowest first {
   for each unassigned task $\tau_x$ {
      determine the largest $\alpha$ among schedulable tasks at priority $Pri$
      by assuming that all unassigned tasks have higher priorities;

      if ( no tasks are schedulable at priority $Pri$) {
         return unschedulable;
      } else{
         assign the schedulable task with the max $\alpha$ with $Pri$;
      }
   }
}
return schedulable;
```

This algorithm is developed based on OPA so that it is also subject to the three conditions presented in Section 2.1.3.3. With a compatible schedulability test, RPA is proved to be optimal and can produce robust priority assignments that can tolerate more additional interference than that of DMPO [35]. However, as a search-based priority assignment algorithm (where priorities are assigned via testing the response time of tasks through each priority level), RPA is only applicable to its compatible schedulability tests while DMPO does not carry such a limitation due to its static priority assignment approach. In Section 2.6.2, a schedulability test that is not compatible with either OPA or RPA is presented with reasons described in details.

2.1.4 Schedulability Analysis

As mentioned in Section 1.1, real-time systems have a strict temporal requirement, where all the tasks in the system must meet their deadlines. To testify whether a given task set can be schedulable under a certain scheduling policy, schedulability analysis techniques are developed to provide a mathematical approach (i.e., by applying a set of equations) for calculating the schedulability of that system. In uniprocessor systems, two major approaches to analyse the schedulability of real-time systems are available for FPS or EDF systems in the form of utilisation-based schedulability test [73] or Response Time Analysis (RTA) [7].

According to [37], there are two important characteristics of a schedulability test: *sufficient* and *necessary*. The term *sufficient* indicates a system which passes the schedulability test is guaranteed that all deadlines will always be met while a *necessary* schedulability test means that failure of the test will eventually lead to deadline misses, at certain point during execution. A sufficient and necessary test is termed as the *exact* schedulability test, which is optimal. As stated in [25], the utilisation-based analysis is not an exact test and can only produce the "yes or no" answer to the schedulability of a given system while RTA is proved to be exact and is able to calculate the worst-case response time of each task. Therefore, as the commonly adopted and the dominant schedulability test [25], this thesis focuses on the Response Time Analysis and its modifications for analysing systems with shared resources. This section describes the techniques of the simplest form of RTA for uniprocessor systems with FPPS assumed and no shared resources.

The Response Time Analysis contains two stages: (1) calculates the worst-case response time of each task via an analytical approach and (2) compares the response time of each task with its deadline. With RTA applied, the worst-case response time is calculated by Equation (2.1), where B_i is the total blocking time incurred by τ_i and $\mathbf{hp}(i)$ returns a set of tasks with a priority higher than that of τ_i.

$$R_i = C_i + B_i + \sum_{\tau_h \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil C_h \tag{2.1}$$

Assuming that each task in the system is assigned with an unique priority, the response time of a given task τ_i is determined by the worst-case execution time, the total blocking time and the interference from higher priority tasks. Even without the presence of shared resources, a task can incur blocking due to the non-preemptive (NP) sections of the underlying RTOS, where a task is prevent from executing due to the system executing a NP section. As such blocking can occur only once during the release of a real-time task, the blocking variable can be simply bounded by the maximum length of the non-preemptive sections of the underlying RTOS, as shown in Equation (2.2), where $\hat{b}$ denotes such maximum length of the NP sections of the RTOS.

$$B_i = \hat{b} \tag{2.2}$$

Function $\sum_{\tau_h \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$ calculates the total interference that τ_i can incur from high priority tasks due to preemptions. For each high priority task, the number of times it can be released during the release of τ_i is determined iteratively via $\left\lceil \frac{R_i}{T_h} \right\rceil$. With an initial response time of C_i, this equation computes the total amount of interference and updates the response time of τ_i until a fixed point is reached.

This simple RTA provides the fundamental technique for analysing a real-time system and derives various forms of schedulability tests to support the analysis of more complicated scenarios, such as systems with the presence of blocking or abitrary deadlines [25]. In Sections 2.4 to 2.6, the RTA-based analysis with the collaboration of resource sharing protocols for both uniprocessor and multiprocessor systems are described in detail.

2.1.5 Summary

This section describes the basic characteristics of real-time tasks, and presents the fundamental concepts and techniques in uniprocessor real-time systems, which includes the real-time system model, scheduling schemes, priority assignments and the schedulability analysis. As described, each of the reviewed techniques is supported by various approaches with different features. Based on the discussion in this section, the research scope of this thesis is narrowed down to the real-time systems with:

- Sporadic task model with constrained deadlines.
- Fixed priority preemptive scheduling.
- RTA-based schedulability analysis.

2.2 Multiprocessor Real-time Systems

As described in Section 1.1.1, in multiprocessor real-time systems, one major challenge is to schedule tasks on multiple processors. This section provides a review of the basic concepts of multiprocessor architectures and the scheduling techniques for real-time multiprocessor systems.

2.2.1 Multiprocessor Architecture

On multiprocessor platforms, several multiprocessing architectures exist according to the features of processors and their memory access model, where each of the multiprocessing architectures is suitable for certain scenarios, categorised as follows [58]:

Symmetric Multiprocessing: Symmetric Multiprocessing (SMP) refers to a multiprocessor architecture where all the processors are homogeneous and share a single main memory under the control of a single operating system. In SMP, each of the processors has full access to all the shared resources via a system bus, such as network and input/output (I/O) devices. From the viewpoint of the memory access model, this architecture is also termed as the Uniform Memory Access (UMA) model, where all the processors use the same memory (i.e., the main memory) and have an identical memory-accessing time. Nowadays, SMP is the most commonly adopted architecture for multiprocessor real-time systems.

Asymmetric Multiprocessing: An Asymmetric Multiprocessing (AMP) system can contain heterogeneous processors with various execution rates, where a task may needs 12 units of time of finish on processor 0 while only requires 10 units of time on processor 2. In AMP systems, each processor can be treated differently with specified duties assigned (e.g.

a processor may be required to only run the operating system while another processor is for manipulating I/O devices only). In addition, a group of processors in AMP usually have its designated memory space, which indicates the non-identical memory-accessing time to all memories. Such a memory access model is referred as the Non-Uniform Memory Access (NUMA) model. Typically, the AMP architecture are adopted in servers.

Clustered Multiprocessing: Clustered Multiprocessing is usually applied in large supercomputers with distributed systems. In such a system, processors are assigned with their local memory and not all memories are accessible by all processors. The communication between processors from different clusters is performed via a network.

As the most commonly applied real-time architecture, the SMP systems is usually assumed in the research of real-time systems while the AMP systems suffer from the analysis issue and the clustered Multiprocessing is barely considered in the real-time resource sharing domain [25]. Therefore, this thesis focuses on systems with the SMP architecture adopted, where all the processors are identical and have the same accessing time to a single shared memory.

2.2.2 Multiprocessor Scheduling Algorithms

Besides the issues considered by the scheduling policies on uniprocessor systems, scheduling schemes on multiprocessor systems also need to address the task dispatching issue, which is the problem of the placement of tasks to processors during execution (i.e., the decisions of which task should execute on which processor at a given time). To address this concern, various scheduling schemes for multiprocessors are proposed with different strategies to dispatch tasks to processors during the execution of the system, summarised as follows:

Global Scheduling: Global scheduling applies a dynamic task allocation approach, where the decisions of task mapping are made during run-time and can be changed dynamically [13]. With global scheduling, the system maintains a single logical global run queue that contains all the executable tasks in the system. When a task becomes executable, it is dispatched immediately to an idle processor (if there exist any). Otherwise, the task still has the chance to execute via preemptions if it has a higher priority than any of the currently executing tasks (assuming FPPS is applied). The unique feature of global scheduling is that it allows tasks to migrate between all the processors in the system so that a waiting task can migrate to an idle processor (if any) or a processor with a lower priority task executing on, instead of waiting for the executing task to finish in the current processor. However, with global scheduling, most of the matured uniprocessor real-time techniques cannot be applied as they do not consider task migrations.

Fully-Partitioned Scheduling: With fully-partitioned scheduling, each task in the system is statically allocated to a processor before run-time and is fixed into the designated processor during its entire lifetime [45]. Unlike global scheduling, migrations are strictly forbidden in this scheduling policy. With fully-partitioned scheduling, a multiprocessor system can be divided into several uniprocessor systems to a certain extent so that matured uniprocessor real-time techniques can be applied, such as the RTA equations. However, adopting this approach requires additional task allocation solutions to statically assign tasks into each processor before run-time.

Semi-partitioned Scheduling Compared to global scheduling, the total utilisation of a system under fully-partitioned scheduling can be relatively low due to the static task allocation approach. Thus, to preserve the advantage of fully paritioned system and to further increase the system utilisation, the semi-partitioned scheduling is proposed [24, 59], which also requires task allocations prior to run-time. However, unlike the fully-partitioned approach, the semi-partitioned scheduling allows some tasks to migrate to a pre-determined processor under certain situation (e.g., running out of budget on the current processor).

Clustered Scheduling The clustered scheduling [95] is proposed as trade-off between the global and partitioned scheduling schemes. With this scheduling policy, processors are divided into several groups (i.e., clusters), and tasks in each cluster is scheduled by the global approach so that a task can migrate to any of the processors in its cluster. In addition, the fully-partitioned approach is applied to clusters, where tasks are not allowed to migrate to processors belonging to other clusters.

Compared to other multiprocessor scheduling schemes, the fully-partitioned scheduling is supported by more matured schedulability tests (i.e., is fully compatible with RTA) and supports the majority of the existing resource sharing protocols [37]. Therefore, this thesis aims at fully-partitioned scheduling schemes, where each partition only contains one processor. However, as we shall see some limited forms of migration may be supported during a resource control protocol.

2.2.3 Task Allocation Schemes

With fully-partitioned dispatching policy assumed, a task allocation scheme must be applied before run-time to statically allocate tasks into each processors without overloading any processor, which is proved to be a bin-packing NP-hard problem with no optimal solutions available [66]. Thus, the heuristic approaches for the bin-packing problem are usually applied to map tasks into processors [37], summarised as below.

- *Worst Fit* (WF): The WF scheme considers all partitions and allocate a task to the partition with the minimum total utilisation.
- *Best Fit* (BF): In contrast to the WF scheme, the BF scheme considers all partitions and allocates a task to the processor that will have the minimal remaining capability (i.e., the maximum utilisation) after this allocation.
- *First Fit* (FF): The FF scheme considers each partition in the index order while allocating each task and assigns the task to the first processor where it can be fitted into.
- *Next Fit* (NF): For the first task, the Next Fit starts searching from the first processor. Then this algorithm searches from the last allocated processor to find the feasible processor for each unallocated task.

In addition, to facilitate allocating, tasks are usually sorted by their utilisations before being mapped into processors by any of the above schemes. Thus, a complete task allocation approach also specifies the task ordering approach. For instance, the WFD approach indicates that tasks are sorted by utilisation non-increasingly and are allocated by the WF method while the NFI algorithm orders the tasks by utilisation non-decreasingly and are mapped via the NF approach.

The heuristic approaches are well-practised with independent tasks, where the response time of a given task is not affected by remote tasks (i.e., tasks on other processors). While allocating a given task τ_x, the RTA equation can be applied to testify whether this and other assigned tasks on the examined partition can be schedulable. If yes, τ_x is assigned to that partition. Otherwise, the algorithm will try to find the next feasible partition based on one of the above approaches. With all tasks assigned with a partition, the heuristic approaches can return a schedulable system. According to [25], with the precondition that the utilisation of each task is lower than 0.5, the utilisation of each partition can reach to 0.63 while remaining schedulable by using the First-Fit task allocation scheme in FPPS systems with RMPO adopted.

2.2.4 Summary

This section presents the basic knowledge of multiprocessor platforms and the concepts for scheduling multiprocessor real-time systems. Based on the discussion, this thesis focuses on the systems with:

- Symmetric multiprocessor architecture.
- Fully partitioned scheduling scheme.

Combining the reviews given in Sections 2.1 and 2.2, the background knowledge and the basic concepts of real-time systems on both the uniprocessor and multiprocessor platforms are presented. From Section 2.3, the concepts, techniques and the related works of resource sharing in real-time domain will be described.

2.3 Resource Sharing Model

In this section, the basic concepts in resource sharing and the fundamental synchronisation approaches are described. Then, reasons are given for the targeted synchronisation approach in this thesis with its issues and concerns described in detail.

2.3.1 Shared Resources

The term *resource* can refer to both the hardware resources such as an I/O device or a processor; and the software resources, such as a single variable, an array or more complicated data structures. With multi-tasking, two or more tasks may request *exclusive access* to the same resource (i.e., a shared resource) simultaneously. The code related to a shared resource is referred as a *critical section*, where the executions must be protected to guarantee the data consistency. Shared resources that are accessed from one processor (e.g., the ones in a uniprocessor system) are referred as *local resources.* In multiprocessor platforms, resources can be accessed from multiple processors in parallel, and are termed *global resources.*

Without proper protections of critical sections, concurrent requests to a shared resource can cause race conditions due to unexpected data updates, and hence, results in corrupted data. For instance, two tasks (τ_1 and τ_2) are performing incremental operations concurrently to a single integer variable with an initial value of 0, where each task will perform the operation 5 times so that the expected output should be 10. However, without synchronisation to the critical section (i.e., the increment operation to the variable), the actual outcome can be less than 10. This is because at the same time, both tasks read the value of the variable (say 5 at this time) and perform the increment operation so that the value 6 is written back in memory by both tasks. Thus, the variable is only incremented once by two incremental operations, which is a typical race condition.

In this thesis, the notation r^k denotes a shared resource with the index k. A task τ_x can issue N_x^k number of requests to r^k during one release. For each resource r^k, c_x^k denotes the worst-case execution time when τ_x accesses r^k. In this thesis, the worst-case execution time of r^k is assumed to be identical for each task i.e., homogeneous cost for executing each resource. Thus, c^k is applied to denote the critical section length of r^k. This assumption is not fundamental but eases presentation. The influence of considering heterogeneous access cost to shared resources is addressed Appendix A. In addition, two functions that describe the resource-usage of tasks introduced in [27] are applied to facilitate analysing systems with shared resources, where function $F(\tau_x)$ gives a set of resources that are used by τ_x while function $G(r^k)$ returns a set of tasks that access r^k. Table 2.2 summarises the notations described in this section.

In addition, with the presence of shared resources, the total worst-case computation time of a task τ_x must be extended to also include the time that τ_x spends on executing each shared resources in $F(\tau_x)$. Therefore, with the

Table 2.2: Notions of Shared Resources

r^k	A given shared resource with index k.
c^k	Critical section length of r^k.
N_x^k	The number of requests issued from τ_x to r^k in one release.
$F(\tau_x)$	The set of resources that are accessed by τ_x.
$G(r^k)$	The set of tasks that request r^k.

presence of shared resources, the total worst-case computation time of τ_x is bounded by $C_x + \sum_{r^k \in F(\tau_x)} N_x^k \cdot c^k$, where C_x denotes the pure computation time of τ_x without accessing any shared resources and $\sum_{r^k \in F(\tau_x)} N_x^k \cdot c^k$ denotes the time that τ_x spends on executing each required resource. Accordingly, the utilisation of τ_x now should be calculated by $U_x = \frac{C_x + \sum_{r^k \in F(\tau_x)} N_x^k \cdot c^k}{T_x}$. This utilisation calculation will be assumed for the rest of the thesis.

2.3.2 Synchronisation Approach

In real-time systems, various synchronisation approaches can be adopted to eliminate race conditions so that the data integrity can be guaranteed, such as the classic lock-based approach and non-blocking methods.

With the lock-base approach, each critical section is protected by a designated lock, where the access to a critical section is only permitted with the corresponding lock acquired. If the lock is occupied, then the requesting task is blocked until the lock is available.

As a major synchronisation approach, various locking primitives are available, such as mutex locks, monitors and semaphores [104]. These locking primitives can be categorised as either being suspension-based or spin-based locks according to the task behaviours while waiting for the lock [22], as summarised below:

Suspension-base locks: If a task requests a lock that is already occupied, the task will give up the processor and become idle until the lock is available. Typically, the waiting tasks are placed into a priority-ordered queue during the period of blocking. Once the lock is released, the task at the head of the queue (i.e., with the highest priority) can become active and acquire the lock.

Spin locks: Tasks with the spin-based locks will busy-wait (spin) instead of becoming idle when the required lock is not available. While spinning, the task keeps executing and continuously checks whether the requested lock is available. Tasks with spin locks are usually served by the First Come First Serve (FCFS) order, but can also be served by the priority order. Spin locks are largely adopted at the kernel level.

In addition, the alternative non-blocking methods including the wait-free and lock-free algorithms are also available, which can prevent race conditions in concurrent executions without using locks [37].

A lock-free algorithm [2] allows tasks to access a resource immediately but requires a copy of the original data before performing any operations. After all the operations associated with the critical section are performed, the task checks whether there exist any conflicts during this access (e.g. unexpected data updates by other tasks). If yes, then the calculations of this access is discarded and the task will re-access the resource with a new copy of the data. By doing so, this algorithms can prevent race conditions, but involves looping due to unsuccessful resource accesses. The wait-free algorithm is an enhanced form of the lock-free algorithm, where neither locks or retry loops are required [99]. However, the wait-free algorithm requires multiple copies of the original data, such as the four-slot mechanism, where four copies of the original data are required to provide independent executions of a single reader and writer [96].

Although there exist researches towards the non-blocking methods for real-time systems [53], such approaches lack effective schedulability analysis [68] and usually require extra memory space, which is usually limited in embedded real-time systems [105]. In addition, as stated in [37], the locking-based protocols are well-accepted and are the dominant approaches for controlling shared resources in real-time systems. For example, protocols such as the Priority Ceiling Protocol and the Stack Resource Policy provide the most appropriate mutual exclusive access to shared resources and are analysable by the RTA equations. These matured lock-based protocols are well-practised and originated the researches for resource sharing solutions in multiprocessor real-time systems.

Based on the above discussion, this thesis focuses on the lock-based approach and aims to propose a generic resource control solution for multiprocessor real-time systems.

2.3.3 Deadlocks and Livelocks

With the lock-based synchronisation approach assumed, the well-known issue of the deadlock must be avoided to guarantee the correctness of the system. Deadlocks can occur when tasks require multiple shared resources in a nested fashion (i.e., nested resource accesses), where two or more tasks are blocked since they both require the resource that is held by each other. Thus, these tasks will never make progress as none of them can get the requested resource. In real-time system, avoiding deadlocks is an essential requirement of the resource sharing protocols.

Another concern with locks is the livelock problem, where each task is waiting (e.g., spinning) for the other to acquire the lock so that none of them gets the lock with no progress being made. In real-time systems, locks are usually served with a strict serving order, such as the priority order and the FCFS order. In addition, the resource-accessing time of each task must be bounded to achieve predictable systems. Thus, the livelock and deadlock problems are prevented in real-time systems by the pre-defined resource serving order. Assuming a schedulable system, each task is guaranteed with the chance to acquire the lock within a bounded period of time.

2.3.4 Priority Inversion

Merely protecting the data integrity and preventing the locking problems are not sufficient to meet the requirements of real-time systems, where the tasks' behaviours while accessing shared resources must be predictable within a bounded resource-accessing time during each resource access. With resource locks adopted, tasks can incur additional delay due to accessing shared resources, which can cause priority inversions through various blocking effects.

The term *priority inversion* indicates the situation where a high priority task is waiting while a low priority task is executing. Consider an uniprocessor system that contains two tasks τ_1 and τ_{10}, where both tasks request the same shared resource r^1. Note that in this thesis, the priority of a task equals to its index and a higher index indicates a higher execution eligibility. If τ_1 is released first and acquires r^1, τ_{10} will be prevented from executing at the time where it request the resource (even if it can preempt τ_1 when being released). Under this case, τ_{10} suffers from priority inversion as it is blocked by a low priority task for accessing an unavailable resource with the blocking period of

c^1.

The priority inversion can become unbounded and can lead to unpredictable amount of blocking time. Consider the same example above, while τ_{10} is being blocked, another task τ_3 is released and preempts τ_1, which is allowed as τ_3 does not request r^1. Under this case, the blocking time of τ_{10} is increased to cope with the execution time of τ_3. In addition, if more tasks with such an intermediate priority are released during the blocking period of τ_{10}, the blocking time of τ_{10} can be further prolonged and can become unpredictable. Under this situation, τ_{10} is said to suffers from the unbounded priority inversion.

The priority inversion phenomenon cannot be completely eliminated due to the difficulty of controlling the time at which a given task can access a shared resource. However, the priority inversion must be bounded to achieve predictable blocking time of each resource access.

2.3.5 Blocking Effects

The real-time resource sharing techniques bound the priority inversion by examining each of the blocking effect that can occurred during the access to shared resources. Typically, there exist four types of blocking effects, as summarised below:

Local blocking: The local blocking occurs when a low priority task blocks a higher priority task on the same processor for accessing a resource. This blocking can occur due to both tasks requesting the same resource, or because the low priority task is executing with another resource non-preemptively or has its active priority boosted to a certain priority level to prevent preemptions [93]. With non-preemptive or the priority boosting approach, the high priority task is blocked at its arrival as preemptions are not allowed. Thus, this blocking is also refereed as the *arrival blocking*.

Transitive blocking: A task, say τ_x, can incur the transitive blocking when being blocked by a resource-holding low priority task, which in turn is preempted by a task with an intermediate priority. Th example used to illustrate the unbounded priority inversion in Section 2.3.4 is a typical form of the transitive blocking. In addition, transitive blocking can also occur with the presence of nested resources. A task, say τ_{10}, can incur transitive blocking if it is blocked by τ_5 due to resource accessing, which is in turn, being blocked by another task τ_1 as τ_5 is trying to access an nested resource that is held by τ_1. Thus, τ_{10} is blocked transitively by τ_1.

Push-through blocking: The push-through blocking occurs when an unrelated intermediate priority task becomes an "innocent victims" due to the competition of locks by other tasks, where a low priority task has its priority boosted to certain priority level due to accessing a resource. This blocking usually happens in systems where a priority boosting technique is applied.

Remote blocking: A task can incur the remote blocking in multiprocessor systems, where it is blocked by remote tasks for resource accessing. The remote blocking can also be in the form of any other blocking effects. A task can incur the *direct remote blocking* when being blocked directly by remote tasks for accessing a global resource; or be blocked *indirectly* by a local high priority task that is accessing a global resource, which in turn is blocked by remote tasks directly. In addition, the remote blocking can also occur in the arrival blocking, where a high priority task is blocked by a low priority task that is accessing a global resource.

To achieve predictable tasks' behaviours while accessing shared resources, the above blocking effects must be eliminated or at least bounded (and hence, the unbounded priority inversion can be addressed) to provide the predictable resource-accessing time in each resource access.

2.3.6 Summary

In this section, the basic concepts of resource sharing and the primitive synchronisation approaches for critical sections have been described. Based on the discussion in Section 2.3.2, this thesis focuses on the system with

- Homogeneous cost for executing a resource.
- The lock-based synchronisation approach.

With locks assumed, issues of applying locks to real-time systems are explained, such as the deadlocks, the priority inversion phenomenon and the various blocking effects. In the next section, the resource sharing protocols for uniprocessor real-time systems will be described, which address the above issues and can provide predictable resource-accessing behaviours.

2.4 Uniprocessor Resource Sharing Protocols

Resource sharing in uniprocessor systems is successfully managed by the matured uniprocessor resource sharing technology, which are well-accepted and practised for decades with several optimal resource sharing policies available [25]. In addition, the RTA equations were extended to support the analysis of FPPS systems with the resource sharing protocols adopted. This section reviews the major approaches of the resource sharing protocols for uniprocessor FPPS systems and the corresponding schedulability test.

2.4.1 Priority Inheritance Protocol

Although adopting the typical non-preemptive sections (i.e., simply disallow preemptions during the critical sections) can provide direct protection to resource-accessing tasks, this approach imposes extra blocking time to the unrelated high priority tasks. To address this concern, the Priority Inheritance Protocol (PIP) [93] was developed to propose a preemption-allowed approach for managing shared resources. PIP is summarised as follows:

- Each task in the system has a *base priority* and an *active priority*. The base priority is the priority that is assigned statically to the task while the active priority is the current priority level that the task is executing with.

- When a high priority task (say τ_2) requests a resource that is held by a low priority task (τ_1), τ_1 then inherits the priority of τ_2 and keeps executing until it releases the lock.

- Later on, if τ_3 is released and also requests the resource, τ_1 will update its active priority again to inherits τ_3's priority, which is the highest priority among all the tasks it blocks.

- If a task has its priority boosted for executing with a resource, its priority will be set back to its previous priority immediately when it releases the resource. The previous priority could either be a priority value inherited from another task due to resource accessing (in nested accesses) or its base priority.

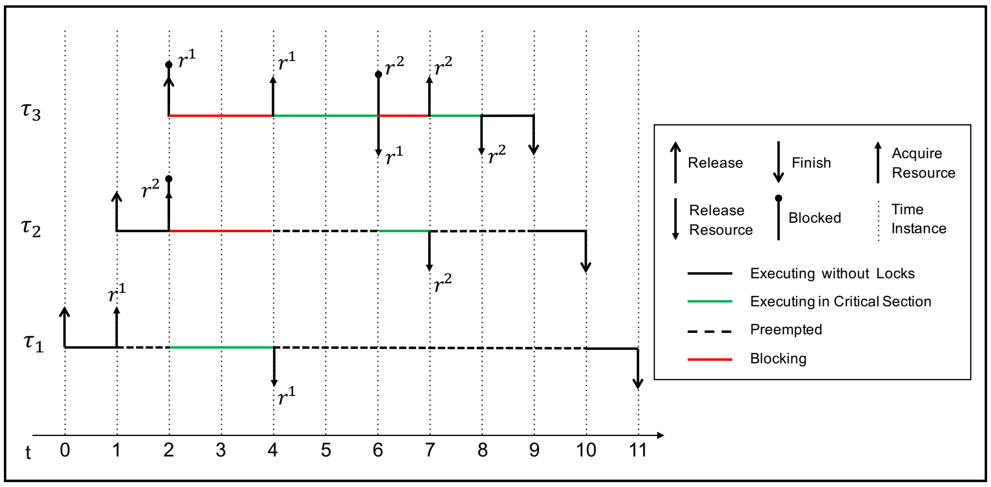


Figure 2.1: Example of the Priority Inheritance Protocol

The example shown in Figure 2.1 demonstrates tasks' behaviours while accessing shared resources with PIP applied, where notation t denotes the units of time. *For the ease of presentation, we assume that multiple events can occur at the same time.* For instance, a task can lock a resource immediately when being released at the same time. As shown in the figure, τ_1 is released at time $t = 0$ and then acquires r^1 at $t = 1$. However, it is preempted by τ_2 immediately as τ_1 now is executing with its based priority (i.e., $Pri(\tau_1) = 1$). At $t = 2$, τ_2 acquires r^2 but is then being preempted by τ_3, which requires both resources in non-nested sequential fashion. However, τ_3 incurs blocking after being released as it requests r^1 immediately, which is held by τ_1. Hence, τ_1 now inherits the priority of τ_3 and resumes its execution with r^1. Note that at this time, τ_1 also blocks τ_2 as it is executing with the highest priority level (now $Pri(\tau_1) = Pri(\tau_3) = 3$) in the system. After τ_1 releases r^1, its priority is restored so that it is preempted by τ_3 at $t = 4$. At $t = 6$, τ_3 finishes its execution with r^1, but is blocked again for requesting r^2 so that τ_2 raises its priority and starts executing the critical section. τ_2 releases r^2 at $t = 7$ so that τ_3 is resumed and starts executing with r^2. τ_3 releases r^2 at $t = 8$ and is then finished at $t = 9$. Then, τ_2 and τ_1 are finished at $t = 10$ and $t = 11$ respectively.

With PIP adopted, the low priority tasks can execute with a boosted priority when they blocks high priority tasks so that the high priority tasks will not suffer from the prolonged blocking due to preemptions from tasks with a intermediate priority (i.e., the unbounded priority inversion is prevented). In addition, the unrelated high priority tasks do not need to incur unnecessary arrival blocking due to the preemptive resource-accessing approach.

However, as illustrated by the example, a task under PIP can incur blocking more than once if it requests more than one resources held by different low priority tasks (see τ_3 in this example). In addition, this protocol cannot eliminate deadlocks as a task that holds a resource (say r^1) and requests an inner resource (say r^2) can be preempted by another task with a higher priority, which then locks r^2 but then request r^1.

2.4.2 Priority Ceiling Protocol

As described in Section 2.4.1, PIP has the issue of deadlocks and can cause several blocking to a resource-accessing task. These remaining issues motivated the development of the Priority Ceiling Protocol (PCP) [93]. In [90], two approaches for realising PCP are described, named as the Original Priority Ceiling Protocol (OPCP) and the Immediate Priority Ceiling Protocol (IPCP).

In OPCP, the notions of *resource ceiling priority* and *system ceiling priority* are introduced, where the resource ceiling for r^k (denoted as $Pri(r^k)$) indicates the highest base priority among all the tasks that require r^k while the system ceiling is the highest resource ceiling priority among all the resources that are currently being accessed. During execution, the system keeps tracking the system ceiling priority and updates its value during each lock acquisition and release. With OPCP adopted, a task can execute normally (according to FPS), but is allowed to acquire a lock only if it has an active priority higher than the current system ceiling. Otherwise, the task is blocked until it is eligible to acquire the resource.

In IPCP, the need to maintain the dynamic ceiling priority is removed. Instead, a task raises its active priority immediately to the corresponding resource ceiling priority each time it acquires a resource, and then executes the critical section with the boosted priority. After the task releases the resource, it restores its priority to the previous priority level. As stated in [90], both OPCP and IPCP have the identical worst-case behaviour. However, IPCP proposes a more elegant solution and can effectively reduce the implementation complexity and the number of context switches.

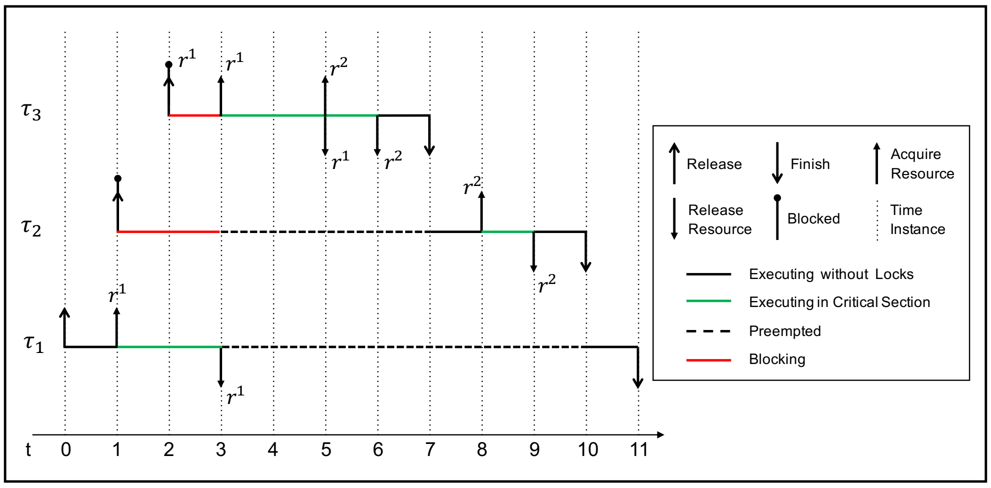


Figure 2.2: Example of the Immediate Priority Ceiling Protocol

To demonstrate that PCP can prevent the issue of multiple blocking in PIP, the example in Figure 2.1 for PIP is applied here with IPCP adopted, as shown in Figure 2.2. With IPCP adopted, the resource ceiling of both r^1 and r^2 is 3 as they are requested by τ_3 (i.e., $Pri(r^1) = Pri(r^2) = 3$). At $t = 0$, τ_1 is released and then acquires r^1 at time $t = 1$. Accordingly, it boosts its active priority immediately to the ceiling priority of r^1 so that its active priority now is 3. Thus, τ_1 blocks τ_2 at $t = 1$ and also blocks τ_3 at $t = 2$ as it is executing with the highest priority. At $t = 3$, τ_1 releases r^1 and restores its priority so that τ_3 is eligible to execute. Thus, it uses r^1 and r^2 and is finished at $t = 7$. Then τ_2 can acquire r^2 and execute. At last, τ_2 and τ_1 are finished at $t = 10$ and $t = 11$ respectively.

Compared to PIP, tasks with PCP can only incur one blocking in each release. Therefore, tasks under PCP can have a shorter response time than that of the tasks with PIP adopted. As shown in the examples, the response time of τ_3 with IPCP adopted (Figure 2.2) is 2 units of time shorter than that of τ_3 under PIP (Figure 2.1) due to the decreased blocking time.

In addition, PCP is a deadlock-free protocol by preventing the formation of the circular resource-requesting chain, where a task that is accessing shared resources cannot be preempted by another task that requests the same resources. Therefore, by addressing the issues with the lock-based synchronisation approach and by limiting the blocking time to only one critical section, PCP is an asymptotically optimal resource sharing solution in uniprocessor FPPS systems [19].

2.4.3 Stack Resource Policy

The Stack Resource Policy [11] (SRP) is proposed as an extension of PCP and can be adopted to both fixed-priority and dynamic priority scheduling schemes, such as FPS and EDF.

As with PCP, the system ceiling and resource ceiling priories are applied in SRP. However, this protocol introduces the notion *preemption level* to each task in the system based on the deadline monotonic scheme, where a task with a shorter relative deadline is assigned with a higher preemption level. With this static metric, SRP is able to work with dynamic scheduling policies. Accordingly, the value of the resource ceiling for each resource and the system ceiling are decided by the static preemption levels rather than dynamic priories.

In SRP, a task is allowed to execute only if it has a preemption level that is higher than the current system ceiling. With this approach, SRP postpones the executions of tasks that can be blocked later so that the tasks can share a same run-time stack. This is because the newly-arrived tasks at the top of the stack has a higher preemption level so that it can preempt the tasks in the lower level of the stack. Thus, with SRP, a task will either be blocked immediately after being released (i.e., arrival blocking) or incurs no blocking at all. Once a task starts executing, the resources required by the task are guaranteed to be available. Due to this property, SRP also successfully prevents the formation of deadlocks.

SRP can achieve the identical worst-case behaviour as that of PCP, where a task can be blocked only once during each release. In FPPS systems, the executions of tasks under SRP are similar to those of tasks with IPCP adopted. For instance, both protocols can lead to the same task executions in the system illustrated in Figure 2.2. At $t = 1$, τ_2 is released but cannot execute as its preemption level is not higher than the current system ceiling, where τ_1 is executing with r^1 and has the highest preemption level in the system. The same situation also occurs at $t = 2$, where τ_3 is blocked immediately after being released. At $t = 3$, τ_1 releases r^1 so that its preemption level is restored to the original value (i.e., the lowest level in the system) and is preempted by τ_3. Finally, all tasks are finished at the same time with that of IPCP adopted.

2.4.4 Schedulability Analysis of Uniprocessor Resource Sharing Protocols

As reviewed in Sections 2.4.2 and 2.4.3, PCP and SRP demonstrate identical worst-case behaviour and can bound the blocking time to one critical section. Therefore, these protocols can be analysed by the same RTA-based schedulability test, assuming the FPPS systems.

Recall Section 2.1.4, the response time of a given task τ_i is computed by Equation (2.1) with the assumption that tasks are running independently (i.e., without shared resources). However, as described in Section 2.3.1, with the presence of shared resources, the total worst-case computation time of τ_i must be extended to also reflect the time it spent on executing each shared resource, which is bounded by $C_i + \sum_{r^k \in F(\tau_i)} N_i^k c^k$.

Equation (2.3) gives the bounding of R_i for τ_i with the presence of shared resources, where function $F(\tau_i)$ gives a set of resources that are required by τ_i and N_i^k represents the number of requests issued from τ_i to r^k in each release (recall notations in Table 2.2).

$$R_i = (C_i + \sum_{r^k \in F(\tau_i)} N_i^k c^k) + B_i + \sum_{\tau_h \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil (C_h + \sum_{r^k \in F(\tau_h)} N_h^k c^k) \qquad (2.3)$$

In addition, with the presence of shared resources, variable B_i must be extended to reflect the potential arrival locking incurred by each task. With either PCP or SRP adopted, τ_i can incur arrival blocking if there exits a lower priority task requesting a resource with a ceiling priority that is equal to or higher than $Pri(\tau_i)$. If there exist multiple such resources, in the worst case, τ_i will incur blocking from the resource with the maximum critical section length as the arrival blocking can occur only once. Let $\hat{c_i}$ denotes the blocking that τ_i can incur during each release and τ_l denotes a lower priority task, Equation (2.4) gives the blocking time that τ_i can incur in each release due to resource sharing.

$$\hat{c_i} = max\{c^k | N_l^k > 0 \land Pri(r^k) \geq Pri(\tau_i)\} \qquad (2.4)$$

Further, as a task can only be blocked once upon arrival, it is blocked either by $\hat{c_i}$ or by the non-preepmitve section from the underlying real-time operating system ($\hat{b}$ in Equation (2.2)). Thus, the total blocking time τ_i can incur with the presence of shared resources in uniprocessor systems can bounded, as given in Equation (2.5).

$$B_i = max\{\hat{c_i}, \hat{b}\} \tag{2.5}$$

This concludes the schedulability test of uniprocessor systems with resources managed by either PCP or SRP. As stated in [25], this analysis is proved to be an *exact* schedulability test for uniprocessor systems. The notations applied in this analysis is summarised in Table 2.3. *Note, for the ease of presentation, the word "schedulability test" or "schedulability analysis" for a given real-time resource sharing protocol in this thesis indicates the analysis of the blocking time of the given protocol based on the RTA equations (see Section 2.1.4) under the FPPS systems.*

Table 2.3: Notations Applied in the Analysis of PCP and SRP

τ_i	A task that is currently been studied by the schedulability analysis.
τ_l	A task with a priority lower than that of τ_i.
$\hat{c_i}$	The arrival blocking incurred by τ_i.
$\mathbf{hp}(i)$	The set of tasks with a priority higher than that of τ_i.
$Pri(r^k)$	The resource ceiling priority of r^k.

2.4.5 Summary

This section provides a review of the major resource sharing protocols in uniprocessor FPPS systems. Among them, PCP and SRP provide the optimal resource sharing solutions with effective schedulability analysis supported. In addition, these protocols directly inspired the development of resource control protocols for multiprocessor systems, described in Section 2.5.

2.5 Multiprocessor Resource Sharing Protocols

In multiprocessor systems, resources can be accessed from multiple processors simultaneously (i.e., global resources), which cannot be managed by the matured uniprocessor resource sharing protocols. To bound the remote blocking for accessing global resources and to prevent deadlocks, many multiprocessor resource sharing protocols have been proposed with various synchronisation approaches. This section reviews the major resource sharing protocols on multiprocessors with their advantages, drawbacks and limitations.

2.5.1 Multiprocessor Priority Ceiling Protocol

The Multiprocessor Priority Ceiling Protocol (MPCP) derived from PCP was the first resource sharing protocol proposed for fully-partitioned multiprocessor systems with fixed priority scheduling [88].

As the forerunner of multiprocessor protocols, the initial approach of MPCP is to turn the multiprocessor resource sharing model into the uniprocessor case by designating a *synchronisation processor* among all *application processors*, where tasks can either execute with local resources or run independently on application processors, but must execute the global resources on the synchronisation processor. Note the tasks that do not require global resources can also execute on the synchronisation processor.

During each access to a global resource, the task is considered to "migrate to" the synchronization processor. However, in reality, tasks can be statically bounded to their designated processors while a thread for executing the critical section of each global resource (i.e., a remote helper that executes on behalf of other tasks) is created on the synchronization processor. While executing with a global resource, the host processor of the task is free so that a local lower priority task can start executing (if it does not request a global resource).

With this approach, the global resources can only be accessed from one processor (i.e., are converted to local resources) so that the system can be regarded as multiple independent uniprocessor systems, where IPCP is adopted in each processor. If multiple tasks issue requests to a global resource simultaneously, they are served in the decreasing order of the base priority, assuming each task has an unique priority. In addition, a priority-boosting mechanism is introduced for global resources to prevent the delay caused by the accesses to local resources, as follows:

- The base priority of a global resource must be higher than the highest based priority of all the tasks in the system (usually plus one).
- The ceiling priority of a global resource is the sum of the highest priority among all tasks in the system plus the highest priority of tasks that use it.

The intuition of this priority boosting mechanism is to give the highest execution eligibility to tasks accessing global resources so that the remote blocking can be bounded. As stated in [88], a task can incur blocking only once for accessing global resources. Once being blocked, the task is suspended on its host processor (i.e., the suspension-based approach) and is added into a priority-ordered queue until it is eligible to acquire the resource (i.e., has the highest priority among tasks in the waiting queue). Meanwhile, nested resource accesses (between either global to global resources, or local to local resources) is allowed in the initial version of MPCP with deadlocks avoided, as accessing an inner-nested resource can only lead to a non-decreasing active priority and a higher execution eligibility.

Figure 2.3 illustrates MPCP with a three-processor system, where tasks are pre-allocated to application processors P_1 and P_2 while P_s is reserved as the synchronization processor. For the ease of presentation, we assume that no tasks are allocated to P_s. The system contains 4 tasks with 4 shared resources, as given in Table 2.4, where resources are accessed in non-nested fashion with the given order by each task. r^3 and r^4 are global resources accessed by all tasks. According to the priority-boosting mechanism, the base priority of r^3 and r^4 should be 5 (i.e., the highest priority plus one) and Table 2.5 gives the ceiling priorities of the resources in Table 2.4. *Note that the accessing cost of a resources is not forced to be identical in the example. This statement remains for the following examples of resource sharing protocols. However, as claimed in Section 2.3.1, homogeneous access cost is assumed for all schedulability tests presented in this thesis for the ease of presentation.*

Table 2.4: Tasks in the Example System of MPCP

Task	Resource Usage	Partition
τ_1	r^1, r^3	P_1
τ_2	r^3, r^2	P_2
τ_3	r^4, r^1	P_1
τ_4	r^4, r^2	P_2

Table 2.5: Ceiling Priorities of Resources in the Example System of MPCP

Resource	Ceiling Priority
r_1	3
r_2	4
r_3	7
r_4	9

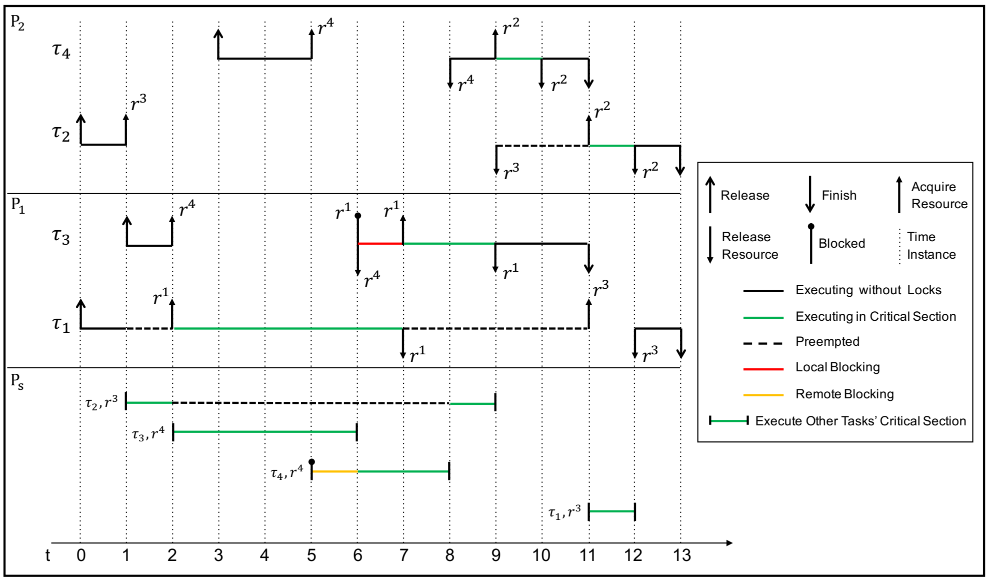


Figure 2.3: Example of the Multiprocessor Priority Ceiling Protocol

- At $t = 0$, τ_1 and τ_2 start to execute on P_1 and P_2 respectively. At $t = 1$, τ_2 requests r^3 so that it executes on P_s and locks r^3 with an active priority of 7.

- Meanwhile, τ_3 is released on P_1 and then requests r^4 at $t = 2$. As τ_3 has a higher active priority (i.e., 9) than that of τ_2, it preempts τ_2 on P_s and starts executing with r^4. While τ_3 is executing on P_s, τ_1 can resumes its execution on P_1 and locks r^1.

- At $t = 3$, τ_4 is released and then requests r^4 at $t = 5$. However, it incurs remote blocking as τ_3 is currently executing with r^4. Hence, τ_4 is suspended and is placed into the prioritised queue.

- τ_3 releases r^4 at $t = 6$ so that τ_4 starts its execution with the resource. Meanwhile, τ_3 incurs local blocking by 1 unit of time on P_1 due to requesting r^1, which is held by τ_1.

- At $t = 8$, τ_4 releases r^4 and resumes it execution on P_2 so that τ_2 continues with r^3 on P_s. τ_2 returns to P_2 at $t = 9$ after released r^3.

- At $t = 11$, both τ_4 and τ_3 finish their executions after the accesses to local resources. Meanwhile, τ_1 requests r^3 so it executes on P_s with an active priority of 7.

- Finally, τ_1 and τ_2 finish their executions at $t = 13$.

As illustrated by Figure 2.3, tasks can only incur one blocking from lower priority tasks when accessing a global resource on the synchronization processor (see τ_4 in the duration from $t = 5$ to $t = 6$), which is achieved by the priority-boosting mechanism and the prioritised resource serving order. This is an important property from the viewpoint of schedulability tests as the remote blocking can be bounded so that response time of each task can be calculated. However, adopting MPCP can lead to a relatively complicated system with considerable run-time overheads either by adopting the remote helpers on the synchronisation processor or forcing tasks to truly migrate during each access to a global resource.

There exist several variants of MPCP. To increase the processor utilisation, the initial MPCP approach was extended to support multiple synchronisation processors (i.e., the extended MPCP), where each synchronisation processor is assigned with one or more global resources [88, 91]. However, the extended MPCP does not support nested accesses between global resources as they can now be accessed from multiple processors simultaneously. In addition, [89] described a generalised version of MPCP (i.e., the generalised MPCP) for shared memory systems, where the need for local agents is removed and tasks on any processor can access a global resource. Later on, the generalised MPCP is supported by the schedulability test proposed in [62]. Another MPCP variant for distrubuted systems (DPCP) is described in [91] and [89], where the remote accessing approach is clarified and the notion of local agent is introduced. Typically, a local agent indicates a task that runs in a cluster and has access to the global resources in its own cluster. If a task from another cluster requires the resource, it issues a request to the corresponding local agent, which will execute with the resource on behalf of the requesting task with highest priority on that processor.

Summarising the above, MPCP is a suspension-based multiprocessor resource sharing protocol with resources served in the priority order. However, the initial version of MPCP only allow one synchronisation processor while the extended and the generalised versions can not support nested resource accesses that involve any global resources. In addition, adopting the notion of synchronisation processors requires either remote helpers or migrations for each global resource access. However, despite that each task can only be blocked once from the analytical viewpoint, serving resources in priority order can lead to a long waiting queue for low priority tasks, as they are often be placed at the end of the prioritised queue while the newly-arrived high priority tasks can always get the resource prior to the low priority tasks. These disadvantages and limintations impose strong limitations towards the usability and run-time efficacy of this protocol, and can lead to highly complicated systems.

2.5.2 Multiprocessor Stack Resource Protocol

While PCP is extended as MPCP for managing global resources, the well-known SRP described in Section 2.4.3 is also revised to support multiprocessor resource sharing, namely the Multiprocessor Stack Resource Protocol (MSRP) [47]. However, even with the name MSRP, SRP is only adopted to manage the accesses to local resources in each partition while a simple and effective approach is proposed to control the requests to global resources.

Instead of suspending unsatisfied resource-requesting tasks, tasks under MSRP perform busy-waiting (i.e., spinning) when contending for a global resource and are non-preemptive during the period of waiting for and executing with a global resource. The following summarises the definitions of this protocol:

- SRP is applied to manage local resources on each processor.
- Each global resource is associated with a FIFO waiting queue.
- If a task requests a global resource, it becomes effectively non-preemptive and begins spinning for the resource. Meanwhile, it is placed at the end of the FIFO queue of the requested resource.
- Once the task becomes the head of the FIFO queue, it is allowed to acquire the resource. During the execution with the global resource, the task remains non-preemptive until the resource is released.
- Limited nested resource accesses are supported, where accesses between local resources and accesses from local resources to global resources on the same processor are allowed. However, accesses from local resource to global resources on different processors, accesses from global resources to local resources and accesses between global resources are not supported due to deadlock concern.

Figure 2.4 illustrates tasks' behaviours when accessing shared resources under MSRP with a two-processor system, which contains four tasks and one

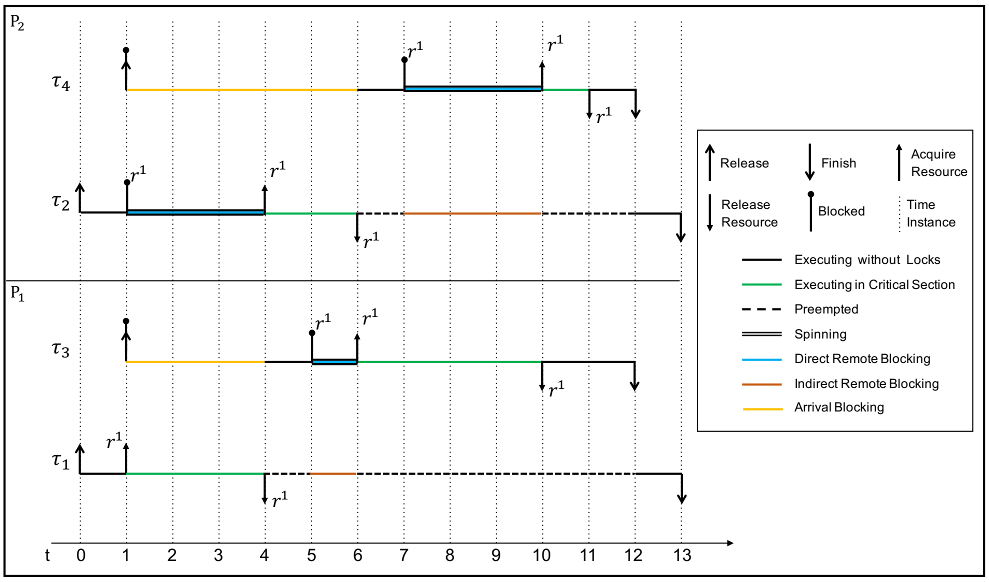


Figure 2.4: Example of the Multiprocessor Stack Resource Protocol

global resource that is required by all tasks. The following describes the execution of the given system.

- τ_1 and τ_2 are released on both processors at $t = 0$ and then both request r^1 at $t = 1$. Assuming τ_1 gets r^1 ahead of τ_2, τ_1 then becomes non-preemptive and executes with r^1 while τ_2 is spinning non-preemtpively (i.e., incurs direct remote blocking).

- At the same time ($t = 1$), τ_3 and τ_4 are also released but they incur arrival blocking immediately as τ_1 and τ_2 are running non-preemptively.

- At $t = 4$, τ_1 releases r^1 so that it is preempted by τ_3 while τ_2 starts executing with r^1 on P_2. τ_3 requests r^1 at $t = 5$ but is not satisfied so that it incurs direct remote blocking and starts spinning. Accordingly, τ_1 also has to cope with this delay as the indirect spin delay from τ_3.

- At $t = 6$, τ_2 releases r^1 so that it is preempted by τ_4 while τ_3 can acquire r^1 and starts executing.

- At $t = 7$, τ_4 requests r^1, which is held by τ_3. Hence, τ_4 incurs direct remote blocking and is spinning for the resource. Meanwhile, τ_2 (which is preempted by τ_4) also incurs this delay as the indirect remote blocking from τ_4 due to accessing r^1.

- τ_3 releases r^1 at $t = 10$ so that τ_4 can lock r^1 and continues. Meanwhile, the indirect remote blocking period of τ_2 is finished.
- τ_4 releases r^1 at $t = 11$, and then both τ_3 and τ_4 are finished at $t = 12$. Finally, τ_1 and τ_2 are finished at $t = 13$.

As illustrated by the example, tasks under MSRP can incur three types of blocking: *direct spin delay*, *indirect spin delay* and *arrival blocking*, described as follows:

- A task τ_x can incur *direct spin delay* when being blocked directly by remote tasks for accessing a resource.
- A task τ_x can incur *indirect spin delay* if it is preempted by a local higher priority task, which in turn is blocked directly by a remote task for accessing an unavailable global resources (i.e., incurs direct spin delay).
- A task τ_x can incur *arrival blocking* upon its arrival if a local lower priority task (denote as τ_{ll}) is accessing a resource r^k with its priority boosted. r^k can be either global or local. In the case of global resource, τ_x has to cope with the potential direct spin delay incurred by τ_{ll} due to accessing r^k.

With the blocking effects of MSRP identified, these blocking can be effectively bounded by the RTA equations described in Section 2.4.4 with minor modifications to reflect the parallel accesses to shared resources. With non-preemptive FIFO spin locks, MSRP guarantees that while accessing a resource, the task can only incur one blocking from each of the remote processors that has tasks requesting the same resource. Thus, the blocking incurred by a task for accessing a resource once can be bounded by the number of processors with tasks requesting the resource.

The following analysis of MSRP is cited from [47] but with the notation style applied in this thesis for consistency. The response time of τ_i under MSRP is bounded by Equation (2.6), where function **lhp** indicates the set of local tasks with a priority higher than $Pri(\tau_i)$ and a new notation $\widehat{C_i}$ is introduced.

$$R_i = \widehat{C_i} + B_i + \sum_{\tau_h \in \mathbf{lhp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \widehat{C_h} \tag{2.6}$$

Differentiated with C_i given in Table 2.1 (which denotes the pure computation time of τ_i without accessing any resources), $\widehat{C_i}$ denotes the pure worst-case computation time of τ_i (i.e., C_i) and the time τ_i spends on each requested resource *with potential delay*, as given in Equation (2.7).

$$\widehat{C_i} = C_i + \sum_{r^k \in F(\tau_i)} N_i^k e^k \tag{2.7}$$

Compared to Equation (2.3), c^k is replaced by e^k in Equation (2.7) to denote the total execution cost for r^k, including the potential delay for accessing the resource from each remote processor. e^k is bounded by Equation (2.8), where function *map* takes a set of tasks and returns a set of processors that those tasks are allocated on and $||$ gives the size of a given set. As shown in Equation (2.8), under MSRP, the blocking time incurred by τ_i for one resource access is bounded by the number of processors with tasks requesting the resource.

$$e^k = |map(G(r^k))| c^k \tag{2.8}$$

With e^k and $\widehat{C_i}$ calculated, the direct spin delay and indirect spin delay incurred by τ_i in one release can be safely bounded. The arrival blocking of τ_i under MSRP can be bounded via revising Equations (2.4) and (2.5), as shown in Equations (2.9) and (2.10), where $\hat{e_i}$ is used to denote the arrival blocking τ_i can incur with potential remote delay and τ_{ll} denotes a local task with a priority lower than τ_i.

$$\hat{e_i} = max\{e^k | N_{ll}^k > 0 \wedge (r^k \text{ is global } \vee Pri(r^k) \geq Pri(\tau_i))\} \tag{2.9}$$

$$B_i = max\{\hat{e_i}, \hat{b}\} \tag{2.10}$$

With MSRP, both local and global resources can cause a task to incur arrival blocking. If r^k is a global resource, the duration of the arrival blocking is bounded by e^k. Otherwise (r^k is local), it can block τ_i only if the ceiling priority of r^k is equal to or higher than $Pri(\tau_i)$, and $e^k = c^k$ as $|map(G(r^k))| = 1$ for local resources.

This concludes the description of MSRP and its schedulability analysis. The new notations adopted in this analysis is summarised in Table 2.6. Note that differentiated with C_i (which denotes the pure worst-case computation time of τ_i without accessing any resources), $\widehat{C_i}$ also includes the time τ_i spends on waiting for and executing with each required resource.

Compared to MPCP (which has various versions with complicated approaches i.e., the initial version with migrations, the distributed version with remote agents and the generalised version with global priority ceilings), MSRP

Table 2.6: Notations Applied in the Analysis of MSRP

τ_{ll}	A local task with a priority lower than that of the task that is currently being studied (i.e., τ_i).
$\widehat{C_i}$	The pure computation time of τ_i (i.e., C_i in Table 2.1) plus the time τ_i spends on waiting for and executing with each requested resource (i.e., with the potential delay for accessing shared resources).
e^k	The total accessing cost of r^k, including the pure execution cost of r^k and the potential delay for accessing this resource.
$\hat{e}_i$	The arrival blocking incurred by τ_i with the potential remote blocking included.
$\mathbf{lhp}(i)$	The set of local tasks with a priority higher than that of τ_i.

effectively bounds the blocking time for accessing global resources with a simple and elegant non-preemptive FIFO spin approach. Unlike the priority-ordered methods, MSRP guarantees that the blocking time of each access is limited to the number of processors that request the resource. Due to this reason, MSRP can be effectively analysed by a simple schedulability test while analysing MPCP systems is much more complicated [106].

This attractive property led to many subsequent studies towards the MSRP systems. In [19], a holistic analysis was proposed to provide a less pessimistic schedulability test than that of the original MSRP analysis (the one described above). Later, [106] presented another analysis based on the Integer Linear Programming technique that further reduces the degree of pessimism when analysing MSRP systems. Detailed descriptions of these schedulability tests are presented in Section 2.6.2.

However, the non-preemptive resource accessing model can be too strong in certain situations. For instance, an unrelated task has to cope with the arrival blocking as long as a task is executing with a global resource on its processor, which also includes the potential delay from remote processors. Therefore, compared to the preemptive approach, high priority tasks (which are usually assigned with a short deadline) are more likely to miss their deadlines in MSRP systems. A detailed discussion towards the preemptive and non-preemptive resource sharing methods is given in Section 3.1.

2.5.3 Flexible Multiprocessor Locking Protocol

The Flexible Multiprocessor Locking Protocol (FMLP) developed in [17] can be applied to either global or fully-partitioned systems, and was the first multiprocessor resource sharing protocol that proposes the idea of managing resources based on their different characteristics, with more than one synchronisation approaches.

In FMLP, resources are classified as either *long resources* or *short resources* based on the length of critical sections, where suspension-based locks are adopted to manage long resources and spin locks are used for short resources. Whether a resource should be regarded as a long or short resource is decided by users. Yet each short resource should has a shorter critical section than that of any long resource. Unlike MPCP and MSRP, FMLP can support nested resource accesses between global resources via *resource groups* and *group locks*. The following summarised the definitions of this protocol.

- A resource group may contains one or more shared resources. However, each group can only contain resources with a same type i.e. either short resources or long resources.

- Each resource group is protected by a designated group lock, and accessing any resource in a resource group must acquire the associated group lock a priori.

- Short resources groups are managed by non-preemptive spin locks and are served in FIFO order. Once a task gets the lock, it remains non-preemptive until it releases the lock.

- Long resource groups are guarded by suspension-based locks, where blocked tasks are suspended and are inserted into a FIFO queue. Once a task locks the resource, the task boosts its active priority to maximum priority among all the tasks that are blocked on the group lock until it releases the lock.

- Non-nested resources are grouped individually while the resources required by nested accesses are placed into the same group.

- A long resource request can contain nested accesses to short resources. However, a short resource request cannot contain any nested requests to long resources.

According to the definitions of FMLP, a task that holds a lock of a short resource group can only issue nested accesses to other short resources, and remains non-preemptive until it releases the group lock. As for a task with a long resource group lock, it can request either long or short resources in nested fashion. When accessing a long resource via nested access, this request will be satisfied immediately as the resource are placed into the same group. If the task issues a nested access to a short resource, the associated spin lock must be acquired so that the task becomes non-preemptive while accessing the short resource. By doing so, FMLP supports the nested resource accesses between global resources and prevents the formation of the circular resource requesting chain i.e., deadlocks are avoided.

Table 2.7: Tasks in the Example System of FMLP

Task	Resource Usage	Partition
τ_1	$r_s^1(r_s^2)$	P_1
τ_2	r_s^2	P_2
τ_3	(r_s^2, r_l^1)	P_1
τ_4	$r_l^1(r_s^2)$	P_2

Table 2.8: Resources in the Example System of FMLP

Group Lock	Resource Group
GL_s	r_s^1, r_s^2
GL_l	r_l^1

To illustrate FMLP, an example system is provided in Table 2.7 and resources in the system are described in Table 2.8, where GL indicates a group lock, r_s denotes a short resource and r_l represents a long resource. The resource access $r_s^1(r_s^2)$ indicates r_s^1 access r_s^2 in the nested fashion while the access (r_s^2, r_l^1) indicates a sequential resource-accessing order without nesting. Figure 2.5 presents the execution of this system under FMLP, as described below.

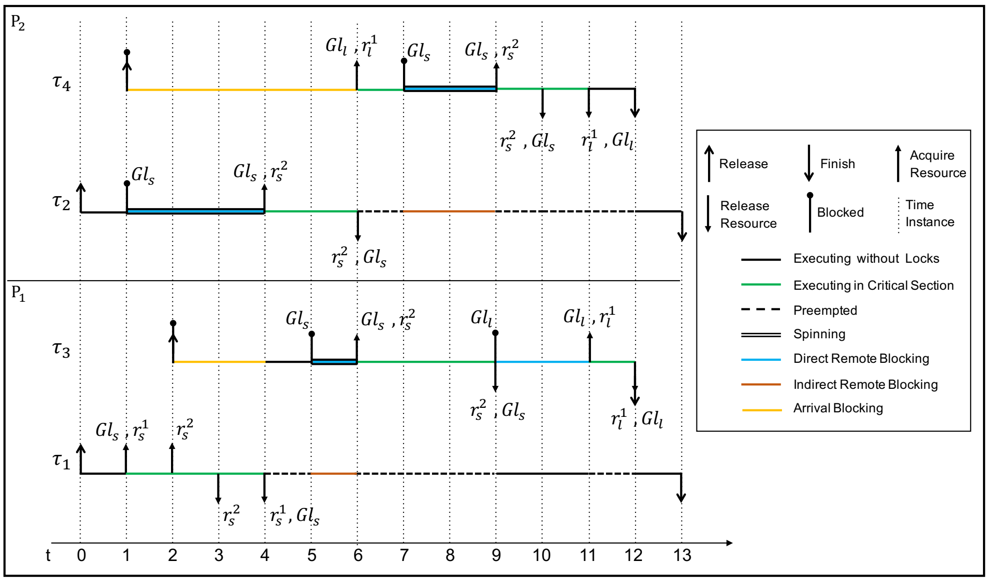


Figure 2.5: Example of the Flexible Multiprocessor Locking Protocol

- At $t = 0$, τ_1 and τ_2 are released on both processors and then request GL_s for accessing r_s^1 and r_s^2 respectively.
- Suppose that τ_1 gets GL_s first, it becomes non-preemtive and acquires r_s^1 while τ_2 starts spinning for GL_s. Thus, τ_3 and τ_4 incur arrival blocking immediately after being released at $t = 2$ and $t = 1$ respectively.
- At $t = 2$, τ_1 issues a nested access to r_s^2 and is satisfied immediately as it holds GL_s so that it acquires r_s^2 and keeps executing non-preemptively. It releases r_s^2 at $t = 3$ and then releases r_s^1 with GL_s at $t = 4$.
- After τ_1 released GL_s, τ_2 gets this lock and then accesses r_s^2. At the same time, τ_1 is preempted by τ_3 on P_1.
- At $t = 5$, τ_3 requests GL_s for accessing r_s^2 so that it starts spinning. Meanwhile, τ_1 also incur this delay as the indirect spin delay form τ_3. τ_3's request is satisfied until τ_2 is finished with r_s^2 and releases GL_s at $t = 6$.
- At $t = 6$, τ_4 locks GL_l and access r_l^1, but is blocked at $t = 7$ for accessing r_s^2. Thus, it spins non-preemptively for r_s^2. Accordingly, τ_2 also incurs this blocking as the indirect spin delay.
- At $t = 9$, τ_3 releases GL_s so that τ_4's request is satisfied. However, τ_3 is

blocked for requesting GL_l so that it is suspended while τ_1 resumes its execution.

- τ_4 releases GL_s at $t = 10$ and unlocks GL_l at $t = 11$ so that τ_3 is resumed with GL_l and r_l^1 acquired.
- At $t = 12$, τ_3 releases r_l^1 and GL_l, and then τ_3 and τ_4 are finished. At $t = 13$, τ_1 and τ_2 are finished as well.

As shown in the example, FMLP supports nested accesses from long resources to short resources, where both types resources can be either local or global resources. According to [22], by adopting the appropriate locks to resources with different critical section length, the impact of resource sharing on schedulability can be reduced. In FMLP, non-preemptive FIFO spin locks are adopted for short resources so that a strong progress of resource execution is guaranteed without incurring extra overheads due to context switches while long resources are managed by suspension-based locks, where waiting tasks can give up the processor and provide the execution opportunities to other tasks.

However, adopting FMLP can lead to a relatively complicated system as both suspension-based and spin-based locking primitives must be available (or be implemented), and the resource serving mechanism for each lock must be realised for the use of this protocol. In addition, tasks under FMLP can incur limitations of both locks, where suspension-based locks can impose frequent context switches while non-preemptive spin locks can lead to prolonged blocking to high priority tasks. Further, with group locks, the degree of parallelism is reduced as the group lock must be acquired before accessing any of resources in a resource group, which serialises the accesses to resources in the same group.

2.5.4 Preemptable Waiting Locking Protocol

As described in Sections 2.5.2 and 2.5.3, despite that non-preemptive spin locks guarantee strong progress of resource execution with bounded blocking time for each resource access, this approach also imposes a considerable amount of blocking time to high priority tasks so that the schedulability of the system can be undermined if critical sections are long. In addition, suspension-based locks introduce frequent context switches with non-negligible run-time overheads, which is a significant cost if critical sections are short. Thus, to minimise the blocking incurred by high priority tasks with non-preemptive spin locks and to avoid the overheads imposed by suspension-based locks, the Preemptable Waiting Locking Protocol (PWLP) is developed in [1] to propose a preemptable spin-based locking approach for both global and fully-partitioned systems.

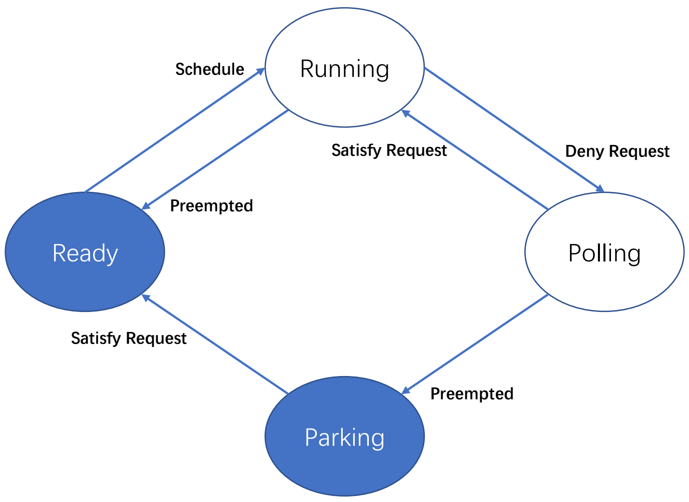


Figure 2.6: Task States with the Preemptable Waiting Locking Protocol

In [1], the execution states of tasks under multiprocessor systems with shared resources are specified, as given in Figure 2.6. When a task is released, it is defined as *Ready* and becomes runnable. Once it is scheduled, its state is changed to *Running* and the task can start its execution. While executing, the task can either be blocked for accessing a shared resource or be preemtped by a local higher priority task. While being preempted, the task sets it states back to *Ready* and waits to be scheduled again. If being blocked for accessing resources, the task performs busy-waiting and enters into the *polling* state. The task can return to *Running* state later if the request is satisfied. However, while spinning, it can also be preemtped (i.e., with the preemptive spinning approach) by a newly-arrived high priority task so that it enters into the *Parking* state. Once the preemptor is finished, the task becomes *Ready* and continues to compete for the resource. Based on this state machine, PWLP defines a set of rules to control accesses to shared resources, summarised below.

- A task τ_x that requests a resource r^k will be added into the FIFO-ordered queue associated with r^k and busy-waits (i.e., spins) with its base priority until its request is satisfied.

- Once r^k is granted, τ_x becomes non-preemptable immediately during the execution with r^k. τ_x restores its base priority after it releases r^k.
- If τ_x is preempted while spinning for r^k (i.e., in *Polling* state), τ_x cancels its request, removes itself from the FIFO waiting queue, and enters into *Parking* state, where it waits for the high priority task to finish.
- Once the preemptor is finished, τ_x changes its state from *Parking* to *Ready* and becomes runnable again. When τ_x is scheduled again, it will re-issue the request to r^k and is placed at the end of the FIFO queue.

As described above, tasks under PWLP should become non-preemptable only when executing with a resource so that the resource execution is protected and cannot be interfered by other tasks. In addition, by spinning at the base priority (i.e., preemptable), the arrival blocking incurred by PWLP tasks is reduced as they can preempt the spinning task immediately instead of incurring blocking with potential remote delay. With this approach, PWLP guarantees the resource execution progress while minimising the arrival blocking of all tasks in the system to one critical section only, which is identical with the uniprocessor case. However, the cancellation mechanism can lead to increased resource waiting time, which results into prolonged direct and indirect spin delay for PWLP tasks. To illustrate the differences between the non-preemptive spin locks and the preemptable spinning approach, the example for MSRP in Figure 2.4 is used here with PWLP applied, as shown in Figure 2.7 and described below.

- τ_1 and τ_2 are released at $t = 0$, and both request r^1 at $t = 1$.
- Assuming τ_1 gets the lock first so that τ_2 is blocked. However, τ_2 is not spinning as it is preempted immediately by τ_4. Meanwhile, τ_3 is released but incurs arrival blocking as τ_1 is executing with r^1 non-preemptively.
- At $t = 2$, τ_4 requests r^1 so that it is placed into the FIFO queue and starts spinning with its base priority.
- τ_1 releases r^1 at $t = 4$ so that τ_4 can lock r^1 and executes. Meanwhile, τ_1 is preemtped by τ_3 as it is now executing with its base priority.
- τ_4 releases r^1 at $t = 5$ and then τ_3 locks r^1.

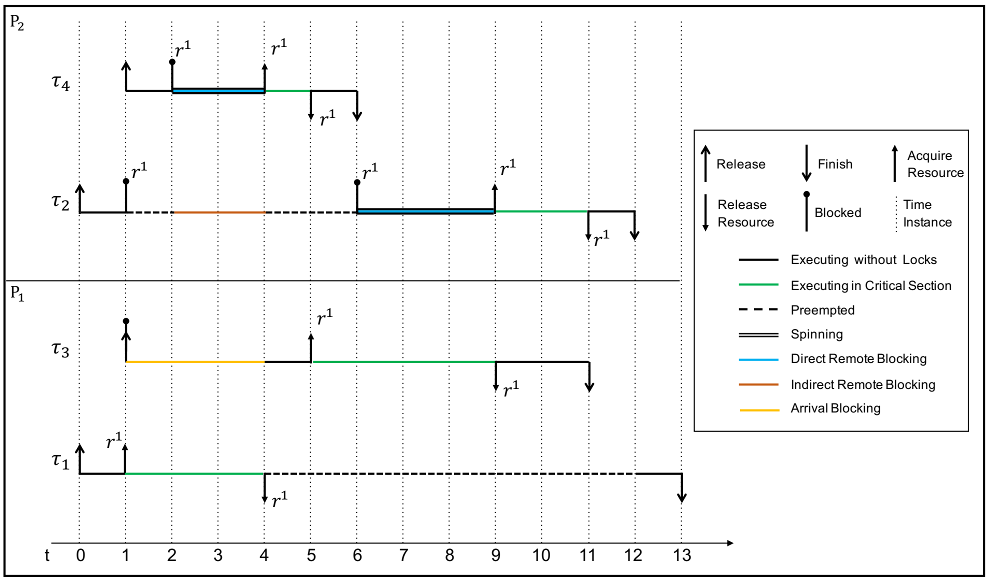


Figure 2.7: Example System of the Preemptable Waiting Locking Protocol

- At $t = 6$, τ_4 is finished and τ_2 is resumes and starts spinning for r^1.
- τ_3 releases r^1 at $t = 9$ so that τ_2's request is satisfied.
- At $t = 11$, τ_2 releases r^1 while τ_3 is finished. Finally, τ_1 is finished at $t = 13$.

As shown in the example, the response time of τ_3 (and τ_4) under PWLP is 6 (and 1) units of time shorter than that of the task with MSRP adopted due to the minimised arrival blocking. However, favouring high priority tasks can result into prolonged response time of low priority tasks, where they can be preempted frequently while waiting for a resource, and hence, incurs more blocking than the non-preemptive spinning approach due to the cancellation mechanism. PWLP was developed assuming resources are accessed in non-nested fashion. Supporting nested resources is not discussed in [1] but group locks are recommended by PWLP's authors to allow nested resource accesses.

2.5.5 Parallel Priority Ceiling Protocol

The Parallel Priority Ceiling Protocol (PPCP) is a suspension-based protocol proposed in [41] for globally scheduled systems with fixed priorities. As described in Section 2.4.2, any low priority tasks under PCP are prevented from executing (and of course, requesting any resources) when a task has locked a resource with a higher ceiling priority. However, this approach can not be adopted to global systems directly due to the presence of multiple processors.

By extending PCP, PPCP proposes a sophisticated resource control approach for global systems, where a configurable number of low priority tasks are allowed to execute and to lock resources under certain situations. To achieve this, PPCP introduces a tuneable parameter α to offer a trade-off between the number of low priority tasks that can execute with shared resources and the blocking incurred by a high priority task due to resource sharing. The following summarises the definitions of this protocol.

- Each priority level in the system is assigned with a tuneable parameter α. The value of α for each priority level is assigned artificially, followed by the rule that the value of α for a low priority level should not be higher than that of α for a high priority level.

- For a given task with a priority level Pri (which has an associated parameter α_{Pri}), it is allowed to lock a shared resource only if the number of the released tasks (including the task itself) with a lower base priority than Pri but have or will have a higher active priority due to resource accessing, is at most α_{Pri} for all Pris.

- If $\alpha_{Pri} = 1$ for all Pris, the behaviour of PPCP is identical with PCP. In contrast, it behaves like PIP if α of each Pri is set to the total number of tasks in the system.

- If a task does not require any resources while there exist an unassigned processor in the system, this task is dispatched to that processor immediately.

- PIP is applied if a task requests an unavailable resource, where the resource-holding task will updates its active priority to the highest priority among all tasks that are blocked on the resource.

- The system keeps tracking the number of tasks with a lower base priority but have or will have a higher active priority for each priority level, and checks whether such a number is larger than the α parameter of the corresponding priority level during each task release, resource access, resource release and task completion.

Table 2.9: Resources in the Example System of PPCP

Resource	Accessing Tasks	Ceiling Priority
r_1	τ_1, τ_6	6
r_2	τ_2, τ_5	5
r_3	τ_3, τ_4, τ_7	7

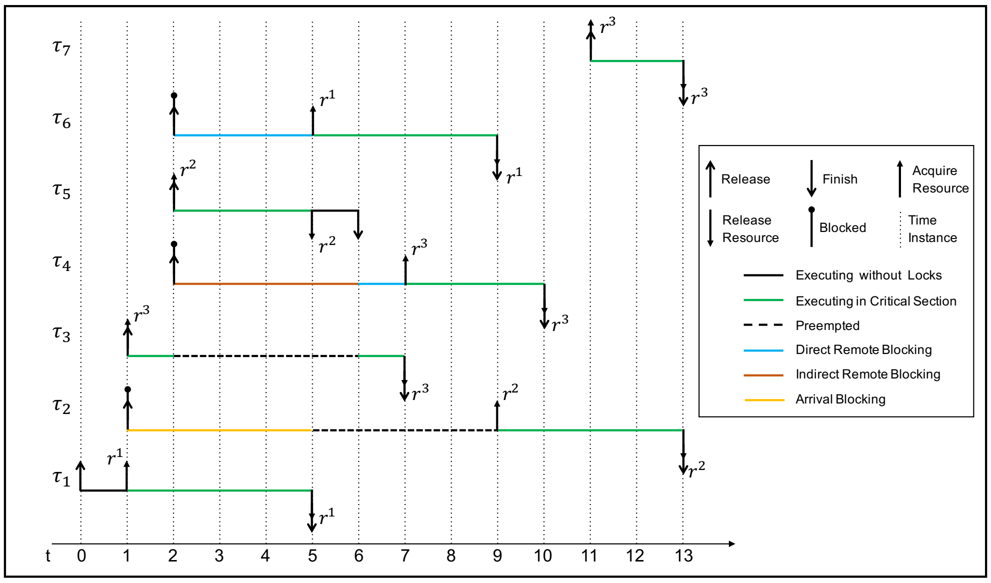


Figure 2.8: Example System of the Parallel Priority Ceiling Protocol

Figure 2.8 provides an example of applying PPCP to a dual-processor system with 7 tasks and 3 shared resources, as described below. The resource usage and ceiling priorities of the shared resources in the system are given in Table 2.9. In addition, the α value for the priority levels that are higher than 4 is set to 3 while α of other priority levels (i.e., priority level 1 to 4) are set to 2. The example provided in Figure 2.8 focuses on the execution status of tasks regardless which processor it is dispatched to, and two tasks can execute at the same time as two processors are available.

- τ_1 is released at $t = 0$ and locks r^1 at $t = 1$. This is allowed as it is the only executing task at this time.
- At $t = 1$, both τ_2 and τ_3 are released and require r^2 and r^3 respectively. At this time, the system detects that r^1, r^2 and r^3 are also requested by τ_6, τ_5 and τ_7 respectively, and each of the tasks has a priority higher than 4. Therefore, for priority level 4, there will be three tasks (τ_1, τ_2

and τ_3) that have a lower base priority, but have or will have an active priority higher than 4 due to resource accessing. However, $\alpha_4 = 2$ so that only two tasks can lock resources. As τ_1 is already executing with r^1, τ_3 is allowed to lock r^3 while τ_2 is prevented from executing (i.e., being blocked upon its arrival).

- At $t = 2$, τ_4, τ_5 and τ_6 are released and request r^3, r^2 and r^1 respectively. These tasks are all eligible to lock resources but r^1 and r^3 are unavailable at this moment. Accordingly, τ_1 and τ_3 raise their priority to 6 and 5 respectively according to PIP. Thus, only τ_5 can execute and acquire r^2 via preempting τ_3, which has the lowest active priority in the system. As a result, τ_4 incurs indirect blocking while τ_6 is blocked directly.

- At $t = 5$ τ_1 releases r^1 and is finished so that τ_6 can lock r^1 and starts executing. Meanwhile, τ_5 releases r^2 and the arrival blocking period of τ_2 is finished but it cannot execute according to fixed-priority scheduling.

- At $t = 6$, τ_5 is finished so that τ_3 resumes its execution with r^3. However, τ_4 still cannot execute as it is now blocked directly by τ_3 for accessing r^3.

- τ_3 releases r^3 at $t = 7$ and is finished. Thus, τ_4 starts its execution and locks r^3.

- τ_6 is finished at $t = 9$ after released r^1 so that τ_2 can start executing as there is only one task (i.e., τ_4) that is executing now.

- At $t = 10$, τ_4 releases r^3 and is finished. τ_7 is then released at $t = 11$ and locks r^3 directly.

- Finally, τ_2 and τ_7 are finished at $t = 13$.

As illustrated by this example, τ_4 incurs blocking for 5 units of time in total. However, with PIP only, τ_2 will get r^2 at $t = 1$ with an active priority of 5 while τ_3 is prevented from executing. Thus, τ_3 has to wait for τ_2 to release r^2 firstly and then wait for τ_5 to finish before it can execute r^3. In this case, τ_4 will incur blocking for 9 units of time (by adding the execution time of τ_2 with r^2) so that the system will finish at $t = 14$. Therefore, PPCP improves the average performance of the system by 1 unit of time in this example via allowing a tunable number of low priority tasks to lock resources

(τ_1 and τ_3 in this case). In addition, PPCP guarantees that the blocking of high priority task is bounded since the number of low priority tasks that can acquire resources is fixed. Due to the same reason, PPCP achieves a deeper parallelism compared to either PIP or PCP.

Unfortunately, PPCP does not support nested resource accesses at all. In addition, applying either PIP (with α set to the total number of tasks for all Pris) or PCP (with $\alpha = 1$ for all Pris) implies this protocol suffers from the same limitations in PCP and PIP. More importantly, although with an interesting resource sharing approach, this protocol is developed explicitly for global scheduling scheme, and hence, cannot be adopted into fully-partitioned systems.

2.5.6 O(m) Locking Protocol

The O(m) Locking Protocol (OMLP) proposed in [20] is a suspension-based protocol and guarantees that each task can incur at most M times of priority inversions for accessing a global resource, where M denotes the number of processors in the system. In OMLP, resources are managed by m-exclusion locks (a locking structure that supports mutual exclusive access of at most M tasks at a given time) and are severed by both priority and FIFO ordering. Basically, tasks contending for a global resource are separated into two queues according to the number of these tasks. If this number is less than M, then all the competing tasks can be directly placed into a global FIFO queue, where tasks are satisfied according to their arrival order. However, if there exist more completing tasks, the lately arrived tasks will be inserted into the priority-ordered queue (assuming that M competing tasks have been inserted into the FIFO queue), where these tasks can join into the FIFO queue later based on priority order if spaces in the FIFO queue becomes available.

This protocol can be adopted to either global or fully-partitioned systems, yet with different definitions due to the differences of these scheduling schemes in nature. For global systems, each resource is associated with a global FIFO queue (Fq) and a priority queue Pq, where the length of the Fq is set to M. The resource will always be granted to the task at the head of the FIFO queue. Once the task acquires a resource, it boosts its active priority to the maximum priority level among all tasks in both Fq and Pq. Upon each resource release, the task at the head of Pq (if it exists) is dispatched to the end of FQ and is suspended if necessary according to FPS.

As for fully-partitioned systems, OLMP introduces the notion of contention token (i.e., a binary semaphore) and assigns a token to each processor. Unlike global OMLP, each global resource is controlled by a FIFO queue with length of M while each contention token is assigned with a priority queue (Pq). A task is allowed to request a global resource only if it acquires the local contention token. If the token is unavailable, the task is suspended and is placed into Pq. After the token is acquired, the task boosts its active priority to the highest priority on its processor, joins into the FIFO queue and is then suspended (if Fq is not empty before adding the task) until the resource is granted. After the resource is released, the task is removed from the FIFO queue, releases the local token and restores its priority to the previous level.

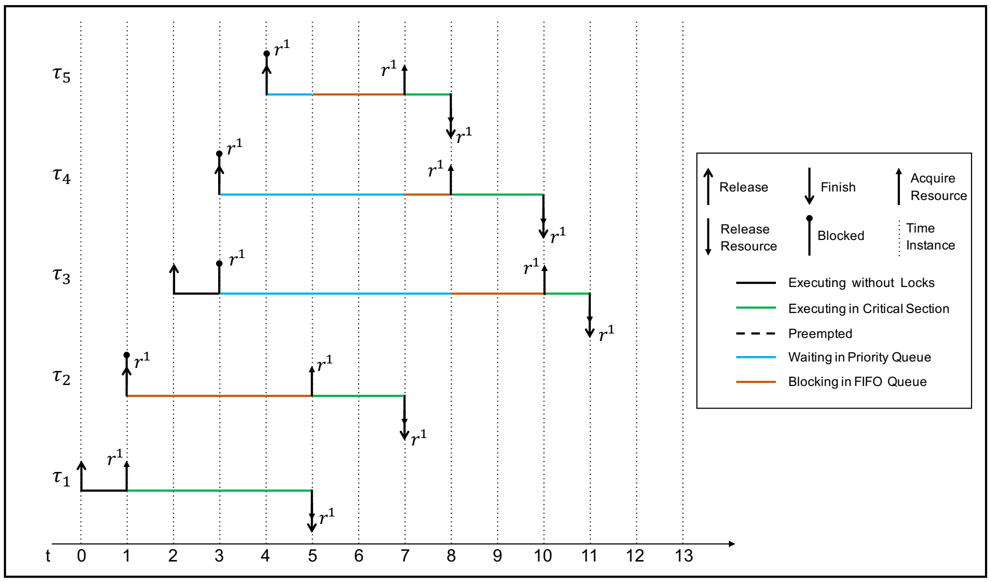


Figure 2.9: Example System of the O(m) Locking Protocol

To illustrate the features of OMLP, a dual-processor globally scheduled system with 5 tasks is given as an example, where all tasks request the same global resource r^1, as shown in Figure 2.9 and described below.

- τ_1 is released at $t = 0$ and then locks r^1 at $t = 1$. This is allowed because the Fq of r^1 is empty before τ_1 is added. Meanwhile, τ_2 is blocked immediately after being released for requesting r^1 so that it is also added into the Fq.

- τ_3 is released at $t = 2$ and then requests r^1 at $t = 3$. However, it cannot join into the Fq as the maximum length is 2 i.e., the number of processors. So that it is inserted into the Pq instead and is suspended.

- Later on, τ_4 and τ_5 are also placed into Pq due to requesting r^1 so that $Pq = \{\tau_5, \tau_4, \tau_3\}$ at $t = 4$.
- At $t = 5$, τ_1 releases r^1 so that it is removed from Fq. Thus, τ_2 gets the lock while τ_5 is dispatched from Pq to Fq.
- At $t = 7$, τ_2 releases r^1 so that τ_5 can acquire the resource after being blocked two times (by τ_1 and τ_2). In addition, τ_4 is moved from Pq to Fq. At this time, $Pq = \{\tau_3\}$ and $Fq = \{\tau_5, \tau_4\}$.
- τ_5 releases r^1 at $t = 8$ so that τ_4 becomes the head of Fq and locks r^1. Consequently, τ_3 now is moved from Pq to Fq.
- At $t = 10$, τ_4 releases r^1 and is finished while τ_3 can finally execute with r^1, which is then finished at $t = 11$.

As shown in the example, each task under OMLP can incur at most M blocking when accessing a global resource. This is achieved by the combination of Fq and Pq. Compared to the suspension-based approach with prioritised ordering (e.g., MPCP), OMLP can provide bounded blocking time for each task in the system under the suspension-based locks. In addition, as stated in [20], nested resource accesses can be supported via group locks. However, compared to MSRP, which also bounds the blocking time to M, OMLP can be less favourable due to high implementation complexity, specially for partitioned OMLP, which requires contention tokens with a set of priority ordering queues. Further, adopting this protocol can also lead to considerable amount of run-time overheads due to frequent context switches and the sophisticated resource sharing techniques.

Later on, an extension of OMLP was proposed in [21] for clustered systems (Clustered OMLP) with a novel approach named *priority donation*. In Clustered OMLP, an one to one donor relationship is established when a priority inversion will occur, where the donor (a newly-arrived high priority task) donates its priority to a task that can cause blocking due to executing with a global resource. This approach is similar with PCP but the priority donation is performed only when a task can actually cause a priority inversion. With this approach, the protocols guarantees that each task in the system can be preemtped only once. However, by favouring low priority tasks, higher priority tasks in Clustered OMLP can have prolonged response time as they may need to donate their priorities due to resource sharing. For the sake of brevity, the details of the Clustered OMLP is not provided here as this thesis focuses on the shared memory systems. A detailed description of this protocols is presented in [21].

2.5.7 Spinning Processor Executes for Preempted Processor

The non-preemptive spin locks (i.e., waiting for and executing with resources non-preemptively) have the advantage of low scheduling cost by delaying the preemptions from high priority tasks, which is an expensive operation [84]. However, this approach can impose considerable amount blocking time to high priority tasks as they must wait for the current resource access to finish before they can start executing. On the other hand, the preemptable spinning approach benefits high priority tasks as they can preempt a spinning low priority task directly. However, the approach (i.e., PWLP in Section 2.5.4) requires that a task must cancel its current request while being preempted and re-request the resource later on when being resumed, which prolongs the waiting time for accessing resources. Under this context, an alternative preemptable spinning approach is proposed in [100] to minimise the arrival blocking incurred by high priority tasks while not interfering the progress of resource accessing, namely the Spinning Processor Executes for Preempted Processor (SPEPP).

SPEPP focuses on the low-level resource sharing i.e., the kernel level so that it can support both global and partitioned scheduling. Notably, a helping mechanism is introduced to decrease the prolonged resource waiting time imposed by the preemptable spinning approach. Under SPEPP, tasks request a shared resource are inserted into a FIFO queue along with an operation block, which stores the operations to be performed with the resource. If a task's turn to acquire a lock comes while being preempted, another task that is spinning on the same lock will execute the operations on behalf of the preempted task. Thus, under SPEPP, the progress of resource execution is guaranteed unless all tasks waiting for the lock are preempted. Once the preempted task is resumed, it will find out that the required operations are already performed so that it can proceed with the output. The definitions of SPEPP are summarised as below:

- A SPEPP resource contains a spin lock with two FIFO queues to store waiting tasks and their operation blocks.

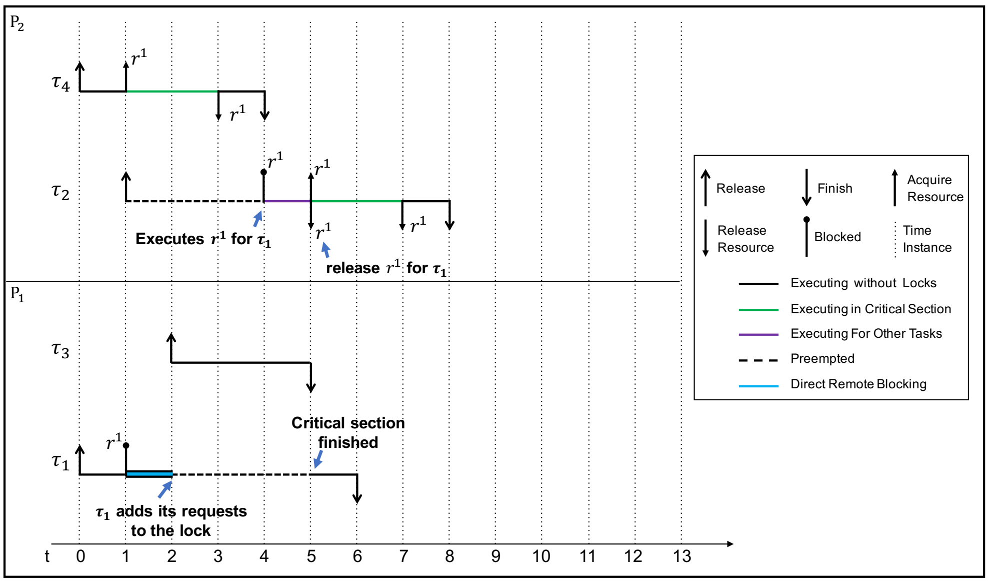
- A task that requests a shared resource will be placed into a FIFO queue along with the associated operation block, which is a memory block that contains all the memory space to store the input and output values of the operations.
- The atomic test and set primitive is used to update the shared resource to indicate the preemption status of waiting tasks.
- Tasks are spinning with their base priorities when waiting for the access to a shared resource. However, critical sections are executed non-preemptively.
- Once a task acquires a resource, it will firstly execute the operations of tasks ahead of it in the FIFO queue and then execute its own operations. The task will not release the lock until its operation is finished.
- The resource-holding task must check whether an interrupt is pending after serving a request to the resource. If so, the task executes the corresponding interrupt handler within a bounded time before executing the operations of the next task.

Unlike the resource sharing protocols reviewed above, which mainly define tasks' behaviours while accessing shared resources (and propose conceptual mechanisms to facilitate resource sharing, such as the tuneable α in PPCP), SPEPP relies on special memory blocks called the *operation blocks.* To illustrate the key idea of this protocol, the pseudo code of the SPEPP algorithm cited from [100] is presented below.

```
shared var OpQueue; // The operation block queue;
shared var SpinLock; // Spin Lock

var opblock;  // operation block

var op; // pointer to an operation block

/* main routine */
enqueue_tail(&opblock, OpQueue);
while (the operations in opblock has not been executed)
do
   acquire_lock(SpinLock);
   op = dequeue_top(OpQueue);
```

```
  execute(op);
  release_lock(SpinLock);
end;
```

The resource-holding task will follow the main routine (i.e., the `while` loop) to execute the operations in `OpQueue` based on FIFO order until the `opblock` (i.e., the operations of the task itself) is executed. After a critical section is executed on behalf of a preempted waiting task, the resource-holding task updates the corresponding shared variable of that waiting task, which will leave the resource-accessing routine later on after being resumed.



P2
τ_4
τ_2
P1
τ_3
τ_1
r^1
Executes r^1 for τ_1
release r^1 for τ_1
Critical section finished
τ_1 adds its requests to the lock
t 0 1 2 3 4 5 6 7 8 9 10 11 12 13
Release
Finish
Acquire Resource
Release Resource
Blocked
Time Instance
Executing without Locks
Executing in Critical Section
Executing For Other Tasks
Preempted
Direct Remote Blocking

Figure 2.10: Example System of the Spinning Processor Executes for Preempted Processor

Figure 2.10 illustrates the execution of a system with SPEPP adopted, where τ_1, τ_2 and τ_4 request r^1, as described below.

- τ_1 and τ_4 are released at $t = 0$ and both request r^1 at $t = 1$. Suppose that τ_4 gets the lock first so that τ_1 is spinning with its base priority while τ_4 is executing with r^1 non-preemptively.

- At $t = 1$, τ_2 is released but cannot execute as it cannot preempt τ_4.

- At $t = 2$, τ_3 is released and can preempt τ_1 as τ_1 is spinning with its base priority. Thus, τ_1 adds its requests and operations (i.e., the `opblock`) to r^1 into the corresponding `opQueue`.

- τ_4 releases r^1 at $t = 3$ and is finished at $t = 4$. Thus, τ_2 can start its execution and requests r^1 immediately, where it finds that τ_1's `opblock` is in the queue. Then, τ_2 firstly executes the operations of τ_1 to r^1.

- τ_2 finishes the operations of τ_1 at $t = 5$ and then starts executing its own operations on r^1.

- At the same time, τ_3 is finished so that τ_1 is resumed, where it finds that its operations on r^1 are finished. Thus, it keeps executing and is finished at $t = 6$.

- At $t = 7$, τ_2 releases r^1 and is then finished at $t = 8$.

SPEPP is significant due to the helping mechanism, where a waiting task can execute the operations on behalf of the preempted spinning tasks. Compared to PWLP, the blocking time for accessing shared resources under SPEPP can be reduced as tasks remain in the resource-accessing routine even being preempted. However, practising SPEPP can lead to complicated spin lock data structures and has a high demand to memory space due to the need of operation blocks. In addition, adopting this protocol imposes considerable amount of run-time overheads as each operation of each resource-requesting task must be recorded into the corresponding operation block when requesting the resource. Finally, nested resource accesses are not allowed under this protocol.

2.5.8 Multiprocessor BandWidth Inheritance protocol

The Multiprocessor BandWidth Inheritance Protocol (M-BWI) is an execution-time server-based resource sharing protocol aims at soft real-time multiprocessor systems with the reservation-based scheduling [44]. This protocol is relevant to this thesis as it proposes a fully preemptive spin-based synchronisation approach with a migration-based helping mechanism that deals with the situation where a task runs out of budgets or is preempted while holding a shared resource.

M-BWI is extended from the BandWidth Inheritance Protocol (BWI) [70], which is a reservation-based protocol for soft or open uniprocessor real-time system. In BWI, the processing time for each task is determined and reserved before run-time so that each task is guaranteed with a specific amount of time to execute via an execution-time server. Typically, a server represents an abstraction that stores the scheduling parameters of a task, including a budget that specifies the maximum processor time allocated to the corresponding task. When a server is scheduled, its corresponding tasks can start executing.

In M-BWI, features in BWI are preserved while new mechanisms are proposed to manage global resources. Under this protocol, tasks perform busy-waits (using its own budget) when requesting a shared resource and are served in FIFO order. If a task is preempted or runs out of budget while holding a resource, it can migrate to a remote sever that has a task waiting for the same resource, and keeps executing by consuming the budget of the remote server until it is preempted again or releases the resource. M-BWI allows nested resource accesses via ordered locks (i.e., each shared resource is assigned with an order), where an access to an inner resource (say r^i) is allowed only if r^i has a greater order number than that of the currently holding resource [69].

Although M-BWI is for open or soft system, the significance of this protocol is that it introduces a novel approach to multiprocessor resource sharing technology that allows a task to use the processor time of another task. By doing so, a resource-holding task that is preempted or runs out its budget can still execute instead of causing a long blocking period to tasks that are waiting for the resource. In addition, compared to the preemptable spinning approach (e.g., PWLP in Section 2.5.4), unrelated high priority tasks under M-BWI incurs no arrival blocking at all as they can preempt lower priority tasks directly even if they are executing with shared resources. Later on, this protocol inspired the development of the Multiprocessor resource sharing Protocol [27] for hard real-time multiprocessor systems, as describe in Section 2.5.9.

2.5.9 Multiprocessor resource sharing Protocol

The Multiprocessor resource sharing protocol (MrsP) proposed in [27] aims at fully-partitioned systems with fixed priorities. The basic features of MrsP are similar to that of MSRP and PWLP, where spin locks are adopted and resources are served in FIFO order. However, MrsP holds a significant property where a helping mechanism is employed to help the resource-holding tasks to keep making progress while being preempted.

Before presenting the protocol, [27] states that on multiprocessor systems, there is a need to serialise the execution of resources due to parallel accesses to a shared resource. Thus, if r^k is served in FIFO order in a system with M processors, the accessing time of the resource for a task is at most $M \times c^k$ in the worst case, as given in Equation (2.8). The suspension-based approaches cannot achieve such a bounding with either prioritised or FIFO order, as more than one task from the same processor can be inserted into the queue. In addition, extra blocking terms must be included to bound the potential priority inversion when a task is suspended while a low priority task is executing with a resource. Therefore, [27] concludes that the bounding of $M \times c^k$ can only be achieved by some forms of the FIFO spin approach.

The non-preemptive FIFO spin locks (e.g., MSRP) does produce the desired bounding, but high priority tasks under this approach could be blocked upon each arrival, and are highly likely to miss their deadlines if critical sections are long. On the other hand, despite that spinning at base priority (e.g., PWLP) minimises the duration of arrival blocking to only one critical section, this approach leads to a prolonged resource-accessing time as tasks that are waiting for a shared resource can be preempted. Therefore, the authors of MrsP focus on the resource ceiling facility to limit the arrival blocking incurred by high priority tasks while minimising the resource-accessing time that can lead to Equation (2.8). The basic features of MrsP is summarised as below.

- Local resources are managed by SRP.
- Global resources are managed by spin locks and are served in FIFO order, where task at the head of the FIFO queue is always the resource-holding task.
- Each global resource is assigned with a set of resource ceiling priorities, one for each processor that contains task that uses the resource. The ceiling priority of a resource for a given processor is the highest priority level of tasks that use the resource on that processor.
- A task boosts its active priority to the corresponding local resource ceiling immediately when it requests a resource. It keeps executing with the ceiling priority during the entire access to the resource.
- After releases the resource, the task restores its priority to the previous priority level.

It is clear that the above definitions cannot lead to the desired bounding as resource-accessing tasks can be preempted when waiting for or executing with the resource. To achieve the blocking bounding, a helping mechanism is adopted in MrsP, where a task waiting to access a resource should be able to undertake the associated computations (i.e., critical section) on behalf of any other tasks accessing the same resource. The objective is that once a resource-accessing task is preempted, it can be helped by other waiting tasks with the wasted spin cycles to keep making progress rather than being held by the scheduler. In the worst case, a task needs to undertake the computations on behalf of all other tasks in the FIFO queue (according to the FIFO order) each time it tries to access a resource, which leads to the bounding of $M \times c^k$ for each resource access. The helping mechanism in MrsP can be summarised as below.

- A task that is waiting for a resource should be able to execute the computations of critical sections for any other tasks that are accessing the same resource.
- A helping task (i.e., a spinning task that helps other tasks to execute) should undertake the computations of other tasks according to the original FIFO order.

In [27], two approaches to realise the helping mechanism are presented. A duplicated execution approach can be adopted in which the access to a resource is independent, that is, given a certain state and input, the critical section produces the same output irrespective of how many times the operation is applied. However, this approach imposes strong restrictions to shared resources and limits the use scenario. A more realistic and commonly adopted approach is to migrate the locally preempted resource-accessing task to a processor where a task is actively spin-waiting to access the resource. After migration, the task is assigned the priority of the helping task and then resumes its execution with the resource on that processor. In practice, the migrated task is usually assigned a priority which is slightly higher than the priority of the helping task so that it can preempt the helping task. After the task releases the resource, it migrates back to its original processor (if necessary).

Combing the preemptive FIFO spin approach and the helping mechanism, MrsP achieves the minimised blocking bound in Equation (2.8) and can be fitted into the RTA-based analysis for MSRP systems (Equations (2.6) to (2.9)) with a minor modification to reflect the limited arrival blocking. Thus, Equation (2.9) is revised to Equation (2.11) as a task in MrsP systems can incur arrival blocking only if there exists a local lower priority task that is accessing a resource with a higher priority ceiling, where $Pri(r^k, P_m)$ gives the local ceiling priority of r^k on P_m and $P(\tau_i)$ returns the processor designated to τ_i.

$$\hat{e_i} = max\{e^k | N_{ll}^k > 0 \wedge Pri(r^k, P(\tau_i)) \geq Pri(\tau_i)\} \tag{2.11}$$

Figure 2.11 illustrates the working mechanism of MrsP by a dual-processor system with four tasks, as described below. The system used in the example is described in Table 2.10 with $Pri(r^1, P_1) = 1$ and , $Pri(r^1, P_2) = 4$ according to the resource usage.

Table 2.10: Tasks in the Example System of MrsP

Tasks	Required Resource	Partition
τ_1	r^1	1
τ_2	r^1	2
τ_3	-	1
τ_4	r^1	2

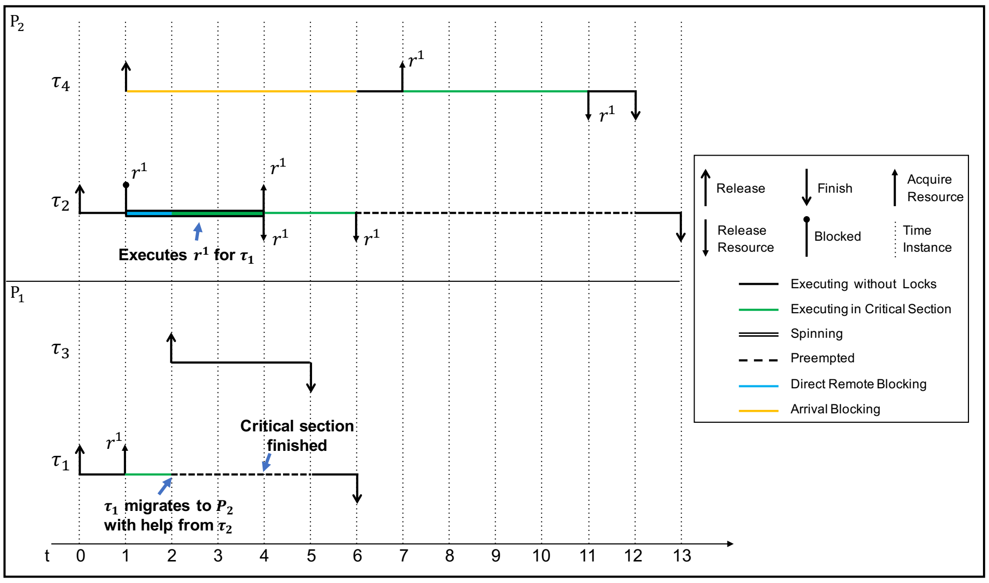


Figure 2.11: Example System of the Multiprocessor resource sharing Protocol

- At $t = 0$, τ_1 and τ_2 are released on both processor and request r^1 at $t = 1$. Suppose that τ_1 gets the lock first so that τ_2 starts spinning with a boosted priority of 4. Meanwhile, τ_4 is released on P_2 but incurs arrival blocking immediately.

- At $t = 2$, τ_3 is released on P_1 and then preempts τ_1 as it has a higher priority. At this point, τ_1 migrates to P_2 and is helped by τ_2 according to the helping mechanism. Thus, τ_1 resumes executing on P_2 with a priority slightly higher than τ_2's active priority (e.g., 5).

- τ_1 releases r^1 at $t = 4$ so that is migrates back to P_1, but still subject to preemption as τ_3 is not finished. Meanwhile, τ_2 can lock r^1 and starts executing.

- τ_3 is finished at $t = 5$ so that τ_1 can resume it execution, which is then finished at $t = 6$. At the same time, τ_2 releases r^1 so that it is preempted by τ_4.

- τ_4 locks r^1 at $t = 7$ and is then finished at $t = 12$. Then, τ_2 is resumed and is finally finished at $t = 13$.

As illustrated by the example, with MrsP, the unrelated high priority tasks do not need to incur arrival blocking upon its release (see τ_3 in this example) while the preempted resource-holding task can keep making progress with the help of remote spinning tasks (i.e., τ_2 in this example). Thus, τ_3 and τ_1 in this example can both have a short response time. If MSRP is adopted, τ_3 will be blocked at $t = 2$ so that its response time is 2 units of time longer than the case with MrsP applied while the response time of other tasks remains the same. In addition, with PWLP adopted, τ_3 still incurs arrival blocking as tasks are executing non-preemptively with resources. However, τ_4 in this case can have a shorter response time as it can preempt τ_2 and then gets r^1 after τ_1 releases the resource.

In [27], the preliminary approaches to support nested resource accesses are described, where either group locks or ordered locks can be adopted. Ordered locks are recommended to avoid the decrease of parallelism. In addition, the analysis that bounds the execution time cost for accessing nested resources are supported, as given in Equation (2.12), where function $V(r^k)$ returns a set of resources that access r^k. As the accesses to r^k in the nested fashion is serialised by the outer resource (only one task can hold a resource at a time), $|V(r^k)|$ can safely bound the number of requests that can cause blocking for accessing r^k.

$$e^k = (|V(r^k)| + |map(G(r^k))|)c^k \tag{2.12}$$

However, as the accessing cost of r^k's inner resources are not studied, this analysis fails to bound the transitively blocking where r^k also requests an inner resource, but in turn, is blocked as that resource is currently being accessed by another resource [49]. In addition, with nested resource accesses allowed, a task can access inner resources while being helped (i.e., on a remote processor), which could lead to a further priority boosting. With the boosted priority, the task could block the currently executing task on its host processor while migrating back with the resource (i.e., incurs preemption again while being helped), which breaks the property of PCP that a task can incur local blocking only once. Therefore, as stated in [49], the current version of MrsP (and its analysis) is insufficient to be adopted in systems with nested resources due to extra local blocking and unbounded transitive blocking time when accessing inner resources.

MrsP is significant due to its preemptive FIFO spin approach with the helping mechanism, which is attractive to systems where the unrelated high priority tasks are assigned with a short deadline. However, as a relatively new protocol, the current version of MrsP contains certain issues in its definitions, which can lead to inaccurate results by the current schedulability test and poor run-time efficiency [109, 110]. In theory, MrsP can achieve the minimised blocking bound due to the helping mechanism. However, introducing migrations can impose considerable amount of run-time overheads to resource-accessing tasks, which leads to longer resource-accessing time than the theoretical value (i.e., $|map(G(r^k))|$) [109]. In addition, with the migration-based helping mechanism, a resource-holding task can be preempted (and hence, can migrate) frequently so that the task can spend more time migrating rather than executing with the resource [110]. Therefore, to guarantee the correctness and to improve the efficiency of resource accessing behaviours in MrsP, the above issues must be addressed before practising this protocol.

2.5.10 Summary and Discussion

This section provided a detailed review of the major multiprocessor resource sharing protocols for real-time systems, and summarised their advantages, drawbacks and limitations with illustrations. Based on the review, the resource sharing protocols on multiprocessors can be classified as two families: the suspension-based family and spin-based family, as shown in figure 2.12, where each protocol has an unique combination of resource classification,

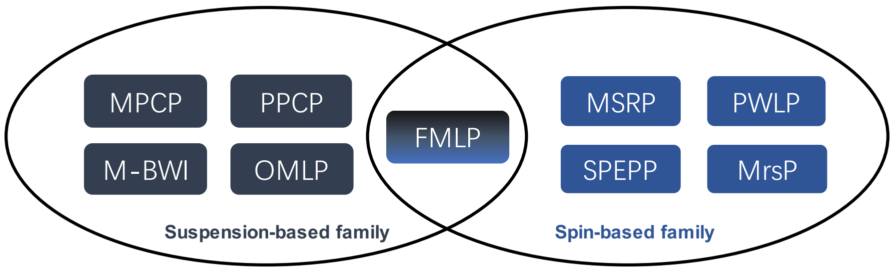


Figure 2.12: Reviewed Protocols in Spin and Suspension-based Families

queueing techniques and resource-accessing priority rules. In addition, some newly-developed protocols (e.g., PWLP, SPEPP and MrsP) also contain an additional mechanism (e.g., the cancellation mechanism and the helping mechanism) to reduce certain blocking terms. Note that as FMLP adopts both spin and suspension-based locks, this protocol belongs to both families.

Table 2.11: Features of Suspension-based Multiprocessor Locking Protocols

Protocol	Resources	Accessing Priority	Queuing Technique	Additional Facility	Nested Resources
MPCP (generalised)	Global & Local	Priority Ceiling	Priority Ordered	-	-
FMLP	Short & Long	Priority Inheritance for Long Resources	FIFO	-	Group Locks
PPCP	-	Priority Inheritance	Priority Ordered	Tunable α	-
OMLP	Global & Local	Priority Inheritance or Non-Preemptive	FIFO & Priority Ordered	-	Group Locks
M-BWI	-	Base Priority of Servers	FIFO	Migration-based Helping	Ordered Locks

Table 2.11 summarises the properties of the multiprocessor locking protocols in the suspension-based family. These protocols classify shared resources as either local and global or long and short resources, or treat them equally. In addition, various resource-accessing priority rules and queuing techniques are adopted in these protocols to manage resource accesses. Among the resource-

accessing priority rules, resource ceiling facility and non-preemptive sections are straightforward and effective with low costs while the priority inheritance facility imposes more run-time overheads due to the need of updating the active priority of the resource-holding task frequently. As for queuing techniques, the prioritised ordering benefits high priority tasks but prolongs the waiting time for low priority tasks while the FIFO ordering achieves a less bounding in general but still cannot achieve a bounded waiting queue, as more than one task in the same processor can join into the queue with suspension-based locks. The bounded blocking time can be achieved by combining both queuing techniques (i.e, FMLP, which provides a bounding of M in systems with M processors), but can lead to a highly-complicated system with considerable run-time overheads compared to the non-preemptive spin locks (i.e., MSRP), which guarantees a shorter blocking bound of $M - 1$. In addition, PPCP and M-BWI propose addition facilities that can reduce certain blocking terms. However, the α facility used by PPCP imposes significant run-time overheads while M-BWI is for soft real-time systems, which is not the focus of this thesis. At last, nested accesses between global resources are supported by FMLP, OMLP and M-BWI via either group locks or order locks.

Resources under spin-based protocols have the same classifications as the suspension-based approach, but are typically served in FIFO order to provide strong progress of resource execution, with various accessing priority rules that benefit certain tasks and resources. Features of the spin-based protocols reviewed in this section are summarised in Table 2.12. The non-preemptive spin locks (i.e., tasks are waiting for and executing with resources non-preemptively) guarantees the bounded blocking for each task but is less favourable for high priority tasks with long critical sections. Spinning at base priority level can benefit high priority tasks but prolongs the blocking time of low priority tasks due to preemptions. Spin locks with a resource ceiling facility (i.e., MrsP) provides a trade-off between the amount of arrival blocking incurred by high priority tasks and the number of tasks that can incur preemptions while accessing resources. In addition, to reduce the impact of preemptions to resource-accessing tasks with preemptable spin locks, additional facilities are proposed in PWLP, SPEPP and MrsP, where the cancellation mechanism avoids the delay for tasks that are waiting behind the preempted task while the helping mechanism (with either the operation blocks or the migration-based approach) can reduce of the prolonged delay due to pre-

Table 2.12: Features of Spin-based Multiprocessor Locking Protocols

Protocol	Resources	Accessing Priority	Queuing Technique	Additional Facility	Nested Resource
MSRP	Global & Local	Non-Preemptive	FIFO	-	-
FMLP	Short & Long	Non-Preemptive for Short Resources	FIFO	-	Group Locks
PWLP	Global & Local	Base Priority for Waiting; Non-Preemptive for Holding	FIFO	Cancel	Group Locks (Recommends)
SPEPP	Global & Local	Base Priority for Waiting; Non-Preemptive for Holding	FIFO	Operation Blocks	-
MrsP	Global & Local	Priority Ceiling	FIFO	Migration-based Helping	Ordered Locks or Group Locks (Preliminary)

emptions incurred by all resource-accessing tasks. As for supporting nested resources, PWLP recommends group locks with no further details given while MrsP lacks of a complete approach to support nested resource access and an effective schedulability test to bound the potential blocking terms for accessing nested resources.

As stated in [37], there exists no optimal resource sharing solution for multiprocessors, where the performance of each protocol varies under different application semantics and resources characteristics. Thus, it is not possible to achieve the best performance by adopting any of the protocols as a generic resource sharing solution for all multiprocessor systems. One obvious metric of choosing an appropriate resource sharing protocol is reported in [22], where the spin locks are preferable for resources with short critical sections while the suspension-based approach can benefit long critical sections. This observation directly motivates the development FMLP. However, this protocol is relatively complicated and can impose considerable amount of run-time overheads due to the combined locking approaches. In addition, the fact that the performance of a locking protocol may also be affected by other factors that have not been studied in previous studies of resource sharing on multiprocessors further complicates the problem of deciding appropriate resource sharing protocols for systems with certain characteristics. For instance, resources with strong contention (where there exist many tasks that request a resource multiple times on different processors) can have a huge impact to the schedulability of PWLP systems as a task can incur prolong blocking due to newly-arrived resource requests each time it re-joins into the resource-accessing routine, after being resumed from preemptions. However, adopting MSRP may provide better schedulability in this case due to the non-preemptive approach.

The discussion above reflects the first two aims of this thesis (as given in Section 1.2), which are the need of a combination of appropriately chosen protocols to reduce the scheduling penalty of managing shared resources with various characteristics and the need for a schedulability test that supports systems with multiple protocols working in collaboration simultaneously. In Chapter 3, the candidate resource sharing protocols for the proposed multiprocessor resource control framework are determined from the reviewed protocols by examining the major factors that can affect the performance of resource sharing protocols and by comparing the locking approaches proposed by above protocols. In addition, a schedulability test framework is also developed to support the analysis of systems that adopt a combination of resource sharing protocols.

2.6 Further Results in Resource Sharing on Multiprocessors

Besides the development of resource sharing protocols, other works had been proposed to facilitate multiprocessor resource sharing by reducing the schedulability penalty for managing shared resources while allocating tasks and by improving schedulability tests for resource sharing protocols. This section describes the previous results that also lead to the aim of this thesis given in Section 1.2, including (1) the need for new resource-oriented task allocation schemes; (2) the need of addressing underlying issues that can undermine the schedulability analysis for the proposed resource control framework; and (3) the need for testifying the optimality and availability of the optimal priority ordering algorithms (i.e., DMPO, OPA and RPA in Section 2.1.3) for the new schedulability test framework.

2.6.1 Resource-aware Task Allocation Schemes

The heuristic approaches are usually adopted to solve the NP-hard bin-packing problem when mapping tasks on multiprocessors and are effective with independent tasks [37] i.e., without the presence of shared resources. Besides the traditional heuristic approaches described in Section 2.2.3 (e.g., the Worst-Fit and Best-Fit schemes), several search-based algorithms are also proposed to facilitate task mapping with improved robustness, flexibility and extensibility of multiprocessor systems [42, 43, 111]. For instance, [43] proposes a task allocation algorithm based on the simulated annealing technique [85] that optimises the extensibility of multiprocessor systems and minimises the changes required for future system upgrading e.g., changes of task priorities and execution times.

However, with shared resources, adopting these task allocation schemes can lead to strong resource completion from multiple processors as shared resources are not taken into account in these algorithms, where tasks requesting the same resource could be allocated to different processors, and hence, incur prolonged blocking. To reduce the overall blocking of systems with shared resources, several task allocation schemes are developed that also take resources into account when grouping tasks into processors [55, 62, 81]. However, some of the algorithms rely on task migrations and uniprocessor locking protocols, such as the algorithms proposed in [55], where tasks that request a resource must migrate to a designated processor during each access while resources are managed by either PCP or NPP. As this thesis focuses on the multiprocessor resource sharing protocols, this section reviews the resource-oriented task allocation schemes that are applicable to the multiprocessor resource sharing protocols reviewed in Section 2.5.

2.6.1.1 Synchronisation-aware Partitioning Algorithm

The Synchronisation-aware Partitioning Algorithm (SPA) proposed in [62] is extended from the Best-Fit scheme with tasks ordered by utilisation non-increasingly (i.e., BFD), which aims to reduce the blocking time due to resource-accessing in multiprocessor systems via localising the globally shared resources. In SPA, the notion of *task bundling* is proposed to facilitate task allocating, where tasks that share the same set of resources are grouped as a *task bundle*, as described below with a given task set that shares a set of resources R.

1. Starts from the first resource (say r^1) in R and inserts the tasks that request this resource (i.e., $G(r^1)$) into the first task bundle. The resources can be ordered by any metrics (e.g., by their indexes), which will not affect the task bundling outcomes as the algorithm aims to bundle tasks that shared the same resources.

2. Gets all the resources required by the tasks in that bundle, and places all the tasks that also require those resources into the same bundle. That is, the task bundling considers the transitive resource sharing. For instance, if $G(r^1) = \{\tau_1, \tau_2\}$ while τ_2 and τ_3 require r^2, τ_3 will also be grouped to that bundle.

3. Repeats step 2 until a constant task bundle is obtained.

4. Generates task bundles for the resources that do not appear in the existing bundles via above strategy.

5. The independent tasks (i.e., the tasks that do not require any resources) will not be placed into any task bundles.

With the task bundles generated, the SPA algorithm starts to allocate these bundles and the independent tasks via the BFD scheme, as described below. The allocation starts with M processors, which can allocate the total utilisation (i.e., U_{total}) of the system (e.g., if $U_{total} = 900\%$ then $M = 9$) theoretically.

1. Firstly, the task bundles are ordered by utilisation non-increasingly and are allocated via the BF approach (i.e., BFD), where a task bundle that cannot fit into a single processor will be examined later on.

2. Then, the independent tasks are allocated via the BFD approach. If a task cannot be mapped to any of the existing processors, a new processor will be added.

3. If a new processor is added, the unallocated task bundles are checked again to see whether a bundle can be mapped to that processor, according to the BFD approach.

After the above steps, only the task bundles that cannot fit into a single processor is left un-allocated, which need to be broken in order to obtain a feasible allocation. As stated in [62], breaking a task bundle indicates transforming its resources into global resources, which imposes the penalty of additional processor utilisation from the viewpoint of utilisation. Then, [62] presents a set of rules specifying the ordering of these task bundles and the approach for breaking them, as described below.

1. Tasks in each bundle are ordered by utilisation non-increasingly.
2. Each task bundle that requires breaking has a cost, which is the sum of the maximum utilisation penalty of all its resources for breaking the bundle into two pieces i.e., $Cost_{tb} = \sum_{r^k \in F(tb)} c^k / min_{\tau_x \in G(r^k)}\{T^x\}$, where tb denotes a task bundle and $F(tb)$ returns a set of shared resources that are required by the bundle tb.
3. Tasks bundles are ordered by their costs in a non-decreasing fashion, and the bundle with the smallest cost is selected to be broken.
4. The selected bundle is broken into two pieces so that the utilisation of one piece is as close as the largest utilisation available among the existing processors, in accordance with the BFD approach. This procedure repeats until this task bundle is allocated.
5. If this allocation is not feasible, a new processor is added and the whole allocation strategy is repeated again, where each unallocated bundles are examined to check where a whole bundle can be fitted into the new processor.

The above described the detailed task grouping and allocating approaches in SPA algorithm. As proved in [62], compared to the traditional task allocation schemes (e.g., the WF heuristic), the total number of globally shared resources can be effectively reduced with SPA adopted so that a higher schedulability can be obtained.

However, SPA aims at localising as many resources as possible despite the characteristics of the resources. Consider the case where MSRP is the only available protocol, and short resources are all localised while long resources are still accessed from many processors, such a system may still contain certain degree of pessimism as MSRP is less favourable for long resources. Such a situation could happen if task bundles that request long resources have a utilisation higher than 100% while each bundle that wants short resources can be allocated into a single processor. In contrast, if localising long resources instead of the short ones, the schedulability penalty for managing shared resources with MSRP could be further minimised as the global resources (i.e., the short ones) are now managed by an appropriate protocol. Therefore, partitioning schemes that minimise the blocking due to certain types of resources via localisation could also benefit resource sharing and could outperform algorithms that merely reduce the number of global resources (assuming an appropriate resource sharing protocol is adopted), and hence, are also desirable.

2.6.1.2 Blocking-aware Partitioning Algorithm

The Blocking-aware Partitioning Algorithm (BPA) proposed in [81] is also extended from BFD, but works explicitly with MPCP. With BPA adopted, each task is assigned with a weight, which denotes its utilisation plus the amount of potential remote blocking that the task can incur for accessing shared resources under MPCP. For brevity, details of calculating the weight are not described in this thesis and are referred to [81]. Similar to BPA, SPA puts the tasks that directly or indirectly share a same set of resources into the same group, which is denoted as a *macrotask*. If a macrotask cannot be fitted into a single processor, it is marked as breakable. Otherwise, it is set to be unbroken. The unbroken marotasks and independent tasks are inserted into a same list and are sorted by their weights in non-increasing order.

After the above steps, SPA performs task allocating in two rounds and selects the partitioning result with less processors required among the outputs generated by both rounds. Both rounds allocate independent tasks and unbroken macrotasks according to BFD, but map breakable macrotasks by different strategies based on an *attraction* value between tasks in the same macrotask. The attraction value of τ_1 to τ_2 denotes the remote blocking that τ_1 can introduce to τ_2 (with MPCP adopted) if they are allocated into different processors, where a larger value represents more blocking time.

Assuming τ_x is the currently to-be-allocated task from a breakable macrotask, the first round of BPA orders other tasks in that macrotask based on the attraction values of those tasks to τ_x in non-increasing order, where τ_x is at the head of the attraction list. Then, the algorithm selects the processor that can fit the most tasks in that attraction list and repeats this procedure until all tasks are allocated.

The second round creates a processor list and identifies the most appropriate processors for τ_x in two steps. First, it inserts the processors that contain tasks from τ_x's macrotask and sorts them by processors attraction to τ_x in non-increasing order. The processor attraction of a given processor P_m to τ_x is the sum of the attraction of tasks on P_m to τ_x. Second, the algorithm orders the processors that do not contain tasks from τ_x's macrotask by utilisation non-increasingly and then places them at the end of the same processor list. Followed by this order, τ_x is allocated to the first processor that it can be fitted into from the processor list. The rationale of this approach is to minimise the blocking incurred by τ_x from tasks in its macrotask as much as possible by finding the processor with the highest attraction that can fit τ_x. If not feasible, the algorithm then checks the following processors in the list.

As shown in [81], systems with BPA adopted demonstrate better schedulability and require less processors than that of under SPA and the traditional task allocation schemes. Compared to SPA, which reduces the number of globally shared resources, BPA is more advanced due to the awareness of blocking and the sophisticated two-round partitioning approach. However, the current version of BPA can only be applied with MPCP assumed. To support other protocols, the corresponding weight and attraction functions must be developed.

As for the resource control framework proposed in this thesis, applying BPA requires extremely complicated weight and attraction functions due to the use of multiple resource sharing protocols. In addition, adding a new candidate resource sharing protocol also requires modifications to BPA's functions, which greatly undermines the usability of this algorithm as well as the proposed framework. Thus, resource-oriented task allocation schemes that are independent from the locking protocols (i.e., can be adopted with any locking protocols assumed) are more desirable for the flexible resource control framework. In future work, we aim to extend the BPA algorithm to support other multiprocessor resource sharing protocols and to support to use of multiple protocols simultaneously in one system. Then, the performance of this algorithm will be investigated with the presence of multiple locking protocols.

2.6.2 Improved Schedulability Tests for Multiprocessor Resource Sharing Protocols

As described in Sections 2.5.2 and 2.5.9, both MSRP and MrsP are supported by the RTA-based schedulability tests, where the cost for accessing a resource (say r^k) is bounded by $|map(G(r^k))| \times c^k$ (i.e., the number of processors with tasks requesting r^k). The rationale of this bounding is that in the worst case, a task can be blocked by each remote processor that requests the resource once in each access. In addition, the interference from high priority tasks incurred by a task is determined by the total worst-case computation time of each higher priority task $(\widehat{C_h})$, which includes the potential blocking for accessing each shared resource. Such techniques are effective and can surely capture the worst-case scenario. However, recently studies toward the schedulability tests of resource sharing protocols have revealed issues underlying these analysing techniques, which can introduce pessimism and undermine the accuracy of the schedulability results.

2.6.2.1 Holistic Analysis

As stated in [19], the blocking bound $|map(G(r^k))| \times c^k$ applied in the original tests of both MSRP and MrsP can be pessimistic due to the assumption that each time a task requests r^k, there will always be a task waiting for r^k on each processor that wants the resource.

Figure 2.13 illustrates the scenario where a critical section is accounted for more than once due to the adoption of this assumption. In the given system, each task requests the same resource, say r^1, a number of times during each release, there τ_1, τ_3 and τ_4 access r^1 3 times while τ_2 and τ_5 request r^1 2 times during each release. As shown in the figure, one arrow indicates one resource access to r^1.

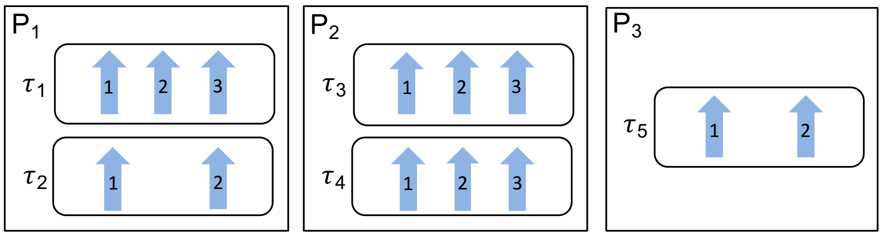


Figure 2.13: Issues of the Original RTA-based Schedulability Tests.

Suppose that during the release of τ_3 or τ_4, other tasks will only be released once (i.e., $\left\lceil \frac{R_3}{T_x} \right\rceil = \left\lceil \frac{R_4}{T_x} \right\rceil = 1$ for any given τ_x in the system). For τ_3 itself, it can be blocked 3 times from P_1 and 2 times from P_3 for accessing r^1, as there are only 2 requests issued from P_3 to r^1 during τ_3's release. However, with the original analysis of either MRSP or MrsP, τ_3 incurs 6 blocking in total as the analysis assumes that each time τ_3 accesses the resource it incurs blocking from both P_1 and P_3 i.e., the processors with tasks requesting the same resource, which accounts for one more critical section into the blocking time. Due to the same reason, τ_4 also incurs 6 times of blocking for accessing the resource with the original analysis adopted. However, in reality, τ_4 will be blocked twice from P_1 while accessing r^1 and its third request will not be blocked at all as other remote requests delay τ_3 directly (thus can only block τ_4 indirectly) and should be accounted for as part of the high priority task interference of τ_4.

This issue is addressed by the holistic analysis proposed in [19], where the blocking time of a task for accessing a resource is bounded via calculating the exact number of critical sections that can be issued from each remote processor during the release of that task. In this analysis, the blocking time incurred by a task τ_x for accessing a resource r^k is analysed by checking whether there exist any remote requests to r^k that are not yet accounted for on each remote processor for each τ_x's access to r^k.

By computing the exact number of requests that each task can issue to each resource, this approach breaks the assumption described above and can provide less pessimistic results than that of the original tests. However, unlike the original tests, applying such an analysis requires full knowledge of the system, including the exact number of requests issued by each task to each resource. In the interest of brevity, the detailed analytical expression and the formal proof of the holistic analysis are not given in this thesis and are referred to [19].

2.6.2.2 Integer Linear Programming-based Analysis

Later on, [106] stated that even if the blocking time can be precisely bounded without over-calculating any critical sections, the original tests still subject to certain degree of pessimism due to the approach of inflating tasks' computation time with blocking (see notation $\widehat{C_i}$ in Section 2.5.2, which denotes the worst-case execution time of τ_i plus the cost for accessing each resource with potential delay).

Consider the same example in Figure 2.13, we now focus on τ_4 and assume that during τ_4's release τ_3 can be released (and preempt τ_4) 3 times so that $\left\lceil \frac{R_4}{T_3} \right\rceil = 3$ while other tasks are released once. Even with the techniques of the holistic analysis adopted (i.e., the blocking is bounded precisely), the interference of τ_4 is $3 \times \widehat{C_3}$, where $\widehat{C_3} = C_3 + 3c^1 + 5c^1$ as τ_3 can only incur 5 times of blocking during 3 accesses to the resource. By doing so, the analysis assumes that each time when τ_3 is released in the context of τ_4, it can be blocked 5 times from P_1 and P_2, which is 15 blockings in total during 3 releases. However, as other tasks are released only once during the release of τ_4, there are at most 7 remote requests that can block τ_3's requests so that 8 critical sections are over-calculated.

To address this issue, [106] proposed a schedulability test framework for spin locks on multiprocessors based on the Integer Linear Programming (ILP) technique. This analysis separates the blocking time from the task's execution time and accounts for this blocking in parameter B_i, which now reflects the total blocking a task can incur during one release, including all potential blocking time due to resource-accessing on multiprocessors (i.e., direct remote blocking, indirect remote blocking and arrival blocking). With a set of constraints applied, this blocking variable B_i can be calculated and safely bounded via a ILP solver with the principle that one remote request can only cause one blocking.

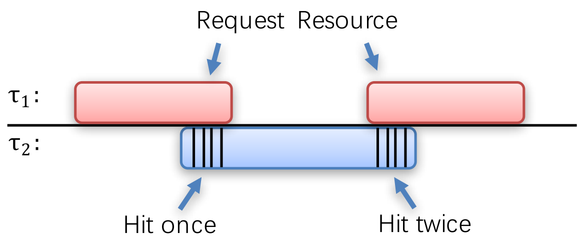


Figure 2.14: The Back to Back Hit Phenomenon.

In addition, [106] pointed out that to accurately account for the number of requests that are issued during the release of a given task, the *back-to-back hit* must be accounted for. This phenomenon is demonstrated in Figure 2.14, where τ_1 can be released only once during the release of τ_2 (i.e., $\lceil \frac{R_1}{T_2} \rceil = 1$) yet can cause one more blocking period (i.e., the black lines in τ_2's execution) due to the resource access in its last release ($\lceil \frac{R_1+R_2}{T_1} \rceil = 2$). To account for the potential blocking from this additional access, the ILP-based analysis computes the number of requests to a given resource by the response time of both the resource requesting task and the task that is currently being calculated, which guarantees the blocking time can be safely bounded.

With the above issues addressed, the ILP-based analysis can provide less pessimistic as well as more accurate schedulability results than both the original tests and the holistic analysis. In [106], 30 constraints are developed to support a wide range of spin locks. With the corresponding constraints adopted, this ILP-based analysis framework can provide schedulability tests to 8 spin-based resource sharing protocols, including MSRP and PLWP described in Section 2.5.

However, due to the use of the ILP technique, this analysis is considerable complicated in its analytical expression and is expensive during practice due to the need for a ILP solver. Section 3.4.4 provides evidence that compared to the original analysis techniques, the use of the ILP solver can take massive computation time while calculating the response time, which could greatly undermine the usability of the analysis even if such computations are usually performed off-line.

In addition, although supporting a wide range of spin-based protocols, this analysis does not consider any helping-based protocols (e.g., SPEPP and MrsP), which are equally important compared to other protocols and have their unique advantages. Thus, applying such an analysis to the resource control framework imposes a strong restriction to the range of available protocols when determining the candidate resource sharing solutions. More importantly, the nature of the resource control framework (where multiple protocols are working together simultaneously) also makes this analysis inapplicable, which can only support the analysis of one protocol at a time.

Combining the discussion above, this thesis aims to develop a new schedulability analysis explicitly for the resource control framework rather than combining or modifying any of the existing tests. The new schedulability test must directly support the analysis of all candidate resource sharing protocols and be able to analyse systems with multiple protocols in use. In addition, the new schedulability analysis must address the above issues to avoid the pessimism as well as to provide accurate results, but should do so without the need for any expensive techniques (e.g., the ILP technique) to avoid massive (or even impractical) computation time when searching for a feasible locking protocol for each resource.

In addition, considering the back-to-back hits could raise new issues to priority ordering algorithms. As described above, with the back-to-back hits taken into account, the response time of a task depends on the response times of potentially all other tasks in the system, including its local lower priority tasks. With such a schedulability test, whether DMPO remains optimal becomes uncertain and whether OPA and RPA can be applied are unknown. Therefore, an investigation should be conducted towards the optimality of

DMPO and the available of OPA and RPA under such a new schedulability test. If DMPO is not optimal (i.e., there exist better priority ordering that cannot be found by DMPO) while OPA and RPA are not compatible with this new schedulability test, a search-based priority ordering algorithm that is independent from the schedulability tests can also facilitate resource sharing on multiprocessors, and hence, is also desirable.

2.7 Summary

This chapter firstly provided the background knowledge and basic concepts of real-time systems that are related to the research proposed in this thesis. Then, the real-time resource sharing model and resource sharing technology for both uniprocessor and multiprocessor systems are discussed in detail, which includes a wide range of resource sharing protocols, resource-aware task allocation schemes and the analysing techniques for systems with shared resources. While presenting this literature review, the task and system model of the research in this thesis is formed with rationales presented, where this thesis focuses on multiprocessor real-time systems with:

- Sporadic task model with constrained deadlines.
- Fixed priority preemptive scheduling.
- Symmetric multiprocessor architecture.
- Fully partitioned scheduling scheme.
- RTA-based schedulability analysis.
- Lock-based synchronisation approach.
- Homogeneous cost for executing a resource.

Note that assuming homogeneous cost for executing a resource is only for the ease of presentation of the new schedulability test proposed in this thesis. The resource control framework can work with heterogeneous resource accesses with materials given in Appendix A.

As described, managing shared resources on multiprocessors can often lead to considerable schedulability penalty due to the prolonged blocking for accessing global resources. Although there exist various multiprocessor locking protocols, each protocol can demonstrate better schedulability than others with certain application semantics and resource characteristics. Therefore, using any of the protocols as a generic resource sharing solution can introduce certain degree of pessimism to the system. In addition, the traditional utilisation-based task allocation schemes could further magnify such schedulability loss due to the unawareness of shared resources. Further, the issues in the original schedulability tests can lead to pessimistic as well as inaccurate schedulability results. Although research efforts have emerged to provide resource-aware task allocation schemes and improved schedulability analysis for certain locking protocols, these newly-proposed techniques either impose strong application restrictions (e.g., can only be used with certain protocols) or has complicated expressions with massive computation time (e.g., by using the ILP techniques).

The above summarises the challenges and issues for managing shared resources on multiprocessor platforms and directly reflects the aim and the hypothesis of this thesis, where the schedulability sacrifice for managing shared resource in multiprocessor systems can be minimised by adopting:

- A combination of appropriately chosen protocols, where each protocol only control certain resources that it can benefit.
- Resource-orientated task allocation schemes with full knowledge of the usage and characteristics of each shared resource and are independent from the multiprocessor resource sharing protocols.

In addition, to provide analysable systems with the use of multiple resource sharing protocols, a schedulability analysis must be supported, which should:

- Directly supports the analysis of each resource sharing protocol that are in use.
- Be able to analysis systems with more than one protocols working simultaneously.
- Addresses the issues identified in [106] for less pessimistic as well as more accurate schedulability results than that of the original schedulability tests.

The requirement of addressing the issues identified in [106] in the new schedulability analysis implies that the back-to-back hit phenomenon must be considered to guarantee accurate analysing results. Thus, the optimality of DMPO and the availability of OPA and RPA must be examined due to the nature of such schedulability tests, where the response time of a task depends on potentially the response time of all other tasks in the system. If DMPO is proved to be not optimal while other searching-based priority ordering algorithms are either inapplicable or not optimal, a new search-based priority assignment algorithm should be proposed to provide feasible priority ordering where the existing priority ordering algorithms cannot. Of course, this priority assignment algorithm must be fully compatible with the new schedulability analysis.

Based on the above discussion, the following chapters propose a Flexible Multiprocessor Resource Sharing (FMRS) framework that can effectively reduce the schedulability loss (i.e., the increase of the response times of tasks due to accessing shared resources) when managing shared resources via integrating the above techniques. First, the candidate resource sharing protocols are determined and the schedulability analysis that meets the above requirements is derived in Section 3. Then, new resource-orientated task allocation schemes based on both the resource-usage and the resource characteristics are developed in Section 4, which are compatible with all locking protocols. In addition, the optimality and availability of the existing optimal priority ordering algorithms are examined under the new schedulability test in this section.Then, a new search-based priority ordering algorithm is developed to facilitate task priority ordering with the presence of shared resources on multiprocessors. Finally, the working mechanism of the complete FMRS framework is presented in Section 5, including the approach for searching the resource sharing, task allocation and priority ordering solutions to achieve a schedulable system.

Chapter 3

Candidate Locking Protocols and Schedulability Tests

This chapter aims to determine the candidate resource sharing protocols and to provide new schedulability analysis for the multiprocessor resource control framework. With the candidate resource sharing protocols determined, new schedulability analysis for each candidate protocol is firstly developed. Then, these tests are combined to form a complete run-time overheads-aware schedulability analysis framework that supports systems with potentially all the candidate protocols working simultaneously. Finally, a set of evaluations are conducted to investigate the schedulability of each candidate locking protocol and to provide evidence that supports the decisions made in this chapter. Materials provided in this chapter directly satisfies the Success Criteria SC-1 given in Section 1.4.

3.1 Deciding the Candidate Multiprocessor Resource Sharing Protocols

Section 2.5 has provided a detailed review for a wide range of multiprocessor resource sharing protocols, and each of them can be a potential candidate resource sharing protocol for the proposed resource sharing framework (namely FMRS). While deciding the candidate resource sharing protocols, a set of problems are encountered, as given below. Via step-by-step reasoning, this section discusses each of the questions and then determines the candidate locking protocols based on the conclusions.

1. Should the resource control framework contains as many candidate resource sharing protocols as possible?

In theory, containing as many candidate resource sharing solutions as possible could provide strong schedulability for a wide range of application semantics with various resource characteristics, given the fact that no locking protocol can dominate others. However, in practice, considering all the reviewed protocols as the candidate resource sharing solutions is impractical as this can result in an extremely complicated run-time system with significant overheads and a highly complicated schedulability analysis framework. To achieve high run-time efficiency and high usability of the proposed resource control framework (i.e., relatively easy to implement and analyse), only a limited number of protocols should be adopted as the candidate resource sharing solutions in FMRS for reducing the schedulability penalty while managing shared resources on multiprocessors.

2. Should the resource control framework employs both the suspension-based and the spin-based locking?

Admittedly, the framework with both locking approaches adopted can demonstrate stronger schedulability with a wider range of critical section length than that of only adopting one locking approach, where the spin locks mainly control short resources while the suspension-based locks focus on long resources. However, adopting both synchronisation approaches can lead to a considerably complicated system with high run-time overheads, where each type of lock could require unique queueing techniques and resource-accessing priority rules. In addition, adopting both approaches also requires the support of both locking primitives from the underlying operating system. If not, users must implement all the locking and queuing primitives before realising the candidate resource sharing protocols, which can undermine the usability of the proposed framework. Thus, to reduce the implementation complexity and run-time costs, either spin locks or the suspension-based locking should be employed in the proposed FMRS framework.

3. Which synchronisation approach should the framework employ?

According to the review given in Section 2.5, it is obvious that neither spinning nor suspension can dominant the other and their performance varies with different resource characteristics, especially with resources that have various critical section length. Compared to spin locks, suspension-based locking is more favourable with long resources, where tasks waiting for a resource is suspended so that other tasks can keeping executing. However, if critical sections are short, the frequent context switches introduced by the suspension-based locks can lead to considerable run-time overheads, which could be even larger than the cost for executing the resource. In contrast, spin locks carry low run-time overheads without the need for switching the resource-requesting tasks and are preferable with short resources. Therefore, as stated in [22], the suspension-based locking approach is *never* favourable to spin locks with short resources assumed.

The suspension-based locks can cause prolonged resource-waiting queue as more than one task can request the same resource (i.e., joins into the resource-waiting queue) on a processor at a given time, which leads to complicated blocking time bounding from the viewpoint of schedulability analysis. In contrast, due to the nature of spinning, spin locks can guarantee a strong progress of resource execution and can effectively bound the resource-accessing queue. With FIFO queuing and non-preemptive resource accessing assumed (e.g., MSRP), the blocking time of tasks for accessing a resource can be effectively bounded to $M-1$ by the original analysis on a multiprocessor system with M processors. Even with the preemptive approach (e.g., MrsP), spin locks can still achieve such a bounding in theory due to the helping mechanism.

In addition, the preemptive spinning approach (e.g., PWLP and MrsP) can improve the performance of spin locks under long critical sections to a certain extent. This is because tasks spinning for a resource can be preempted by local higher priority tasks, and hence, offer valuable processor time to tasks that are more urgent to execute. Section 3.4.1 provides a detailed discussion of spin locks with long resources and presents experimental evidence showing that the preemptive spin locks can demonstrate strong schedulability with long critical sections.

Furthermore, the fact that spin locks are widely available at the kernel level and are largely employed in practice also reflects the superior of this locking approach [25, 37]. For instance, the Automotive Open System Architecture (i.e., AUTOSAR) for automotive electronic control units has explicitly mandated the use of spin locks for managing shared resources [40, 46]. According to [22], the suspension-based locking should be avoided for global resources under fully-partitioned systems. As demonstrated in [22], with the current analysing techniques (e.g., the RTA-based analysis), the suspension-based approach is *never* preferable to spin locks from the viewpoint of schedulability and schedulability test.

Summarising the above discussion, the candidate resource sharing protocols should be based on spinning only, but should with various features (e.g., resource accessing priority rules) to provide a generic resource sharing solution for a wide range of application semantics and resource characteristics in multiprocessor systems.

4. With Spin-based locking decided, should the spin locks served in the FIFO order, or the priority order, or both?

In addition to the FIFO spin-based protocols reviewed in Section 2.5, there also exist several spin-based protocols with priority ordering employed [106]. However, adopting the prioritised ordered spin locks can prolong the waiting time of low priority tasks, and hence, can jeopardise their timing requirements. Therefore, to further reduce the implementation complexity (i.e., one queueing techniques only) and to achieve a shorter bounding of the resource-accessing queue in general (where low priority tasks can be benefitted while the blocking time of high priority tasks is also safely bounded), the FMRS framework proposed in this thesis focuses on spin-based protocols with FIFO order assumed.

The Candidate FIFO Spin-based Resource Sharing Protocol

Combing the conclusions above, the scope of the candidate resource sharing protocols is narrowed down to the FIFO spin-based protocols. From the reviewed protocols in Section 2.5, three protocols are decided as the candidate resource sharing protocols for the proposed resource control framework, which are MSRP, PWLP and MrsP. These three protocols basically cover all the features in the reviewed spin-based protocols (see Table 2.12), which include various resource accessing priority rules (i.e., non-preemptive in MSRP, base priorities in PWLP, and ceiling priorities in MrsP) and additional facilities (i.e., the cancellation mechanism and the helping mechanism).

Among these candidate protocols, the non-preemptive spinning approach (i.e., MSRP) can provide strong schedulability with short critical sections while the preemptive approaches (i.e., PWLP and MrsP) can demonstrate better schedulability with long critical sections. In addition, as discussed in

Section 2.5.10, PWLP is preferable with low degree of either the parallelism or resource contention while the schedulability of MSRP systems can be less affected with the increase of the number of processors or the frequency of resource access. Section 3.4.1 provides experimental evidence that verifies the above statements.

With these three protocols working together simultaneously, the framework should be able to provide a strong schedulability to a wide range of application semantics and resource characteristics, assuming an appropriate candidate locking protocol is applied to each resource. In Chapter 5, the criteria and techniques of deciding the appropriate resource sharing protocol for each shared resource are presented. Note that although only MSRP, PWLP and MrsP are adopted in the resource control framework in this thesis, other protocols can be easily integrated into the resource control framework as long as a RTA-based schedulability test is supported and is integrated into the schedulability analysis framework. Section 3.3 demonstrates how a schedulability test of a protocol can be easily integrated into the schedulability analysis framework of the resource control framework proposed in this thesis.

3.2 New Schedulability Tests for MSRP, PWLP and MrsP

With the candidate resource sharing protocols determined, each of the protocols must be supported by a schedulability test so that a schedulability analysis framework can be developed for systems with potentially all the candidate locking protocols in use. In addition, as stated in Section 2.7, the schedulability test of each candidate locking protocols must address the issues identified in [106] to achieve less pessimistic as well as more accurate schedulability results than that of its original test (if it exists).

According to the discussion given in Section 2.6.2.2, it is clear that the ILP-based analysis contains the most advanced analysing techniques among all the existing schedulability tests and directly supports the analysis of MSRP and PWLP systems (with the corresponding constraints applied). However, the use of the ILP technique is considerably expensive from viewpoint of both implementing the analysis and computing the response times in practice. In addition, the ILP-based analysis does not take any run-time costs into account, which can undermine the accuracy of the analysis, where tasks that have passed the test can miss their deadlines in practice due to the unexpected run-time costs incurred from either the underlying operating system or the locking protocol in use. Such a case is most likely to happen with MrsP adopted, where migrations are required but the cost of migrations is not analysed and safely bounded.

Combing the discussion in Section 2.6.2 and above, new schedulability tests for each of the candidate locking protocols are developed in this section, which should satisfy the following requirements:

R-1 The new schedulability tests must not rely on the assumption that a task that requests a resource can incur blocking from each remote processor that requires the resource during each access to that resource.

R-2 The new schedulability tests must not rely on the analysing technique that inflates the execution time of tasks with their blocking time.

R-3 The new schedulability tests should take the potential blocking due to the back-to-back hit phenomenon into account.

R-4 The new schedulability tests should take the run-time overheads from both the underlying operating systems and the locking protocol into account, especially the cost of potential migrations in MrsP systems. *Note, the term "run-time overheads" used throughout this thesis does not include the costs due to the underlying RTOS (e.g., the costs of reloading cache and accessing memory).* This thesis focuses on analysing the costs for adopting real-time resource sharing protocols.

R-5 The new analysis for each candidate protocol should satisfy the above requirement without the use of any analysing techniques that are expensive and time-consuming (e.g., the ILP technique) .

In Section 3.4, evaluations are presented to verify the necessity of the requirements given above, which should be satisfied by the new schedulability tests. In addition, as described in Section 2.5.10, all the candidate protocols impose certain limitations when supporting nested resources, where MSRP does not support nested resource accesses between global resources at all, PWLP recommends the use of group locks but with no further details given, and MrsP lacks a complete approach to bound all potential blocking for accessing nested resources. *For the ease of presentation and in the interest of* *brevity, we assume that a task can only access one resource at a time. That is, we will focus on non-nested resource accesses in the following chapters.* However, we acknowledge that nested resource access is highly relevant. In Appendix A, the resource control framework is extended to support the use of nested resources via group locks and the analysis for nested resource access is developed.

3.2.1 Analysing MSRP Systems

We start with the most straightforward protocol among all the candidate locking protocols (i.e., MSRP) and set the format of the schedulability tests that the following protocols should comply with to facilitate the integration of all the schedulability tests. The analysis keeps the philosophy of the original RTA-based equations of MSRP (see Section 2.5.2), but with new techniques to precisely bound the blocking terms with the implementation overheads of MSRP and the run-time costs from the underlying operating system accounted for. In contrast to the ILP-based analysis, which bounds all the blocking terms by the blocking variable B, we aim to precisely bound the three blocking effects identified in Section 2.5.2 separately (i.e., the *direct spin delay*, the *indirect spin delay* and the *arrival blocking*), and then fit them into the RTA equations without inflating the task's execution time while avoiding the use of the potentially expensive techniques (e.g., the ILP technique) to meet the requirements listed above.

Equation (3.1) gives the response time of task τ_i, where the blocking effects are reflected by three parameter: E_i is the total resource accessing time of τ_i with direct spin delay accounted for; $I_{i,h}$ indicates the indirect spin delay incurred by τ_i from a local high priority task τ_h; and the arrival blocking is accounted for in B_i. Note that in our new analysis, C_i is the pure computation time of τ_i without accessing any resource and function $\lceil \frac{R_i}{T_h} \rceil \cdot C_h$ gives the pure computation interference from a local high priority task τ_h without accessing resources. In addition, this analysis separates the direct spin delay E_i and the indirect spin delay $I_{i,h}$ from the task's execution time, where function $\lceil \frac{R_i}{T_h} \rceil \cdot C_h + I_{i,h}$ is adopted when calculating the total interference (i.e., with blocking accounted for) form a local high priority task τ_h rather than the function $\lceil \frac{R_i}{T_h} \rceil \cdot \widehat{C_h}$ adopted in the original MSRP test.

$$R_i = C_i + E_i + B_i + \sum_{\tau_h \in \mathbf{hpl}(i)} (\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h}) \tag{3.1}$$

Direct and Indirect Spin Delay

We start by bounding the total resource-accessing time with direct spin delay E and the indirect spin delay I incurred by τ_i. The following two equations share a similar format but take different inputs, as shown in Equations (3.2) and (3.3), where $e_x^k(l,\mu)$ gives the accessing time (with direct spin delay) to resource r^k that task τ_x can incur within the duration l and a release jitter μ. By given different duration and jitter length, the function gives a different bounding as τ_x can be released a different number of times (so that a different number of requests) within the given period. Therefore, with the blocking variables E and I separated from the task's execution time and calculated independently via different inputs, our analysis does not rely on inflating execution time so that the requirement R-2 given in Section 3.2 is satisfied.

$$E_i = \sum_{r^k \in F(\tau_i)} e_i^k(R_i, 0) \tag{3.2}$$

$$I_{i,h} = \sum_{r^k \in F(\tau_h)} e_h^k(R_i, R_h) \tag{3.3}$$

Equation (3.2) gives the total resource accessing time of τ_i. For τ_i itself, $l = R_i$ and $\mu = 0$ so that we will only account for resource requests in one release. As for the indirect spin delay (see Equation (3.3)), $l = R_i$ and $\mu = R_h$ so that the back-to-back hit can be accounted for when computing the total number of requests issued from a high priority task τ_h to r^k in the context of τ_i (i.e., during τ_i's release). To achieve a fine-grained schedulability test, we analyse the resource accessing time of a task in each individual access so that $e_x^k(l,\mu)$ is further expanded as:

$$e_x^k(l,\mu) = \sum_{n=1}^{N_x^k(l,\mu)} e_x^k(l)(n) \tag{3.4}$$

where $N_x^k(l,\mu) = \left\lceil \frac{l+\mu}{T_x} \right\rceil \cdot N_x^k$ gives the number of requests τ_x can issue to resource r^k with the back-to-back hit included and $e_x^k(l)(n)$ gives the time of τ_x's n-th access to r^k within a duration l. With the back-back hit phenomenon accounted for by introducing the duration l and the jitter μ, the requirement R-3 can be satisfied.

To reflect the worst-case scenario, a higher priority task should incur blocking before any local low priority tasks do, as the spin delay incurred by high priority tasks is propagated to all local lower priority tasks as interference.

Thus, when computing the direct spin delay that τ_x can incur for accessing r^k, the requests from a remote processor should delay τ_x's local higher priority tasks prior to τ_x (if they request r^k). This discussion leads to the following observations, where $Nh_x^k(l) = \sum_{\tau_h \in \mathbf{hpl}(x)} N_h^k(l, R_h)$ gives the number of requests issued by local high priority tasks, $Np_m^k(l) = \sum_{\tau_j \in \tau(P_m)} N_j^k(l, R_j)$ gives the number of requests issued from a remote processor m, $\tau(P_m)$ gives a set of tasks allocated on processor m and $(f(x))_a$ denotes $max\{f(x), a\}$ for the ease of presentation.

Theorem 1. *The maximum number of requests on a remote processor m that* ***may*** *block τ_x directly for accessing r^k within the duration l is bounded by $NS_{x,m}^k(l) = (Np_m^k(l) - Nh_x^k(l))_0$.*

Proof. Let N_S^{may} denote the number of requests from a remote processor that may block τ_x. If $N_S^{may} > NS_{x,m}^k(l)$, then there exist remote requests that can block both τ_x and a higher priority task on τ_x's processor that requests r^k directly, which is not possible as one request can only cause one blocking. Otherwise (where $N_S^{may} < NS_{x,m}^k(l)$), certain requests that may block τ_x are not accounted for. ■

Theorem 2. *The number of direct spin delays that τ_x* ***can*** *incur for accessing r^k from a remote processor m within the duration l and jitter μ is $min\{NS_{x,m}^k(l), N_x^k(l, \mu)\}$.*

Proof. Let N_S^{can} denote the number of spin delay that τ_x can incur. If $N_S^{can} = NS_{x,m}^k(l) \wedge NS_{x,m}^k(l) > N_x^k(l, \mu)$, there exists a remote request that can block τ_x multiple times. In contrast, where $N_S^{can} = N_x^k(l, \mu) \wedge N_x^k(l, \mu) > NS_{x,m}^k(l)$, there exist more than one requests on a remote processor that can block the same access of τ_x. Under MSRP, neither case is possible. ■

To examine the blocking time in each resource access, we assume that *the first access to a resource incurs as much spin delay as possible.* This assumption will not introduce any pessimism as the total spin delay a task can incur remains identical. Accordingly, equation $e_x^k(l)(n)$ can be constructed to compute the time for each access (see Equation (3.5)), where n is bounded to $[1, N_x^k(l, \mu)]$ by Equation(3.4) and one extra c^k is accounted for the access by τ_x itself. Let $(f(x))_a^b$ denote $min\{max\{f(x), a\}, b\}$, where a and b are positive integers with $a \leq b$.

$$e_x^k(l)(n) = \sum_{P_m \neq P(\tau_x)} (NS_{x,m}^k(l) - n + 1)_0^1 \cdot c^k + c^k \tag{3.5}$$

In τ_x's n-th access, requests from a remote processor m can block τ_x only if there still exists unaccounted requests on m i.e., $(NS^k_{x,m}(l) - n + 1)_0 \geq 1$. Upon one access, there can be at most one request on a remote processor that can cause the spin delay and hence $(NS^k_{x,m}(l) - n + 1)^1_0$.

With Equations (3.4) and (3.5), the direct spin delay in E and the indirect spin delay I can be computed. As proved, our approach guarantees that each critical section will only be accounted for once and does not rely on inflating task's computation time. In addition, with the back-to-back hit considered, the new equations can provide less pessimism and more accurate spin delay bounding than that of the original MSRP analysis. Therefore, the issues discussed in Section 2.6.2.2 are addressed (i.e., has met the requirements R-1 to R-3).

Compared to the ILP-based analysis (which only gives the total amount of spin delay for each task), we provide a fine-grained analysing technique that is able to give the spin delay incurred for each individual resource access. Further, in contrast to the ILP-based analysis, the new equations keep the philosophy of the original MSRP analysis and can be much less expensive when either implementing or practising this analysis without the need for an ILP solver (i.e., requirement R-5). The above statements are confirmed later on by experiments given in Section 3.4.

Arrival Blocking

The arrival blocking is accounted for by parameter B_i, as given in Equation (3.6), where $\hat{e_i}$ gives the maximum arrival blocking that τ_i can incur and is calculated by Equation (3.7).

$$B_i = max\{\hat{e_i}, \hat{b}\} \tag{3.6}$$

$$\hat{e_i} = max\{|\alpha^k_i| \cdot c^k | r^k \in F^A(\tau_i)\} \tag{3.7}$$

Equation (3.7) firstly identifies resources that can cause τ_i to incur arrival blocking (i.e., $F^A(\tau_i)$) and then gives the maximum blocking time among the resources in $F^A(\tau_i)$. Under MSRP, a resource r^k can cause arrival blocking to τ_i if (1) r^k is a global resource and will be accessed by a local lower priority task (i.e., τ_{ll}) or (2) r^k is a local resource that is required by τ_{ll} with a ceiling priority equal to or higher than τ_i's priority, as given by Equation (3.8). Note that for a local resource r^k on P_m, $Pri(r^k, P_m) = Pri(r^k)$.

$$F^A(\tau_i) \triangleq \{r^k | N^k_{ll} > 0 \land (r^k \text{ is global} \lor Pri(r^k, P(\tau_i)) \geq Pri(\tau_i))\} \tag{3.8}$$

The arrival blocking can be computed without the knowledge of the exact task that causes such a blocking. For any resource (either local or global) in $F^A(\tau_i)$, it can cause a local blocking of c^k. For a global resource r^k, there can be at most one request from each remote processor that can cause τ_i to incur arrival blocking transitively. Therefore, by identifying the number of such processors, the arrival blocking can be computed. Let $P(\tau_i)$ denotes τ_i's processor and α_i^k be the set of processors with requests to r^k that cause arrival blocking to τ_i (including $P(\tau_i)$), where

$$\alpha_i^k \triangleq \{P_m | NS_{i,m}^k(R_i) - N_i^k > 0 \wedge P_m \neq P(\tau_i)\} \cup P(\tau_i) \tag{3.9}$$

Similar to Equation (3.5), a request to r^k from a remote processor that can block a lower priority task on τ_i's processor only if the remote request does not cause any delay yet (including τ_i) i.e., $NS_{x,m}^k(l) - N_i^k > 0$. Otherwise (where $NS_{x,m}^k(l) - N_i^k \leq 0$), this remote request (if exists) will be calculated more than once because it is already accounted for in the spin delay of τ_i. In addition, $P(\tau_i)$ should also be accounted for in α_i^k to include the local blocking issued by τ_{ll} to r^k. For a local resource r^k in $F^A(\tau_i)$, $\alpha_i^k = \{P(\tau_i)\}$. With α_i^k computed for each resource in $F^A(\tau_i)$, the arrival blocking of τ_i is obtained, as shown in Equation (3.7).

The above equations can provide precise bounding of the blocking variables E, I and B of a given task τ_i so the the response time of τ_i can be computed. Note that in our analysis, we account for the spin delay before computing arrival blocking while in practice a task will incur arrival blocking firstly. However, our approach does not break any statements above as the total number of requests that can block τ_i is fixed and our approach provides an easier way to account for all the blocking effects. With the new analysing techniques for bounding the blocking variables, our analysis addresses the issues reported by the ILP-based analysis[1].

This analysis is independent of the priority assignment scheme and is not fixed to any specific hardware architecture. Similar to ILP-based analysis [106], the blocking time of a given task in our analysis depends on the response time of potentially all tasks in the system. With an initial response time, say C_i, the analysis computes all the blocking variables and updates the response times of all tasks in the system iteratively and alternately until

[1]The experiment `IdenticalTest` in the testing program `https://github.com/RTSYork/SchedulabilityTestEvaluation` shows that our new MSRP test achieves the identical response time for each task as the ILP-based analysis does with MSRP constraints adopted.

a fixed-point is reached.(i.e., the response time and the blocking variables remain the same after further calculations). As proved, the new analysis satisfies the requirements R-1 to R-3 and R-5.

Incorporating the Run-time Overheads

The above presents the theoretical response time analysis of MSRP systems. In practice, the response time can be larger than the theoretical value due to the overheads incurred from both the protocol implementation and the underlying operating system. To guarantee the accuracy of the schedulability results, such costs should be taken into account in the analysis.

Admittedly, the actual run-time cost that a task can incur largely depends on the real hardware platforms and operating systems. Yet by treating each of the costs as a constant upper bound, the maximum run-time overheads a task incur during run-time can safely bounded. This section presents the techniques of incorporating the run-time overheads of MSRP systems into the newly-proposed schedulability test.

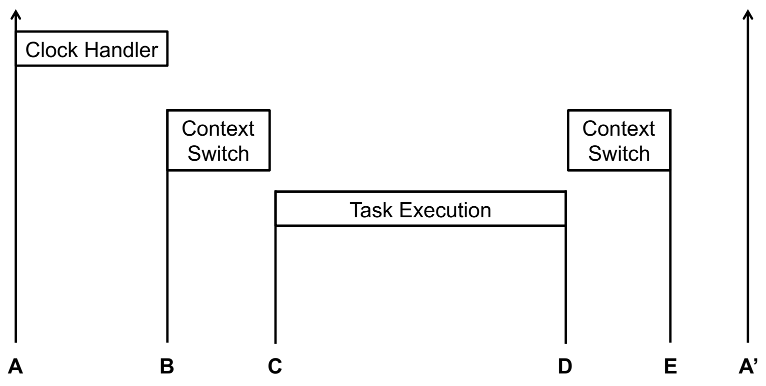


Figure 3.1: Events from the Operating System During a Task's Release [25].

In MSRP systems, the run-time overheads that incurred by tasks mainly include the cost of obtaining and releasing a lock, and the context switches due to task releases and preemptions. Figure 3.1 cited from [25] illustrates the major events occurred in the underlying operating system during the lifetime of a task's release (say τ_i).

When τ_i's turn to release arrives, the corresponding clock interrupt will be fired and the interrupt handler will move τ_i from the sleeping queue to the ready queue, where it waits to be scheduled (i.e., event A). Assuming τ_i has the highest priority among all the ready tasks, the scheduler will be invoked to schedule τ_i to execute (i.e., event B). If there is an executing task, this task will be cleaned up and switched away. τ_i starts its execution at event C and finishes at event D, where it could be preempted several times by newly-released higher priority tasks. When τ_i is finished, it will be cleaned up and be switched away by the scheduler (event E). Then the system schedules the next ready task to execute (if any) and keeps waiting for the next clock interrupt (i.e., event A'). Once τ_i is preempted during the point C to D, it incurs the overheads from all events given in Figure 3.1.

According to the description above, to account for the cost due to the potential context switches τ_i can incur during each release, Equation (3.1) is extended to Equation (3.10), as given below.

$$R_i = CX_1 + C_i + E_i + B_i + \sum_{\tau_h \in \mathbf{hpl}(i)} \Big(\left\lceil \frac{R_i}{T_h} \right\rceil \cdot (CX_2 + C_h) + I_{i,h} \Big) \tag{3.10}$$

where CX_1 denotes the cost of events A and B (i.e., releases the task and schedules it to execute), which will occur before the real execution of τ_i. If τ_i is preempted while executing, it will incur extra overheads caused by the event A, B and E, which is denoted as CX_2. With these two variables determined, the run-time overheads incurred by τ_i due to major scheduling events from the underlying system can be bounded.

The cost for obtaining and releasing a MSRP lock mainly includes the overheads for raising and restoring the priorities of the resource accessing tasks, and manipulating the FIFO queues, which are performed in the function `lock()` and `unlock()`. Such costs are denoted as C^{lock}_{MSRP} and C^{unlock}_{MSRP} respectively, where they can be easily integrated into the cost for accessing a resource via a new notation C^k, as given below.

$$C^k = C^{lock}_{MSRP} + c^k + C^{unlock}_{MSRP} \tag{3.11}$$

Accordingly, Equations (3.5) and (3.7) are revised as the following to incorporate with the overheads of the locking protocol.

$$e^k_x(l)(n) = \sum_{P_m \neq P(\tau_x)} (NS^k_{x,m}(l) - n + 1)^1_0 \cdot C^k + C^k \tag{3.12}$$

$$\hat{e_i} = max\{|\alpha^k_i| \cdot C^k | r^k \in F^A(\tau_i)\} \tag{3.13}$$

With the above equations, the run-time overheads incurred by tasks in MSRP systems can be bounded so that requirement R-5 can be satisfied. Note that the above equations only provide an overall approach for incorporating the run-time overheads.

To practice this analysis, the underlying hardware and a real-world operating system must be provided and the cost for each event in the worst case should be measured. The exact measuring approach of CX_1 and CX_2 largely depends on the scheduling structure of the given operating system while the

Table 3.1: Notations Introduced in the New MSRP Analysis

τ_j	A remote task.
τ_h	A high priority task.
$\tau(P_m)$	Tasks allocated to P_m.
$P(\tau_x)$	Partition of τ_x.
C^k	The cost for executing resource r^k with implementation overheads.
CX_1, CX_2	The cost of context switches of the operating system.
E_x	Total resource-accessing time of τ_x to all resources.
$I_{x,h}$	Indirect spin delay incurred by τ_x from a local high priority task.
$e_x^k(l, \mu)$	Total resource-accessing time of τ_x accessing r^k during the time l with a jitter μ.
$e_x^k(l)(n)$	Resource-accessing time of τ_x's n-th access to r^k during the time l.
α_i^k	A set of partitions with requests that can cause τ_i to incur arrival blocking.
$N_x^k(l, \mu)$	Number of requests of τ_x to r^k during the time l with a jitter μ.
$Nh_x^k(l)$	Number of requests of τ_x's higher priority tasks to r^k during the time l.
$Np_m^k(l)$	Number of requests issued from tasks on P_m to r^k during the time l.
$NS_{x,m}^k(l)$	The maximum number of requests on a remote processor m that could block τ_x directly for accessing r^k within the duration l.
$(f(x))_0$	Function $f(x) >= 0$.
$(f(x))_a^b$	Function $min\{max\{f(x), a\}, b\}$.

costs of `lock()` and `unlock()` depends on the real implementation of the protocol. In Appendix B, the above run-time cost variables are measured under the Litmus^{RT} Real-Time Operating System [19, 30] based on the protocol implementations given in Appendixes C and D.

Summary

This concludes the new run-time overheads-aware schedulability test for MSRP, where the new notations introduced in this analysis are summarised in Table 3.1. As discussed before, this analysis satisfies all the requirements listed in Section 3.2, and is able to provide less pessimistic as well as more accurate schedulability results than that of its original analysis without the need for any expensive and time-consuming analysing techniques.

In addition, this analysis provides the basic techniques for analysing systems with FIFO spin locks, which can be directly applied when analysing certain blocking terms in PWLP and MrsP systems. Finally, a complete template for bounding the blocking variables under the FIFO spin-based protocols is presented. To guarantee the analysing correctness and to facilitate the integration of the analysis framework, the schedulability tests developed for the following candidate locking protocols should comply with the theorems and the schedulability test format proposed in this section.

3.2.2 Analysing PWLP Systems

For PWLP systems, the above MSRP analysis can be adopted with certain modifications to reflect the preemptive spinning approach and the cancellation mechanism. With PWLP applied, the resource-waiting queue is prolonged as a task that is spinning for a resource can be preempted so that it is removed from the FIFO queue. Once the task is resumed, it re-requests the resource and re-joins into the end of the FIFO queue.

In addition, by spinning with base priorities, tasks in PWLP systems can incur arrival blocking for one critical section only, which is identical with the uniprocessor case. This section firstly bounds the additional resource-waiting time in PWLP and then computes the arrival blocking via extending the new MSRP analysis. In addition, the run-time overheads incurred by tasks in PWLP systems are discussed.

Costs of the Cancellation Mechanism

As proved by Theorems 1 and 2 in Section 3.2.1, if a task incurs no preemptions while waiting for a PWLP resource, its worst-case resource-accessing time is identical with that of MSRP due to the FIFO spinning approach, where Equations (3.1) to (3.5) can be directly applied for bounding the spin delay. However, by spinning with base priorities, additional blocking time must be taken into account to cope with the cancellation mechanism, where Equation (3.1) is extended as follows with S_i being introduced to represent such additional blocking.

$$R_i = C_i + E_i + B_i + \sum_{\tau_h \in \mathbf{hpl}(i)} (\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h}) + S_i \tag{3.14}$$

As the cancellation mechanism is triggered by preemptions, a helper function NoP_i is firstly introduced to provide a safe bounding of the number of preemptions that can occur during the release of τ_i, denoted as $NoP_i = \sum_{\tau_h \in \mathbf{hpl}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil$, where each release of τ_h in $\mathbf{hpl}(i)$ can cause a preemption to either τ_i or a τ_i's local higher priority task in the worst-case. Thus, the maximum number of the re-requests to shared resources (due to preemptions) of τ_i and its local higher priority tasks can be bounded, where each preemption in NoP_i can cause a retry in the worst case. The reason to consider the preemptions (so that the potential re-requests) to τ_i's local higher priority tasks when computing τ_i's S_i is because this blocking can propagate to τ_i, where τ_i has to wait for such a high priority task (which is preempted by a higher priority task, and hence, triggers the cancellation mechanism) to finish before it can be resumed.

Once a preemption occurrs during the release of τ_i while the preempted task (either τ_i or a task in $\mathbf{hpl}(i)$) is waiting for a resource (say r^k), the amount of blocking time for re-requesting r^k depends not only on the number of remote requests to r^k that is not being accounted for in the analysis (i.e., not being considered as the direct spin delay or indirect spin delay of τ_i yet) but also the critical section length of r^k. If there exist no unaccounted remote requests to r^k, this preemption will not cause any additional blocking to τ_i.

To reflect the worst-case scenario, this analysis searches for the resources that can cause the most amount of blocking to τ_i by each preemption. Let L_i^k denotes a list of extra blocking incurred by τ_i for tasks (τ_i itself or its higher priority tasks) re-accessing r^k due to each preemption that can occur during the release of τ_i, where

$$L_i^k = \{ \sum_{P_m \neq P(\tau_i)} (NS_{i,m}^k(R_i) - N_i^k - n + 1)_0^1 \cdot c^k | 1 \leq n \leq NoP_i\} \tag{3.15}$$

For the n-th preemption during τ_i's release that preempts a task waiting for r^k, the amount of blocking incurred by τ_i due to re-requesting r^k from a remote processor P_m is bounded by $(NS_{i,m}^k(R_i) - N_i^k - n+1)_0^1 \cdot c^k$, where $NS_{i,m}^k(R_i) - N_i^k$ represents the remote requests on P_m that have not being accounted into either the direct or the indirect blocking of τ_i, as proved in Section 3.2.1. Note that such a value can be 0 if there exist no more unaccounted remote requests on P_m upon a given preemption.

The above presents the approach for computing the additional blocking caused by all preemptions due to accessing one resource. As briefly described before, τ_i can incur such additional blocking not only when accessing a global resource itself, but also by a local high priority task (say τ_h) that preempts τ_i and requests a global resource, which in turn, is preempted by another higher priority task. In this case, the additional blocking incurred by τ_h will transitively block τ_i as well. Such blocking can also be accounted for by the above equation as preemptions to τ_i's local higher priority tasks can also be reviewed as preemptions to τ_i. However, resources that are neither requested by τ_i nor τ_i's local higher priority tasks will not cause any additional blocking to τ_i. Accordingly, the resources that can cause τ_i to incur the additional blocking under PWLP can be identified, as given below, where $F^S(\tau_i)$ denotes such resources and τ_{lh} represents the local tasks that have a higher priority than τ_i.

$$F^S(\tau_i) \triangleq \left\{ r^k | (N_i^k > 0 \vee N_{lh}^k > 0) \wedge r^k \text{ is global} \right\} \tag{3.16}$$

To capture the maximum additional blocking time that τ_i can incur, the blocking times caused by each resource in $F^S(\tau_i)$ are computed with the assumption that all preemptions have occurred while τ_i or τ_{lh} is waiting for the resource (i.e., L_i^k). Then, the blocking values from each L_i^k list are merged into a single list and are sorted by the non-increasing order to facilitate bounding such retry cost under this protocols, as shown in Equation (3.17), where LS_i gives the list of blocking due to re-requesting each of the resource in $F^S(\tau_i)$ upon all preemptions and $\{\}^{dList}$ denotes a list with its elements ordered non-increasingly.

$$LS_i = \{L_i^k |\ r^k \in F^S(\tau_i)\}^{dList} \tag{3.17}$$

Accordingly, the total amount of the additional blocking that τ_i can incur under PWLP in the worst-case can be safely bounded via summing up the first NoP_i values in the LS_i list, as given in Equation (3.18), where $LS_i(n)$ gives the n-th element in the list LS_i.

$$S_i = \sum_{n=1}^{NoP_i} LS_i(n) \tag{3.18}$$

Via extending the response time equation with a new notation S_i, the above equations provide a complete approach for bounding the additional blocking incurred by tasks due to the cancellation mechanism with PWLP adopted.

Arrival Blocking

Another difference between MSRP and PWLP is the amount of arrival blocking that a task can incur. Both protocols share the same set of resources that can cause the arrival blocking (see Equation (3.8)) as they require non-preemptive execution while holding a resource. However, as tasks are spinning with base priorities under PWLP, they will not incur any remote blocking when being blocked upon their arrival, where they can directly preempt an executing low priority task even if they are waiting for a global resource. Hence, the arrival blocking can be easily bounded by the Equation (3.19), where a task can be blocked by the longest critical section among resources in $F^A(\tau_i)$.

$$\hat{e_i} = max\{c^k | r^k \in F^A(\tau_i)\} \tag{3.19}$$

The above presents an approach for calculating the theoretical response time (i.e., without the run-time costs) of tasks in a PWLP system, including the blocking due to the cancellation mechanism. Similar with the new MSRP analysis, this test gives the identical response time for each task as that of the ILP-based analysis with PWLP constraints adopted (i.e., the testing program `IdenticalTest` mentioned in Section 3.2.1). Yet, the new tests avoids the use of the expensive ILP technique, which ease the implementation of the analysis and require less computation time to deliver the schedulability results. Such features are curial for the usability of the proposed FMRS framework (see Chapter 5 for explanations). In Section 3.4.4, the time consumption of the newly-proposed analysis and the ILP-based tests will be investigated.

Run-time Overheads

The approach to bound the overheads of context switches and locking is the same with that of MSRP, where CX_1 and CX_2 are introduced to Equation (3.14) while C^k is adopted instead of c^k in Equations (3.5), (3.15) and (3.19). However, C^k under PWLP should be revised as Equation (3.20) to cope with the overheads from PWLP's `lock()` and `unlock()` functions.

$$C^k = C^{lock}_{PWLP} + c^k + C^{unlock}_{PWLP} \tag{3.20}$$

In addition, with PWLP adopted, preemptions can trigger the cancellation mechanism so that the preempted task is removed from the resource-waiting queue. Once the preempted task is resumed, it should re-request the resource (i.e., re-joins into the FIFO queue) so that additional run-time overheads are imposed. The cost for these two operations can be denoted by one notation C_{retry} and be bounded together via extending Equation (3.17), where a preemption can cause the cancellation of the request and the subsequent retry.

$$LS_i = \{C_{retry} + L^k_i |\ r^k \in F^S(\tau_i)\}^{dList} \tag{3.21}$$

Note that although this analysis bounds the cost of cancelling a request and the retry together as one notation, the cancellation operation is likely to happen inside the scheduler before the preempted task is switched away, and hence, could lead to a slightly higher cost of the context switch procedure in theory (i.e., CX_1 and CX_2) even if this protocol is not in use. However, the cancellation mechanism can be effectively modelled into an `if` statement with a single variable to control whether a cancellation is required. Therefore, such an intrusion can be trivial to the total cost of a context switch, and hence, can be ignored without jeopardising the schedulability results. This concludes the new schedulability test of PWLP. The notations introduced in this new schedulability analysis are summarised in Table 3.2.

3.2.3 Analysing MrsP Systems

Unlike PWLP, tasks in a MrsP system share the identical FIFO queue with that of MSRP due to the helping mechanism, where Equations (3.1) to (3.7) and (3.9) can be directly applied to bound the theoretical response time of MrsP tasks. However, as tasks in MrsP are accessing resources with the ceiling priority, Equation (3.8) requires modifications to reflect the set of resources that can cause MrsP tasks to incur the arrival blocking, where τ_i can be blocked upon its arrival by r^k only if r^k is required by local lower priority tasks with a ceiling equal to or higher than $Pri(\tau_i)$.

$$F^A(\tau_i) \triangleq \{r^k | N^k_{ll} > 0 \wedge Pri(r^k, P(\tau_i)) \geq Pri(\tau_i)\} \tag{3.22}$$

With the equations mentioned above, a complete response time of MrsP can be formed. In addition, the approach for incorporating the overheads of context switches, and locking and unlocking resources is also similar to that of the MSRP analysis, where Equations (3.10) to (3.13) can be directly applied with the notation C^k revised to reflect to overhead of MrsP's `lock()` and `unlock()` functions, as given below.

$$C^k = C^{lock}_{MrsP} + c^k + C^{unlock}_{MrsP} \tag{3.23}$$

Migrations in MrsP

The current definition of the MrsP helping mechanism carries a certain degree of pessimism under the situation where the resource holder is preempted and

Table 3.2: Notations Introduced in the New PWLP Analysis

C_{retry}	The implementation overheads for cancelling a resource request and re-accessing a resource.
S_i	The additional blocking of τ_i due to the cancellation mechanism of PWLP.
NoP_i	The number of preemptions τ_i can incur during one release.
L^k_i	A list of additional blocking times incurred by τ_i for re-accessing r^k due to each preemption.
LS_i	A list of additional blocking times (ordered decreasingly) that τ_i can incur for re-accessing shared resources in $F^S(\tau_i)$.
$F^S(\tau_i)$	The set of resources that can cause τ_i to incur the additional blocking under PWLP.
$L(n)$	The n-th element in the given list L
$\{\}^{dList}$	A list with the elements ordered non-increasingly by their values.

there exist a large number of potential helpers each of which resides in processors where there is one or more high priority tasks with short periods. This can result in the resource-holding task suffering frequent migrations. Under such a situation, the task can spend more time migrating than it does executing with the resource so that the resource-accessing time can be significantly prolonged with the efficiency of the protocol significantly undermined.

To avoid frequent migrations, we introduce a short non-preemptive section into the MrsP helping mechanism so that *upon each migration with a resource, the holder is allowed to execute non-preemptively (NP) for a short time before it inherits the corresponding resource ceiling priority.* The NP-section can provide guaranteed progress to resource holders and can reduce the number of migrations effectively, especially when high priority tasks are released frequently. The only side effect of this approach is that any newly released high priority tasks have to cope with the cost of one NP-section before it can preempt the holder and execute. However, the length of the NP-section (i.e., C_{NP}) can be configured so that the high priority tasks are still able to meet their deadlines. As a default it can be the maximum time of the NP-sections in the hosting operating system (i.e., $\hat{b}$). Our analysis presented below bounds the cost of the migration with this approach. In Section 3.4.3, evidence is given to demonstrate improved efficiency of MrsP with the NP-section adopted.

Migration Cost Analysis

As described above, migrations are required in this protocol due to the helping mechanism, which usually require updating data structures (e.g., run queues) in the underlying operating system and reloading caches. Such operations can impose non-negligible run-time overheads to MrsP systems and should not be ignored by MrsP analysis. In addition, the implementation overheads of the helping mechanism should also be accounted for to achieve a more accurate and complete schedulability test for MrsP. In this section, a migration cost analysis for MrsP is developed via treating the migration cost as a constant upper bound (e.g., C_{mig} in this thesis) and bounding the maximum number of migrations a task can perform during each release due to accessing shared resources under MrsP.

To capture the worst-case scenario, we assume that *a preempted resource holder can migrate to any valid processor* (i.e., a processor that has a task spinning for the resource or the holder's original processor). In addition, as shown in the above analysis, for any resource-requesting task τ_x, it can incur a different amount of spin delay upon each access to a resource so that its migration targets can also be different during each resource access. Thus, the migration cost should be computed based on each individual access to each resource. We firstly identify the set of migration targets for a given task τ_x.

Theorem 3. *In τ_x's n-th access to r^k within a duration l, the set of migration targets for τ_x is $mt_x^k(l)(n) \triangleq \{P_m | P_m \neq P(\tau_x) \wedge NS_{x,m}^k(l) - n + 1 > 0\} \cup P(\tau_x)$.*

Proof. A remote processor m is a valid migration target for τ_x's n-th access to r^k only if there exists a request to r^k from processor m that is not already accounted for during l (i.e., $NS_{x,m}^k(l) - n + 1 > 0$ from Equation (3.5)). In addition, τ_x's original processor should be included as τ_x may migrate back to $P(\tau_x)$ when it is preempted on a remote processor. ■

In addition, when τ_x incurs arrival blocking by a low priority task, the blocking task may also incur migration cost, which in turn delays τ_x. The migration targets of the low priority task can be identified by the set α_x^k (the set of remote processors with requests that can cause τ_x to incur arrival blocking) in Equation (3.9).

As tasks inherit the resource ceiling when accessing a MrsP resource, the potential preemptors on each migration target can be identified. With a given set of migration targets (denoted by mt) and a resource r^k, the migration targets with preemptors $mtp(mt, r^k)$ is:

$$mtp(mt, r^k) \triangleq \{P_m | P_m \in mt \wedge hpt(r^k, P_m) \neq \emptyset\} \quad (3.24)$$

where $hpt(r^k, P_m)$ gives a set of tasks on processor m that have a priority higher than the resource ceiling of r^k. Note that $mtp(mt, r^k)$ is a subset of the given migration targets mt and can be empty.

As presented above, migration targets are identified based on whether there will be a request from the remote processor. Thus, on each migration target, there exists one request issued to the resource and they share the same set of migration targets. To bound the migration cost that a task τ_x can incur when accessing a resource, we examine the migration cost of each request issued from the migration targets. Let $Nmig$ be the number of potential migrations. We summarise the following observations where a limited number of migrations can be triggered when a request is issued from processor P_m to resource r^k with a given set of migration targets mt:

Lemma 1. *$Nmig = 0$ if $P_m \notin mtp(mt, r^k)$.*

Proof. The request issued from processor P_m incurs no migrations if there exists no preemptors on that processor. ■

Lemma 2. *$Nmig = 0$ if $\{P_m\} = mt$.*

Proof. No matter how many times the request from P_m can be preempted on its processor, there will be no migrations if there exists no other migration targets. ■

Lemma 3. *$Nmig = 2$ if $\{P_m\} = mtp(mt, r^k) \wedge |mt| > 1$.*

Proof. In the case where the request can only be preempted on its original processor P_m, the requesting task can migrate to other migration targets without further preemptions. Once the task releases the resource, it migrates back to P_m. ■

In a more general case where there exist more than one migration targets with potential preemptors, the number of migrations have to be bounded by the release of all potential preemptors. Unfortunately, we are not able to track the state of the current processor of the resource holder constantly as no assumption can be made about the migration destination in the worst case. Thus, we have to assume that each release of the high priority task can cause a preemption with a subsequent migration. Because of this, *our analysis provides a safe upper bound of the migration cost rather than a precise worst-case bounding.* However, by applying the NP-section and by identifying the exact set of migration targets, the pessimism of the analysis can be effectively reduced, as shown by experiments in Section 3.4.3.

In the case where the resource-requesting task's processor P_m satisfies $P_m \in mtp(mt, r^k) \wedge |mt| > 1$, the migration cost of that single request is bounded by the releases of high priority tasks on each migration target, denoted by $Mhp(mt, r^k)$, where C_{mig} represents the overheads of one migration.

$$Mhp(mt, r^k) = C_{mig} \cdot \Big(\sum_{P_m \in mtp(mt, r^k)} \big(\sum_{\tau_h \in hpt(r^k, P_m)} \left\lceil \frac{c^k + Mhp(mt, r^k)}{T_h} \right\rceil \big) + 1 \Big) \tag{3.25}$$

The equation accounts for the total number of releases of all the potential preemptors on each migration target within the duration of one resource computation time with migration cost considered $c^k + Mhp(mt, r^k)$. Through iteration, the equation can give a fixed migration cost that the requesting task can incur based on the given set of migration targets. To cope with the situation where the next holder needs to wait for the current holder to migrate away before it can acquire the resource, one extra migration is included.

On the other hand, with the NP-section adopted, the migration cost in a single access can also be bounded by the length of the NP-sections, denoted by Mnp^k, as given by Equation (3.26), where C_{np} represents the length of the NP-section. Note that in our analysis we assume the length of NP-section as a positive integer value (by default $C_{np} = \hat{b}$).

$$Mnp^k = C_{mig} \cdot (\left\lceil \frac{c^k}{C_{np}} \right\rceil + 1) \tag{3.26}$$

In the case where the holder can be preempted frequently, this equation can give a more acceptable number of migrations that a MrsP resource holder can incur. Unlike Equation (3.25), this equation does not rely on iterations as the NP-section is for the resource execution only and does not include the cost of migrations. Therefore, $\left\lceil \frac{c^k}{C_{np}} \right\rceil$ can provide a safe bounding on the number of migrations with NP section applied. Combing Equations (3.25) and (3.26) we give the following lemma, where the request is issued from processor m:

Lemma 4. *$Nmig = min\{Mhp(mt, r^k), Mnp^k\}$ if $P_m \in mtp(mt, r^k) \wedge |mtp(mt, r^k)| > 1$.*

Proof. In the case where $Mnp^k < Mhp(mt, r^k)$, the resource holder is protected by the NP section while some of the preemptions are delayed so that $Nmig = Mnp^k$. In contrast (where $Mhp(mt, r^k) \leq Mnp^k$), the holder often can execute for an amount of time longer than C_{np} after migrations without the effect of NP sections. Thus, $Nmig = Mhp(mt, r^k)$. ■

In addition, note that c^k is still in use instead of C^k (i.e., c^k plus protocol implementation overheads) in Equations (3.25) and (3.26) as preemptions should not be allowed during functions `lock()` and `unlock()` to guarantee the correctness of tasks' behaviours while requesting or releasing a MrsP resource (e.g., manipulating FIFO queues, changing priorities and updating MrsP data structures). Such NP-sections should be accounted for in $\hat{b}$ as the blocking is caused by the underlying operating system upon tasks' arrival.

Combining Lemma 1 to 4, we give the total migration cost a task can incur. In the worst case, the task has to cope with the migration cost of all the requests in the FIFO queue, including the migration cost of those resource requests. Let $Mig(mt, r^k)$ be the total migration cost that a task can incur for accessing r^k with a given set of migration targets mt:

$$Mig(mt, r^k) = \sum_{P_m \in mt} \begin{cases} 0, & \text{if } P_m \notin mtp(mt, r^k) \vee \{P_m\} = mt \\ 2 \cdot C_{mig}, & \text{if } \{P_m\} = mtp(mt, r^k) \wedge |mt| > 1 \\ min\{Mhp(mt, r^k), Mnp^k\}, & \text{otherwise} \end{cases} \tag{3.27}$$

With the migration cost analysis constructed, we integrate this with the response time analysis presented above to form a complete run-time overheads-aware schedulability analysis for MrsP systems. Firstly, the migration cost should be integrated into the equation that bounds the spin delay (see Equations (3.4) and (3.28)). The set of migration targets are identified previously by $mt_x^k(l)(n)$.

$$e_x^k(l, \mu) = \sum_{n=1}^{N_x^k(l,\mu)} (e_x^k(l)(n) + Mig(mt_x^k(l)(n), r^k)) \tag{3.28}$$

In addition, the migration cost also needs to be accounted for when bounding the arrival blocking. The set of migration targets here are given by α_i^k. Equation (3.29) gives the arrival blocking with the migration cost integrated. In the case where r^k is a local resource, $Mig(\alpha_i^k, r^k) = 0$ as $\alpha_i^k = \{P(\tau_i)\}$.

$$\hat{e_i} = max\{|\alpha_i^k| \cdot C^k + Mig(\alpha_i^k, r^k) | r^k \in F^A(\tau_i)\} \tag{3.29}$$

Finally, as we adopt the NP-section for migrations, an extra blocking effect should be accounted for. If the length of the NP-section is configured as the maximum NP-section length in the hosting operating system ($\hat{b}$), no further modifications to the equations are required. Otherwise (where $C_{np} > \hat{b}$), for any given task τ_i, it has the risk to incur such a blocking (denoted by $\hat{np_i}$) as long as it has a priority equal or higher than the lowest ceiling priority of the global resources on its processor:

$$\hat{np_i} = \begin{cases} C_{np}, & \text{if } Pri(\tau_i) \geq min_{\{r^k \text{ is global}\}} Pri(r^k, P(\tau_i)) \\ 0, & \text{otherwise} \end{cases} \tag{3.30}$$

Same with the arrival blocking, such a blocking happens before the execution of τ_i and can only happen once. Therefore, Equation (3.6) should be modified to reflect this extra blocking.

$$B_i = max\{\hat{e_i}, \hat{np_i}, \hat{b}\} \tag{3.31}$$

This concludes the work of MrsP Schedulability Analysis. The notations introduced by this analysis are summarised in Table 3.3. Combining the response time analysis and the migration cost analysis together, we provide an improved and more complete schedulability analysis tool for MrsP with the awareness of implementation and run-time costs, especially the cost of migrations.

Table 3.3: New Notations in the New MrsP Analysis

C_{mig}	The cost of one migration.
C_{np}	The length of the NP section.
$\hat{np}_i$	The blocking that τ_i can incur due to the NP section.
$mt_x^k(l)(n)$	The migration targets of τ_x's nth access to r^k within the duration l.
$mtp(mt, r^k)$	A set of migration targets with tasks that can preempt tasks that accessing resource r^k in the given set of migration targets mt.
$Mig(mt, r^k)$	The total migration cost a task can incur for accessing r^k with the given set of migration targets mt.
$Mhp(mt, r^k)$	The migration cost of a single access to r^k bounded by the releases of high priority tasks on the given set of migration targets mt.
Mnp^k	The migration cost of a single access to r^k bounded by the length of the NP section.
$hpt(r^k, P_m)$	tasks on partition m that have a higher priority than the ceiling of r^k.

3.2.4 Summary

This section presents new schedulability tests for the candidate resource sharing protocols in the proposed multiprocessor resource control framework, where each of the analysis is developed based on Theorems 1 and 2 in Section 3.2.1 with the back-to-back hit phenomenon accounted for. In addition, the new schedulability tests include the major run-time overheads from both the underlying operating system and the protocols. Finally, the new schedulability analysis preserves the philosophy of the original schedulability tests of these protocols and avoids the use of potentially expensive analysing techniques.

Therefore, the proposed schedulability tests satisfy all the requirements listed in Section 3.2.

3.3 A Flexible Schedulability Test Framework

With the schedulability tests for the candidate resource sharing protocol developed, a complete schedulability analysis framework for systems with potentially all the candidate locking protocols in use can be formed via integrating each of the schedulability test. As the schedulability tests is developed based on the same format, merging these analysis is relatively straightforward, but is essential to achieve analysable systems with the proposed FMRS framework adopted.

3.3.1 The Response Time Equation

The response time of τ_i under such a system is given in Equation (3.32). To achieve a simple analytical expression while bounding the spin delay, the migration cost of τ_i in E_i and $I_{i,h}$ due to MrsP resources is separated as an independent variable, denoted by MC_i.

$$R_i = CX_1 + C_i + E_i + B_i + \sum_{\tau_h \in \mathbf{hpl}(i)} \Big(\left\lceil \frac{R_i}{T_h} \right\rceil \cdot (CX_2 + C_h) + I_{i,h} \Big) + S_i + MC_i \quad (3.32)$$

This is because by treating the additional costs incurred by PWLP and MrsP as independent variables (i.e., S_i and MC_i respectively), the analytical expression for bounding E_i and $I_{i,h}$ can be identical with any of the protocols adopted, where PWLP and MrsP can have the same resource-accessing queue with that of MSRP if ignoring the additional costs due to the preemptive spinning approach.

In addition, note that with all three protocols in use, the overheads of context switches (i.e., CX_1 and CX_2) can be higher than that of only adopting one protocol, especially with the helping mechanism in MrsP. The exact bounding of CX_1 and CX_2 in this analysis is measured in Appendix B.2 by considering the underlying hardware platform, a real-world operating system and implementations of all the candidate locking protocols.

3.3.2 The Direct and Indirect Spin Delay

We start with bounding the blocking variables E_i and $I_{i,h}$ with the presence of potentially more than one locking protocols. As described above, the spin delay incurred due to each protocol is identical if the additional cost is separated away. Thus, E_i and I_i can be simply bounded via Equations (3.2) to (3.4) and (3.12), as proved in the above sections.

As the protocols can carry different implementation overheads in their own `lock()` and `unlock()` functions, the notation C^k is in Equation (3.12) now denote the cost for executing r^k with the overheads incurred from its designated locking protocol. With the proposed FMRS framework adopted, each resource in a given system will be controlled by one of the candidate locking protocols; depending on the decisions made by the framework. Accordingly, C^k is now bounded by Equation (3.33), where function R_p returns a set of resources that are managed by a given locking protocol p.

$$C^k = c^k + \begin{cases} C^{lock}_{MrsP} + C^{unlock}_{MrsP}, & \text{if } c^k \in R_{MrsP} \\ C^{lock}_{PWLP} + C^{unlock}_{PWLP}, & \text{if } c^k \in R_{PWLP} \\ C^{lock}_{MSRP} + C^{unlock}_{MSRP}, & \text{otherwise} \end{cases} \tag{3.33}$$

Note that the sets R_{MSRP}, R_{PWLP} and R_{MrsP} will never intersect with each other (i.e., an element in a given set will never belongs to other sets), but should be equal to the total resources in the system (i.e., R) when these sets are combined, where

$$R \triangleq R_{MSRP} \cup R_{PWLP} \cup R_{MrsP} \tag{3.34}$$

In addition, If not all protocols are in use (say PWLP is not applied) based on the decisions from the resource control framework, then $R \triangleq R_{MSRP} \cup R_{MrsP}$. However, the framework should guarantee that each resource in the system is managed by a designated locking protocol.

3.3.3 Spin Delay from the Additional Facilities

Besides the direct spin delay and the indirect spin delay, tasks with PWLP and MrsP adopted can also incur additional blocking due to the cancellation mechanism and the migration-based helping mechanism, denoted as S_i and MC_i respectively in Equation (3.32).

We firstly present the bounding of variable S_i. As with the equations given in Section 3.2.2, S_i is bounded via Equation (3.18) while $F^S(\tau_i)$ is given by Equation (3.16). However, the equation that computes the list LS_i is revised to consider PWLP resources only, as given in Equation (3.35).

$$LS_i = \{C_{retry} + L^k_i |\ r^k \in R_{PWLP} \wedge r^k \in F^S(\tau_i)\}^{dList} \tag{3.35}$$

In addition, C^k in Eequation (3.33) is applied to Equation (3.15) to incorporate the overheads of `lock()` and `unlock()` functions in PWLP, as given below. This is feasible because r^k in this equation belongs to the set R_{PWLP} (i.e., $r^k \in R_{PWLP}$).

$$L_i^k = \{ \sum_{P_m \neq P(\tau_i)} (NS_{i,m}^k(R_i) - N_i^k - n + 1)_0^1 \cdot C^k | 1 \leq n \leq NoP_i \} \tag{3.36}$$

As for the variable MC_i, it can be formed via separating the migration cost from Equation (3.28), where the notation $MIG_x^k(l, \mu)$ is introduced to denote the amount of migration cost caused by τ_x (which can be either τ_i, i.e., the task that is being studied, or a local higher priority task τ_h) for accessing r^k within the given duration l and jitter μ.

$$MC_i = MIG_i^k(R_i, 0) + \sum_{\tau_h \in hpl(i)} MIG_h^k(R_i, R_h) \tag{3.37}$$

The equation for bounding $MIG_x^k(l, \mu)$ is formed via extracting the migration cost bounding given in Equation (3.28), where Mig gives the migration cost for one access to a resource. In this equation, resource r^k is specified to be a resource that is controlled by MrsP and is required by τ_x to guarantee the correctness of the analysis.

$$MIG_x^k = \sum_{r^k \in R_{MrsP} \wedge r^k \in F(\tau_x)} \sum_{n=1}^{N_i^k(l,\mu)} Mig(mt_i^k(l)(n), r^k) \tag{3.38}$$

With the above equations established, the equations given in Section 3.2.3 can be applied to calculate variable Mig directly without any modifications, and hence, the migration cost MC_i can be safely bounded.

3.3.4 The Arrival Blocking

As a task can only be blocked once upon each arrival, there will be only one factor (a resource managed by a designated locking protocol, the NP-section in MrsP, or the NP-section in the underlying operating system) that can actually cause the arrival blocking to τ_i. To capture the worst-case scenario, the arrival blocking that incurred by τ_i should be the maximum value among all these factors, as given in Equation (3.39).

$$B_i = max\{B_i^{MSRP}, B_i^{PWLP}, B_i^{MrsP}, \hat{np_i}, \hat{b}\} \tag{3.39}$$

where B_i^p gives the maximum blocking time τ_i can incur with resources that are managed by protocol p and $\hat{np_i}$ denotes the blocking due to the NP-section with MrsP adopted. As described in Section 3.2.3, the variable $\hat{np_i}$ can be bounded by Equation (3.40), but should specify that the resource is under MrsP's control, revised as follow.

$$\hat{np_i} = \begin{cases} C_{np}, & \text{if } r^k \in R_{MrsP} \wedge Pri(\tau_i) \geq min_{\{r^k \text{ is global}\}} Pri(r^k, P(\tau_i)) \\ 0, & \text{otherwise} \end{cases} \tag{3.40}$$

As for B_i^p, this variable is computed independently for each protocol because the analysing techniques vary with different protocols adopted. For MSRP resources, B_i^{MSRP} can be bounded by the following equation, which is similar with Equation (3.13) but with MRSP resources specified.

$$B_i^{MRSP} = max\{|\alpha_i^k| \cdot C^k | r^k \in R_{MSRP} \wedge r^k \in F_{NP}^A(\tau_i)\} \tag{3.41}$$

To simplify the analytical expression, function $F_{NP}^A(\tau_i)$ is introduced to denote the set of resources that can cause arrival blocking to τ_i under protocols where resources are accessed or executed non-preemptively (i.e., MSRP and PWLP), as given in Equation (3.42), where α_i^k can be bounded via Equation (3.9) directly.

$$F_{NP}^A(\tau_i) \triangleq \{r^k | N_{ll}^k > 0 \ \wedge (Pri(r^k, P(\tau_i)) \geq Pri(\tau_i) \vee r^k \text{ is global})\} \tag{3.42}$$

With $F_{NP}^A(\tau_i)$ defined, the arrival blocking caused by PWLP resource can be simply bounded via the following equation.

$$B_i^{PWLP} = max\{C^k | r^k \in R_{PWLP} \wedge r^k \in F_{NP}^A(\tau_i)\} \tag{3.43}$$

As described in Section 3.2.3, The set of resources that can cause τ_i to incur arrival blocking with MrsP adopted is different from that of MSRP and PWLP due to the ceiling facility. In addition, tasks with this protocol can incur prolonged blocking due to the migration cost, and hence, should also be accounted for in variable B_i^{MrsP}, as given below.

$$B_i^{MrsP} = max\{|\alpha_i^k| \cdot C^k + Mig(\alpha_i^k, r^k) | r^k \in R_{MrsP} \wedge r^k \in F_{Ceiling}^A(\tau_i)\} \tag{3.44}$$

where α_i^k can be computed via Equation (3.9) while $Mig(\alpha_i^k, r^k)$ can be bounded by Equation (3.27) directly. The notation $F_{Ceiling}^A(\tau_i)$ is introduced to denote the set of resources that can cause τ_i to incur arrival blocking with the ceiling priority resource accessing rule applied (i.e., MSRP in this framework), which is identical with Equation (3.22), as given below.

$$F^A_{Ceiling}(\tau_i) \triangleq \{r^k | N^k_{ll} > 0 \land Pri(r^k, P(\tau_i)) \geq Pri(\tau_i)\} \tag{3.45}$$

With all the variables in B_i computed, the arrival blocking that τ_i can incur with potentially all the candidate locking protocols in use can be safely bounded.

3.3.5 Summary

Table 3.4: Notations in the New Analysis Framework

MC_i	The total migration cost incurred by τ_i in the spin delay.
$MIG^k_i(l, \mu)$	The amount of migration cost caused by τ_x for accessing r^k within the given duration l and jitter μ.
R	The shared resources in the given system.
R_{MSRP}	The resources that are controlled by MSRP.
R_{PWLP}	The resources that are managed by PWLP.
R_{MrsP}	The resources that are controlled by MrsP.
B^{MSRP}_i	The arrival blocking caused by the MSRP resources.
B^{PWLP}_i	The arrival blocking caused by the PWLP resources.
B^{MrsP}_i	The arrival blocking caused by the MrsP resources.
$F^A_{NP}(\tau_i)$	The set of resources that can cause τ_i to incur arrival blocking with the non-preemptive resource-accessing rule.
$F^A_{Ceiling}(\tau_i)$	The set of resources that can cause τ_i to incur arrival blocking with the ceiling priority resource accessing rule applied.

The above has presented a complete run-time overheads-aware schedulability test framework for systems with multiple locking protocols in use. The notations introduced in the new schedulability analysis framework are summarised in Table 3.4. This analysis is developed via integrating the new schedulability test of each candidate locking protocol developed in Sections 3.2.1 to 3.2.3, where each of the analysis is developed based on Theorems 1 and 2 in Section 3.2.1 with the back to back hit phenomenon accounted for. In addition, the new analysis framework includes the major run-time overheads from both the underlying operating system and the protocols, especially the cost of migrations with MrsP adopted. Finally, the new schedulability preserves the philosophy of the original schedulability tests of these protocols and avoids the use of potentially expensive analysing techniques. Therefore, the proposed schedulability test framework developed in this section satisfies all the requirements listed in Section 3.2.

This schedulability framework provides the foundation of the proposed resource control framework. While determining the appropriate locking protocol for each resource in a given system, this schedulability test must be adopted to verify whether such decisions can lead to a schedulable system. In Chapter 5, the techniques for determining the resource sharing solutions are presented. In Chapter 6, a set of experiments are conducted to investigate the efficiency and the performance of the flexible multiprocessor resource sharing framework based on the schedulability analysis framework developed in this section.

3.4 Investigating the Schedulability of the Candidate Resource Sharing Protocols

With the new schedulability tests developed for each candidate resource sharing protocol in Section 3.2, a set of experiments are conducted to investigate (1) the schedulability between the original tests and the new tests; (2) the schedulability of the candidate locking protocols; (3) the impact of the run-time overheads to the schedulability results and (4) the time consumption of the new schedulability tests and the ILP-based analysis. The evaluations performed in this section provide evidence that directly demonstrates the necessity of the requirements R-1 to R-5 given in Section 3.2. The code for the evaluations performed in this section can be accessed via `https://github.com/RTSYork/SchedulabilityTestEvaluation`. To conduct this evaluation, the following schedulability tests are implemented:

- The original schedulability tests of MSRP and MrsP.
- The new response time analysis of MSRP, PWLP and MrsP without any run-time overheads.
- The complete run-time overheads-aware schedulability tests of MSRP, PWLP and MrsP with scheduling and locking overheads accounted for.
- A pluggable migration cost analysis for MrsP.

To compare the time consumption of the schedulability tests developed in Section 3.2 and the ILP-based analysis proposed in [106], the implementation of the ILP-based analysis from the SchedCAT project [18] is integrated into the testing program via JNI. To provide randomly generated systems for analysing, a system generation tool is developed to generate systems with different application semantics and resource characteristics configurations.

The experimental setup for investigating the schedulability tests in this thesis is similar with that of the ILP-based analysis work [106], which covers a wide range of system settings in real-time automotive applications. In this thesis, we consider platforms with $M = [2, 24]$ processors, where systems with $M \leq 8$ are radially available now while $M > 8$ gives the forward-looking scenario. The system contains n tasks with a total utilisation U and $U = 0.1n$. Tasks are allocated to each processor via the WF algorithm described in Section 2.2.3. Periods of tasks on each processor are randomly chosen between $[1ms, 1000ms]$ in a log-uniform distribution fashion. In this evaluation, we assume that the deadline of the tasks are equal to their periods ($D = T$). The utilisation of each task is computed based on the UUnifast-Discard algorithm proposed by Bini and Buttazzo [14] and hence the total computation time for each task (including the time it spends on executing each required resource, denoted as C'_x for τ_x) can be computed. The system supports 1000 priority levels. The priorities of the tasks in a given system is assigned via the DMPO algorithm prior to allocation.

In addition, as with the settings adopted in [106], tasks in each system share either $M/2$, M or $2M$ resources. A wide range of critical section length (L): $[1\mu s, 15\mu s]$, $[15\mu s, 50\mu s]$, $[50\mu s, 100\mu s]$, $[100\mu s, 200\mu s]$, $[200\mu s, 300\mu s]$ and $[1\mu s, 300\mu s]$ is supported. A real value parameter κ is introduced to specify the number of tasks on each processor that can access to resources (i.e., $\lfloor \kappa \cdot n \rfloor$), where $\kappa \in$ [0.0,1.0]. A task will issue requests to a number of randomly chosen resources, but limited by $[1, M]$. The number of requests is randomly decided between $[1, A]$, where $A = [1, 41]$. Let C^r_x be the total resource computation time of τ_x. For a given task τ_x, with its resource usage generated, the pure computation time C_x can be computed, where $C_x = C'_x - C^r_x$. We enforce that $C'_x - C^r_x \geq 0$. Note, unless specified, the word "random" in this evaluation indicates the values are generated randomly with an uniform distribution.

At last, the statistical significance of the experimental results presented in this section is illustrated via an ANOVA analysis, which demonstrates a confidence level of 95% of the observations (and claims) made based on the experiments. The detailed approach of conducting the ANOVA analysis is presented in Appendix E.

3.4.1 Schedulability Comparison

We first investigate the theoretical response time of the candidate resource sharing protocols under systems with various application semantics and resource characteristics, including (a) work load of the system U (i.e., $0.1n$); (b) parallelism m; (c) critical section length L and (d) resource contention A (i.e., the frequency of resource access). The schedulability tests adopted in this evaluation includes the original MSRP analysis [48] (MSRP-original); the original MrsP analysis [27] (MrsP-original); the new MSRP analysis (MSRP-new); the new PWLP analysis (PWLP-new); and the new MrsP analysis (MrsP-new) developed in Section 3.2. In this evaluation, the schedulability of the above analysis is examined by testing 1000 system produced by the task generation tool with the system settings given below.

Note, there exist a large amount of possible combinations of the system settings given above (e.g., n, M, L and A). In the interest of brevity, we only present the results under certain system settings, which can effectively demonstrate the schedulability difference of the evaluated schedulability tests. This remains in the following thesis.

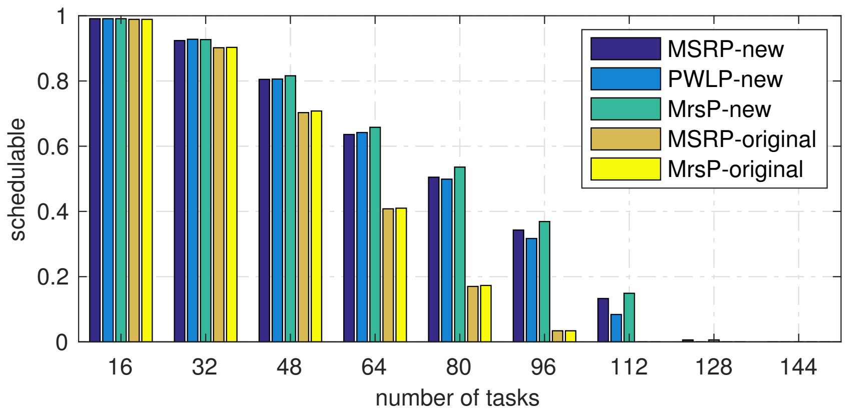


Figure 3.2: Schedulability for $M = 16$, $U = 0.1n$, $\kappa = 0.4$, $A = 2$, $L = [15\mu s, 50\mu s]$, and M Shared Resources.

(a) *Varying n and M:* With a low resource contention ($A = 2$ and $\kappa = 0.4$) and short critical section length ($L = [15\mu s, 50\mu s]$), MrsP demonstrates a

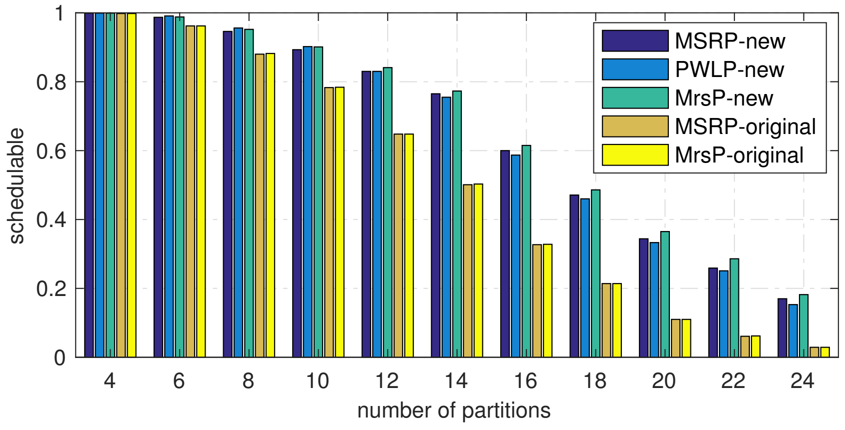


Figure 3.3: Schedulability for $n = 4M$, $U = 0.1n$, $\kappa = 0.4$, $A = 3$, $L = [15\mu s, 50\mu s]$, and M Shared Resources.

better schedulability in theory than other spin locks, as shown in Figure 3.2, where the Y axis gives the percentage of the schedulable systems among the 1000 systems generated based on the system setting given in the X axis. By further incrementing n, the original analysis of both MSRP and MrsP gives a much lower schedulability than that of our new analysis, which shows the reduced pessimism of our new schedulability tests. Another interesting observation is that PWLP shows similar schedulability with that of MSRP when $n \leq 64$, but is outperformed by MSRP with $n \geq 80$. This is because with more tasks, tasks have a higher chance to be preempted while waiting for a PWLP resource so that more additional blocking is imposed due to the cancellation mechanism.

A similar trend between MrsP and MSRP is observed when increasing M (see Figure 3.3). However, PWLP in this experiment offers the best schedulability when $M \leq 12$ due its relatively low arrival blocking. Yet with a further increasing of M, both MrsP and MSRP give a better schedulability than that of PWLP (when $m \geq 14$) as the spin delay can be bounded to M in theory.

(b) *Varying A:* As shown in Figure 3.4, PWLP has the best schedulability and MSRP is worse than both MrsP and PWLP in the case where $A = 1$, as tasks incur limited preemptions within a short resource-accessing time and such a cost is more likely to be less than the arrival blocking that tasks can suffer under MSRP or MrsP. However, with a further increment of A (and an increased risk to preempt resource-accessing tasks), PWLP becomes the worst with an observable difference compared to the other two protocols.

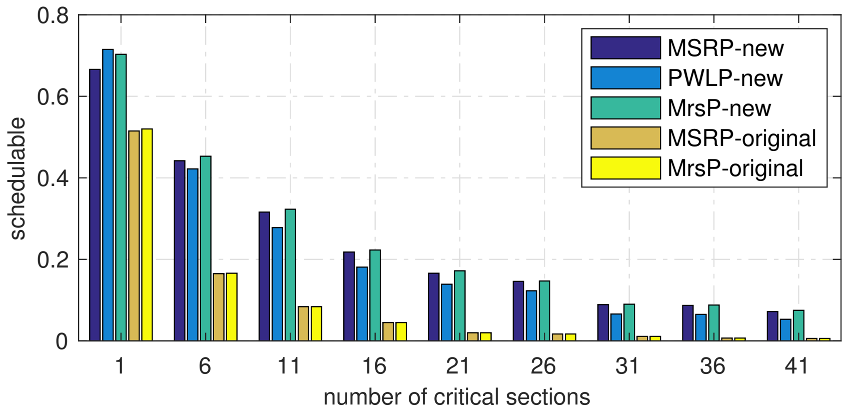


Figure 3.4: Schedulability for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.4$, $L = [15\mu s, 50\mu s]$, and M Shared Resources.

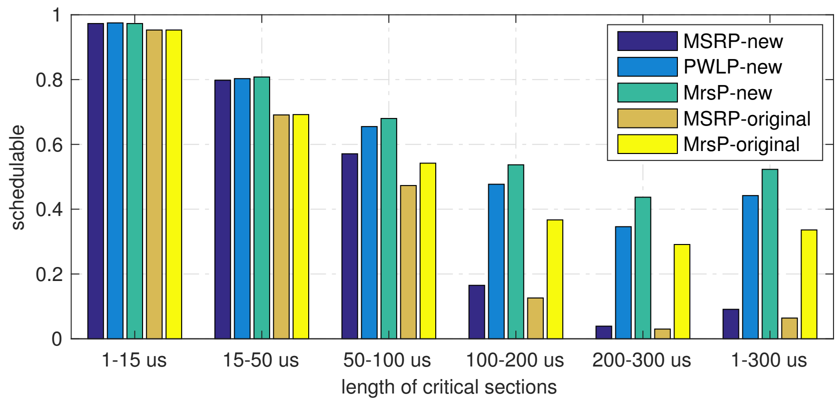


Figure 3.5: Schedulability for $M = 16$, $n = 48$, $U = 0.1n$, $\kappa = 0.4$, $A = 2$ and M Shared Resources.

(c) *Varying L:* With an increasing length of critical sections, we observed the schedulability of MSRP locks decreases dramatically while MrsP provides the best schedulability among all tested locks (see Figure 3.5). With MSRP, the highest priority tasks have to cope with the largest arrival blocking, and hence, can easily miss their deadlines if long critical sections are adopted. In contrast, although with a longer spin delay, PWLP locks only incur a local blocking so that it can offer a higher schedulability than that of MSRP. Under MrsP, tasks can incur a limited amount of arrival blocking due to the ceiling facility and can have a shorter spin delay than that of PWLP. Thus, MrsP can offer a better schedulability with long critical sections than both MSRP and PWLP can achieve. Due to the same reason, MrsP can also demonstrate the best schedulability among all examined protocols in a more realistic case (i.e., $L = [1\mu s, 300\mu s]$), where a system could contain both short and long resources.

From the experiments we observed that the newly-developed schedulability tests can demonstrate much better schedulability than that of the original tests. Such observation directly supports the necessity of the requirements R-1 to R-2 given in Section 3.2, which aim to reduce the degree of the pessimism of the schedulability results, as described in [106].

Theoretically, MrsP offers a better (at least identical) schedulability than MSRP in all cases because both protocols have an identical spin delay but MrsP can guarantee a shorter arrival blocking at most times. In addition, as observed, both MSRP and MrsP are less efficient than PWLP in systems with low resource contention or less partitions due to adopting either the non-preemptive accessing or the resource ceiling facility approach. More importantly, for long resources, the preemptive spinning approaches are able to provide a much better schedulability than the non-preemptive approach. Admittedly, one can argue that for long critical sections, the suspension-based locks should be applied rather than spin locks. However, as revealed by the experiments, both the PWLP and MrsP can be considered applicable to long critical sections by offering an acceptable schedulability ratio, where MrsP gives a better schedulability in theory. As described before, the confidence level of such observations is 95% (see Appendix E for proof).

3.4.2 The Back-to-Back Hits

This section provides an experiment demonstrating the necessity of the requirement R-3 in Section 3.2, where the back-to-back hits should be accounted for to increase the accuracy of the proposed schedulability tests. The experiment is conducted by varying A with the same system settings adopted in Figure 3.4, as given in Figure 3.6. The schedulability tests that are examined in this experiment including the proposed schedulability tests for the candidate locking protocols (i.e., MSRP, PWLP and MrsP) and the modified schedulability tests that do not take the back-to-back hits into account, denoted as MSRPˆ, PWLPˆ and MrsPˆ, where the parameter μ in equation $N_x^k(l, \mu) = \left\lceil \frac{l+\mu}{T_x} \right\rceil \cdot N_x^k$ is always 0.

As shown in the figure, the schedulability tests with the back-to-back hits

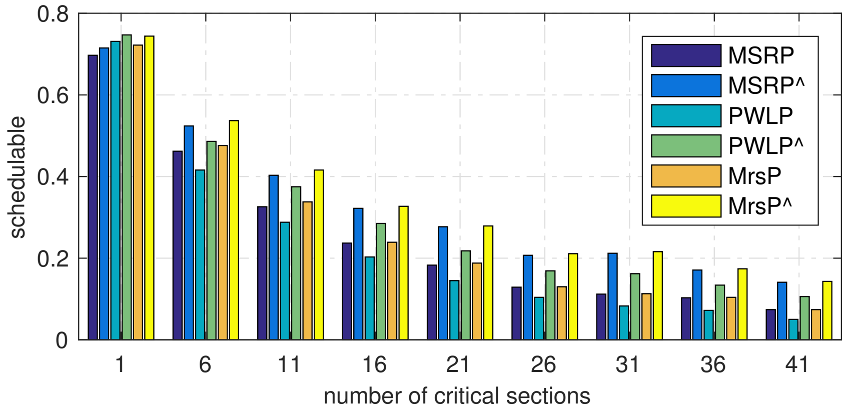


Figure 3.6: Schedulability for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.4$, $L = [15\mu s, 50\mu s]$, and M Shared Resources.

accounted for demonstrate non-trivial schedulability differential (i.e., more accurate schedulability results) than that of the tests without the back-to-back hits in all cases. In addition, by incrementing A, such differential between these two types of analysis is further increased. As described in Section 2.6.2.2, the back-to-back phenomenon can cause a task to incur extra blocking. This section provides experimental evidence that illustrates this phenomenon and proves the necessity of the requirement R-3 in Section 3.2, which mandates the calculation of the blocking due to the back-to-back hits in the new schedulability tests proposed in this thesis. The statistical significance of the results in this experiment is given in Appendix E via the ANOVA analysis with a confidence level of 95%.

3.4.3 Run-time Overheads

Now we study the impact of the run-time overheads to the schedulability results with the analysis developed in Section 3.2. The experiment is conducted by varying the critical section length L. In addition, we present evidence of an improved efficiency of MrsP by the controlled migration behaviours due to the NP-section. The approaches for bounding the run-time overheads due to both the operating system and locking protocols are illustrated in Appendix B under Litmus^{RT} with all candidate locking protocols implemented (see Appendixes C and D for the implementation details). Table 3.5 summarises the worst-case bounding of the run-time cost variables introduced in the newly-developed schedulability tests measured from Appendix B, and will be adopted in this experiment.

Table 3.5: The Run-time Costs of the Candidate Protocols under Litmus^{RT}

Variables	Worst-case Cost	Variables	Worst-case Cost
CX_1	5606 ns	C^{unlock}_{MSRP}	602 ns
CX_2	10,240 ns	C^{lock}_{PWLP}	1255 ns
C_{retry}	1663 ns	C^{unlock}_{PWLP}	602 ns
C_{mig}	8378 ns	C^{lock}_{MrsP}	1272 ns
C^{lock}_{MSRP}	979 ns	C^{unlock}_{MrsP}	1642 ns

The schedulability analysis examined in this experiment includes (1) new MSRP test without run-time overheads (MSRP); (2) new MSRP test with run-time overheads (MSRP*); (3) new PWLP test without run-time overheads (PWLP); (4) new PWLP test with run-time overheads (PWLP*); (5) new MrsP analysis without run-time overheads (MrsP); (6) new MrsP analysis with run-time overheads, including the cost of migrations but without the protection of the NP-section (MrsP*); and (7) new MrsP analysis with NP section adopted, including run-time overheads and the NP-section adopted (MrsP-np*). The analysis "MrsP*" is modified from the analysis in Section 3.2.3 by taking the functions Mnp^k and $\hat{np}_i$ out of Equations (3.27) and (3.31) respectively. When "MrsP-np" is in use, the length of the NP sections are set differently based on a given generated system to achieve the best schedulability, and hence is not presented. As described in Section 3.2.3, this length can be tuned for each individual system to achieve the best schedulability.

From the experiment in Figure 3.7, we firstly observed that the schedulability tests with the run-time overheads accounted for can demonstrate more accurate schedulability results than the theoretical response time analysis do (e.g., MSRP vs. MSRP* and PWLP vs. PWLP*), especially MrsP, where the cost of migrations leads to the protocol barely impractical without the NP-sections. Such an observation reveals the necessity of incorporating the run-time overheads into schedulability tests. However, with the NP-section adopted (i.e., MrsP-np*), the efficiency of the helping mechanism is significantly improved, where the schedulability of MrsP* is much lower than MrsP-np* in all cases. Compared to PWLP*, MrsP-np* is less favourable when applied to short critical sections as one single migration has a cost of $8.378\mu s$ in this experiment, where MrsP-np* provides a low schedulability

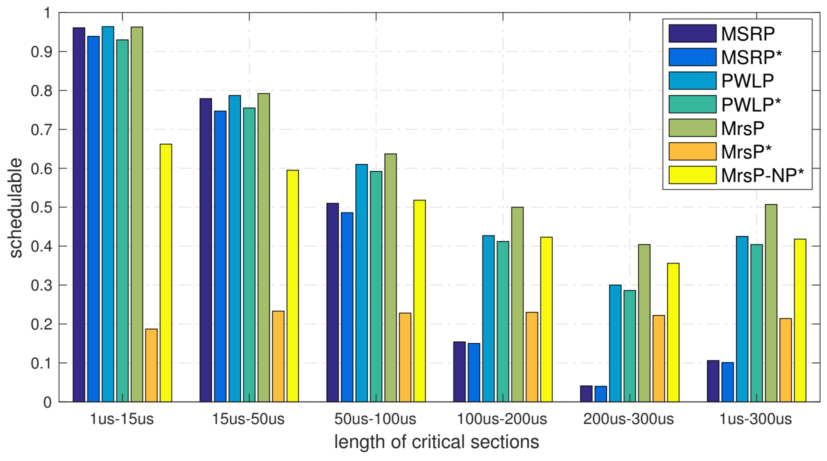


Figure 3.7: Schedulability for $M = 16$, $n = 48$, $U = 0.1n$, $\kappa = 0.4$, $A = 3$ and M Shared Resources.

with $L = [1\mu s, 50\mu s]$. However, when $L \geq 100\mu s$, MrsP with the NP-section adopted shows a better schedulability than both MSRP* and PWLP*, including the case of $L = [1\mu s, 300\mu s]$, which again proves that MrsP works better with long critical sections. By taking the run-time overheads into account while analysing all candidate locking protocols, we have revealed the real schedulability of these protocols (with a confidence level of 95%). The above discussion and evaluation demonstrate the necessity of requirement R-4 given in Section 3.2.

3.4.4 Time Consumption

The last experiment conducted in this section is to investigate and to compare the time consumption of our newly-developed schedulability tests and the ILP-based analysis. As the computing time of a given test largely depends on the exact system being generated, there can be huge differences between the computation times under a same system setting. To illustrate the overall time consumption of the schedulability tests in general, 1000 systems will be generated by each system setting and an average computing time of each analysis under each system setting is reported, as given in Tables 3.6 and 3.7, where “MSRP-ilp” denotes the ILP-based analysis with MSRP constraints adopted and “PWLP-ilp” denotes the ILP-based analysis for PWLP. Table 3.6 gives the time consumption by varying n while Table 3.7 shows the results with varied M.

As given in both tables, the time consumption of all schedulability tests increases while incrementing either n or M. Among the newly-developed tests in this thesis, MrsP-new requires more time to compute the response times due to the additional migration cost analysis while MSRP-new and PWLP-new require similar computation time to deliver the results. In contrast, the ILP-based MSRP and PWLP analysis takes much more time to deliver the schedulability results, where the ILP-based tests require more than 1 second to finish while the newly-developed tests only require about 13 milliseconds to compute the responses times of all tasks under systems with $M = 16$ in Table 3.7. One interesting observation is that with a shorter L, all tests require larger computation time to deliver the schedulability results, see $n = 80$ in Table 3.6 and $M = 16$ in Table 3.7 (with $L = [15\mu s, 50\mu s]$ and $[1\mu s, 15\mu s]$ respectively). This is because that with a shorter critical section length, the response time of tasks will have a smaller increment under each recursion calculation so that more recursions could be required to either get fixed response times or reach the deadlines of tasks (i.e., where the test are finished).

Table 3.6: The Time Consumption for Analysing Systems with $M = 16$, $U = 0.1n$, $\kappa = 0.4$, $A = 2$, $L = [15\mu s, 50\mu s]$, and M Shared Resources.

n	MSRP-new	PWLP-new	MrsP-new	MSRP-ilp	PWLP-ilp
48	1.24 ms	0.96 ms	3.33 ms	139.6 ms	137.9 ms
64	1.92 ms	1.59 ms	5.69 ms	228.6 ms	233.9 ms
80	1.37 ms	0.87 ms	2.21 ms	252.8 ms	328.9 ms
96	1.85 ms	0.99 ms	3.16 ms	318.6 ms	441.9 ms
112	1.98 ms	1.01 ms	3.35 ms	347.4 ms	618.2 ms

Table 3.7: The Time Consumption for Analysing Systems with $n = 5M$, $U = 0.1n$, $\kappa = 0.4$, $A = 2$, $L = [1\mu s, 15\mu s]$, and M Shared Resources.

M	MSRP-new	PWLP-new	MrsP-new	MSRP-ilp	PWLP-ilp
4	0.19 ms	0.17 ms	1.98 ms	37.1 ms	28.7 ms
8	1.72 ms	1.22 ms	13.3 ms	361.8 ms	359.5 ms
12	4.86 ms	3.42 ms	11.0 ms	972.7 ms	1168 ms
16	6.83 ms	4.72 ms	13.3 ms	1299 ms	1700 ms

Tables 3.8 and 3.9 present the increase rate of the computation cost of each analysis given in Tables 3.6 and 3.7. As shown in both tables, the computation

Table 3.8: The Increase Rate of the Time Consumption in Table 3.6.

n	MSRP-new	PWLP-new	MrsP-new	MSRP-ilp	PWLP-ilp
$48 \to 64$	155 %	166 %	171 %	164 %	170 %
$64 \to 80$	71 %	55 %	39 %	111 %	141 %
$80 \to 96$	135 %	114 %	143 %	126 %	134 %
$96 \to 112$	107 %	102 %	106 %	109 %	140 %

Table 3.9: The Increase Rate of the Time Consumption in Table 3.7.

M	MSRP-new	PWLP-new	MrsP-new	MSRP-ilp	PWLP-ilp
$4 \to 8$	905 %	718 %	672 %	975 %	1253 %
$8 \to 12$	283 %	280 %	83 %	267 %	325 %
$12 \to 16$	141 %	138 %	121 %	134 %	146 %

cost of each analysis does not follow the linear pattern (i.e., positive proportional) when incrementing either n or M. In general, the computation cost of the ILP-based analysis demonstrates a slightly higher increase rate compared to that of the newly-developed analysis (the first three analysis). Based on these tables, the new analysis proposed in this thesis requires less computation cost (and has lower increase rates by giving higher systems setting parameters) than the ILP-based analysis.

In addition, we observe that the computation cost of the new analysis does not increase monotonically with the increase of the system parameter settings (see n from $64 \to 80$ in Table 3.8). This is because the systems being analysed are generated randomly (including the resource usage, see the experimental setup described at the beginning of Section 3.4). Thus, systems with low resource contention (i.e., systems with limited number of tasks accessing resources just a few times) could be generated under high system setting parameters, which could require less cost to analyse than the ones generated under low system setting parameters but with a strong resource contention. However, even under such case, the costs of the ILP-based analysis keep increasing due to its computation approach (i.e., the use of the ILP solver), which needs to establish the constraints (e.g., 8 constraints for analysing MSRP systems) for each task and each resource access, and hence, requires a large amount of calculations.

The similar trend can also be observed in Table 3.9, where the computation costs of the new MrsP analysis decrease when increasing M from 8 to 12 while other analysis has increased computation time. Unlike the new MSRP and PWLP analysis, the new MrsP schedulability analysis contains a migration cost analysis, which relies on recursive calculations, but only requires a few computations in each recursion (each recursion can add additional migration costs to the response time of that task, see Equations 3.25 to 3.29 in Section 3.2.3). Therefore, there could be the case that a given task fails to meet its deadline after a few recursions in these equations above so that the analysis of the whole system is terminated. Such computations usually require less costs than the recursive calculation in Equation 3.1, which has a more complicated computation mechanism and requires more time to finish. Therefore, the costs of the new MrsP analysis decrease while increasing M in this experiment.

The above explains the decreased trend observed in these experiments. Note, the computation costs and the increase rate presented above are by no means absolute values. This experiment aims only to illustrate that the new schedulability analysis developed in this thesis requires less costs than the ILP-based analysis in general. When increasing certain system setting parameters, there is no guarantee that the cost of the new analysis will follow a fixed trend as (1) the systems are generated randomly (including the resource usage) and (2) several termination mechanisms are added into the implementations of the analysis for improved run-time efficiency, which can terminate the analysis immediately as long as a a deadline miss is found (see implementation details in `https://github.com/RTSYork/SchedulabilityTestEvaluation`). Thus, there could be the case that the computation costs of certain analysis (especially the new ones, which does not reply on a ILP solver) decreases when increasing system setting parameters. However, as discussed, the above experiment is sufficient to illustrate that the ILP-based analysis is more expensive than the new analysis in a general case.

Admittedly, the time consumed by the ILP-based analysis for executing once is definitely acceptable. For systems with a locking protocol pre-defined, the ILP-based analysis provides a valuable analysing tool that can be adopted to analyse a wide-range of protocols (i.e., 8 types of spin locks in total). However, as for the proposed FMRS framework, where the resource sharing protocol for each resource can be different, adopting this analysis can lead to significant or even unacceptable time consumption as searching for a feasible resource control solution usually requires the schedulability test to execute many times (see Chapter 5 for the mechanism of searching feasible resource control solutions), even if such calculations are performed off-line (i.e., before the execution of the system).

For instance, for a typical system with 16 processors and 16 shared resources, there exist $3^{16} \approx 4.3 \times 10^7$ possible resource control solutions with the three candidate locking protocols, where a large number of possible solutions will be tested to search for a feasible solution, assuming no candidate protocols in FMRS can schedule this system. Accordingly, adopting the ILP technique as the schedulability test in such a search-based approach can lead to tremendous time consumption while the expenses of the schedulability analysis framework proposed in Section 3.3 are more acceptable. The experiment and discussion given in this section demonstrate the necessity of requirement R-5 given in Section 3.2.

From the experiments and the discussions give above, the necessity of the 5 requirements listed in Section 3.2 are certified. With all the requirements satisfied, the proposed run-time overheads-aware schedulability tests for the candidate resource sharing protocols are less pessimistic as well as more accurate than that of their original schedulability tests. In addition, with the NP-section adopted, the efficiency of the helping mechanism in MrsP is significantly improved. More importantly, we have proved that the preemptive spinning approach (especially MrsP) can demonstrate strong schedulability with long critical sections. Finally, by avoiding the use of any complicated and time-consuming analysing techniques, our new schedulability tests can be practised without incurring massive computation expenses. As illustrated by the ANOVA analysis, the above claims are made with a confidence level of 95%.

By building upon these schedulability tests, the schedulability analysis framework developed in Section 3.3 also carries the above features and provides the analysing tool for the FMRS framework proposed in this thesis with high usability.

3.5 Summary

This chapter firstly decided the candidate resource sharing protocols for the resource control framework proposed in this thesis via step-by-step reasoning, which is supported by experimental results. Then, new schedulability tests are developed for each candidate locking protocol with new analysing techniques that reduce the pessimism and improve the accuracy of the schedulability results. With the new schedulability tests, a complete run-time overheads-aware schedulability analysis framework is created for systems with potentially all the candidate locking protocols in use, which is crucial to achieve analysable systems with the proposed framework adopted. With the materials given in this chapter, the success criteria SC-1 in Section 1.4 is satisfied. The contributions presented in this chapter is summarised as below.

- New analysing techniques for systems with MSRP, PWLP or MrsP adopted.
- The NP-section for MrsP's helping mechanism and a pluggable migration cost analysis for the migration-based helping mechanism in MrsP.
- A complete run-time overheads-aware schedulability analysis framework for systems with multiple protocols in use.
- An investigation towards the schedulability of the MSRP, PWLP and MrsP; the impact of run-time overheads; and the expenses for using the proposed schedulability tests.

In addition, in the interest of brevity, additional contributions that are related to the multiprocessor resource sharing protocols are not presented in this chapter and are referred to Appendixes, as summarised below.

- Analysing techniques for heterogeneous and nested resource accesses (Appendix A).
- The techniques for bounding the run-time overheads incurred from both the underlying operating systems and the implementations of the candidate locking protocols (Appendix B).
- Fully functional MSRP, PWLP and MrsP implementations in the P-FP scheduler under Litmus^{RT} (Appendix C).
- An investigation towards the correctness and efficiency of the MrsP in fully-partitioned systems (Appendix D).

Chapter 4

Task Allocation and Prioritisation

In addition to the multiprocessor resource sharing protocols and schedulability tests studied in Chapter 3, the task allocation schemes can also have a significant impact to the schedulability of multiprocessor systems with shared resources (see discussion in Section 2.6.1). In this chapter, new resource-oriented task allocation algorithms are proposed to facilitate resource sharing on multiprocessors and to provide candidate task allocation solutions for the resource control framework proposed in this thesis. In addition, as described in Section 1.2, another major factor that can affect the performance of resource sharing on multiprocessor systems is task priority ordering. In this chapter, a formal proof is presented to demonstrate that the DMPO algorithm is not optimal with the new schedulability tests developed in Chapter 3. Then, a new search-based priority ordering algorithm that is fully compatible with the new schedulability tests in Chapter 3 is proposed and is evaluated with the priority ordering algorithms reviewed in Section 2.1.3. Materials provided in this section satisfies the success criteria SC-2 and SC-3 given in Section 1.4.

4.1 Resource-Oriented Task Allocation Schemes

As discussed in Section 2.6.1, the traditional task utilisation-based allocation schemes (e.g., the WF algorithm adopted in the experiments in Section 3.4) cannot benefit resource sharing as tasks are mapped without the knowledge of shared resources, where tasks accessing the same resource could be allocated to different processors so that a significant amount of remote blocking can be imposed. In Section 2.6.1, the SPA and BPA resource-oriented task allocation schemes are reviewed, where the BPA algorithm is an analysis specific approach (i.e., requires the weight and attraction functions of the locking protocol adopted) while the SPA algorithm attempts to localise shared resources based on the utilisation of task bundles.

As illustrated in Section 3.4, the schedulability of a system with the candidate resource sharing protocols adopted can vary under different application semantics and resource characteristics, where MSRP is favourable with short resources; MrsP demonstrates the best schedulability with long resources; and PWLP is more favourable with a low degree of parallelism and low resource contention. Therefore, compared to the SPA algorithm, which simply decreases the number of globally shared resources, localising the resources that are less favourable for a given candidate locking protocol could further increase the schedulability of the system. For instance, the schedulability of a MSRP system can be further increased with the long resources localised (i.e., with the blocking from long resources reduced) while localising the competitive resources (i.e., the resources that are requested most frequently) can benefit PWLP systems.

In this section, three new resource-oriented task allocation schemes are developed based on the heuristic task mapping approaches described in Section 2.2.3. The new task allocation algorithms aim to not only reduce the number of remote requests to globally shared resources but also take the resource characteristics into account, where each proposed task mapping algorithm can benefit certain candidate resource sharing protocols via reducing the remote blocking of the resources that the protocol is less favourable with. Then, a set of experiments are conducted (1) to investigate the schedulability of systems with the proposed task allocation schemes and each of the candidate locking protocols adopted and (2) to compare the performance of the newly-developed task allocation schemes and the existing task mapping algorithms.

For the flexible multiprocessor resource sharing framework proposed in this thesis (where multiple locking protocols are working together simultaneously), the task mapping algorithms proposed for this FMRS framework should be independent from the resource sharing protocols i.e., algorithms such as BPA should not be adopted in this particular work. That is, each of the task allocation schemes proposed in this thesis should be applicable to systems regardless of the locking protocols adopted (although it may favour one protocol more than another). Such a requirement also provides convenience when adding more candidate locking protocols to the resource control framework in the future.

4.1.1 Resource Contention Fit

The first resource-orientated task allocation scheme proposed in this thesis aims to minimise the blocking from the competitive resources (i.e., the resources that are accessed most frequently) among all the shared resources, named as the Resource Contention Fit (RCF). Unlike the SPA algorithm (which relies on the notion of *task bundles* and the Best-Fit Decreasing (BFD) mapping approach), new mechanisms are proposed in the RCF algorithm to facilitate localising the shared resources in a given system.

In the SPA algorithm, the resource-requesting tasks are placed into a set of task bundles, where tasks that share the same set of resources are bundled together, including the transitive resource sharing (i.e., if τ_1 and τ_2 want r^1 while τ_2 and τ_3 request r^2, these three tasks will be bundled together). However, for a system with many tasks requesting multiple resources (i.e., a high value of κ introduced in Section 3.4), adopting such an approach can lead to extremely heavy task bundles (i.e., with a high utilisation), which need to be split into many pieces so that the whole bundle can be allocated into processors. However, with task bundles with extremely high utilisations, many processors could be required to allocate one single bundle. Therefore, a high remote blocking time can be imposed to tasks in these bundles, and hence, can jeopardise the schedulability of the whole system.

Based on the discussion above, in the RCF algorithm, a task grouping approach is firstly proposed to generated lighter task groups than that of the task bundling method in the SPA algorithm, as described below.

1. Each resource (say r^k) has a total number of requests issued by each task in one release, denoted as $N^k = \sum_{\tau_x \in G(r^k)} N_x^k$.

2. Resources are ordered by N^k in a non-increasing order. If the N^k values are equal, the algorithm then checks the resource utilisations and the index values (i.e., resource id) sequentially to break ties.

3. Tasks are grouped via each shared resource. The grouping starts from the first resource (i.e., the one with the biggest N^k among all shared re-

sources), and all tasks that request this resource will be grouped together as a task group.

4. A task that already belongs to a task group will not be grouped again when examining the subsequent shared resource.

5. The tasks that do not require any resources belong to no task groups and will be allocated independently after all task groups are allocated.

With the above steps, the task group of each shared resource can be identified, where a task group of a resource can be empty if no tasks require that resource or all its requesting tasks are already grouped by other resources.

Compared to the task bundling in SPA, this task grouping approach does not consider the transitive resource sharing so that lighter task groups (i.e., with a lower amount of total task utilisation) could be generated in general (see Section 2.6.1.1 for the task bundling approach in SPA). This is obvious as each task group in the RCF algorithm contains the tasks that require only one resource while a task bundle in the SPA algorithm contains tasks that share the same set of resources. The rationale of starting from the resource with the largest N^k and grouping all tasks that request this resource (no matter whether these tasks request other resources) is to facilitate reducing the blocking from the most competitive resources, as described in detail after the allocating approaches are presented.

The following explains the mechanism of allocating the task groups and independent tasks (i.e., tasks that do not require resources) to a system with M processors under the RCF algorithm.

1. The tasks in each task group are ordered by their utilisations in a non-decreasing order (ties broken by deadlines).

2. Staring from the resource group of the first resource (i.e., the one with the biggest N^k) and P_0, the tasks in each task group are mapped to processors according to the Next-Fit Increasing (NFI) algorithm, where tasks are ordered by utilisation non-decreasingly and are mapped by the NF approach described in Section 2.2.3.

3. Finally, the independent tasks are allocated by the Worst-Fit Decreasing (WFD) algorithm, where ties are broken by task id.

4. If a task cannot be fitted into any processor, the algorithm is finished with no allocation solutions being found.

The above presents the complete approach of allocating tasks in the RCF algorithm. Note that although a fixed number of processors is assumed in the above description, this algorithm can also take an initial number of processors (usually $\lceil U_{tot} \rceil$ processors, where U_{tot} is the total utilisation of all tasks) and adds extra processors later on if a task cannot be allocated to any existing processors.

The intuition of allocating the task groups via the NFI heuristic is that, compared to other heuristic approaches (i.e., the WF, BF and FF algorithms), the NFI algorithm is more likely to allocate tasks in a group into a limited number of processors, as this algorithm always starts from the previously-allocated processor and then checks the next processor if the given task cannot be allocated. Further, with tasks ordered by utilisation non-decreasingly, more tasks in a task group could be allocated to a single processor. In addition, the independent tasks are mapped via the WFD algorithm. Such an approach could require less processors (or could increase the success rate of this algorithm with a fixed number of processors assumed), as the independent task with the biggest utilisation is allocated first so that other tasks (with a smaller utilisation) are more likely to be fitted into the remaining spaces (if any) of the existing processors under the WF heuristic.

With the RCF algorithm adopted, the blocking imposed by the competitive resources can be reduced as the requesting tasks can be allocated to a limited number of processor, or even into a single processor. This algorithm is fully independent from the resource sharing protocols i.e., can be adopted with any locking protocols assumed. In general, this algorithm could benefit all the multiprocessor locking protocols as the blocking from certain shared resources can be reduced. Especially, this algorithm is favourable with PWLP adopted, which performs better than other candidate locking protocols with low remote resource contention (see Section 3.4). In Section 4.1.3, experiments are conducted to investigate the schedulability of systems with this algorithm and each of the candidate locking protocols adopted.

4.1.2 Resource Length Fit

In Section 3.4, the experiments have showed that the length of the critical sections has a significant impact to the schedulability of the candidate locking protocols. As observed, MSRP is better than other candidate locking protocols with short resources, but has a poor schedulability when managing long resources due to its non-preemptive FIFO spinning approach. In addition, MrsP is favourable with long resources but demonstrates the worst schedulability among all the candidate locking protocols if the resources are short. According to this observation, two resource-oriented task allocation algorithms are proposed in this section, which aim to localise shared resources based on the length of critical sections, named as the Resource Length Fit-Long (RLF-L) and the Resource Length Fit-Short (RLF-S) respectively.

Both algorithms aim to minimise the blocking due to certain resources, where the RLF-L aims to reduce the blocking from long resources while the RLF-S attempts to decrease the blocking due to resources with short critical sections. The intuition is that, although these allocation algorithms are independent from resource sharing protocols, the RLF-L could benefit MSRP while the RLF-S should be more favourable with MrsP adopted based on the experiments given in Section 3.4. In these two task mapping algorithms, the notion of *task groups* in Section 4.1.1 is adopted for grouping the resource-requesting tasks. Thus, the number of processors required to allocate a task group could be less than that of a task bundle, and hence, leads to less blocking time. In these two algorithms, resources are ordered by the length of their critical sections, as described below.

- **RLF-L:** Resources are ordered by their length of critical sections (i.e., c^k) in a *non-increasing* order, where ties are broken by resource utilisations and resource id sequentially.
- **RLF-S:** Resources are ordered by their length of critical sections (i.e., c^k) in a *non-decreasing* order, where ties are broken by resource utilisations and resource id sequentially.

The rest of the task grouping and allocating mechanisms in the RLF-L and RLF-S algorithms are identical with that of the RCF algorithm, where the tasks are grouped from the first resource and all the tasks that require this resource are grouped. Then, tasks are mapped into each processor by the same approach proposed in the RCF algorithm, where the task groups are allocated via the NFI algorithm and then, the independent tasks are mapped by the WFD algorithm.

The above presents the approaches of the RLF-L and RLF-S algorithms. With these task allocation schemes adopted, the blocking due to long (or short) resources can be reduced. Similar with the RCF scheme, both algorithms are independent from the locking protocols and could benefit resource sharing in general as the blocking due to certain resources can be reduced. Especially, based on the observation from the experiments in Section 3.4, the RLF-L and RLF-S algorithms could benefit MSRP and MrsP respectively. In Section 4.1.3, experiments are conducted to investigate the performance of the newly-developed resource-oriented task allocation algorithms under each candidate locking protocol.

4.1.3 The Impact of Task Allocation on Multiprocessor Systems with Shared Resources

In this section, a set of experiments are conducted to (1) compare the performance of the newly-developed resource-oriented task allocation schemes with the existing task mapping algorithm; and to (2) compare the schedulability of systems with the new task mapping algorithms under each candidate locking protocol. In total, 8 task allocations are examined in this evaluation, including the traditional task utilisation-based allocation schemes in Section 2.2.3 (the WF, BF, FF, NF algorithms), the SPA algorithm reviewed in Section 2.6.1.1 and the newly-proposed resource-oriented task mapping algorithms in this chapter (i.e., the RCF, RLF-L and RLF-S algorithm).

The experiments are conducted via varying L and A respectively (i.e., the range of critical section length and the frequency of resource accesses). In the following experiments, the schedulability of each candidate locking protocol is investigated with the above task allocation schemes adopted. In addition, the system generation tool described in Section 3.4 is adopted in the experiments and the DMPO algorithm is adopted for assigning task priorities. The test program of the experiments in this section can be accessed via `https://github.com/RTSYork/SchedulabilityTestEvaluation`. Similar to the experiments given in Section 3.4, the statistical significance of the experimental results in this section demonstrates a confidence level of 95%, as calculated in Appendix E.

For the allocation schemes that involve the BF, FF and NF algorithms, a maximum utilisation bound (i.e., U_{max}) is applied to avoid overloading processors. Without this bounding, a processor could be assigned with a high utilisation (although less than 1), which might lead to deadline misses of the tasks in that processor. As described in [25], a task set with a combined utilisation lower than 0.693% is always schedulable on a single processor under the

FPPS scheduling policy and the DMPO algorithm with no shared resources. Based on the discussion above, this bounding is set to $U_{max} = 0.6$ initially (see Figure 3.2, where most systems are unschedulable when $\frac{U_{tot}}{M} > 0.6$) and is raised to $U_{max} = \frac{U_{tot}}{M}$ if $\frac{U_{tot}}{M} > 0.6$ in the following experiments. However, for a task that has an utilisation higher than the given U_{max}, the bounding is ignored for this task to achieve a feasible allocation (but the total utilisation of that processor must not exceed 1). To facilitate experimenting, we enforce that each generated task set can be successfully allocated by each of the examined allocation scheme.

Varying Critical Section Length L

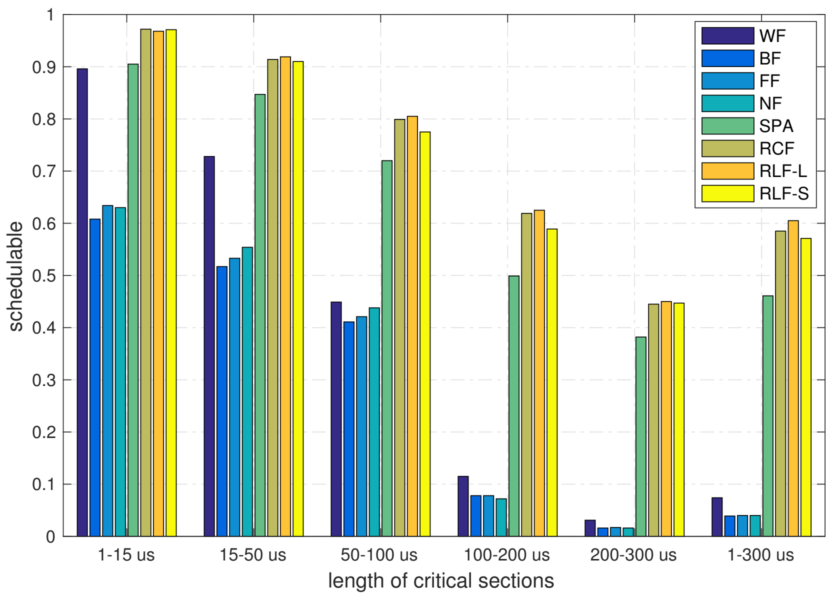


Figure 4.1: Schedulability of MSRP for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

Figure 4.1 illustrates the schedulability of MSRP systems with each of the above allocation schemes adopted via varying L. Firstly, among the traditional task allocation schemes (i.e., the WF, BF, FF, NF), the MSRP system with the WF algorithm adopted demonstrates better schedulability in most cases. The reason is that with the WF algorithm adopted, tasks are always allocated to the processor with the least utilisation. Accordingly, processors are less likely to be overloaded so that a high schedulability can be achieved. However, when $L = [200\mu s, 300\mu s]$, where blocking can be significant due to long critical sections, the efficiency of the WF algorithm is decreased and is similar to other traditional heuristic approaches. In addition, among the commonly adopted heuristic approaches for localising shared resources in the resource-oriented task allocations described in this thesis (e.g., the BF algorithm in SPA and the NF algorithm in our task allocation schemes), the NF algorithm demonstrates better performance in general as a group of task can be allocated to a limited number of processors. Such an observation certifies the decision of adopting the NF algorithm in our resource-oriented task allocation schemes.

Now we focus on the performance of the resource-oriented task mapping algorithms under MSRP, also shown in Figure 4.1. Firstly, we observed that systems with the resource-oriented task allocation schemes adopted demonstrate better schedulability than that of under the traditional allocation algorithms. Among the resource-oriented schemes, the SPA algorithm is outperformed by our newly-proposed algorithms in all cases. In addition, we observed that with MSRP, the RLF-L algorithm can outperform the RLF-S algorithm in general, which again proves that MSRP is more favourable with short critical section length. Another interesting finding is that MSRP systems under the RCF and RLF-S algorithms can demonstrate better schedulability than that of with RLF-L adopted when $L = [1\mu s, 15\mu s]$. Such an observation indicates that neither algorithm can dominate others with MSRP systems. Nonetheless, the experimental results clearly indicate that our newly-proposed task allocation schemes can benefit resource sharing in MSRP systems and can outperform the existing task mapping algorithms in general.

Figures 4.2 and 4.3 give the schedulability of PWLP and MrsP with each of the task allocation schemes adopted by increasing the range of critical section length. As with the above experiment, the WF algorithm has the best performance among the traditional schemes in most cases while the NF scheme is more favourable compared to the BF and FF algorithms with either PWLP or MrsP adopted in general. In addition, the SPA algorithm is outperformed by our new schemes with either protocol adopted in all cases. Surprisingly, in PWLP systems, the RCF algorithm is better than other algorithms in many cases but is outperformed by the RLF-L algorithm under $L = [1\mu s, 300\mu s]$ with an observable difference (see Figure 4.2). The intuition is that the RLF-L algorithm aims to localise long resources, which can cause considerable amount of blocking time. With a wide range of critical section length (where there

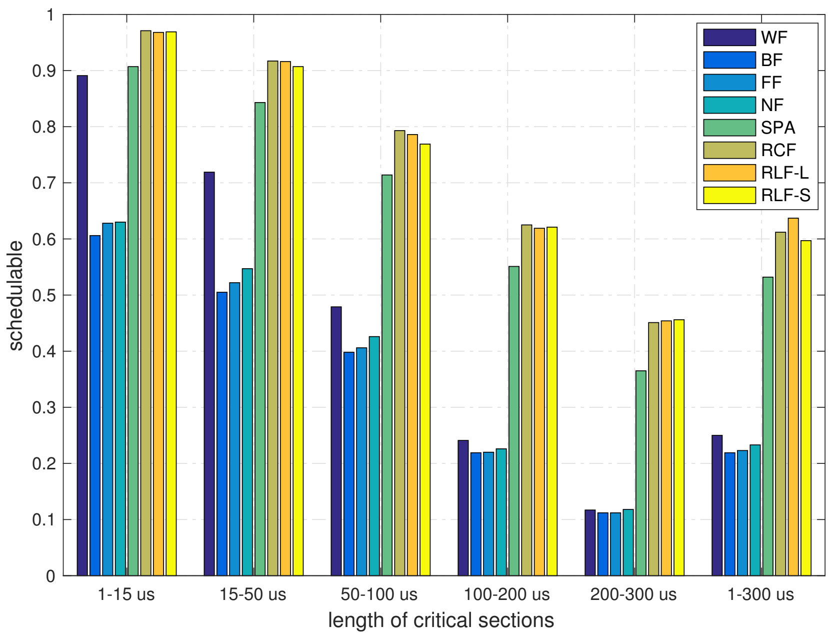


Figure 4.2: Schedulability of PWLP for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

exist both short and long resources), some of the competitive resources could be short resources, which impose less blocking compared to the long resources. Thus, although RCF is more favourable with PWLP systems in general, the RLF-L scheme can provide better performance with $L = [1\mu s, 300\mu s]$ under the above system setting.

A similar observation is obtained in the experiment with MrsP adopted (Figure 4.3), where systems have better schedulability with the RLF-L algorithm than systems with other allocation schemes under $L = [1\mu s, 300\mu s]$ due to the same reason discussed above. Yet, with $L = [100\mu s, 200\mu s]$ and $[200\mu s, 300\mu s]$, where all the resources are assigned with a relatively long critical section, the RLF-S algorithm demonstrates the best performance among the newly-developed algorithms, which supports the hypothesis that the RLF-S algorithm is favourable with MrsP adopted. In addition, another interesting result is observed, where the RCF algorithm demonstrates the best schedulability under $L = [15\mu s, 50\mu s]$ and $[50\mu s, 100\mu s]$. This is because the RCF algorithm allocates the most competitive resources into a limited number of processors so that the number of migration targets of a MrsP task for access-

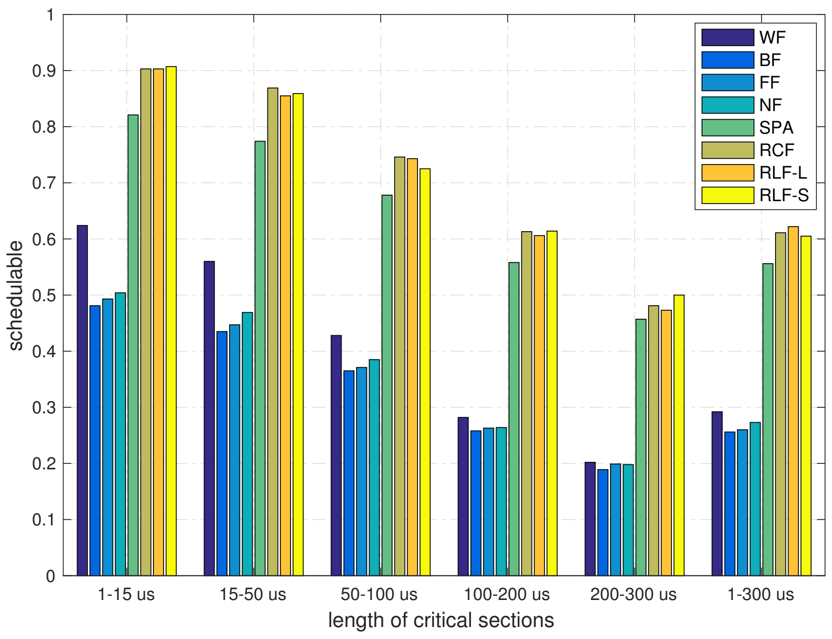


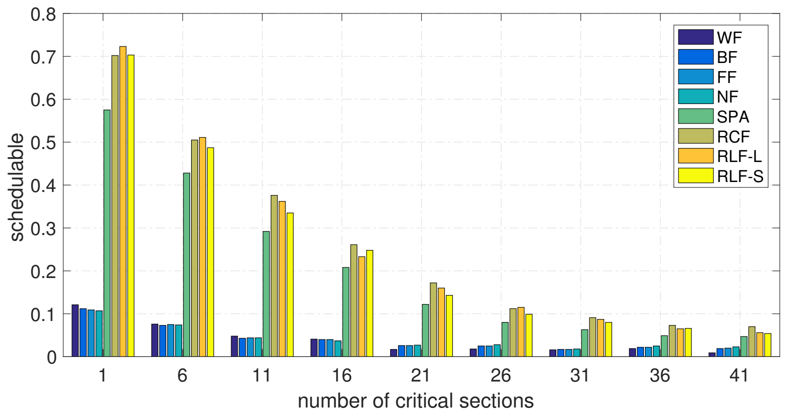
Figure 4.3: Schedulability of MrsP for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

ing such resources can be decreased. Accordingly, the migration cost incurred by these tasks are effectively reduced, and hence, a higher schedulability can be achieved.

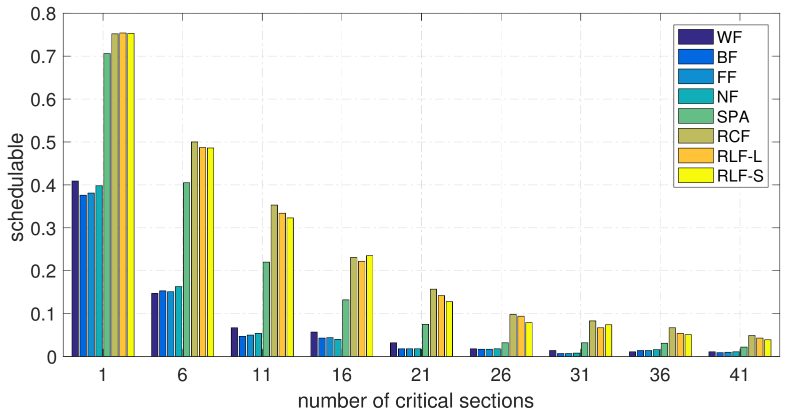
Varying Resource Access Frequency A

Figure 4.4 presents the schedulability of the candidate locking protocols under each task allocation scheme via varying A. As with the experiments above (which are conducted with varied critical section length), the resource-oriented task allocation algorithms perform better than the traditional task mapping algorithms while the SPA algorithm is outperformed by the new task allocation schemes proposed in this thesis under each protocol in all cases.

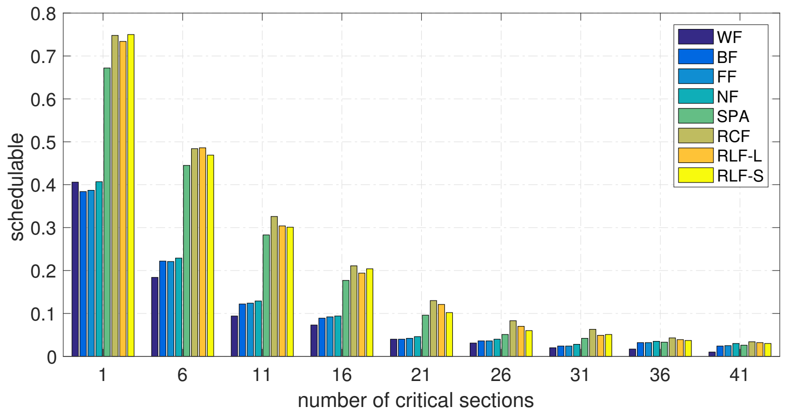
By cross comparing Figures 4.4a, 4.4b and 4.4c, we observed that with the traditional allocation algorithms adopted (e.g., the WF and BF schemes), the candidate locking protocols demonstrate similar schedulability as the experiments conducted in Section 3.4, where MSRP is less favourable with long critical sections ($L = [100\mu s, 200\mu s]$ in this experiment) while MrsP can demonstrate the best schedulability with long resources under each of the traditional



(a) MSRP Systems



(b) PWLP Systems



(c) MrsP Systems

Figure 4.4: Schedulability of Systems for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $L = [100\mu s, 200\mu s]$ and M Shared Resources.

task allocation scheme.

In addition, although PWLP can outperform MSRP under resources with low competitive (i.e., where $A = 1$ and $A = 6$), by further incrementing A, the schedulability of PWLP is decreased significantly and is outperformed by MSRP when $A > 21$ under all examined task allocation schemes in most cases. Compared to PWLP, the schedulability of both MSRP and MrsP under the tested allocation schemes are less affected by increasing the frequency of resource accesses. Further, we observed that the WF and NF algorithms perform better than other traditional schemes in most cases with each candidate locking protocol, which again supports the decision of adopting the NF algorithm in our new resource-oriented algorithms.

Now we focus on the resource-oriented task mapping algorithms by cross comparing their performance under each protocol. With $A = 1$ (i.e., a low resource contention), the RLF-L or RLF-S algorithm performs better than the RCF algorithm, where the RLF-L algorithm provide best schedulability for MSRP and PWLP while the RLF-S algorithm outperforms other task mapping approach with MrsP adopted. However, by further incrementing A (where $A > 1$), the performance of the RCF algorithm can outperform other task mapping schemes in most cases. The reason is that, with $L = [100\mu s, 200\mu s]$ (where the minimum critical section length is 100 μs), increasing the number of resource access can impose considerable amount of blocking to the system, where the blocking due to a resource r^1 with $c^1 = 100\mu s$ and a high contention can easily exceeds the blocking from a resource with a length of $200\mu s$ and a low contention. Therefore, reducing the blocking time (i.e., adopting the RCF algorithm) due to competitive resources under the given system setting can lead to better performance in general than other task mapping algorithms.

Interestingly, we observed that compared to the traditional task allocation schemes, MSRP can demonstrate better tolerance with long critical sections while PWLP can be less affected with a high resource-accessing frequency under the resource-oriented task mapping approaches, especially with the new task allocation schemes developed in this thesis. For instance, MSRP provides the best schedulability while MrsP is the worst under long resources with strong contention in general (see cases where $A > 11$ in Figure 4.4 for all protocols). In addition, by cross comparing the schedulability of the evaluated protocols in Figures 4.1, 4.2 and 4.3 with $L = [1\mu s, 15\mu s]$, we also observed that MrsP demonstrates the largest schedulability boost by replacing the traditional task allocation schemes to the resource-oriented ones. Such observations show that the resource-oriented task allocation schemes can facilitate multiprocessor resource sharing under the candidate locking protocols in general, and can benefit each candidate resource sharing protocol with resources that it is less favourable with.

Summary and Discussion

Summarising the above, the experimental results presented in this section have showed that resource-oriented task allocation schemes developed in this section can benefit resource sharing in multiprocessor systems with the candidate locking protocols adopted. In general, systems under the newly-developed schemes can demonstrate better performance than that with the existing task allocation schemes adopted. As revealed by the experiments, even though each newly-developed allocation scheme is favourable with certain candidate protocols, neither scheme can dominate others under all situation (even for a single locking protocol). In addition, compared to our task mapping algorithms, the SPA algorithm is less favourable under each experiment presented in this section.

However, it is insufficient to claim that the SPA algorithm is dominated by the new ones. Consider, in the case where each task bundle can be allocated into a processor, SPA can outperform the new algorithms as our algorithms mainly focus on minimising the blocking due to certain resources rather than localising all the shared resources. Further, although the traditional task allocation schemes are outperformed by the resource-oriented ones in general, there can be the situation where a traditional mapping algorithm can lead to a schedulable system while the resource-oriented schemes cannot even provide a feasible task allocation. For instance, for a system where all tasks require the same resource (which is also the most competitive resource among all the shared resources), the RCF algorithms will behave exactly the same as the NFI algorithm, which allocates the tasks with a small utilisation prior to the heavy ones (i.e., tasks with a high utilisation) via the NF approach. However, compared to the WFD algorithm, mapping the heavy tasks at the end of the allocation is more likely to cause an allocation failure, as tasks with a high utilisation are more difficult to fit into the remaining space of processors (after all other tasks are allocated) than the ones with a low utilisation. Therefore, with such a task set, the WFD algorithm could return a feasible task allocation and lead to a schedulable system while the RFC algorithm cannot even allocate the given tasks successfully. In addition, such situations can also happen with the RLF-L and RLF-S algorithms adopted, assuming that resource (the one that is requested by all tasks) has the longest (or shortest) critical section among all the shared resources in the above example.

Accordingly, as there exists no task allocation scheme that can dominate other task mapping approaches examined in this evaluation (with the candidate locking protocols assumed), all the task allocation schemes studied in this section will be considered as the candidate task allocation schemes for the proposed resource control framework, including four traditional task utilisation-based allocation schemes and four resource-oriented task mapping approaches, as described in Chapter 5 in detail.

4.2 Priority Ordering for Fully-Partitioned Systems with Shared Resources

In addition to the resource sharing protocols and the task allocation schemes, the priority ordering algorithms specify the execution order of tasks in each processor, and hence, can also have a huge impact to the schedulability of multiprocessor systems with shared resources. As reviewed in Section 2.1.3, the DMPO is well-practised and can provide optimal priority ordering solutions for sporadic tasks with constrained deadlines while the OPA and RPA algorithms are search-based and can provide optimal priority ordering solutions for wider application semantics (e.g., tasks with release offsets and arbitrary deadlines).

However, as described in [38], the optimality of a given priority ordering algorithm holds only for a given schedulability analysis. For the DMPO algorithm, it has been proved to be optimal with the RTA equations in Section 2.1.4 for uniprocessor systems without the presence of shared resources [38]. In addition, with the schedulability test of PCP and SRP (i.e., the blocking analysis of PCP and SRP based on the RTA equations under FPPS systems) assumed, it has been proved in [16] that DMPO remains optimal for uniprocessor systems with shared resources managed by either PCP or SRP. However, with the schedulability tests for multiprocessor locking protocols assumed (e.g., the original analysis of MSRP in Section 2.5.2 and the new MSRP test developed in Section 3.2.1), whether the DMPO algorithm remains optimal has not been studied yet. As for the OPA and RPA algorithms, whether these search-based priority ordering algorithms are compatible with our new schedulability tests also requires investigation.

In this section, the optimality of the DMPO algorithm for fully-partitioned systems with shared resources is investigated under both the original and the new schedulability tests of the candidate multiprocessor resource sharing protocols determined in Chapter 3. In addition, we demonstrate that the existing search-based priority ordering algorithms are not compatible with the new schedulability tests developed in Section 3.2. Then, a new search-based priority ordering algorithm is proposed that is fully compatible with the new schedulability analysis. Finally, the schedulability of systems with the new search-based priority ordering algorithm is investigated and is compared with the existing compatible priority ordering algorithms (e.g., DMPO) under the new analysis. The material presented in this section satisfies the success criteria SC-4 in Section 1.4.

4.2.1 The Optimality of DMPO in Multiprocessor Systems with Shared Resources

This section investigates the optimality of DMPO for fully-partitioned systems with shared resources managed by the candidate locking protocols. As with the proof given in [38], the standard technique for proving the optimality of a priority ordering algorithm P is adopted in this section, where *the algorithm P is said to be optimal with a given schedulability test S if any task sets that are deemed to be schedulable with some priority assignment policy W under test S are also schedulable with algorithm P and test S adopted.* The proof is conducted by induction, where a schedulable task set with priority ordering W is transformed to the priority ordering of P while guaranteeing that each task remains schedulable during the transformation. The following gives the base case and inductive step [38] for proving the optimality of the priority ordering algorithm P on a fully-partitioned system with M processors and n tasks.

Base Case: The priority ordering algorithm W is assumed to be schedulable for a given task set τ (where $|\tau|= n$) with M processors under schedulability test S, where W^x denotes the schedulable priority order for the task set τ.

Inductive step: In the priority ordering W^x, a pair of adjacent tasks are chosen with their priorities swapped to form a new priority ordering, denoted as W^{x-1} (see Figure 4.5). Then, proof is presented to demonstrate that no tasks have missed their deadlines under test S due to this priority swapping. For a task set with n tasks, at most $x = n(n+1)/2$ priority swapping (i.e., as with the OPA algorithm) are required to transfer the priority ordering from W^x to P (i.e., $W^1 = P$). If no tasks have missed their deadlines under S during the entire priority reordering process, there will be no task sets that are schedulable with W but are not schedulable with P adopted, and hence, proves the optimality of the priority ordering algorithm P under S with the task model of the given task set τ.

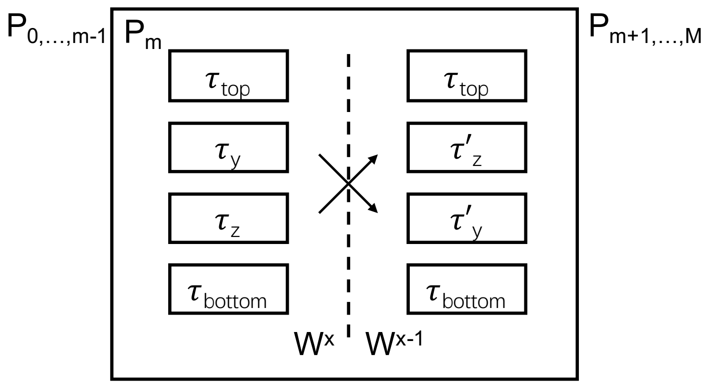


Figure 4.5: A Priority Swap.

The system given in Figure 4.5 will be adopted to conduct the proof below. The system contains M processors and each processor is assigned with a set of tasks. A priority transfer is performed between τ_y and τ_z, where τ_{top} and τ_{bottom} denote a set of local higher and lower priority tasks respectively. To differentiate the response times of τ_y and τ_z under priority ordering W^x and W^{x-1}, R'_y and R'_z are used to denote the response times of these tasks after the priority swapping (i.e., the priority ordering W^{x-1}).

In addition, the system contains a set of shared resources that are managed by a given resource sharing protocol with a schedulability test S, where functions $F(\tau_x)$ and $G(r^k)$ given in Section 2.5.9 are adopted here to describe the usage of shared resources, where $F(\tau_x)$ gives a set of resources required by τ_x while $G(r^k)$ returns a set of tasks that access r^k. That is, our proof is conducted with a generalised resource usage of the above system (i.e., without any exact resource usage assumed).

Note that the priority ordering policy W does not comply with the DMPO algorithm. With DMPO adopted, increasing priorities are assigned in the reverse ordering of deadlines, which indicates that for any given pair of tasks (say τ_1 and τ_2) in the given system, $D_1 < D_2$ so that $Pri(\tau_1) > Pri(\tau_2)$. However, with the priority ordering policy P adopted, there exists at least one pair of tasks (say τ_1 and τ_2) that $D_1 < D_2$ and $Pri(\tau_1) < Pri(\tau_2)$. In this proof, we assume that $D_y > D_z$ and $Pri(\tau_y) > Pri(\tau_z)$ under the priority ordering W^x. In addition, no assumptions are made between the relationship of the priority and deadline for any other tasks in the system.

4.2.1.1 Investigating the Optimality of DMPO with the Original MSRP and MrsP Analysis

As MSRP and MrsP are developed with original schedulability tests provided [27, 47], we investigate the optimality of the DMPO algorithm on fully-partitioned systems with either MSRP or MrsP adopted under their original schedulability tests given in Sections 2.5.2 and 2.5.9, as given in the theorem below.

Theorem 4. *The Deadline Monotonic Priority Ordering is optimal in fully partitioned multiprocessors with shared resources under the original schedulability test of either MSRP or MrsP.*

Proof. We prove that DMPO is optimal under the original analysis of MSRP and MrsP individually.

(1) Under MSRP:

The complete original analysis of MSRP is given in Section 2.5.2. With MSRP's original analysis assumed, the response time of a given task is determined by the independent task properties only and the cost for accessing a resource r^k is always $|map(G(r^k))| \times c^k$, as shown in Equation (2.8).

We firstly prove that $R'_z \le D_z$ after the priority swap. Under W^x, τ_z incurs the interference of τ_y, including the indirect spin delay due to τ_y's resource accessing, which is $\left\lceil \frac{R_z}{T_y} \right\rceil \widehat{C_y}$ and is equal to

$$\left\lceil \frac{R_z}{T_y} \right\rceil (C_y + \sum_{r^k \in F(\tau_y)} N_y^k \times |map(G(r^k))| \times c^k) \tag{4.1}$$

according to Equations (2.7) and (2.8). After the priority swap (i.e., in W^{x-1}), τ_z will not incur such interference as it now has a higher priority. Thus, the interference of τ_z is reduced after the priority swap, where it can only be preempted by τ_{top} under W^{x-1}.

However, after the priority swap, τ_z could incur an increased arrival blocking. As described in Section 2.5.2, a resource r^k can cause task τ_x to incur arrival blocking if r^k is requested by τ_x's local lower priority tasks and r^k is either a global resource or a local resource with a ceiling priority at least equals to $Pri(\tau_x)$. According to Equation (2.9), the resources that can cause τ_z to incur arrival blocking (i.e., $F^A(\tau_z)$) under W^x can be identified as

$$F^A(\tau_z) \triangleq \{r^k | r^k \in F(\tau_{bottom}) \wedge \big(r^k \text{ is global} \vee Pri(r^k) \geq Pri(\tau_z)\big)\} \tag{4.2}$$

where $r^k \in F(\tau_{bottom})$ indicates that r^k is requested by τ_z's local lower priority tasks under W^x. In addition, the condition $Pri(r^k) \geq Pri(\tau_z)$ (for the case where r^k is a local resource) can be further specified as $r^k \in F(\tau_{top}) \cup F(\tau_y) \cup F(\tau_z)$. Thus, the resources that can cause τ_z to incur arrival blocking under W^x in the system given in Figure 4.5 are

$$\{r^k | r^k \in F(\tau_{bottom}) \wedge \big(r^k \text{ is global} \vee r^k \in F(\tau_{top}) \cup F(\tau_y) \cup F(\tau_z)\big)\} \tag{4.3}$$

Accordingly, in the priority ordering W^{x-1}, the resources that can cause τ_z to incur arrival blocking can also be identified by the following function.

$$\{r^k | r^k \in F(\tau_y) \cup F(\tau_{bottom}) \wedge \big(r^k \text{ is global} \vee r^k \in F(\tau_{top}) \cup F(\tau_z)\big)\} \tag{4.4}$$

The calculations in Equations (4.3) and (4.4) demonstrate that the resources that can cause τ_z to incur arrival blocking can be different under these two priority orderings. For global resources, it is clear that the number of global resources that can cause arrival blocking to τ_z after the swapping can be increased, where such resources are $r^k \in F(\tau_{bottom})$ in W^x and are $r^k \in F(\tau_y) \cup F(\tau_{bottom})$ under W^{x-1}, assuming r^k is a global resource.

In addition, as shown in Equations (4.3) and (4.4), the local resources that can block τ_z upon its arrival under W^x can be identified by

$$r^k \in F(\tau_{bottom}) \wedge r^k \in F(\tau_{top}) \cup F(\tau_y) \cup F(\tau_z) \wedge r^k \text{ is local} \tag{4.5}$$

while such resources under W^{x-1} are

$$r^k \in F(\tau_y) \cup F(\tau_{bottom}) \wedge r^k \in F(\tau_{top}) \cup F(\tau_z) \wedge r^k \text{ is local} \tag{4.6}$$

By comparing the Equations (4.5) and (4.6), the local resources that are only required by τ_y and τ_{bottom} cannot cause τ_z to incur arrival blocking under

W^{x-1}. However, the local resources that are requested by τ_y and tasks with a priority equal to or higher than $Pri(\tau_z)$ (e.g., $G(r^k) = \{\tau_{top}, \tau_y\}$, $G(r^k) = \{\tau_y, \tau_z\}$ or $G(r^k) = \{\tau_{top}, \tau_y, \tau_z\}$) can now block τ_z upon its arrival under W^{x-1} while cannot cause such blocking to τ_z under W^x.

Based on the above discussion, in the worst case, τ_z's arrival blocking can increase after the priority swap. Further, if the arrival blocking of τ_z does increased in W^{x-1}, such a resource (either global or local) that causes the blocking must be a resource that is requested by τ_y. Therefore, according to Equation (2.9), in the worst case (i.e., the arrival blocking of τ_z under W^{x-1} is increased), τ_z's arrival blocking can be calculated as

$$max\Big\{|map(G(r^k))|\times c^k | r^k \in F(\tau_y) \wedge \Big(r^k \text{ is global} \vee G(r^k) \neq \{\tau_y, \tau_{bottom}\}\Big)\Big\} \tag{4.7}$$

where $\Big(r^k \text{ is global} \vee G(r^k) \neq \{\tau_y, \tau_{bottom}\}\Big)$ indicates that r^k is either a global resource or a local resource that is not shared only by τ_y and τ_{bottom}. In addition, $|map(G(r^k))|= 1$ if r^k is a local resource.

Now, recall the decreased indirect spin delay of τ_z after the priority swap in Equation (4.1) (which is $\left\lceil \frac{R_z}{T_y} \right\rceil \cdot (\sum_{r^k \in F(\tau_y)} N_y^k \cdot |map(G(r^k))| \cdot c^k)$), it is clear that the potential increase of the arrival blocking of τ_z after the swap is at most equal to the decrease of its indirect spin delay, where the arrival blocking can occur only once while τ_y could access that resource multiple times during each release. In addition, as τ_z will not incur the interference of τ_y's pure computation time (i.e., C_y) after the priority swap, the increase of the arrival blocking of τ_z is always less than the decrease of its interference after the priority swap i.e., $R'_z < R_z \leq D_z$.

On the other side, if τ_z's arrival blocking is not increased after the priority swap, it is still schedulable as it incurs less interference while has the same amount of direct spin delay and a non-increased arrival blocking. Therefore, we can conclude that $R'_z < R_z \leq D_z$ (i.e., τ_z is schedulable under W^{x-1}) in the system given in Figure 4.5 regardless of the exact resource usage.

Now we prove that $R'_y \leq D_y$. This can be achieved by examining R_z and R'_y. By applying Equation (2.6), the response time of τ_z under W^x and τ_y under W^{x-1} is given below.

$$R_z = \widehat{C_z} + B_z + \left\lceil \frac{R_z}{T_y} \right\rceil \widehat{C_y} + \sum_{\tau_h \in \tau_{top}} \left\lceil \frac{R_z}{T_h} \right\rceil \widehat{C_h} \tag{4.8}$$

$$R'_y = \widehat{C_y} + B_y + \left\lceil \frac{R'_y}{T_z} \right\rceil \widehat{C_z} + \sum_{\tau_h \in \tau_{top}} \left\lceil \frac{R'_y}{T_h} \right\rceil \widehat{C_h} \tag{4.9}$$

where $\widehat{C_y}$, $\widehat{C_z}$ and $\widehat{C_h}$ are constant values under the original analysis of MSRP (for τ_x, $\widehat{C_x} = C_x + \sum_{r^k \in F(\tau_x)} N_x^k \times |map(G^k)| \times c^k$).

Firstly, according to Equation (2.9), under W^{x-1}, the resources that can cause τ_y to incur arrival blocking can be identified by the following equation.

$$F^A(\tau_y) \triangleq \{r^k | r^k \in F(\tau_{bottom}) \wedge \left(r^k \text{ is global} \vee r^k \in F(\tau_{top}) \cup F(\tau_z) \cup F(\tau_y)\right)\} \tag{4.10}$$

Compared to the resources that cause τ_z to incur arrival blocking under W^x (see Equation (4.5)), it is clear that τ_z in W^x and τ_y in W^{x-1} can be blocked upon their arrival by the same set of resources, and hence, leads to the same amount of arrival blocking (i.e., B_z in W^x equals to B_y in W^{x-1}) by Equation (2.9), denote as B below.

To facilitate the comparison, we firstly ignore the interference from the tasks in τ_{top} and will consider this interference later on. Such an approach is valid because the amount of interference from high priority tasks increases monotonically with the response time, where $R_1 \geq R_2$ then $\left\lceil \frac{R_1}{T_x} \right\rceil \widehat{C_x} \geq \left\lceil \frac{R_2}{T_x} \right\rceil \widehat{C_x}$ for τ_x, which further proves that $R_1 \geq R_2$. With the interference due to the tasks in τ_{top} ignored, the following calculations are obtained:

$$\begin{aligned} R_z &= \widehat{C_z} + B + \left\lceil \frac{R_z}{T_y} \right\rceil \widehat{C_y} \\ &= \widehat{C_z} + B + N_p \times \widehat{C_y} \end{aligned} \tag{4.11}$$

For τ_z in W^x, as no assumption can be made between R_z and T_y so that τ_y could preempt τ_z more than once, where N_p denotes the number of preemptions τ_z can incur from τ_y under W^x and $N_p = \left\lceil \frac{R_z}{T_y} \right\rceil \geq 1$.

Now we calculate R'_y with an initial value of $R'_y = \widehat{C_y} + B$ by iterative calculations, where

$$\begin{aligned} R'_y &= \widehat{C_y} + B + \left\lceil \frac{R'_y}{T_z} \right\rceil \widehat{C_z} \\ &= \widehat{C_y} + B + \left\lceil \frac{\widehat{C_y} + B}{T_z} \right\rceil \times \widehat{C_z} \\ &= \widehat{C_y} + B + 1 \times \widehat{C_z} \end{aligned} \tag{4.12}$$

Firstly, recall the value of R_z computed in Equation (4.11), it is clear that $\widehat{C_y} + B < R_z \leq D_z \leq T_z$ so that $\left\lceil \frac{\widehat{C_y}+B}{T_z} \right\rceil = 1$. With further iterations, R'_y is at most equal to R_z (i.e., $R'_y \leq R_z$), as τ_y can preempt τ_z at least once under priority ordering W^x. Thus, the calculation of R'_y is finished with a fixed value of $\widehat{C_y} + B + \widehat{C_z}$, assuming no interference from tasks in τ_{top} is imposed. Thus,

$$\begin{aligned} R_z - R'_y =& \widehat{C_z} + B + N_p \times \widehat{C_y} - \widehat{C_y} - B - \widehat{C_z} \\ =& (N_p - 1) \times \widehat{C_y} \geq 0 \end{aligned} \tag{4.13}$$

From the calculation, it is clear that $R'_y \leq R_z$ with the the interference from τ_{top} ignored. Now we consider such interference incurred by τ_z in W^x and τ_y in W^{x-1}. As $R'_y \leq R_z$ in above calculations, for each τ_h in τ_{top}, $\left\lceil \frac{R'_y}{T_h} \right\rceil \widehat{C_h}$ is at most equal to $\left\lceil \frac{R_z}{T_h} \right\rceil \widehat{C_h}$, which further proves that $R'_y \leq R_z$ with the interference of tasks in τ_{top} accounted for. Therefore, $\left\lceil \frac{R'_y}{T_z} \right\rceil$ will always be 1 as $R'_y \leq R_z \leq T_z$. Accordingly, τ_y is schedulable after the priority swap as $R'_y \leq R_z \leq D_z < D_y$ with any resource usage assmed.

The above proof is conducted by the same strategy of the proof for DMPO in the uniprocessor case given in [38]. In this proof, we provide evidence that swapping the priorities of two adjacent tasks that are schedulable under priority W^x can also be schedulable (i.e., in W^{x-1}) under the original MSRP analysis. By swapping all the adjacent tasks with the incorrect priority order in each processor in Figure 4.5 according to DMPO, a schedulable system with the DMPO algorithm can be obtained under the original MSRP analysis.

(2) Under MrsP:

Now we prove that DMPO is also optimal under the original analysis of MrsP described in Section 2.5.9, which is similar with the analysis of MSRP in Section 2.5.2. The only difference is that under MrsP, the resources that can cause tasks to incur arrival blocking is determined by the ceiling priority of shared resources, where

$$F^A(\tau_i) \triangleq \{r^k | N_{ll}^k > 0 \wedge Pri(r^k, P(\tau_i)) \geq Pri(\tau_i)\} \tag{4.14}$$

according to Equation (2.11). However, such a difference in the analysis will not undermine the optimality of DMPO under the MrsP's original analysis. According to the above equation, the set of resources that can cause arrival blocking to τ_y and τ_z under both priority orderings with MrsP adopted can be identified, where in the priority ordering W^x

$$F^A(\tau_y) \triangleq \{r^k | r^k \in F(\tau_{bottom}) \cup F(\tau_z) \wedge r^k \in F(\tau_{top}) \cup F(\tau_y)\} \tag{4.15}$$

$$F^A(\tau_z) \triangleq \{r^k | r^k \in F(\tau_{bottom}) \wedge r^k \in F(\tau_{top}) \cup F(\tau_y) \cup F(\tau_z)\} \tag{4.16}$$

while under priority ordering W^{x-1}

$$F^A(\tau_y) \triangleq \{r^k | r^k \in F(\tau_{bottom}) \wedge r^k \in F(\tau_{top}) \cup F(\tau_y) \cup F(\tau_z)\} \tag{4.17}$$

$$F^A(\tau_z) \triangleq \{r^k | r^k \in F(\tau_{bottom}) \cup F(\tau_y) \wedge r^k \in F(\tau_{top}) \cup F(\tau_z)\} \tag{4.18}$$

Similar with the MSRP case, τ_z could incur an increased arrival blocking due to a resource that is requested by τ_y (i.e., in $F(\tau_y)$) after the priority swap (see Equations (4.16) and (4.18)). Therefore, as proved before, the increase of τ_z's arrival blocking will always be less than the decrease of its interference in W^{x-1}. Thus, τ_z remains schedulable after the priority swap i.e., $R'_z < R_z \leq D_z$ under the original schedulability test of MrsP.

In addition, as shown by Equations (4.16) and (4.17), the set of resources that can cause τ_z in W^x and τ_y in W^{x-1} to incur arrival blocking under MrsP are identical. Thus, the calculations of R_z and R'_y under MrsP's original analysis are identical with the MSRP case. Therefore, it is also clear that $R'_y \leq R_z \leq D_z < D_y$ after the priority swap. Accordingly, the DMPO algorithm is optimal under the original analysis of MrsP in fully-partitioned systems.

This concludes the proof of the optimality of the DMPO algorithm under the original schedulability tests of either MSRP or MrsP in fully-partitioned FPPS systems. ■

4.2.1.2 Investigating the Optimality of DMPO with the New MSRP, PWLP and MrsP Analysis

Now we investigate the optimality of DMPO under the new schedulability tests of the candidate locking protocols developed in Section 3.2. From the viewpoint of analytical expression, a major difference between the original and new schedulability tests is that in the new tests, the response time of a given task depends potentially on the response times of all the tasks in the system. In this work, we prove that the DMPO algorithm is not optimal with new schedulability tests adopted by providing a counterexample, as described in the following theorem and proof.

Theorem 5. *The Deadline Monotonic Priority Ordering is* ***not*** *optimal in fully partitioned multiprocessors with shared resources under the new schedulability tests of MSRP, PWLP and MrsP developed in Chapter 3.*

Proof. It is sufficient to prove the above theorem if DMPO is not optimal under the new schedulability test of MSRP (see Section 3.2.1), as the new analysis of other candidate locking protocols are developed based on this schedulability test. To conduct the proof, a three-processor system is presented with a specific resource usage, where a priority swap is occurred between τ_2 and τ_3 on P_1, as given in Figure 4.6. In this example, $Pri(\tau_3) > Pri(\tau_2) > Pri(\tau_1)$ under priority ordering W^x while $Pri(\tau_2) > Pri(\tau_3) > Pri(\tau_1)$ in W^{x-1}. Table 4.1 gives the task property and resource usage in processor P_1 of the system. In addition, there exist sufficient requests to r^1 and r^2 from both P_0 and P_2 that can block each resource access issued from P_1, which implies that the cost of accessing a resource r^k from P_1 is always $3 \cdot c^k$.

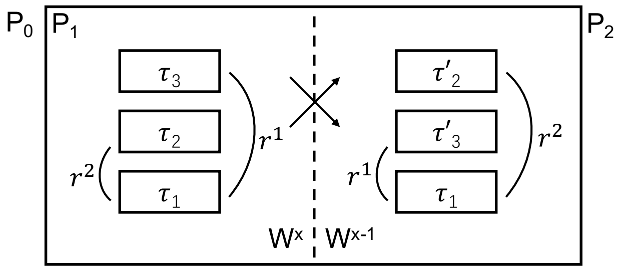


Figure 4.6: A Priority Swap with Shared Resources.

Table 4.1: Task Property and Resource Usage in P_1 of the System in Figure 4.6

Task (τ_x)	C_x	T_x	D_x
τ_3	1	27	27
τ_2	1	17	17
Resource (r^k)	c^k	$G(r^k)$	N_x^k
r^1	1	$\{\tau_1, \tau_3\}$	$N_1^1 = 1,\ N_3^1 = 1$
r^2	2	$\{\tau_1, \tau_2\}$	$N_1^2 = 1,\ N_2^2 = 1$

In this proof, we focus on τ_2 and τ_3 and demonstrate that τ_3 can miss its deadline due to the priority swap in the system given in Figure 4.6, which provide direct evidence that supports the theorem. As shown in Figure 4.6, $F(\tau_2) \triangleq \{r^2\}$ and $F(\tau_3) \triangleq \{r^1\}$. In addition, with MSRP adopted, τ_2 and τ_3 under both priority orderings can incur arrival blocking from the same set of resources (i.e., $F^A(\tau_2) = F^A(\tau_3) \triangleq \{r^1, r^2\}$).

We first give the response time of τ_3 and τ_2 under priority ordering W^x

based on Equation (3.1), where

$$\begin{aligned}
R_3 =& C_3 + E_3 + B_3 \\
=& C_3 + e_3^1(R_3, 0) + max\{|\alpha_3^1|\cdot c^1, |\alpha_3^2|\cdot c^2\} \\
=& 1 + 3 \times 1 + 3 \times 2 \\
=& 10 \\
R_2 =& C_2 + E_2 + B_2 + \left\lceil \frac{R_2}{T_3} \right\rceil \cdot C_3 + I_{2,3} \\
=& C_2 + e_2^2(R_2, 0) + max\{|\alpha_2^1|\cdot c^1, |\alpha_2^2|\cdot c^2\} + \left\lceil \frac{R_2}{T_3} \right\rceil \cdot C_3 + e_3^1(R_2, R_3) \\
=& 1 + 3 \times 2 + 3 \times 2 + \left\lceil \frac{R_2}{27} \right\rceil \cdot 1 + \left\lceil \frac{R_2 + 10}{27} \right\rceil \cdot (3 \times 1) \\
=& 17
\end{aligned}$$

As the cost for accessing a resource r^k from P_1 always $3 \times c^k$, $e_2^2(R_2, 0) = 3 \times c^1$ and $e_3^1(R_2, R_3) = \left\lceil \frac{R_2+R_3}{T_3} \right\rceil 3 \times c^2$ while $|\alpha_3^1| = |\alpha_3^2| = |\alpha_2^1| = |\alpha_2^2| = 3$ (see Equations (3.2), (3.3) and (3.9) in Section 3.2.1). From the above calculations, both τ_3 and τ_2 can meet their deadlines under priority ordering W^x, where $R_3 = 10$ and $R_2 = 17$ while $D_3 = 27$ and $D_2 = 17$.

We now present the calculations of the response times of τ_2 and τ_3 under priority ordering W^{x-1}, where

$$\begin{aligned}
R_2' =& C_2 + E_2 + B_2 \\
=& C_2 + e_2^2(R_2', 0) + max\{|\alpha_2^1|\cdot c^1, |\alpha_2^2|\cdot c^2\} \\
=& 1 + 3 \times 2 + 3 \times 2 \\
=& 13 \\
R_3' =& C_3 + E_3 + B_3 + \left\lceil \frac{R_3'}{T_2} \right\rceil \cdot C_2 + I_{3,2} \\
=& C_3 + e_3^1(R_3', 0) + max\{|\alpha_3^1|\cdot c^1, |\alpha_3^2|\cdot c^2\} + \left\lceil \frac{R_3'}{T_2} \right\rceil \cdot C_2 + e_2^2(R_3', R_2') \\
=& 1 + 3 \times 1 + 3 \times 2 + \left\lceil \frac{R_3'}{17} \right\rceil \cdot 1 + \left\lceil \frac{R_3' + 13}{17} \right\rceil \cdot (3 \times 2) \\
=& 30
\end{aligned}$$

As shown by the above calculations, $R_3' = 30$ under W^{x-1} but $D_3 = 27$. Therefore, τ_3 has missed its deadline after the priority swap, and hence, breaks the optimality of the DMPO algorithm under new MSRP test.

Similarly, as the new schedulability tests of other candidate locking protocols are developed based on the new MSRP test and have the same approach for calculating response times (i.e., the back-back hit computing mechanism), the DMPO algorithm is not optimal under these schedulability tests as well due to the proof given above. Under the new schedulability tests of either PWLP or MrsP (see Sections 3.2.2 and 3.2.3), a high priority task that experienced the priority swap (has its priority swapped with a low priority task) also incurs the additional blocking from either the cancellation mechanism or the migration-based helping mechanism due to the increased number of preemptions incurred after the swap, and hence, results in a further increased response time.

Summarising the above, we conclude that the DMPO algorithm is **not** optimal on fully partitioned FPPS systems under the new schedulability tests of MSRP, PWLP and MrsP developed in Chapter 3, including the analysis framework proposed in Section 3.3. ■

In this section, we have investigated the optimality of DMPO under both the original and newly-developed schedulability tests of the candidate locking protocols in the proposed multiprocessor resource sharing framework. As given by Theorem 4, the DMPO algorithm remains to be optimal under the original analysis of either MSRP or MrsP. However, according to the theorem 5, the optimality of DMPO is undermined under the new schedulability tests, which will be adopted in the FMRS framework. Section 4.2.4 provides experimental evidence that again demonstrates that there exist systems that are not schedulable with DMPO adopted but are feasible under other priority ordering algorithms.

4.2.2 The Compatibility of OPA and RPA with New Schedulability Tests

For the search-based algorithms reviewed in Section 2.1.3 (i.e., OPA and RPA, where RPA is developed based on the OPA algorithm), their optimality holds as long as the given schedulability test is compatible (i.e., meets the three conditions described in Section 2.1.3.3). This is because such algorithms search through all the possible priority ordering solutions via $n(n+1)/2$ calculations for each processor and can guarantee to deliver a schedulable priority ordering solution, assuming there exists one. Thus, it is clear that the OPA and RPA are optimal with the original schedulability tests for the candidate locking protocols (e.g., the analysis for MSRP and MrsP in Sections 2.5.2 and 2.5.9), which are compatible with OPA and RPA as the response time of each task under the original tests depends only on the independent task properties (e.g., C_x and T_x of τ_x).

Unfortunately, these search-based priority algorithms cannot be applied to our new schedulability tests proposed in Section 3.2. The major reason is that, with the newly-proposed schedulability tests, the response time of each task depends potentially on the response times of all other tasks in the system. Recall the new analysis of MSRP in Section 3.2.1, where the function $N_x^k(l, \mu) = \left\lceil \frac{l+\mu}{T_x} \right\rceil \cdot N_x^k$ is introduced to account for the back-to-back hits from local higher priority tasks and remote tasks during the release of the task that is currently being studied (say τ_i). The jitter parameter μ in this function is either 0 (for τ_i itself) or R_x (for a local higher priority task or a remote task). With the back-to-back hits calculation mechanism, the response time of tasks in a system must be calculated iteratively and alternatively until the fixed response times of all tasks are obtained, assuming the system is schedulable. Therefore, such a mechanism violates the conditions of adopting the OPA and RTA algorithms, where the response time of a given task must depend only on the independent task properties.

In addition, to apply our new schedulability tests, each task in the system must be explicitly assigned with a priority. This is because our analysis provides a fine-grained analysing approach, where the indirect spin delay of a given task is calculated via examining the blocking time incurred by each local higher priority task (starting from the highest priority task) during the release of the currently-studied task. Therefore, as the new schedulability analysis is assumed in the proposed resource control framework, the OPA and RPA algorithms cannot be directly applied in this work.

Nonetheless, with the jitter parameter μ replaced by an independent task property when $\mu \neq 0$ (such as D_x) and with each task assigned with an initial priority (which can be achieved via the static priority ordering algorithms, such as DMPO), these search-based algorithms can be applied as the analysis now satisfies the conditions of use for both the OPA and RPA algorithms. However, the concern with such an approach is that by replacing R_x with D_x, the degree of pessimism of the schedulability tests will increase as $R_x \leq D_x$ in a schedulable system, which could produce unschedulable results for systems that are actually feasible. We denote such approach as OPA-D and RPA-D. As compromises must be made to the new analysis, the optimality of these priority ordering algorithms can be undermined. In Section 4.2.4, evaluations are conducted to investigate the performance of OPA-D and RPA-D and to provide experimental evidence that these compromised priority ordering approaches under the new schedulability tests are not optimal.

4.2.3 The Slack-based Priority Ordering (SBPO)

As proved in Section 4.2.1, the DMPO algorithm is not optimal with our new schedulability tests assumed, which implies that there could exists a priority ordering that is able to schedule a task set while the DMPO algorithm cannot. As for the OPA and RPA algorithms, the pessimism of the new schedulability tests is increased to be compatible with these search-based priority ordering algorithms (see OPA-D and RPA-D in Section 4.2.2) so that their optimality can be undermined. Thus, with the new schedulability tests assumed, there can be the case where a system that is actually schedulable with certain priority ordering, but cannot be scheduled by either the static (i.e., DMPO) or search-based (i.e., OPA-D and RPA-D) priority ordering algorithms reviewed in Section 2.1.3.

Based on the above discussion, this section presents a new search-based priority ordering algorithm as a candidate priority ordering solution for the FMRS framework proposed in this thesis, named as the Slack-based Priority Ordering (SBPO) algorithm. This search-based priority ordering algorithm shares the similar philosophy with that of the OPA and RPA algorithms, but with a different realising approach (1) to be fully compatible with the new schedulability tests of the candidate locking protocols; and (2) to decrease the pessimism introduced for adopting the OPA and RPA algorithms to the new schedulability tests (i.e., the OPA-D and RPA-D approaches). Below gives the pseudo code for the SBPO algorithm with the new schedulability test in Section 3.2 assumed, denoted as S.

```
Initialise priorities of tasks on each processor by DMPO;

For each processor $P_m$, starting from $P_0${
  For each priority level $Pri$, lowest first {
    For each unexamined task $\tau_x$ on $P_m$ {
      Assign $\tau_x$ with priority $Pri$;
      $R_x$ = Get_Response_Time(task $\tau_x$);
      Get the additional slack of $\tau_x$ by $D_x - R_x$;
```

```
            Restore the initial priority of $\tau_x$;
        }
        Assign priority $Pri$ to the task with biggest slack[1];
    }
    Get response times of all tasks in $P_m$ with the new priority
    ordering by test $S$, where $R = D$ for all the unexamined tasks and
    $R = D$ if $R > D$ for all tasks on $P_m$;
}

Get response times of all tasks in the system with the priority
ordering via test $S$;

if (the system is schedulable)
    return true;
else
    return false;
```

where function `Get_Response_Time()` is introduced to provide a special response time calculation approach for obtaining the remaining slack of τ_x, as given below.

```
long Get_Response_Time(task $\tau_x$){
    Assuming $R = D$ for each unexamined remote task;

    Computing the response times of all tasks in $P_m$ iteratively and
    alternately via test $S$, the calculation ends when $R$ has reached to
    $\eta \cdot D$[2] (in case where $R > D$) or $R$ is fixed for each task in $P_m$
    expect $\tau_x$;

    return $R_x$;
}
```

Similar to the RPA algorithm with $E(\alpha, w, i) = \alpha$ (see in Section 2.1.3.4), for a processor P_m and a given priority level Pri, the SBPO algorithm checks the response times of all the unexamined tasks on P_m with priority Pri (while the priorities of other unexamined tasks in the system are initialised by DMPO) and assigns the priority Pri to the task with the largest remaining slack (i.e., $D_x - R_x$ for τ_x). However, unlike the RPA-D approach, where

[1]If tasks have the same slack, the task with biggest deadline is assigned with the priority.

[2]The extension parameter η is introduced to extend to response time calculation to $\eta \cdot D_x$ for a task τ_x.

the response times of all other tasks are assumed to be their deadlines while examining the response time of a given task, the SBPO algorithm holds such an assumption only for the unexamined remote tasks.

For instance, when assigning priorities to tasks in P_5, the updated response times of tasks in P_0 to P_4 are used when computing the response times of tasks in P_5 rather than the deadlines. By doing so, more accurate response times of tasks in P_5 could be obtained as higher input values can only lead to an equal or higher response time in our analysis (i.e., increasing monotonically, as our analysis is based on the RTA analysis in Section 2.1.4). Such an approach is safe, as the response time of tasks from P_0 to P_4 are computed by assuming the response times of the unexamined tasks as their deadlines, which indicates that the response times of tasks from P_0 to P_4 computed are at least equal to their actual response times under the given priority ordering and our new schedulability tests.

In addition, with the RPA-D approach adopted, the algorithm returns with no feasible priority ordering solutions being found if no tasks can be scheduled for a given priority level. In contrast, the SBPO algorithm allows the situation where all tasks miss their deadlines for a given priority level and can still obtain their remaining slacks via function `Get_Response_Time()`, which provides a special approach to calculate response times of tasks in τ_x's processor for obtaining their remaining slacks. Firstly, to increase the accuracy of the response time calculation of a given task τ_x, the response times of all tasks in τ_x's processor are calculated (with their currently assigned priorities) iteratively and alternately in function `Get_Response_Time()`, and will be applied in the calculations of R_x instead of the deadlines.

Calculating Remaining Slacks

Arguably, examining the remaining slacks of tasks that have missed their deadlines is somewhat meaningless. However, as the response times of the unexamined tasks are assumed to be their deadlines, there can be the case where the response times of all the examined tasks are higher than their deadlines under a given priority level, but some of the tasks are actually schedulable. Recall the mechanism for calculating the back-to-back hits, where the number of requests issued from a remote task τ_j to r^k during the release of τ_i is $N_j^k(R_i, R_j) = \left\lceil \frac{R_i+R_j}{T_j} \right\rceil \cdot N_j^k$. Now, assume that both tasks are schedulable (i.e., fixed response times are obtained for both τ_i and τ_j), where $R_i + R_j < T_j$ while

$R_i = D_i$, replacing R_j to D_j in this function will introduce the back-to-back hits from τ_j. Such a phenomenon can block the requests of τ_i to r^k so that more blocking can be imposed to τ_i. Accordingly, $R_i > D_i$ in this response time calculation due to the pessimism introduced by replacing R_j by D_j.

Under such a situation, the SBPO algorithm aims to assign appropriate priorities to these tasks (i.e., the tasks that have missed their deadlines according to the calculations) and to increase the possibility of obtaining a schedulable priority ordering for the system. However, with response times higher than deadlines, the remaining slack $D - R$ is difficult to reflect the robustness of a task under a priority level. For instance, for a given priority level Pri, both τ_1 and τ_2 have missed their deadlines after one iterative calculation, where the slack (i.e., $D_1 - R_1$) of τ_1 is slightly higher than that of τ_2. However, with further iterations, the slack of τ_1 becomes much lower than that of τ_2. In this example, if the slacks are obtained by the calculation where both tasks have just missed their deadlines, τ_1 will be assigned with priority Pri so that the system is likely to be unschedulable as the slacks obtained could not reflect the robustness of the tasks.

To obtain the remaining slacks of tasks under this situation, an extension parameter η is introduced to extend the iterative response time calculations for tasks on P_m when they missed their deadlines, where the calculation ends when the response times have reached to $\eta \cdot D$ for all the deadline-missed tasks except τ_x. The intuition is that for a given priority level, there could be several tasks where their response times are just slightly higher than their deadlines after one iteration. However, with further iterative calculations, huge variances could appear between the slacks of tasks, where the task with the highest slack is relatively strong at this priority level compared to other tasks. Such an approach is also adopted in [87], where the response time calculation of tasks is extended to $n \times D$ for tasks that have missed their deadlines to determine the system that is closest to be schedulable among a set of given systems, where n is a positive integer.

Admittedly, there can be the case where the slacks of two tasks are higher than each other alternatively under each iterative calculation. However, in a general case, extending the iterative response time calculations to a certain extent is more likely to reveal or magnify the differences between the slacks of tasks that has already missed their deadlines. Therefore, by this approach, tasks are more likely to be assigned with appropriate priorities so that the possibility for obtaining a feasible priority ordering solution could be increased. The value of η must be a positive integer and can be decided by users, where a higher η could lead to more accurate remaining slacks of tasks that have missed their deadlines but requires more computation time in a general case. As with the setting adopted in [87], the extension parameter η in this work is set to 5.

With the above approach, the priorities for tasks in each processor can be assigned via comparing the remaining slacks for all tasks under each priority level. With each task assigned with a priority by the SBPO algorithm, the response time of all the tasks in the system based on the new priority ordering are calculated under the schedulability test S to check whether the priority ordering is feasible. If the system is schedulable with the assigned priority ordering, the SBPO algorithm obtains a feasible priority ordering for the given system. Otherwise, the algorithm returns with no priority ordering soliton being found.

Processor Ordering

Note that in this work, the processors are ordered by their indexes and the SBPO algorithm always starts from the first processor (i.e., processor P_0). We acknowledge that other processor orderings are also possible, such as by the total utilisation or the number of tasks on each processor. However, the accuracy of the response times of tasks in a given processor largely depends on the response times of the remote tasks that share the same set of resources. Such a response time dependency is two-way, which implies that among a set of processors that share the same resources, starting the response time calculations from either processor has to assume that the response times of the resource-requesting tasks on other processors are their deadlines.

Therefore, it is difficult to specify an efficient processor ordering merely based on the application semantics and resource usage (i.e., the independent properties of the given system). In addition, no guaranteed benefits can be obtained by ordering processors based on either the utilisation or the number of tasks. Therefore, similar with the existing search-based priority ordering algorithms, we start the priority assignment from the first processor i.e., P_0. In this work, we focus on presenting a practicable SBPO algorithm that is fully compatible with the new schedulability tests. Further optimisation towards this priority ordering algorithm is subject to future work.

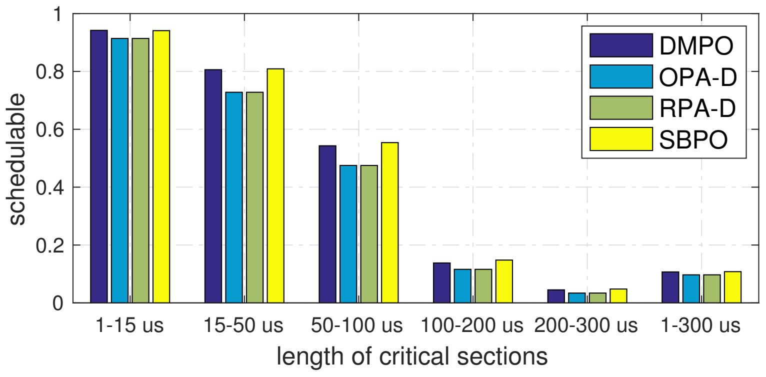
Summary

This section presents a new search-based priority ordering algorithm that is fully compatible with the schedulability tests proposed in Chapter 3.2. As discussed above, compared to the OPA-D and RPA-D approaches, the pessimism of our new algorithm could be reduced due to the modified response time calculation approach. In Section 4.2.4, the schedulability of systems with the SBPO algorithm adopted is investigated and is compared with other priority ordering algorithms discussed in this chapter.

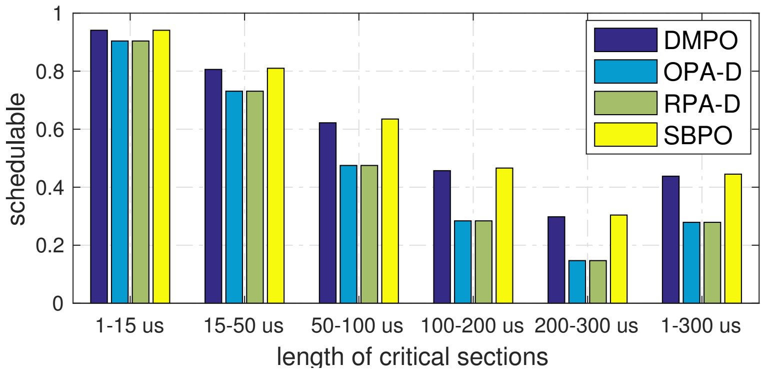
4.2.4 The Impact of Task Prioritising on Multiprocessor Systems with Shared Resources

The above sections has discussed the major existing priority ordering approaches (i.e., the DMPO, OPA-D and RPA-D algorithms) and has developed a new search-based priority ordering algorithm that is fully compatible with the schedulability tests developed in Section 3.2. In this section, experiments are conducted with these priority ordering algorithms to (1) investigate the schedulability of systems with the candidate locking protocols under each of the priority ordering algorithms and to (2) compare the performance between these priority assignment algorithms. The experiments are conducted with the WF algorithm adopted. The test program can be accessed by `https://github.com/RTSYork/SchedulabilityTestEvaluation`. The statistical significance of the experimental results presented in this section is given in Appendix E, demonstrating a confidence level of 95%.

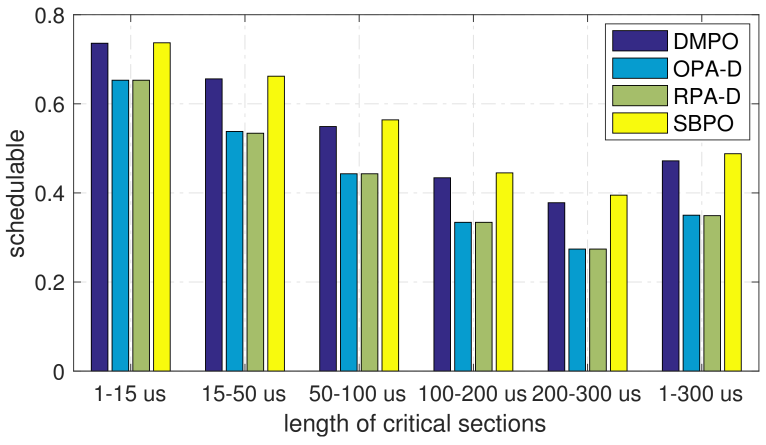
Figure 4.7 illustrates the schedulability of systems with each priority ordering algorithm adopted under the new schedulability test of each candidate locking protocol via varying L (the range of critical section length). As observed, the performance of DMPO is better than both the OPA-D and RPA-D approaches (where the performance of OPA-D and RPA-D is similar) under each protocol, but is outperformed by SBPO in most cases. This observation indicates that (1) the DMPO algorithm is not optimal under the new schedulability tests, where there exist systems that SBPO can schedule but DMPO cannot; (2) the compromises made in the new analysis for the OPA-D and RPA-D approaches in Section 4.2.2 introduce considerable pessimism and can significantly affect the performance of these searching-based priority ordering algorithms; and (3) the SBPO approach effectively reduces the pessimism due



(a) MSRP Systems



(b) PWLP Systems



(c) MrsP Systems

Figure 4.7: Schedulability of Systems for $M = 16$, $n = 48$, $U = 0.1n$, $\kappa = 0.4$, $A = 2$ and M Shared Resources.

to the compromises made to the new schedulability tests and can demonstrate the best performance among all the evaluated priority ordering algorithm in general under any of the candidate locking protocols.

However, although the performance of SBPO is better than DMPO in general with each candidate locking protocols adopted, the schedulability differential of systems under these priority ordering algorithms is relatively small or even unobservable under some cases (e.g., the MSRP systems with $L = [15\mu s, 50\mu s]$ and the PWLP systems with $L = [200\mu s, 300\mu s]$). In addition, although the SBPO algorithm demonstrates equal or better performance compared to other priority ordering algorithms under each candidate locking protocol, whether SBPO can schedule all the systems that other algorithms can also achieve (i.e., the dominance of SBPO) remains unanswered.

Therefore, to further investigate the performance of these priority ordering algorithms, an experiment is conducted under the new MrsP analysis to obtain the exact percentage of systems where the priority ordering A (e.g., SBPO) can find a schedulable system but another algorithm B (e.g., DMPO) cannot in 10,000 systems, as given in Table 4.2, where A & $!B$ indicates the systems that are schedulable under algorithm A but are unfeasible with algorithm B adopted. The priority ordering algorithms that are examined in this experiment include the DMPO, RPA-D and SBPO algorithms. In addition, the performance of OPA-D is similar with that of RPA-D in most cases in our experiments, and thus, is not presented in this experiment to ease the presentation.

Table 4.2: The Percentage of Schedulable MrsP Systems with $M = 16$, $n = 48$, $U = 0.1n$, $\kappa = 0.4$, $A = 2$ and M Shared Resources.

L in μs	DMPO & !RPA-D	!DMPO & RPA-D	DMPO & !SBPO	!DMPO & SBPO	RPA-D & !SBPO	!RPA-D & SBPO
$[1, 15]$	7.1	0	0.1	0	0	7
$[15, 50]$	11.3	0.3	0.1	0.7	0	11.6
$[50, 100]$	10	0.9	0.3	1.8	0	10.6
$[100, 200]$	10.3	0.3	0.5	1.1	0	10.6
$[200, 300]$	10.4	0.8	0.3	1.9	0	11.2
$[1, 300]$	12.6	0.4	0.3	1.6	0	13.5

Similar to the results in Figure 4.7, the DMPO and SBPO algorithms perform better than that of the RPA-D algorithm, and SBPO can schedule more systems than that of DMPO in most cases. However, as revealed by Table 4.2, no priority algorithm can dominate all other algorithms among the examined priority ordering algorithms, which illustrates that all the examined priority ordering algorithms are not optimal for the new schedulability tests developed in this thesis. Note that although there exist no systems where the RPA algorithm can schedule while the SBPO cannot in this experiment, it is insufficient to claim that SBPO can dominate RPA in all cases. However, based on the experimental results, the RPA algorithm is highly unlikely to provide feasible priority ordering to a system where both the DMPO and SBPO algorithms cannot schedule. Based on the above discussions, only the DMPO and SBPO algorithms will be adopted into the proposed resource control framework as the candidate task priority ordering solutions (see Chapter 5 for details).

4.3 Summary

This chapter has investigated the impact of task allocation and priority ordering on multiprocessor systems with shared resources managed by the candidate resource sharing protocols determined in Chapter 3.

For task allocation, three new resource-oriented task allocation schemes (i.e., RCF, RLF-L and RLF-S) are proposed that can reduce the blocking due to certain shared resources, where each new task mapping algorithm could benefit certain candidate resource sharing protocols. Accordingly to the experiments given in Section 4.1.3, as there exist no optimal task mapping solutions, each of the evaluated task allocation schemes is included as a candidate task allocation scheme for the proposed resource control framework, as described in Chapter 5 in detail.

As for task priority ordering, we proved that the DMPO algorithm is not optimal under the newly-developed analysis due to the back-to-back hits calculation mechanism. In addition, we showed that the OPA and RPA algorithms are not applicable to the new schedulability analysis unless comprises are made to the schedulability tests, which however, introduce considerable pessimism and can largely affect the performance of both the OPA and RPA algorithms. Then, a new search-based priority ordering algorithm that is fully compatible with the new schedulability tests is developed. The experiments given in Section 4.2.4 demonstrates that the SBPO algorithm has a better performance in general than other examined priority ordering approaches, but there exists no optimal priority ordering algorithm for the new analysis.

With the new resource-oriented task allocation schemes and the new search-based priority ordering algorithm developed, this chapter provides the candidate task allocation and priority ordering solutions for the resource control framework proposed in Chapter 5 and have satisfied the success criteria SC-2 and SC-3 given in Section 1.4. Below summarises the major contributions made in this chapter:

- New resource-oriented task allocation schemes that take resource characteristics into account and are independent from the resource sharing protocols.
- A formal proof that demonstrates the DMPO algorithm is optimal in fully-partitioned FPPS systems with shared resources under the original schedulability tests of MSRP and MrsP.
- A proof by a counterexample that demonstrates the DMPO algorithm is not optimal in fully-partitioned FPPS systems with shared resources under the new schedulability tests of MSRP, PWLP and MrsP proposed in this thesis.
- A new search-based priority ordering algorithm that is fully compatible with the new schedulability tests for the candidate locking protocols of the proposed resource control framework.
- An investigation towards the impact of task allocation and priority ordering on fully-partitioned systems with shared resources managed by each of the candidate locking protocols.

Chapter 5

FMRS - A Framework for Scheduling Resource-Sharing Tasks in Fully-Partitioned Systems with Fixed Priorities

Combining the outcomes of Chapters 3 and 4, a Flexible Multiprocessor Resource Sharing framework (FMRS) is proposed in this chapter to provide generic resource control and task scheduling solutions for fully-partitioned systems with shared resources. A discussion of the flexibility of the proposed framework is given in Section 5.4 after the compete FMRS framework is presented. The framework is based on Genetic Algorithm (GA) technology [79] and aims to provide a combination of feasible resource sharing, task allocation and priority ordering solutions to any given systems with shared resources under the fully-partitioned platform with the FPPS scheme. Notably, a novel technique for managing shared resources on multiprocessors is proposed in FMRS, where each resource in the system is controlled by an appropriately chosen resource sharing protocol designated by the framework so that the blocking time due to resource-accessing can be less than that of adopting a single protocol for all the shared resources.

This chapter firstly summarises the candidate resource sharing, task allocation and priority ordering solutions for the proposed resource control framework. Then, the motivation and objective of the new resource control technique for fully-partitioned systems is presented. In addition, a brief description of the GA technology for optimisation and complex search problems is presented and the rationale of adopting this heuristic approach for multiprocessor resource sharing issue is discussed. With the above described, we present the detailed approaches of the genetic algorithm-based FMRS framework, including the new resource control technique. Finally, the complete working process of this GA-based multiprocessor resource sharing framework is described with the GA parameter settings summarised based on the suggestions from the literature and the discussions presented in this section.

5.1 Motivation

In Chapters 3 and 4, we have demonstrated that a significant impact can be imposed to the schedulability of multiprocessor systems due to resource sharing via the typical resource control and task scheduling approach, where a single resource sharing protocol is adopted to managed all the resources in the system with traditional task allocation schemes (e.g., the WF scheme) and priority ordering algorithms (e.g., the DMPO algorithm). To investigate the performance of managing shared resources on multiprocessors, three major factors that can directly affect the schedulability of multiprocessor systems with shared resources have been examined, which are multiprocessor resource sharing protocols, task allocation schemes and priority ordering algorithms. Summarising the conclusions in the above chapters, the candidate resource sharing protocols, task allocation schemes and the priority ordering algorithms for the proposed resource control framework for fully-partitioned systems with fixed priorities are given below.

- **Resource sharing:** MSRP, PWLP, MrsP.
- **Task allocating:** WF, BF, FF, NF, SPA, RCF, RLF-S, RLF-L.
- **Task prioritising:** DMPO, SBPO.

Unlike the task allocation and priority ordering solutions (which aim at the whole system), the resource sharing protocols are determined for each individual shared resource in the FMRS framework, where a resource can be managed by any of the candidate locking protocols. Such an approach is motivated by the observations obtained from the experiments in Section 3.4, where each candidate resource sharing protocol can benefit resources with certain characteristics under a given system setting. The intuition is that with each resource managed by the favourable resource sharing protocol, the blocking time of the system can be reduced compared to using a single locking protocol to control all the resources that have various characteristics in a given system.

For instance, as shown in Figure 3.7, under the given system setting, MSRP (i.e., MSRP* in the experiment) can demonstrate better schedulability with $L = [1\mu s, 15\mu s]$ than others, PWLP (i.e., PWLP*) can provide the best schedulability with $L = [15\mu s, 100\mu s]$ and MrsP (i.e., MrsP-NP*) is the better with $L = [100\mu s, 300\mu s]$. Accordingly, with this particular system setting and shared resources with $L = [1\mu s, 300\mu s]$, even though MrsP demonstrates the best schedulability in Figure 3.7, a better schedulability could be achieved by managing the resources with $L = [1\mu s, 15\mu s]$ by MSRP, resources with $L = [15\mu s, 100\mu s]$ by PWLP and the resources with $L = [100\mu s, 300\mu s]$ by MrsP respectively. The schedulability analysis framework developed in Section 3.3 will be adopted in this framework to analyse system with multiple candidate locking protocols in use.

In addition, as demonstrated in Chapter 4, there exist no optimal solutions among the candidate task allocation and priority ordering algorithms for fully-partitioned applications under the new schedulability analysis, where the performance of a task allocation scheme or a priority ordering algorithm largely depends on the application semantics, resource characteristics and the exact resource usage. Accordingly, with the appropriate task allocation and priority ordering algorithms chosen among all the candidate solutions (which include the new task allocation and priority ordering algorithms proposed in this thesis) for a given system, better schedulability of the fully-partitioned applications with shared resources could be obtained than that of with the existing task allocation and priority ordering algorithms adopted.

Therefore, the overall motivation of this multiprocessor resource sharing framework is to provide feasible resource sharing, task allocation and task prioritisation solutions to any given systems, where each resource in the given system is managed by a designated resource sharing protocol. More specifically, with the new resource control technique, task allocation and priority ordering algorithms proposed in this thesis, this framework aims to provide schedulable solutions to systems where the traditional resource control approach with the existing task allocation and priority ordering algorithms cannot. In the following sections, the detailed approach of realising FMRS is presented.

5.2 Heuristic Searching

The above section presents the motivation of the FMRS framework. However, as discussed above, there exist no optimal resource sharing protocols, task allocation schemes and priority ordering algorithms for multiprocessor systems with shared resources under the newly-developed schedulability tests. In addition, as shown by the experiments in Chapter 4, the performance of a candidate algorithm (e.g., a locking protocol) depends not only on the characteristics of the given system, but also on other candidate solutions (e.g., a task allocation scheme and a priority ordering algorithm) that are adopted to that system.

Therefore, to guarantee that feasible resource sharing, task allocation and task prioritisation for a given system can be obtained (assuming they exist), the system has to be analysed by the schedulability analysis framework in Section 3.3 under each possible combination of the resource sharing, task allocation and priority ordering solutions in the worst case, which is unrealistic from the viewpoint of computation expenses as there could exist a significant number of such combinations for a system (i.e., $3^{|R|} \times 8 \times 2$, where $|R|$ denotes the number of shared resources in a given system) . For instance, there exist $3^{16} \times 8 \times 2 = 688,747,536$ possible combinations of the resource sharing, task allocation and priority ordering solutions for a system with 16 resources. In addition, as the performance of a given candidate algorithm depends on many variables (e.g., the application semantics, resource characteristics, resource usage and other algorithms that are adopted to the system), it is impossible to develop a set of static rules that can specify the appropriate resource sharing, task allocation and task prioritisation solutions for a given system.

Based on the above discussion, to provide generic resource control and task scheduling solutions for fully-partitioned systems with shared resources, the heuristic approaches that aim at the optimisation and complex search problems should be adopted. In this work, the genetic algorithm (GA) technology is employed.

5.2.1 The Genetic Algorithm

The genetic algorithm is a meta-heuristic search-based approach inspired by the progress of natural selection, which reflects the principle of "survival of the fitness" [79]. The genetic algorithm belongs to the evolutionary computation methods and is commonly adopted to address the optimisation, dynamic planning and complex search problems e.g., the bin packing problem. The basic approach of the genetic algorithm is to simulate the natural evolution progress, where strong species (i.e., better solutions) can emerge and survive during the process of the selection, evolution and elimination.

A typical genetic algorithm starts with a set of randomly generated solutions (i.e., the first generation) to a particular problem, where each solution (i.e., a *gene*) consists of a set of *chromosomes* representing the detailed approach of that solution. Each gene has a *fitness value* computed by the *fitness function*, which provides the metric of the performance of a given solution for the targeted problem. The fitness value represents the likelihood of a given solution for achieving the objective.

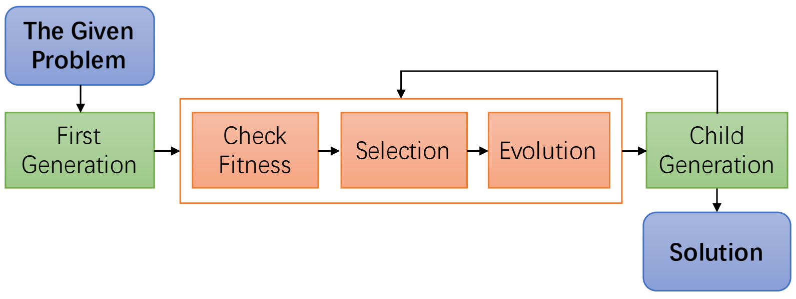


Figure 5.1: The Basic Workflow of a Genetic Algorithm.

With the first generation, iterative calculations are performed to produce further generations via a set of selection and evolution operations, where each iterative calculation produces one generation that contains a set of potential solutions to the given problem. Figure 5.1 illustrates the basic workflow of a genetic algorithm. According to the fitness values of each gene, the best individuals among the current generation are selected as the parents for generating the child population by the *crossover mating* and *mutation* operators. With crossover mating, each individual from the child populations shares the chromosomes of both their parents, which are the best individuals selected from the parent generation. Thus, the children are also likely to be good solutions for the targeted problem. By continuous evolving the best solutions of the last generation, better solutions could emerge compared to their parents. In addition, while generating new populations, the mutation operator is usually applied to change certain chromosomes of randomly chosen individuals in the new generation so that the neighbour solutions with different features can also be explored. Finally, the genetic algorithm can be finished either with a maximum generation limit specified or when a feasible solution is obtained for the given problem.

The above presents the basic rationale and workflow of the genetic algorithm. In practice, the detailed mechanisms for practising this search-based technique (e.g., the approach for generating the first population, and the methods for selection, evolution and mutation) usually vary depend on the actual given problems. The detailed approaches of the GA method adopted in our FMRS framework (which aims at the resource sharing and task scheduling issues on fully-partitioned systems with shared resources) are presented in the following sections.

5.2.2 Rationale

Besides the genetic algorithm, there also exist other heuristic-based searching algorithms for optimisation problems, such as the Simulated Annealing (SA) algorithm [60]. The SA algorithm is a probabilistic technique that searches for an approximate global optimal solution among a significant amount of candidate solutions. Unlike the genetic algorithm (which is an adaptive approach and can generate better solutions through iterative calculations [79]), the SA algorithm is based on the discrete searching technique rather than the evolutionary methods.

Both the GA and SA algorithms have been successfully practised in addressing practical issues in real-time systems. In [6,77,87], the GA technique is adopted to search for feasible task allocation and priority ordering solutions for real-time systems with the presence of network communications while the SA algorithm has also been employed to facilitate allocating tasks while improving the robustness, flexibility and extensibility of real-time systems [42,43]. In addition, in [87], the effectiveness of the GA with various parameter settings (e.g., the population size and the crossover rate) was investigated and the parameter setting that can demonstrate a high performance was reported and was suggested for readers.

As for FMRS, we do not enforce that a designated heuristic approach must be adopted i.e., either the SA technique or the GA approach can be adopted in this framework. However, as reported in [31,80,86], although both approaches can address the problem, the GA approach could lead to better results with less computation time required than the SA algorithm in their experiments. Therefore, the GA technique is adopted to FMRS in this work. The investigation of the effectiveness of the SA technique for the proposed flexible multiprocessor resource sharing framework is subject to future work.

5.3 Framework Design

This section presents the detailed design of the GA-based Flexible Multiprocessor Resource Sharing framework, including the chromosome representation, the fitness functions, the selection approach and the evolution methods required by the GA technique. The framework takes a task set with the exact resource usage known a priori as the input, and aims to return a schedulable system (if achievable) with each resource managed by a locking protocol and each task assigned with an allocation and a priority. A fully-functional implementation of the FMRS framework can be accessed via `https://github.com/RTSYork/FIFOSpinLockFramework`, which contains the implementations of the analysis framework in Section 3.3, the candidate task allocation and priority ordering algorithms, and a typical GA solver. For brevity, the implementation details will not be presented in this thesis.

Note that unlike the works proposed in [43, 87], where the task allocation and priority ordering solutions of a system are obtained via examining a significant number of possible allocations and priorities for each individual task in the system, this work aims to identify feasible task allocations and priority ordering algorithms among the candidate solutions for the whole system, as the FMRS framework mainly focuses on the resource sharing issues for fully-partitioned FPPS systems.

5.3.1 Fitness Functions

As presented in Section 5.2.1, a fitness function is required to measure the quality of the generated solutions so that good solutions in a given generation can be identified. Similar to the GA-based task mapping approach in [87], two fitness functions are adopted in our resource control framework to measure the quality of a given solution, which contains a set of resource sharing protocols, a task allocation scheme and a priority ordering algorithm. Both fitness functions are derived from the schedulability analysis framework proposed in Section 3.3, but with different calculation approaches for measuring the solutions from different metrics.

Function F_D

The first fitness function aims to identify the tasks that miss their deadlines with the given solution. With this function adopted, the response time calculation of a given task finishes immediately if it misses its deadline. The whole function returns when the response time of each task is fixed (less or equal to its deadline) or is higher than its deadline. This function checks through all the tasks in the system and then returns the number of the deadline-missed tasks under the given solution. We denote this function as F_D. Among a set of solutions generated by the GA solver for a given system, the solutions with a lower F_D value (i.e., with less tasks that missed their deadlines) are better solutions in general by intuition and will be selected for further evolution. In the case where $F_D = 0$, the feasible resource sharing, task allocation and priority ordering solutions for the given system are obtained and the search is finished immediately.

Function F_D^η

However, in the case where two or more solutions have the same value (higher than 0) under the function F_D, it is not possible to tell which solution is better. Therefore, the second fitness function is introduced, which has a response time calculation approach similar to that of the SBPO algorithm presented in Section 4.2.3. The second fitness function extends the response time calculation of the analysis framework to $\eta \cdot D$, where the function returns if the response time of each task τ_x is either fixed in further iterations or higher than $\eta \cdot D_x$. Hence, the second fitness function is denoted as F_D^η. As with the setting adopted in [87], η is set to 5 in this work. With a given solution adopted, this function returns the sum of $R_x - D_x$ of each unschedulable task τ_x in the system. The intuition is that with the same F_D value, the solutions with a lower value of F_D^η could has a better performance (i.e., is closer to the objective) in a general case. Accordingly, $F_D = 0$ implies that $F_D^\eta = 0$ as all tasks in the system are schedulable.

Given a set of solutions, the function F_D is adopted firstly, where the solution with the least F_D value is considered as the best solution. In the case where multiple solutions have the same F_D value, the second fitness function is then applied to these solutions, where a lower F_D^η value represents a better performance. Note that the metric from the function F_D overwhelms the function F_D^η. For instance, a solution S_1 is better than S_2 as long as $F_D(S_1) < F_D(S_2)$ regardless the values computed by F_D^η of these two solutions.

5.3.2 Chromosome Representation

In order to perform the genetic algorithm, the candidate resource sharing, task allocation and priority ordering algorithms should be encoded as a set of chromosomes to facilitate the evolution. Before presenting the chromosome encoding for the candidate algorithms, we firstly decide the issues that should be addressed by the adaptive searching approach. As described in Section 5.1, there exist 3 locking protocols, 8 task allocation schemes and 2 priority ordering algorithms as the candidate solutions for a given system, where the locking protocols aim at each individual resource while the task allocation and the priority ordering algorithms target at the whole system.

Firstly, it is clear that the resource sharing issue should be addressed by the adaptive evolutionary method, as there could exist a significant number of possible resource sharing solutions for a given system (see discussion in Section 5.2). For the task allocation issue (where there exist 8 candidate solutions), it is considerably expensive to analyse the system under each resource sharing solution generated by the GA solver and each candidate task allocation algorithm, assuming a given priority ordering algorithm is adopted. Therefore, to reduce the computation expenses, the task allocation problem is also addressed by the GA approach in this framework. However, as for the priority ordering issue (where there only exist 2 candidate algorithms), we decide that the priority ordering issue will not be addressed by the GA technique, where each solution generated by the GA solver (i.e., a locking protocol for each resource and a task allocation scheme for the system) is analysed under both candidate priority ordering algorithms. Such an approach can improve the efficiency of the proposed framework as the searching range is reduced. However, the fitness functions are adopted with only one priority algorithm assumed in order to be able to correctly compare each combination of resource sharing and task allocation solutions. In this work, the priority ordering algorithm adopted for computing the fitness values is set to the first candidate priority ordering algorithm by the index values i.e., the DMPO algorithm in this work. In practice, this setting can be configured by users conveniently.

For this framework, the candidate solutions can be effectively encoded by a set of integer values. The chromosome values of the candidate resource sharing

Table 5.1: Chromosome Values of the Candidate Resource Sharing Protocols

Resource Sharing Protocol	Value
MSRP	1
PWLP	2
MrsP	3

Table 5.2: Chromosome Values of the Candidate Task Allocation Schemes

Task Allocation Scheme	Value	Task Allocation Scheme	Value
WF	1	SPA	5
BF	2	RCF	6
FF	3	RLF-L	7
NF	4	RLF-S	8

protocol and the task allocation schemes are given in Tables 5.1 and 5.2. Note that the traditional task allocation schemes are performed with tasks ordered by utilisation non-increasingly. The intuition is that with the heavy tasks (i.e., tasks with a high utilisation) allocated first, the tasks with a low utilisation can have a higher chance to be fitted into the remaining spaces of the processors so that a higher success ratio of these task mapping algorithms can be obtained, compared to the approach that starts the allocation from the low utilisation tasks.

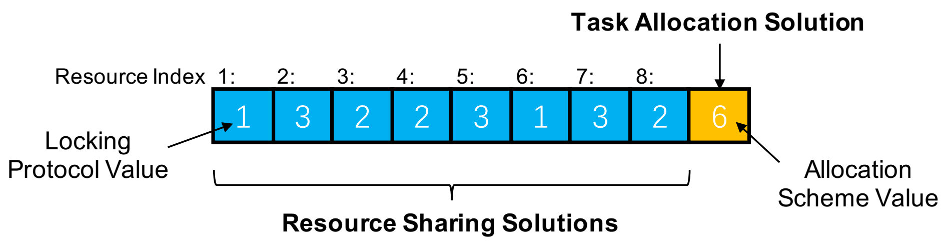


Figure 5.2: The Chromosome Representation for the Candidate Solutions.

An example of the chromosome representation for a combination of the resource sharing and task allocation algorithms (i.e., a solution generated by GA) for a system with 8 resources is shown in Figure 5.2, which is an array with 9 integers. As shown in the figure, the first 8 integers in the array indicate the resource sharing solutions of that system, where the n^{th} integer represents the chromosome value of the resource sharing protocol adopted to the resource r^{n+1} (the resource indexing starts from 1). In addition, the last integer in the array indicates the chromosome value of the task allocation scheme. Note that although the candidate resource sharing protocols and some of the candidate task allocation schemes share the same index values, this will not cause any issue as the task allocation solution is always placed at the end of the chromosome queue.

Such a chromosome encoding approach is simple, but is effective according to the discussion in [87] and is sufficient for the FMRS framework proposed in this thesis. For a system with 16 shared resources, an array with 17 integers will be generated for each solution produced by the framework. As described in Section 3.4, this thesis considers systems with up to $M \times 2$ shared resources and the maximum value of M is set to 24, which indicates a maximum chromosomes array size of 49. The system settings adopted in this thesis are similar with the settings employed in [106] for investigating the resource sharing in the AUTOSAR profile [46] for automotive electronic control units so that a wide range of application semantics in real-world applications can be covered in the system settings of this thesis.

5.3.3 Generation and Population

As an evolutionary computation method, the GA technique relies on a set of iterative calculations, where each iterative calculation can produces a child generation that contains a set of potential solutions to the given problem. By providing the first generation, the GA can produce future generations via a set of selection and evolution operations described in Section 5.2.1, where the size of a given generation is specified by users via the parameter *population size.*

The First Generation

To start the evolution process, the first generation must be provided for evolving future generations. Typically, the first generation is usually a set of randomly generated solutions to the given problem [54]. In our framework, the initial generation is produced by two steps, as described below.

- Firstly, each solution that contains a single resource sharing protocol (i.e., the traditional resource sharing technique) and a task allocation scheme that can allocate the given task set is encoded as a gene in the first generation e.g., $\{1, 1, 1, 1, 3\}$ for a system with 4 resources. For any given systems, there exist at most 3×8 such solutions, assuming the given task set can be allocated by each candidate task allocation scheme.

- Then, the rest of the individuals in the first generation is produced randomly, where the resource sharing solution of each resource is randomly decided by `Random.nextInt(Integer.MAX_VALUE) % 3 + 1` and the task allocation solution is randomly determined between the candidate task allocation schemes that can allocate the given task set. For instance, if a task set can be mapped by `allocations=`$\{$`1,2,5,6,7,8`$\}$, the task allocation solution is then decided by `allocations.get(Random.nextInt(Integer.MAX_VALUE) % allocations.size())`.

Considering the traditional resource sharing technique in this framework is worthwhile. If a system can be schedulable with only one locking protocol adopted, this approach is more preferable than the new resource control technique (where multiple locking protocols are in use), as a simpler system can be obtained with an easier analysing technique required. In addition, compared to the randomly generated solutions, such resource sharing solutions could have a better fitness under certain situations, and hence, are valuable for the evolution process. Furthermore, adopting the traditional resource control approach as the solutions of the first generation guarantees that the performance of the best solution in each generation in the framework is equal to or higher than the traditional resource sharing solutions. This is because the best solution of the current generation will be passed to the next generation directly in our framework to guarantee the quality of each generation being produced. The detailed selection and evolution approaches are presented later on in Section 5.3.4.

Population Sizing

For each generation (including the first generation produced above), the population size must be specified to provide a searching range for the GA solver, where a small population size cannot explore enough potential solutions so that the efficiency of the resource control framework is undermined while a huge population size can significantly increase the computation expenses (especially with expensive fitness functions adopted), and hence, can lead to a poor usability. As reported in [61], decreasing the population size can increase the speed of optimisation to a certain point. However, after that point, the optimisation speed is slowed down due to the pre-matured convergence, where the solutions in a given generation becomes identical after only a few evolutions even with a randomly generated initial population due to a low population size (i.e., a poor population diversity).

In [87], the population size is tested with a value of 400 and 1000 respectively. With no doubt, the GA with a higher population size is more likely to obtain the feasible solution to the given problem as a wider range of potential solutions can be explored. However, from the viewpoint of run-time efficiency, increasing the population size is a tradeoff between

- increasing the possibility of obtaining a feasible solution
- decreasing of the computation time efficiency.

Compared to the fitness functions employed in the GA in [87], the fitness functions adopted in the FMRS framework is relatively complicated and time-consuming. As described in Section 5.3.1, the fitness functions adopted in the this GA-based framework is an extension of the schedulability analysis framework proposed in Section 3.3, which integrates each individual schedulability test in Section 3.2. Therefore, the population size is set to 500 in this work due to the concern of computation expenses. Such a setting may not be optimal but is sufficient to demonstrate the performance differences between this new resource control framework and the traditional resource control and resource sharing techniques for fully-partitioned systems with shared resources.

The above discussion also verifies the rational for developing the new schedulability analysis framework presented in Section 3.3 as the fitness functions of this GA-based framework rather than modifying the existing ILP-based analysis in [106] to support the analysis of systems with multiple protocols in use. As shown in the experiments of the time consumption for the new schedulability tests and the ILP-based analysis presented in Section 3.4.4, the ILP-based analysis is considerably expensive compared to our new analysis. In addition, such a time consumption can become significant with the response time calculation extended to $\eta \cdot D$, where η is set to 5 in this work and can be further increased by users in practice. Thus, if the ILP-based analysis was assumed in FMRS framework, the fitness functions can be much more time consuming than that of our analysis, which can greatly undermine the usability of the framework.

The Maximum Generation Limit

In addition, with the GA technique adopted, a set of generations will be produced to obtain the feasible solutions for the given system. As described above, the search is finished either with a feasible solution obtained for the given system or it has reached the pre-defined maximum generation limit. As suggested in [87], the maximum generation limit (i.e., the parameter specifies the ending point of the GA-based framework when no feasible solution is found) in this work is set to 500, which offers a reasonable searching range of the potential solutions for a given problem.

5.3.4 Selection

With the first generation obtained, individuals in this population will be selected for producing the next generation. As stated in [82], the selection method determines by how much the knowledge of the current generation can be utilised, where a high selection pressure can facilitate the selection of solutions with a high fitness value while a low pressure can maintain the diversity of the next generation. Figure 5.3 illustrates the whole selection and evolution procedure in our GA-based framework, where the green blocks represent the selection methods and the red blocks denote the evaluation approaches. A detailed description of the evolution approaches is presented in Section 5.3.5.

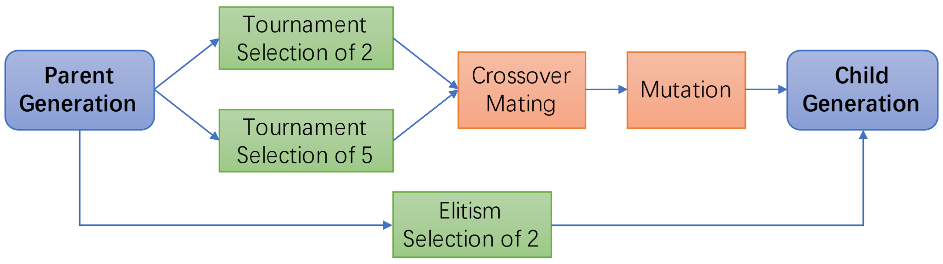


Figure 5.3: The Selection and Evolution Methods in the GA-based Multiprocessor Resource Sharing Framework.

In [29] and [4], the tournament selection method is described, where a given number (say n) of individuals are randomly selected from the current generation and the individual with the highest fitness will be passed directly to the next generation, where n reflects the selection pressure. In [87], two tournament selections are adopted to select the parents for generating the child solutions via the crossover operation, where the first tournament selection has a low pressure so that the population diversity can be guaranteed and the second selection applies a high pressure to obtain the individuals with a high fitness. In FMRS, the selection approach described in [87] is applied. As investigated and suggested in [10, 78, 87], in this framework, the size of the first tournament selection is set to 2 (i.e., a low selection pressure for diversity) while the second tournament selection has a selection size of 5 (i.e., a high selection pressure for quality).

Besides the tournament selection, the elitism selection approach [39] is also adopted in the selection process in our framework, which directly advance n individuals with the highest fitness in the current generation into the next generation. This approach prevents the situation where the newly produced generations have a lower quality than the previous generations via the crossover or mutation process, which can undermine the efficiency of the GA searching. With the elitism selection approach adopted, it guarantees that the quality of a child generation is always at least equal to that of the previous generations (recall the discussion for considering the traditional resource sharing technique in Section 5.3.3). However, as observed from the experiments in [87], a high elitism size (i.e., many elites are passed directly to the next generation) can lead to the situation where the elites dominate other solutions in further evolution so that the optimisation process is converged too early. Therefore, as suggested in [87], the size of the elitism selection in our framework is set to 2.

5.3.5 Crossover and Mutation

This section discusses the crossover and mutation operators in genetic algorithms, which are the major approaches for evolution [54]. As stated in [97], these operations have a significant impact to the diversity and convergence of the generations being produced and can reduce the premature convergence to a local-optimal instead of the global-optimal solutions.

The Crossover Operator

As shown in Figure 5.3, the crossover operator is applied after two individuals are selected from the parent generation for producing a child. In [98], two crossover approaches are described, which are the one-point crossover and the two-point crossover operations, as illustrated in Figure 5.4.

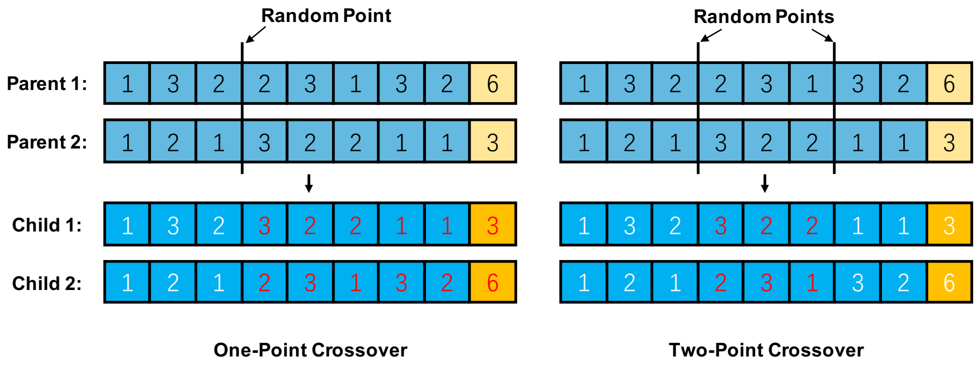


Figure 5.4: The One-Point and Two-Point Crossover Operations.

With the one-point crossover method adopted, a random point is determined in the chromosome array of the parents and two children are generated by exchanging the chromosomes after the given point. As for the two-point crossover, two random points are determined in the parent's gene and the children is generated by exchanging their chromosomes between the given two points.

As shown in the figure, both crossover operations can generate two children, where the one with a higher fitness value will be evolved to the next generation in our framework. As reported in [87], the two-point crossover operation can demonstrate better efficiency than the one-point crossover operation in most cases. Therefore, the two-point crossover operation is adopted in the proposed resource control framework.

In addition, it is not necessary that each pair of selected individuals must be processed by the crossover operation. This is controlled by the *crossover rate* parameter, which has a range of $[0, 1.0]$. As stated in [51], this parameter is curial to the effectiveness of the GA solver, where a higher crossover rate provides a higher diversity but slows down the speed of convergence (i.e., more potential solutions can be explored).

With a low crossover rate adopted, the individuals from the parent generation are more likely to be advanced into the child generation directly, and hence, undermines the efficiency of evolution to a certain degree. In [87], the effectiveness of the GA is tested with the crossover rate of 0.5 and 0.8, and a better performance is observed with the crossover rate of 0.8 assigned. Therefore, as suggested in [87], the crossover rate in the propose GA-based framework is assigned to 0.8 for a high efficiency in general.

The Mutation Operator

The mutation operation is conducted after the entire child generation is obtained via the selection and crossover operations, where each individual in that generation has a chance to mutate according to the *mutation rate*. As described in [51], the mutation rate should be assigned with a small value as a high mutation rate can cause the GA more likely to be a purely random approach. As investigated and suggested in [51], the mutation rate is set to 0.01 in the FMRS framework.

In addition, giving an individual selected to mutate, a certain number of the chromosomes in that individual will be updated (i.e., mutated) with random values generated by the approach for producing random solutions in Section 5.3.3. The number of chromosomes that can mutate in an individual is set to $\lceil 0.1 \times (|R|+1) \rceil$, where $|R|$ gives the number of shared resources in the given system. That is, only one chromosome will be chosen to mutate for a gene with a size less than or equal to 10. By doing so, the individuals will not be changed too much due to the mutation operator while the similar solutions of a given individual can be explored.

5.3.6 Complete Working Process and Parameter Settings

With the detailed approaches for adopting the genetic algorithm in the proposed resource control framework described in the above section, this section describes the complete working process of the GA-based FMRS framework for solving the resource sharing, task allocation and priority ordering issues in fully-partitioned FPPS systems with shared resources. In addition, the GA parameter settings for the proposed FMRS framework are summarised based on both the suggestions from the literature and the discussions given in the above section.

With a given task set that shares a set of resources and the candidate solutions specified, the framework firstly identifies the task allocation schemes that can provide feasible task allocations for the given task set among the provided solutions. Then, the first generation is produced, which contains the traditional resource control solutions (i.e., with one locking protocol only) and a set of randomly generated solutions via the approach in Section 5.3.3. Among the resource sharing, task allocation and priority ordering issues targeted by the proposed framework, only the resource sharing and task allocation issues are addressed by the GA-based searching approach, and each solution produced by GA will be examined under both candidate priority ordering algorithms i.e., DMPO and SBPO. However, the fitness values of the solutions are computed with the DMPO algorithm assumed in this work for comparability.

Generation 1:	Best Fitness: ($F_D = 3, F_D^5 = 1342$)
	Best Individual: {3, 3, 3, 2, 2, 2, 2, 3, 3, 3, 2, 3, 1, 2, 2, 2, 0}
Generation 2:	Best Fitness: ($F_D = 2, F_D^5 = 1287$)
	Best Individual: {3, 3, 3, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 2, 2, 2, 0}
Generation 3:	Best Fitness: ($F_D = 1, F_D^5 = 4925$)
	Best Individual: {3, 3, 3, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 3, 2, 2, 0}
Generation 4:	Best Fitness: ($F_D = 1, F_D^5 = 4925$)
	Best Individual: {3, 3, 3, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 3, 2, 2, 0}
Generation 5:	Best Fitness: ($F_D = 1, F_D^5 = 4719$)
	Best Individual: {3, 3, 3, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 3, 2, 3, 0}
Generation 6:	Best Fitness: ($F_D = 1, F_D^5 = 329$)
	Best Individual: {3, 1, 3, 2, 2, 3, 2, 2, 3, 3, 3, 1, 1, 3, 2, 3, 0}
Generation 7:	Best Fitness: ($F_D = 1, F_D^5 = 23$)
	Best Individual: {3, 1, 3, 2, 2, 3, 2, 1, 3, 3, 3, 1, 1, 3, 2, 3, 0}
Generation 8:	Best Fitness: ($F_D = 0, F_D^5 = 0$)
	Best Individual: {3, 2, 3, 2, 2, 3, 2, 1, 3, 3, 3, 1, 2, 3, 2, 3, 0}

Feasible Solution Obtained at **8th** Generation: **{3, 2, 3, 2, 2, 3, 2, 1, 3, 3, 3, 1, 2, 3, 2, 3, 0}**

Figure 5.5: A Successful Run of the GA-based Resource Control Framework.

Then, a set of iterative calculations are performed to produce the descendant generations via a set of selection and evolution operations, where each iterative calculation produces a generation with a quality that is at least equal to that of its ancestors due to the adoption of the elitism selection method. Thus, with further iterations, better solutions that are closer to the feasible solutions could emerge via the GA-based searching technique. Finally, the framework returns if the feasible resource sharing, task allocation and priority ordering solutions for the given system are obtained or the pre-defined maximum generation limit has been reached. Figure 5.5 provides an example of a successful run of the GA-based framework with a given system that cannot be schedulable with the traditional resource control approach (also the reason that the GA begins to evolve). Through selecting and evolving, better solutions are produced in each generation (except the 4th generation), and a feasible solution (i.e., with $F_D = 0$ and $F_D^5 = 0$) is obtained eventually in the 8th generation with the WF and DMPO algorithms pre-defined as the task allocation and priority ordering solutions.

Table 5.3: The GA Parameter Settings of the Resource Control Framework

GA Parameters		**Settings**
Population Size		500
Max Generation		500
Chromosome Encoding		*index*
Crossover	Rate	0.8
	Method	*two-point*
Mutation	Rate	0.01
	Bound	$\lceil 0.1 \times (\|R\|+1) \rceil$
Fitness Functions		F_D, F_D^5

The above presented the complete working process of the GA-based multiprocessor resource sharing framework. To practice this framework, the GA parameters must be assigned. In this section, each GA parameter is assigned with a value either based on the suggestions from the literature e.g., the elitism size and the mutation rate or by the discussions presented in the above section i.e., the population size. Table 5.3 summarised the GA parameter settings that are assigned to the GA-based framework. In Chapter 6, the experiments that investigate the performance of this new FMRS framework is conducted with these settings adopted.

Admittedly, the GA parameter settings adopted in the FMRS framework in this thesis may not be optimal e.g., the population size of 1000 is found to be better than 500 in [87]. However, as discussed above, such parameter settings are sufficient to demonstrate the performance differences between our resource control framework and the traditional resource control and task scheduling technique for fully-partitioned FPPS systems with shared resources (see Chapter 6), where a single locking protocol is adopted for all resources with the existing priority ordering and task allocation algorithms employed. Investigating the performance of the FMRS framework under various GA parameter settings is interesting, but is not objective of this thesis, which focuses on a new resource control technique for fully-partitioned systems. The investigation towards the optimisation of the GA parameters in this framework is subject to the future work.

5.4 Summary

In this chapter, a resource control framework for the fully-partitioned platform was presented, namely the Flexible Multiprocessor Resource Sharing framework. The FMRS aims to provide feasible resource sharing, task allocating and priority ordering solutions to any given task sets with shared resources. Besides the traditional resource control technique and the existing task allocation and priority ordering algorithms, this framework provides new solutions to the resource sharing, task allocation and priority ordering issues for multiprocessors systems. Notably, with FMRS adopted, each resource in the system is managed by a resource sharing protocol designated by the framework. To provide such solutions, the genetic algorithm is adopted in FMRS to facilitate the searching process. The schedulability analysis developed in Section 3.3 is adopted to analyse the schedulability of systems with multiple resource sharing protocols working in collaboration simultaneously. In Chapter 6, the performance of FMRS is investigated and is compared with the traditional resource control and task scheduling technique for fully-partitioned systems with shared resources under various system settings.

This framework is flexible. First, during practice, users can specify a given task allocation and/or priority ordering algorithm to the framework so that the framework will only focus on the unsolved issue (or issues). For instance, if the SPA task allocation algorithm is mandated for a given work, FMRS can be configured to focus on addressing the resource sharing and priority ordering issues only with the SPA algorithm assumed as the task allocation solution. In addition, the candidate algorithms in FMRS can be changed easily. The existing candidate algorithms can be directly removed without further actions required. Meanwhile, new candidate multiprocessor resource sharing protocols, task allocation schemes and priority ordering algorithms can be added into the resource control framework conveniently, as long as the new locking protocols are supported with a schedulability test, and new the task allocation and priority ordering algorithms are independent from the resource sharing protocols and schedulability analysis. Section 3.3 has demonstrated that a schedulability test of a locking protocol can be easily integrated into the analysis framework.

Finally, although the current version of this framework aims to find feasible resource control and task scheduling solutions, it can be extended to further optimise the system (assuming that feasible solutions are obtained) for improved robustness and scalability. The work proposed in [43] has demonstrated the approach for optimising the task allocation of a given system for improved robustness and extensibility via a heuristic-based search algorithm. In addition, as proved in Section 4, there exist no optimal task allocation and priority ordering algorithms on multiprocessors with the new schedulability tests adopted. Therefore, this framework can be further extended to compute the task allocation and priority ordering solutions by heuristic searching (similar to the approaches proposed by [43, 87]) instead of via the candidate algorithms, which could provide better performance for managing shared resources on multiprocessors. However, as the first version, the FMRS framework aims to provide feasible resource control solutions via the novel resource control and task scheduling approaches. The investigation of the above discussion is subject to future work.

Summarising the above, the material provided in this chapter has satisfied the success criteria SC-4 given in Section 1.4 with the contributions given below.

- A novel resource control technique for managing shared resources in fully-partitioned systems, where each shared resource can be controlled by a designated multiprocessor resource sharing protocol.

- A Flexible Multiprocessor Resource Sharing framework for the fully-partitioned FPPS systems that takes a task set with shared resources as the input, and aims to searche for a schedulable system (via the new schedulability analysis in Section 3.3) with each resource controlled by a designated resource sharing protocol and each task assigned with a priority and a processor.

- A GA-based approach to search for feasible resource sharing solutions for multiprocessor systems with shared resources.

Chapter 6

Evaluating the Multiprocessor Resource Sharing Framework

In Chapter 5, the FMRS framework for scheduling resource-sharing tasks under fully-partitioned systems with fixed priorities was proposed, which aims to provide feasible resource sharing, task allocation and priority ordering solutions (under the run-time overheads-aware schedulability analysis framework in Section 3.3) for any given task sets via a genetic algorithm. This framework uses a new technique for managing shared resources on multiprocessors, where multiple resource sharing protocols (i.e., MSRP, PWLP and MrsP) can work in collaboration, and each of the protocols only manage certain shared resources designated by the GA-based framework.

In addition, with FMRS adopted, the allocations of the tasks in the given system are generated by one of the task allocation schemes among the WF, BF, FF, NF, SPA, RCF, RLF-L and RLF-S algorithms, and the priorities of the tasks are assigned via the DMPO or the SBPO algorithms. The decisions of which task allocation and priority ordering algorithms should be adopted to a given system are made by the GA-based framework. Among the candidate solutions, the RCF, RLF-L, RLF-S and SBPO algorithms are developed in Chapter 4 and are illustrated to be better than the existing algorithms (e.g., the WF, SPA and DMPO algorithms respectively) in the general case.

In this chapter, the performance of this multiprocessor resource control framework is investigated and is compared with the typical resource sharing and task scheduling approaches on multiprocessors, where only one locking protocol is adopted for a system with the existing algorithms for task allocation and priority ordering. Firstly, we investigate the efficiency of the newly-proposed resource control approach, where more than one resource sharing protocols are adopted to managed the shared resources in a given system. Then, we evaluate the performance of the complete FMRS framework, where the resource sharing, task allocation and priority ordering solutions are all decided by the GA-based framework.

The experimental setup is similar to the one adopted in Chapters 3 and 4, where the experiments are conducted under a set of systems generated with various settings (e.g., critical section length and the frequency of resource accesses) via the system generation tool described in Section 3.4. For each system setting, 1000 systems will be generated and their schedulability is examined under the evaluated algorithms (i.e., the traditional and the new approaches for managing shared resources and scheduling resource-sharing tasks in fully-partitioned systems). This evaluation method is also adopted in [106] to investigate the performance of the spin locks in the AUTOSAR profile with the ILP-based analysis adopted. In addition, the GA parameters in the resource control framework are configured as the settings given in Table 5.3.

The test programs of the experiments presented in this chapter can be accessed via `https://github.com/RTSYork/FIFOSpinLockFramework`, which contains the implementations of

- The schedulability analysis framework that supports analysing systems with multiple resource sharing protocols (i.e., MSRP, PWLP and MrsP) in use.
- The candidate task allocation schemes (i.e., the WF, BF, FF, NF, SPA, RCF, RLF-L and RLF-S algorithms) and the priority ordering algorithms (i.e., the DMPO and SBPO algorithms).
- The GA-based FMRS framework that takes a task set with shared resources as the input, and aims to provide feasible resource sharing, task allocation and priority ordering solutions (if achievable) to the given task set.
- A system generator that can produce a set of tasks and shared resources with detailed resource usage based on the given system settings.

In addition, there exist a large number of combinations of different system settings (e.g., various number of tasks, processors and the resource-accessing behaviours). In the interest of brevity, we present the experimental results with the system settings that can effectively demonstrate the performance difference between the evaluated algorithms. In Appendix E, the statistical significance of the experimental results presented in this chapter is analysed, which demonstrates a confidence level of at least 95%. The material provided in this section satisfies the success criteria SC-5 given in Section 1.4.

6.1 Investigating the Performance of the New Resource Control Technique

In this section, we focus on investigating the performance of the new resource control technique proposed in Chapter 5. The resource control approaches evaluated in this section include the typical resource control approach (where only one protocol i.e., either MSRP, PWLP or MrsP is adopted to a given system) and the newly-proposed technique, which uses a combination of protocols to manage the shared resources in one system. We will firstly compare the schedulability of systems with either the traditional or the FMRS framework adopted. Then, we investigate the success rate of this new resource control technique on systems that are not schedulable with a single locking protocol adopted.

To provide fair comparison, the WF and DMPO algorithms are adopted to all the generated systems as the task allocation and priority ordering solutions. This applies to the FMRS framework as well (which contains the new resource control approach), where the WF and DMPO algorithms are pre-assigned to the framework so that it focuses on searching for the feasible resource sharing solutions only. The schedulability of systems with a single locking protocol adopted is obtained via the schedulability analysis developed in Section 3.2, and the analysis framework proposed in Section 3.3 is employed for analysing systems with the new resource control technique employed. All these tests take the run-time overheads of both the operating system (in this case, the Litmus overheads presented in Appendix B) and the protocol implementations into account.

6.1.1 Schedulability Comparison

Figures 6.1 to 6.4 present the percentage of schedulable systems among the generated systems with the traditional and new resource control techniques adopted, where the stack of the “brown” and “yellow” bars represent the total percentage of schedulable systems by using the FMRS framework. The brown bar denotes the systems that are schedulable with a single locking protocol (i.e., any of the candidate locking protocols adopted in FMRS) while the yellow bar indicates the systems that are unschedulable by each of the candidate locking protocols (i.e., the typical single locking protocol approach) but are feasible under the new resource control technique in FMRS.

For each system setting, 1000 systems are generated and tested by each resource sharing approach. In Figures 6.1 to 6.3, the critical section range $L = [1\mu s, 300\mu s]$ is assumed to simulate applications that are more realistic, where a system can have resources with both short and long critical sections. The experiment for investigating systems with varied range of critical section length is presented in Figure 6.4.

Varying n and M

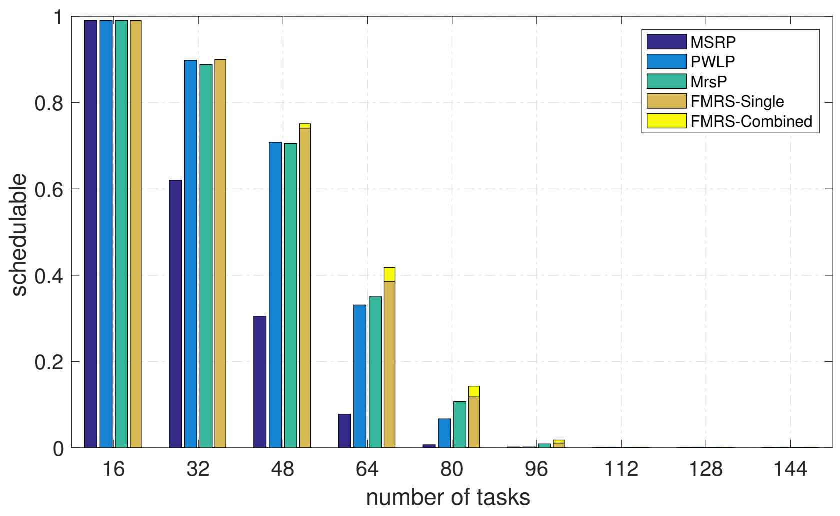


Figure 6.1: Schedulability of Systems for $M = 16$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

The first experiment is conducted via varying the number of tasks in the system (i.e., n) on a 16-processors platform, as given in Figure 6.1, and it presents the percentage of schedulable systems under the single protocol approach (i.e., with MSRP, PWLP and MrsP respectively) and the FMRS (i.e., the stacked bars) adopted respectively.

From the figure we firstly observed that by increasing n, there exists a decreasing trend in the schedulability of systems under each examined resource control approach (including our new framework), where all the resource control approaches can hardly schedule any systems with $n \geq 96$. Among them, the schedulability of systems with MSRP adopted decreased dramatically compared to other approaches and can hardly schedule any systems with $n \geq 80$. Such an observation provides evidence again that the performance of MSRP can be undermined greatly with the presence of long critical sections. Among the candidate locking protocols, PWLP demonsrates the best schedulability with $n = 32$ due to its fully-preemptable mechanism and MrsP outperforms others with $n = 64$ as this protocol is favourable with long critical section. Such phenomenon is investigated and is explained in details in Section 3.4. In addition, the schedulability of each candidate resource sharing protocols shown in the following figures is also as expected (see Section 3.4 for detailed explanations).

With $n = 16$ and $n = 32$, the performance of the FMRS framework is similar with that of the single protocol approach. This is because with a low system utilisation, the penalty for accessing shared resources cannot cause a significant impact to the schedulability of the systems, and hence, adopting any of the examined approaches will not lead to an obvious schedulability difference. However, by further increasing n (i.e., with $n = \{48, 64, 80\}$), FMRS demonstrates better performance (i.e., the stacked "brown" and "yellow" bars combined) than the single protocol approach. This phenomenon is caused by the following two reasons:

1. Accessing shared resources under these system settings now has a direct impact on the schedulability of the generated systems, where there exist systems that can only be schedulable with certain resource sharing protocols. For instance, some systems are only schedulable by MSRP while there also exist systems that are feasible only with MrsP adopted. Thus, as FMRS also considers the single protocol approach (i.e., will check whether the given system is schedulable under each candidate locking protocol), it can have a higher percentage of schedulable systems (i.e., the "brown" bar) than that of each candidate locking protocol (i.e., the first three bars) via including the systems that can be schedulable with any of the protocols adopted. However, this observation cannot lead to any general conclusions.

2. There exist systems that cannot be schedulable with any of the candidate locking protocols, but are schedulable by using the new resource control technique (i.e., the yellow bar), where multiple locking protocols are adopted to manage certain shared resources designated by FMRS . This is the key observation that demonstrates the combined resource control technique employed in the FMRS framework is effective and has a better performance than the single protocol approach.

The above discussions can be observed in Figure 6.1 with $n = 48$ (where there exist systems that can be schedulable only with a certain locking protocol adopted), $n = 64$ and $n = 80$ (where there exist systems that cannot be schedulable with a single locking protocol, but is feasible under the new resource control technique). However, with $n \geq 96$, each examined resource control approach can hardly schedule any systems under the given system settings due to the high system utilisation.

This experiments provides evidence that the new resource control technique has a performance at least equal to or better than the traditional resource control techniques under the tested system settings. Especially, with $n = 64$ and $n = 80$, where the blocking time is a major factor that can affect the schedulability of systems, and the new approach becomes effective and can schedule systems that are not feasible with the traditional resource sharing technique adopted.

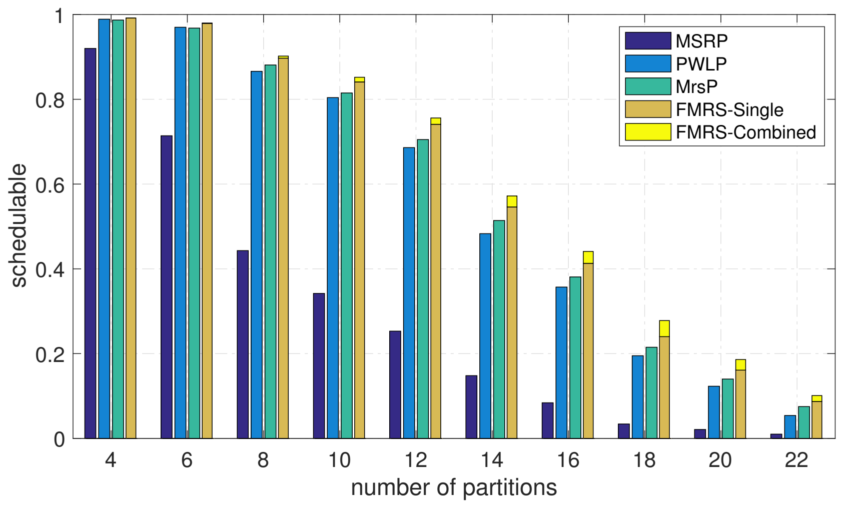


Figure 6.2: Schedulability of Systems for $n = 4M$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

The observation given above can also be obtained in Figure 6.2, where each resource control approach is evaluated with varied M (i.e., the number of processors) and $n = 4M$. With $M = \{4, 6, 8\}$, the remote blocking is relatively low so that there exists no obvious difference between the traditional and new resource control approaches. As described above, the reason that the "brown" bar is slightly higher than the first three bars (i.e., with each candidate locking protocol adopted) is that the FMRS framework also considers this resource control approach.

By further increasing M, the remote blocking time becomes higher and has a direct impact on the schedulability of the generated systems, where the performance of each evaluated resource control approach decreases continuously. However, in such cases, the new resource control technique becomes effective (i.e., the yellow bar) and can schedule systems where each of the candidate resource sharing protocols cannot. Such an observation becomes obvious with $M \geq 14$. With $M = 18$, the new resource control technique has a percentage of schedulable systems similar to that of MSRP.

However, with $M > 18$, the performance of the new resource control technique begins to decrease. The reason is that under such system settings, MSRP can hardly schedule any systems and is basically dominated by PWLP and MrsP. Therefore, adopting the new resource control approach under such cases could not obtain a high efficiency as systems that are not feasible with either PWLP or MrsP adopted also has a small chance to be schedulable by other resource control approaches (i.e., MSRP or the new resource control approach).

Varying A and L

Now we investigate the performance of the new resource control technique by varying the frequency of resource access (i.e., A) and the range of critical section length (i.e., L). Figure 6.3 presents the evaluation results under varied frequency of resource access with $n = 64$ and $M = 16$. In this experiment, the new resource control technique demonstrates a high performance (i.e., the yellow bar) in most cases. However, unlike the results given in Figures 6.1 and 6.2, there exists no obvious relationship between the performance of the new resource control technique and the value of A. The reason is that although MSRP is the worst among all candidate locking protocols under the most cases (given that $L = [1\mu s, 300\mu s]$), the performance of this protocol is less affected

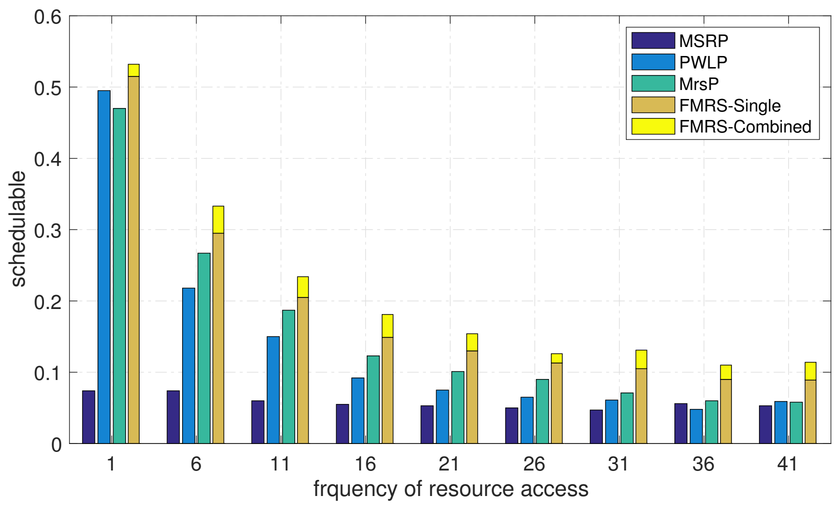


Figure 6.3: Schedulability of Systems for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

by increasing the frequency of resource access, where all the candidate locking protocols demonstrate similar performance with $A \geq 36$. Therefore, each of the candidate locking protocols is effective with certain resources under the given system settings i.e., neither protocol is dominated by others. Thus, the new resource control approach can demonstrate a more effective performance in each A tested in this figure.

Figure 6.4 presents the experimental results by varying L. Firstly, we observed that the new resource control approach is not effective with $L = [1\mu s, 15\mu s]$, $L = \{15\mu s, 50\mu s\}$ and $L = \{200\mu s, 300\mu s\}$. The reason of such a phenomenon is that with $L = [1\mu s, 15\mu s]$ and $L = \{15\mu s, 50\mu s\}$, MSRP and PWLP basically dominate MrsP under the given system settings while the systems that are schedulable under either MSRP or PWLP are similar. Therefore, systems that are not feasible with either MSRP or PWLP adopted are highly unlikely to be schedulable by other resource control approaches (i..e, MrsP and the new resource control approach). Similarly, with $L = \{200\mu s, 300\mu s\}$, MSRP and PWLP can hardly schedule any given systems if they cannot be schedulable by MrsP, and hence, makes the new resource control technique less favourable as only one protocol is effective among all the candidate locking protocols.

Table 6.1 provides evidence that supports the above discussion, where the value of A & !B indicates the percentage of systems that are schedulable

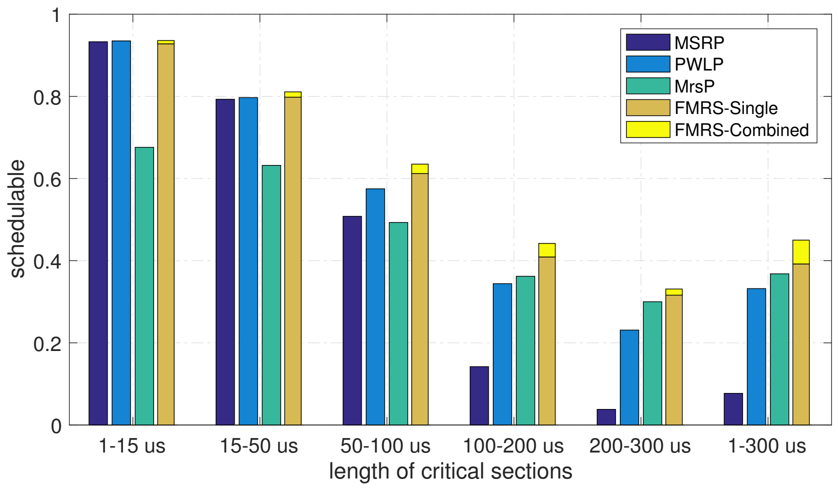


Figure 6.4: Schedulability of Systems for $M = 16$, $n = 64$, $U = 0.1n$, $A = 3$, $\kappa = 0.3$ and M Shared Resources.

with locking protocol A adopted but are not feasible under protocol B among 1000 systems generated for each L setting. With $L = [1\mu s, 15\mu s]$ and $L = [15\mu s, 50\mu s]$, the systems that can be schedulable by either MSRP and PWLP are similar (e.g., MSRP & !PWLP = 0.2% and !MSRP & PWLP = 0.7% in $L = [1\mu s, 15\mu s]$) while both protocols completely dominate MrsP i.e., !MSRP & MrsP = !PWLP & MrsP = 0%. Similarly, with $L = [200\mu s, 300\mu s]$, MrsP almost dominate both MSRP and PWLP (i.e., MSRP & !MrsP = 0.5% and PWLP & !MrsP = 0.7%).

However, with $L = [100\mu s, 200\mu s]$, although MrsP has the best performance, both MSRP and PWLP can schedule certain systems that other protocols cannot, which indicates that all the candidate locking protocols are effective under the given system settings. Therefore, the new resource control technique has a better performance with $L = [100\mu s, 200\mu s]$ (see Figure 6.4) than with the L settings discussed above (e.g., $L = [1\mu s, 15\mu s]$).

Finally, with $L = [1\mu s, 300\mu s]$, although MrsP and PWLP almost dominant MSRP in Table 6.1, adopting a combination of the candidate locking protocols to manage resources with various critical section can benefit the resource-accessing tasks, where MSRP is favourable with short resources while MrsP can benefit resources with a long critical section. Therefore, as shown in Figure 6.4, the new resource control technique demonstrates the best performance with $L = [1\mu s, 300\mu s]$ among all the settings.

Table 6.1: Schedulability of Systems with $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

L in microseconds	MSRP & !PWLP	!MSRP & PWLP	MSRP & !MrsP	!MSRP & MrsP	PWLP & !MrsP	!PWLP & MrsP
$[1, 15]$	0.2 %	0.7%	24.6%	0%	25.1%	0%
$[15, 50]$	1.7 %	1.2%	17.2%	0%	16.7%	0%
$[50, 100]$	3.3 %	11.8%	8%	10%	7.4%	0.9%
$[100, 200]$	2.8 %	24%	2.3%	25.8%	2.4%	4.7%
$[200, 300]$	0.7 %	16.5%	0.5%	24.5%	0.7%	8.9%
$[1, 300]$	0.9 %	26.9%	0.9%	30.1%	2.3%	5.5%

Summarising the above, this section has investigated the performance of the new resource control approach under various application semantics and resource characteristics. As demonstrated, systems under the resource control framework has a better schedulability than those with a single protocol adopted in most cases. Based on the discussions presented above, the new resource control approach can effectively reduce the schedulability loss due to managing shared resources that have various characteristics (e.g., the critical section length and the frequency of resource access). However, with resources that have similar characteristics, the effectiveness of the proposed resource control technique can be undermined. With such systems, a certain locking protocol could overwhelm others so that there is no need to adopt multiple resource sharing protocols.

Note, this resource sharing approach (i.e., applying a combination of resource sharing protocols to one system) does not necessary reduce the blocking time of the system. By searching for a feasible resource sharing solution, this approach aims to guarantee that each task in the system can meet its deadline rather than focusing on reducing the blocking time of the system. Although the blocking time of certain tasks does decrease with FMRS adopted, this is usually achieved by imposing additional blocking to other tasks (i.e., the ones that have met their deadlines). Thus, with FMRS adopted, there can be the case that the total blocking incurred by the system is increased (compared to the system with only one protocol applied), yet the system can become schedulable. This is also the main reason for the evaluation approach, which investigates the schedulability of the randomly generated systems rather than examining their blocking time directly. This evaluation approach is also adopted in [15,106] for investigating and comparing the performance of several spin-based resource sharing protocols on multiprocessors.

6.1.2 Success Rate

The above experiments have compared the performance of the traditional and new resource control techniques, and have illustrated the efficiency of the FMRS framework. However, such experiments cannot reveal the full expressive power of the new resource control approach. In this section, we investigate the percentage of the systems that are schedulable with the new resource control technique adopted among the systems that are deemed to be infeasible with the traditional resource control techniques. For each system setting, 1000 systems that cannot be schedulable with any of the candidate locking protocols are generated and are tested under the new resource control technique.

Table 6.2: Success Rates of Systems with $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

Critical Section Length	**Success Rate**
$L = [1\mu s, 15\mu s]$	0.1%
$L = [15\mu s, 50\mu s]$	0.3%
$L = [50\mu s, 100\mu s]$	4.7%
$L = [100\mu s, 200\mu s]$	5.0%
$L = [200\mu s, 300\mu s]$	3.6%
$L = [1\mu s, 300\mu s]$	8.3%

Tables 6.2 presents the success rates of the new resource sharing technique with varied L. Firstly, as shown in the table, the new resource control technique is not effective with $L = [1\mu s, 15\mu s]$ and $L = [15\mu s, 50\mu s]$, where few systems that are not feasible under the traditional resource control technique can become schedulable with the new technique adopted. Such results are not surprising due to the discussion given in Section 6.1.1, where MSRP and PWLP dominant MrsP under these L settings while systems that are schedulable with either MSRP or PWLP adopted are similar (see Table 6.1).

However, the new resource control approach becomes effective with $L =$

$[50\mu s, 100\mu s]$ and $[100\mu s, 200\mu s]$, where 4.7% and 5% of the unschedulable systems can become feasible with the resource control framework adopted under the corresponding L setting. The reason for this phenomenon is that with the given L settings, each candidate locking protocol is effective with certain resources (see Table 6.1) so that adopting them into a system can reduce the schedulability sacrifice effectively. In addition, this experiment also reveals that even with $L = [200\mu s, 300\mu s]$ (where MrsP almost dominates other candidate locking protocols), the new resource control approach can still schedule 3.6% of the given systems. Finally, with resources that have a critical section length across $[1\mu s, 300\mu s]$, the proposed resource control technique demonsrates the best efficiency among all the settings and can schedule 8.3% of the given systems, which are deemed to be unschedulable with any of the candidate locking protocols adopted.

Table 6.3: Success Rates of Systems with $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

Frequency of Resource Access	**Success Rate**	**Frequency of Resource Access**	**Success Rate**
$A = 1$	5.5%	$A = 26$	4.5%
$A = 6$	7.9%	$A = 31$	5.6%
$A = 11$	7.4%	$A = 36$	4.3%
$A = 16$	5.4%	$A = 41$	4.4%
$A = 21$	5.7%		

Table 6.3 provides results with varied frequency of resource accesses. As shown in the table, the new resource control technique are effective under each tested A, where up to 7.9% of the infeasible systems can become schedulable with the new approach adopted. The reason behind this phenomenon is also discussed in Section 6.1.1, where each candidate locking protocol is effective with certain resources under the given system settings.

With the experiments given above, we have revealed the full expressive power of the new resource control framework, where up to 8.3% of the unschedulable systems under the single locking protocol approach can because feasible with our resource control technique adopted under the evaluated system settings. The experiments again demonsrates that this new technique is effective and is better than the traditional resource sharing techniques.

6.2 Investigating the Performance of the Complete Resource Control Framework

Now we investigate the performance of the complete FMRS framework, which searches for feasible resource sharing, task allocation and priority ordering solutions for scheduling a given task set with shared resources in fully-partitioned systems under the FPPS scheme. In the experiments presented below, the performance of this new multiprocessor resource sharing framework is compared with the typical multiprocessor resource sharing and task scheduling (i.e., task allocation and priority ordering) approaches for tasks with shared resources under the fully-partitioned platform, where a single locking protocol (either MSRP, PWLP or MrsP) with the existing task allocation (i.e., WF, BF, FF, NF or SPA) and priority ordering (i.e., DMPO) algorithms are adopted. Then, we investigate the success rate of the complete FMRS framework for systems are not schedulable by any of the traditional multiprocessor resource sharing and task scheduling approaches.

In this evaluation, we do not enforce that the generated systems must be allocatable by all the candidate task allocation schemes so that the impact of the success rate of these task mapping algorithms is also considered. However, the success rate of the task allocations is not a major factor that affects the results. Under the system settings tested in this section, all the candidate task allocation schemes demonstrate a high success rate (a success rate of 1) except the SPA algorithm, which has a success rate of 95.8%, 94.1%, 92.0% and 91.4% respectively among 1000 generated tasks under the system settings in Figure 6.5 with $n \geq 6$. Note, the word “success” used for the task allocation schemes here merely indicates that a given task set can be successfully allocated rather than the task set can be schedulable under a given task allocation scheme.

6.2.1 Schedulability Comparison

Figures 6.5 to 6.8 present the percentage of the schedulable systems among 1000 systems generated for each system setting. In these figures, the first three bars represent the percentage of the schedulable systems with each candidate locking protocol adopted under any of the existing task allocation schemes (i.e., WF, BF, FF, NF, SPA) and priority ordering (DMPO) algorithms while the last bar (i.e., the stacked bars) gives the percentage of the schedulable systems with the FMRS framework adopted, where the "brown" bar denotes the schedulable systems with the typical resource-sharing task scheduling approaches (FMRS also considers these approaches) and the "yellow" bar indicates the systems that are infeasible under the traditional approaches but are schedulable with the new resource control and (or) the new scheduling techniques (i.e., the RCF, RLF-L, RLF-S and SBPO algorithms) in FMRS.

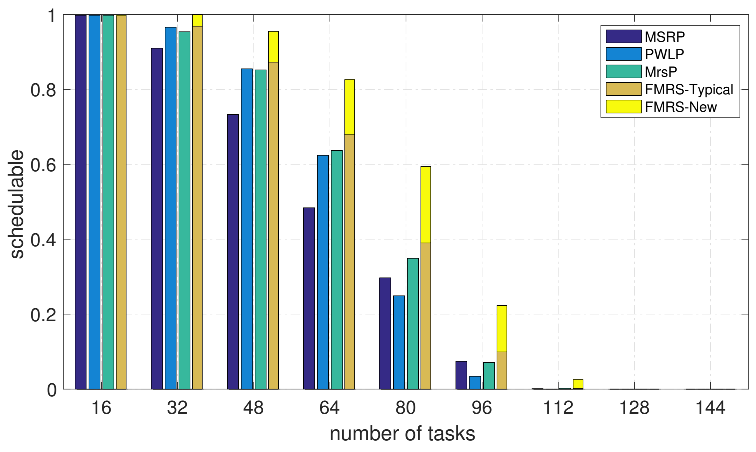


Figure 6.5: Schedulability of Systems for $M = 16$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

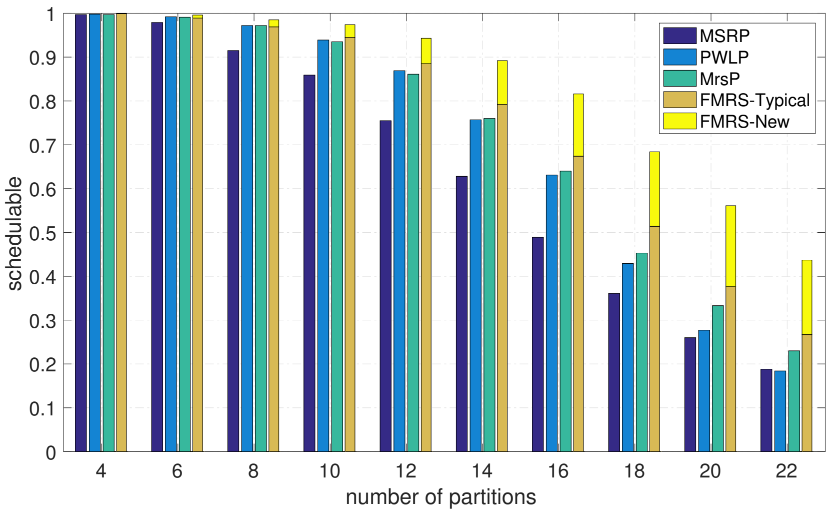


Figure 6.6: Schedulability of Systems for $n = 4M$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

The performance of the complete FMRS framework under varied number of tasks n and processors M is illustrated in Figures 6.5 and 6.6 respectively. Firstly, by cross comparing these two figures with Figures 6.1 and 6.2 (the experiments with only the WF and the DMPO algorithms assumed), we observed that the performance of each candidate locking protocol is improved by considering all the existing task allocation schemes, specially the SPA scheme. Then, with the newly-proposed task allocation and priority ordering algorithms included, the performance of the FMRS framework is further boosted, where a large number of systems that are not schedulable with the typical approaches become feasible under FMRS (i.e., the yellow bar) in most cases. Especially, in the cases where any of the resource control approaches can hardly schedule any given systems or has a low performance (e.g., $n = 96$ in Figure 6.1 and $M = 22$ in Figure 6.2), systems with the complete FMRS framework adopted can still achieve a strong schedulability due to the high effectiveness of the new task allocation and priority ordering algorithms proposed in this thesis.

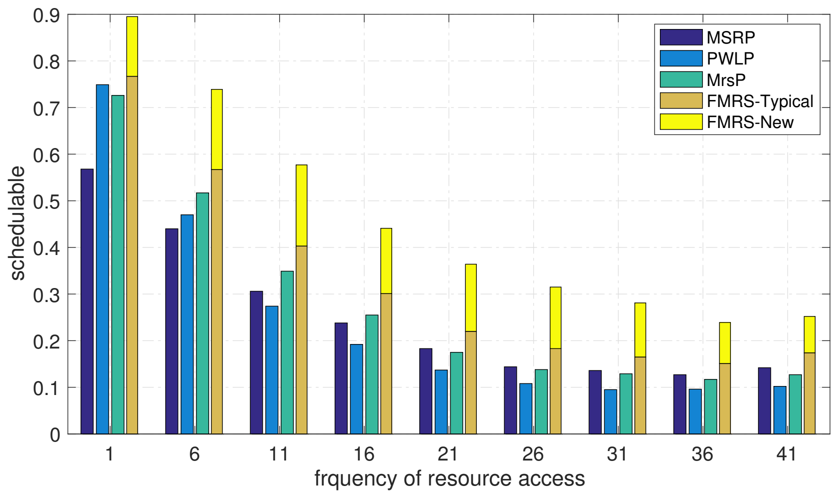


Figure 6.7: Schedulability of Systems for $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

Figure 6.7 again illustrates the efficiency of the FMRS framework, where the framework is evaluated under various frequency of resource accesses. As shown in this figure, this newly-proposed resource control framework demonstrates a high performance under each tested resource-accessing frequency, where systems with the new framework adopted have a much better schedulability than the ones with the typical approaches adopted in all cases. With the new task allocation and priority ordering algorithms, the impact of the increased frequency of resource access to the schedulability is effectively reduced (especially the RCF algorithm, see Sections 4.1.1 and 4.1.3 for details) so that each protocol can remain effective under most cases, especially PWLP, which is relatively vulnerable with a high resource-accessing frequency compared to both MSRP and MrsP. In addition, the fact that there exist both short and long resources in the given systems also makes MSRP and MrsP valuable candidate resource sharing solutions for these systems. Therefore, a high performance of resource sharing in the evaluated systems is achieved as each issue (i.e., resource sharing, task allocation and priority ordering) can be effectively addressed with the FMRS framework adopted.

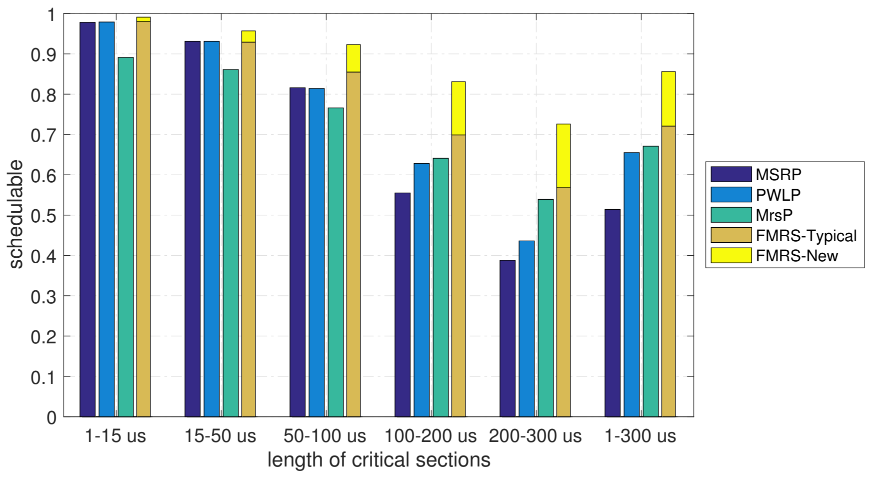


Figure 6.8: Schedulability of Systems for $M = 16$, $n = 64$, $U = 0.1n$, $A = 3$, $\kappa = 0.3$ and M Shared Resources.

Figure 6.8 gives the experimental results under various range of critical section length L. As observed, although each candidate locking protocol demonsrates a strong schedulability under the given test setting by considering all the existing task allocation schemes (especially the SPA algorithm), adopting the complete FMRS framework can still achieve a better performance in most cases. Firstly, similar to Figure 6.4, with $L = [1\mu s, 15\mu s]$ and $L = [15\mu s, 50\mu s]$, the new resource control and task scheduling approaches in FMRS do not demonstrate a huge performance difference compared to the typical approaches. Under such system settings, the blocking time for accessing shared resources can be effectively reduced by the typical multiprocessor resource-sharing task scheduling approaches so that the systems can demonstrate a high schedulability under each of the examined typical approaches, where almost all the given systems can be schedulable by each evaluated resource control with task scheduling approaches. However, by further increasing L i.e., with more blocking time imposed, the FMRS framework becomes effective and demonstrates a better performance than the traditional approaches, where a large amount of systems that are infeasible under the traditional approaches are schedulable with the newly resource control and task scheduling approaches adopted.

In addition, unlike the results given in Figure 6.4 (where the new resource control technique does not demonsrates a high effectiveness under $L = [50\mu s, 100\mu s]$ and $[200\mu s, 300\mu s]$), a high performance is achieved under the these settings with the complete FMRS framework adopted mainly due to the new resource-oriented task allocation schemes proposed in this thesis, which (1) reduces the blocking time in general and (2) maintains the effectiveness of each candidate locking protocol with resources that it is not favourable with (see discussions in Section 4.1.3 and evidence in Section 6.2.2). Accordingly, the new resource sharing technique remains effective under the complete FMRS framework as each candidate locking protocol is a valid resource sharing solution to certain shared resources in the given systems. Evidence that supports this discussion is presented in Section 6.2.2.

Summarising the above, this section provides clear evidence that the FMRS framework can effectively reduce the schedulability penalty for managing shared resources on multiprocessors due to the new resource sharing, task allocation and priority ordering solutions proposed in this thesis. As demonstrated, the FMRS framework has a better performance than the typical resource control (i.e., the single protocol approach) and task scheduling (i.e., the existing task allocation and priority ordering algorithms) techniques for fully-partitioned systems with shared resources in most cases.

6.2.2 Success Rate

Now we investigate the success rate of the complete FMRS framework with systems that cannot be schedulable by the typical multiprocessor resource control and task scheduling approaches. For each system setting, 1000 systems that cannot be schedulable by any of the candidate locking protocol under the existing task allocation and priority ordering algorithms will be generated and analysed with FMRS adopted.

Table 6.4 presents the success rate of FMRS under varied L and the percentage of the schedulable systems (among the 1000 given systems for each L) with the new resource control, task allocation and priority ordering techniques adopted. As shown in this table, the complete FMRS framework achieves a high success rate under all the tested settings, where up to 50.8% (and at least 31.4%) of the systems that are not feasible with the traditional approaches adopted can become schedulable under FMRS. Compared to Table 6.2 (which only adopts the new resource control approach), the success rates shown in this experiment are significant higher by employing the new task allocation and priority ordering algorithms, and reveals the full expressive power of the complete FMRS framework.

Table 6.4: Success Rates of Systems $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

L in microseconds	**Success Rate**	**New Resource Control**	**New Task Allocation**	**New Priority Ordering**
[1, 15]	50.8%	1.1%	48.7%	2.3%
[15, 50]	46.5%	1.5%	45.5%	1.9%
[50, 100]	43.2%	6.7%	38.1%	2.7%
[100, 200]	36.9%	8.3%	33.9%	2.2%
[200, 300]	31.4%	5.4%	29.2%	2.5%
[1, 300]	37.0%	11.9%	31.3%	3.1%

In addition as shown in the table, each of the newly-proposed resource sharing, task allocation and priority ordering solutions in FMRS is effective, especially the candidate task allocation schemes, which can schedule up to 48.7% of the given systems. Notably, by cross comparing Tables 6.4 and 6.2, we observed that the new resource control technique has a better performance under the complete FMRS framework, especially with $L = [1\mu s, 300\mu s]$, where up to 11.9% of the given systems (i.e., the unschedulable systems with the traditional approaches) are feasible with the new resource control technique adopted. The reason for this phenomenon is that with the new task allocation and priority ordering algorithms adopted, the blocking time due to resource-sharing can be effectively decreased so that the impact of the resource characteristics to the performance of the resource sharing protocols are also reduced.

Accordingly, as each of the candidate locking protocols is an effective solution under most cases, a high performance of the new resource control technique can be achieved.

Table 6.5 provides evidence that supports the above discussion by investigating the dominance of each candidate locking protocols under all the candidate task allocation schemes in FMRS, where A & ! B indicates the percentage of systems (among 1000 generated systems) that can be schedulable by protocol A while are infeasible with protocol B adopted under any of these task allocation schemes adopted, which include the newly-proposed RCF, RLF-L and RLF-S algorithms.

Table 6.5: Schedulability of Systems with $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$ and M Shared Resources.

L in microseconds	MSRP & !PWLP	!MSRP & PWLP	MSRP & !MrsP	!MSRP & MrsP	PWLP & !MrsP	!PWLP & MrsP
$[1, 15]$	0.2 %	0.1%	4.8%	0%	4.7%	0%
$[15, 50]$	0.3%	0.5%	5.3%	0%	5.5%	0%
$[50, 100]$	1.3 %	1.2%	4.5%	0.1%	4.3%	0.3%
$[100, 200]$	2.2 %	4.0%	2.8%	4.4%	2.5%	2.3%
$[200, 300]$	4.1 %	3.5%	2.5%	6.1%	1.8%	6.0%
$[1, 300]$	2.8 %	5.9%	3.3%	6.0%	2.9%	2.5%

Firstly, under $L = [1\mu s, 15\mu s]$ and $[15\mu s, 50\mu s]$, although both MSRP and PWLP basically dominate MrsP, the percentage of both MSRP & !MrsP and PWLP & !MrsP is largely reduced compared the results shown in Table 6.1, which is conducted with the WF scheme only. This phenomenon indicates that with the new allocation schemes, MrsP demonsrates a better performance with short resources (resources that MrsP is not favourable with) compared to this protocol under the WF scheme.

By further increasing L, we observed that each of the candidate locking protocol cannot dominate another, and all these resource sharing solutions are effective (i.e., can schedule certain systems where others cannot) when managing certain resources under each tested L. Compared to the results in Table 6.1, the dominance of MrsP is reduced while the performance of MSRP and PWLP is increased under systems with either long resources (e.g., $L = [200\mu s, 300\mu s]$) or resources that have varied critical section length (i.e.,

$L = [1\mu s, 300\mu s]$) by considering all the candidate resource allocation schemes. Therefore, a high effectiveness of the new resource control technique can be achieved in the complete FMRS framework as each of the candidate resource control solution remains effective in most cases.

Table 6.6: Success Rates of Systems $M = 16$, $n = 64$, $U = 0.1n$, $\kappa = 0.3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

Frequency of Resource Access	**Success Rate**	**Frequency of Resource Access**	**Success Rate**
$A = 1$	54.9%	$A = 26$	16.2%
$A = 6$	34.7%	$A = 31$	13.9%
$A = 11$	29.1%	$A = 36$	10.4%
$A = 16$	20.0%	$A = 41$	9.4%
$A = 21$	18.5%		

Table 6.7: Success Rates of Systems $n = 4M$, $U = 0.1n$, $\kappa = 0.3$, $A = 3$, $L = [1\mu s, 300\mu s]$ and M Shared Resources.

Number of Processors	**Success Rate**	**Number of Processors**	**Success Rate**
$M = 4$	79.7%	$M = 14$	43.2%
$M = 6$	62.8%	$M = 16$	37.4%
$M = 8$	52.1%	$M = 18$	34.1%
$M = 10$	45.4%	$M = 20$	27.6%
$M = 12$	48.3%	$M = 22$	23.2%

Finally, Tables 6.6 and 6.7 present the success rates of the complete FMRS framework (with the new resource control, task allocation and priority ordering techniques only) under varied resource-accessing frequency and degree of parallelism. The results given in these two figures again demonstrate a high effectiveness of FMRS under the tested system settings in most cases, where up to 54.9% and 79.7% systems that are infeasible under the traditional approaches can become schedulable with FMRS adopted in these two experiments. The reasons for this phenomenon have been discussed in above sections. Notably, even in the cases where each of the resource sharing protocol has a low performance (e.g., $M = 22$ and $A = 41$ in Figures 6.6 and 6.7

respectively), the FMRS framework is still effective and can schedulable up to 9.4% and 23.2% of the given systems (the unschedulable systems under the traditional approaches) in the above experiments.

6.3 Summary

This chapter has presented a comprehensive investigation towards the performance of the Flexible Multiprocessor Resource Sharing framework proposed in this thesis and has satifised the success criteria SC-5 given in Section 1.4. Firstly, Section 6.1 has evaluated the effectiveness of the new resource control technique for multiprocessor systems under various application semantics and resource characteristics, where a combination of resource sharing protocols are adopted into a single system for managing shared resources. Then in Section 6.2, the performance of the complete FMRS framework is investigated, which also provides the task allocation and priority ordering solutions to the given systems with the newly-proposed resource-oriented task allocation and the search-based priority ordering algorithms included.

As illustrated by the experiments, the new resource control and new task scheduling approaches adopted in the FMRS framework can effectively reduce the schedulability penalty due to accessing shared resources in fully-partitioned systems, and has a better performance than the traditional resource control and task scheduling approaches for multiprocessor systems with shared resources. In addition, with FMRS adopted (especially the complete FMRS framework), a significant amount of systems that are deemed to be unschedulable under the traditional multiprocessor resource control and task scheduling approaches are feasible due to the minimised schedulability loss for accessing shared resources.

Further, although the FMRS framework only considers the spin-based synchronisation approach (recall the decisions made in Section 3.1), this resource control framework can demonstrate strong performance with long critical sections assumed (see Figures 6.4 and 6.8) by (1) including the preemptable spin-based protocols (i.e., PWLP and MrsP) as the candidate resource control solutions and (2) adopting the new resource-oriented task allocation schemes and the search-based priority ordering algorithm as the candidate task mapping and priority ordering solutions. A detailed discussion of spin-based synchronisation approach with long critical sections is presented in Section 3.4

with evidence demonstrating that the preemptable spin-based locking approach has a strong performance under long critical sections. In addition, with the new resource-oriented task allocation schemes (especially the RLF-L scheme, which aims to reduce the blocking due to long critical sections) and the new search-based priority ordering algorithm adopted, a further performance boost of FMRS can be achieved in general (so that better support for long resources as well). Therefore, this FIFO spin-based multiprocessor resource control framework can provide a strong support for resources with a long critical section.

Based on this evaluation, we confirm that the newly-proposed FMRS framework is effective and is better than the typical resource control and task scheduling approaches for fully-partitioned systems with the presence of shared resources in the general case.

Chapter 7

Conclusion

In this thesis, we have investigated three major factors (i.e., multiprocessor resource sharing protocols, task allocation schemes and priority ordering algorithms) that can directly affect the schedulability of multiprocessor systems in the presence of shared resources. Then, a Flexible Multiprocessor Resource Sharing framework (namely FMRS) is proposed for scheduling the resource-sharing tasks in fully-partitioned systems under the FPPS scheme, which aims to provide a combination of feasible (i.e., schedulable) resource sharing, task allocation and priority ordering solutions to any given task sets under the fully-partitioned platforms.

In Chapter 2, a detailed review of the works related to this thesis was firstly presented, which includes the descriptions of the background of real-time systems, the major resource sharing protocols for multiprocessor platforms, the schedulability tests for multiprocessor resource sharing protocols, the utilisation-based and resource-oriented task allocation schemes, and the major priority ordering algorithms. Based on this review, the objective of this thesis is given in Section 1.4 as a set of success criteria (which form the complete Flexible Multiprocessor Resource Sharing framework that can facilitate resource sharing in fully-partitioned systems), as restated below.

SC-1 *A new schedulability analysis framework that can be applied to systems with the presence of multiple resource sharing protocols, which includes a response time analysis that can provide more accurate results than that of their original analysis, and a pluggable run-time overheads analysis that takes the run-time costs from both the underlying operating system and the resource sharing protocols into account.*

The success criterion SC-1 is satisfied in Chapter 3. In this chapter, the candidate resource sharing protocols for the proposed resource control framework were firstly determined (i.e., MSRP, PWLP and MrsP). Then, new schedulability tests for each candidate resource sharing protocols were developed with novel techniques for analysing the blocking time due to accessing shared resources. In addition, a run-time overheads analysis was developed for each locking protocol and can be integrated into the new schedulability tests effectively, which includes the costs of context switches, protocol implementations and the additional facilities carried in PWLP and MrsP (i.e., the cancellation mechanism and the helping mechanism).

Finally, Section 3.3 integrated the new schedulability tests (combined with the run-time overheads analysis) and developed a schedulability analysis framework that supports analysing systems with the presence of multiple candidate locking protocols. This analysis framework is flexible as it does not mandate the presence of all the supported protocols while new protocols can be effectively integrated into this framework as long as an RTA-based analysis is presented.

SC-2 *Resource-oriented task allocation schemes that are independent from the resource sharing protocols, where each task allocation scheme assigns tasks to processors based on certain characteristics of the shared resources, such as the length of critical sections and the degree of resource contention.*

This success criterion is satisfied in Chapter 4. In this Section 4.1, three new resource-oriented task allocation schemes were developed, namely RCF, RLF-L and RLF-S, where each allocation scheme aims to reduce the blocking time due to accessing resources with certain characteristics (i.e., the frequency of resource accesses and the length of critical sections). The new allocation schemes map tasks based on the resource characteristics and task utilisations, and are not subject to any specific resource sharing protocols (i.e., can be adopted with any locking protocols assumed).

SC-3 *A new priority ordering algorithm that inherits the philosophy of the OPA algorithm i.e., search-based, but is fully compatible with the schedulability tests where DMPO is not optimal and OPA cannot be applied,* *such as the one in [106] and the new analysis framework in SC-1.*

Chapter 4 also satisfies the success criterion SC-3. In Section 4.2.3, the Slack-Based Priority Ordering Algorithm was developed based on philosophy of OPA and its variant RPA to provide a search-based priority assignment approach. To be compatible with the schedulability tests that have response time dependency (e.g., the new schedulability tests in Chapter 3 and the ILP-based analysis in [106], where the response time of a task depends on the response times of potentially all other tasks in the system), such dependency is removed by SBPO while examining the remaining slack of each task, and is reconsidered later on when computing the response time of each task in the given system.

SC-4 *A flexible multiprocessor resource sharing framework that takes a system as the input, and aims to search for a schedulable solution (with the new schedulability analysis in SC-1) of resource sharing, priority ordering and task allocating issues to the given system, which include a combination of locking protocols to control each resource in the system, a task allocation scheme that can benefit resource sharing and a feasible priority ordering decided via examining all the candidate solutions provided by this framework.*

This is satisfied in Chapter 5. With the candidate resource sharing, task allocation and priority ordering solutions determined in Chapters 3 and 4, a complete framework (i.e., FMRS) for scheduling resource-sharing tasks in fully-partitioned systems with fixed priorities was developed. By giving a set of tasks and resources with detailed resource-usage, the framework aims to search for a schedulable system with each task assigned with an allocation and a priority, and each resource managed by a designated candidate locking protocol. The decisions of which resource sharing protocols, task allocation scheme and priority ordering algorithm should be adopted to a given system is computed off-line (i.e., before run-time) by the genetic algorithm technique with the analysis framework in Section 3.3 as the fitness functions.

SC-5 *An evaluation with evidence that the resource sharing framework proposed in SC-4 demonstrates at least equal or better schedulability than that of the typical real-time resource control approaches, where one resource sharing protocol is adopted to manage all shared resources in a* *system with the existing task allocation and priority ordering approaches applied.*

The success criterion SC-5 is satisfied in Chapter 6. In this chapter, the newly-proposed resource control framework is evaluated under systems with various application semantics and resource characteristics. In Section 6.1, the performance of the new resource control technique was investigated and compared with the typical resource control technique, where a single resource sharing protocol is adopted to manage all the shared resources in a system. The investigation demonstrates that the new resource control technique is effective and can lead to better schedulability compared to the single protocol approach in most cases.

In Section 6.2, the performance of the complete Flexible Multiprocessor Resource Sharing framework was investigated, where the framework also provides task allocation and priority ordering solutions with the newly-proposed task allocation and priority ordering algorithms included. The experiments provide clear evidence that the schedulability sacrifice due to accessing shared resources on the fully-partitioned platform can be effectively reduced with the new resource control framework adopted in most cases so that better schedulability can be obtained compared to the typical approach for scheduling resource-sharing tasks in fully-partitioned systems, which adopts only one locking protocol with the traditional task allocation and priority ordering algorithms to the systems.

Based on the discussion above, the materials presented in this thesis have met each of the success criteria given above, which demonstrated the thesis hypothesis given in Section 1.3, as restated below.

With shared resources, the schedulability of a multiprocessor real-time system can be undermined due to the considerable amount of blocking time. Such schedulability penalty can be reduced by adopting (i) a combination of appropriately chosen resource sharing protocols, where each protocol only controls certain resources; (ii) new resource-orientated task allocation schemes with full knowledge of the usage and characteristics of each resource; and (iii) a search-based priority assignment that is compatible with schedulability tests where the Deadline Monotonic Priority Ordering is *not optimal and Audsley's Optimal Algorithm cannot be applied. The decisions of which resource sharing protocols, task allocation scheme and priority ordering algorithm that can lead to a schedulable system are made off-line by a genetic algorithm.*

7.1 Major Contributions and Key Findings

To investigate the performance of multiprocessor resource sharing, the impact of shared resource control, task allocation and priority ordering to the schedulability of multiprocessor systems with shared resources has been studied and a generic FMRS framework that can facilitate resource sharing for fully-partitioned systems has been developed in this thesis. During this research, improvements, new algorithms and novel approaches for scheduling tasks with shared resources have been proposed to improve the efficiency of resource sharing on multiprocessors. This section summarises and reviews the major contributions and key findings of this thesis.

Multiprocessor Resource Sharing Protocols

The multiprocessor resource sharing protocols define the behaviours of tasks when accessing shared resources and have a direct impact to the schedulability of systems with shared resources. To determine the appropriate resource sharing protocols for the proposed resource control framework, a comprehensive review of the resource sharing protocols for multiprocessor systems is firstly presented in Section 2.5, which contains detailed descriptions of 9 major multiprocessor locking protocols. Based on this review and the rational presented in Section 3.1, we decided to focus on the protocols with FIFO spin locks as the candidate resource sharing solutions for the FMRS framework, which are MSRP, PWLP and MrsP. Then, these candidate resource sharing protocols were studied with the following contributions.

- New analysing techniques for systems with MSRP, PWLP or MrsP adopted, which are less pessimistic and more accurate compared to their original analysis and require less computation expenses compared to the tests (e.g., the ILP-based analysis in [106]) with relatively expensive analysing techniques. In addition, the new schedulability tests are extended to support analysing the heterogeneous and nested resource accesses (via group locks) for a wider use scenario.

- The NP-section for MrsP's helping mechanism and a pluggable migration cost analysis for MrsP's new analysis, which provides more efficient migration behaviours and bounds the cost of migrations for more accurate schedulability results respectively.
- A complete run-time overheads-aware schedulability analysis framework for systems with multiple protocols in use, which contains the techniques for bounding the run-time overheads incurred from both the underlying operating systems and the protocol implemenations.
- Fully functional MSRP, PWLP and MrsP implementations under the P-FP scheduler in Litmus^{RT}.
- An investigation towards the correctness and efficiency of the MrsP in fully-partitioned systems.
- An investigation towards the schedulability of MSRP, PWLP and MrsP, including the impact of the run-time overheads and the expenses for using the proposed schedulability tests.

With above studies, we have revealed the performance of the examined resource sharing protocols (via increasing the accuracy of schedulability results and accounting for the run-time overheads), where no resource sharing protocol can dominate other candidate protocols and the performance of each locking protocol largely depends on the given application semantics, resources characteristics and the resource usage.

As observed, MSRP is favourable with short resources while MrsP can benefit resources with a long critical section. As for PWLP, its performance can be less affected with various critical section length, but is sensitive to the degree of parallelism and the frequency of resource access, where it can demonstrate strong performance with a low degree of parallelism and a low frequency of resource access. In addition, we observed that with short resources, MSRP basically dominates MrsP while MSRP can hardly schedule any systems where MrsP cannot under resources with a long critical section. However, with resources that have varied critical section length, MrsP is clearly the best choice based on the experiments given in this thesis.

Such observations directly motivated the development of the new resource control technique and the new resource-oriented task allocation schemes employed in the FMRS framework.

Resource-Oriented Task Allocation Schemes

According to the literature review, allocating tasks into a fully partitioned system is a bin-packing NP-hard problem and is usually addressed by heuristic approaches, such as the Worst-Fit and the Best-Fit allocation schemes. However, these algorithms allocate tasks merely based on utilisations and do not consider the usage of shared resources, which can lead to high remote blocking as tasks that share a same resource can be mapped into multiple processors. In addition, there exist several task allocations (i.e., the SPA and BPA algorithms) that take shared resources into account when allocating tasks. However, BPA is an analysis specific algorithm (i.e., requires the weight and attraction functions of the locking protocol adopted) and currently only supports MPCP while SPA is subject to a certain degree of pessimism as it only considers the total utilisation of each shared resource. This discussion leads to the following contribution:

- Three new resource-oriented task allocation schemes (i.e., the RCF, RLF-L and RLF-S algorithms) that aim to reduce the remote blocking due to certain type of resources by taking the resource characteristics into account and are fully independent from resource sharing protocols.

As demonstrated by the evaluation, these newly-proposed task allocation schemes demonstrate better performance (with the candidate locking protocols and the DMPO algorithm assumed) than the existing task allocation schemes (including the SPA algorithm) in all cases. In addition, we observed that although these algorithms are developed as generic task allocation solutions, each of them can benefit certain locking protocols. With an appropriate task allocation scheme adopted, the resource sharing protocols can demonstrate a strong performance even with the shared resources that are less favourable for that locking protocol. For instance, the RLF-L reduces the blocking time caused by resources with a long critical section so that the performance of MSRP is boosted with RLF-L adopted even being applied to systems with the presence of such resources.

Priority Ordering for Multiprocessors Resource-Sharing Tasks

The priority ordering of tasks in systems with shared resources defines the execution eligibility of each resource-accessing task so that it can directly affect the performance of multiprocessor resource sharing. During the research towards the priority assignments for multiprocessor resource-sharing tasks, the following were observed.

- The DMPO algorithm is optimal in fully-partitioned FPPS systems with shared resources under the original schedulability tests of MSRP and MrsP.
- The DMPO algorithm is **not** optimal in fully-partitioned FPPS systems with shared resources under the new schedulability tests of MSRP, PWLP and MrsP proposed in this thesis.
- The existing search-based algorithms (e.g., the OPA and RPA algorithms) are not applicable to the new schedulability tests, where compromises must be made for compatibility. However, such comprises introduce considerable amount of pessimism and significantly undermine the performance of these priority ordering algorithms.

Based on the above findings, this thesis proposes:

- A search-based priority ordering algorithm named the Slack-based Priority Ordering algorithm that is fully compatible with the schedulability tests where the response time of a task depends on the response times of potentially all other tasks in the system.

According to the evaluation, there exist no optimal priority ordering algorithms (including the SBPO algorithm) in fully-partitioned systems with shared resources under the new schedulability tests. However, as observed, the SBPO algorithm demonsrates a better performance than other priority ordering algorithms in most cases.

New Approach for Scheduling Resource-Sharing Tasks in Fully-Partitioned Systems

With the above findings and contributions, a complete framework for scheduling resource-sharing tasks in fully-partitioned systems is developed in this thesis and is named as the Flexible Multiprocessor Resource Sharing framework. In FMRS, 3 locking protocols (MSRP, PWLP and MrsP), 8 task allocation schemes (the WF, BF, FF, NF, SPA, RCF, RLF-L and RLF-S algorithms) and 2 priority ordering algorithms (the DMPO and SBPO algorithms) are used as the candidate resource sharing, task allocation and priority ordering solutions scheduling any given task sets with shared resources on fully-partitioned platform with FPPS scheme.

Notably, with the observations obtained in the research of multiprocessor resource sharing protocols, a novel resource control approach for multiprocessors is proposed in this framework, where each resource in this system can be controlled by one of the candidate locking protocols designated by the framework. In addition, as there exist no optimal solutions for resource control, task allocation and priority ordering issues in multiprocessor systems with shared resources, a genetic algorithm is adopted to provide a heuristic-based approach for searching for the feasible solutions to these issues for any given systems.

As demonstrated in Chapter 6, the schedulability penalty for managing shared resources in fully-partitioned systems can be reduced effectively under FMRS in most cases so that better schedulability could be achieved compared to the typical resource control and task scheduling approaches. With the FMRS framework adopted, systems that are deemed to be unschedulable under the typical resource control and task scheduling approaches could become feasible due to the new approaches for scheduling resource-sharing tasks on multiprocessors.

7.2 Future Work

This section describes possible future research towards the work presented in this thesis, as described below.

Analysing Nested Resource Accesses with Ordered Locks

In Section 3.2, new schedulability tests of MSRP, PWLP and MrsP are developed based on the assumption of homogeneous and non-nested accesses. Later on, this restriction is removed in Appendix A, where the new tests are extended to support heterogeneous and nested resource accesses. To manage nested resources, the group locks are adopted instead of the ordered locks as they are more schedulability test friendly (i.e., only requires minor modifications to the schedulability tests).

However, considering the ordered locks is worthwhile, especially in multiprocessor systems, as managing a group of shared resources by one lock (i.e., a group lock) serialises the accesses to nested resources so that the degree of parallelism can be undermined. In [49], the preliminary approach for analysing

MrsP systems with the presence of nested resources under the ordered locks are proposed, which demonstrates the feasibility of supporting the analysis of ordered nested resources in our new schedulability tests. In future, we aim to extend the schedulability analysis of MSRP and PWLP to also support the accesses to ordered nested resources for a wider use scenario.

Extending the BPA Algorithm

As described in Section 2.6.1.2, the BPA task allocation scheme is an analysis-specific algorithm, where the weight and attraction functions of the resource sharing protocol must be provided if this scheme is adopted to a system with that protocol assumed. The current version of the BPA algorithms supports the use of MPCP [81]. As the FMRS framework contains multiple resource sharing protocols that are not yet supported by the current version of BPA, the allocation schemes that are independent from the locking protocols are adopted in current version of FMRS.

However, by examining the exact blocking time (via the weight and attraction functions) of the unallocated tasks with a given allocation, this algorithm can effectively reduce the remote blocking time due to accessing shared resources and can provide better performance than the SPA algorithm [81]. Thus, it is worthwhile to compare the performance of the BPA algorithm with the new task allocation schemes proposed in this thesis by extending BPA to support other locking protocols (e.g., MSRP PWLP and MrsP studied in this thesis). As the schedulability analysis of these resource sharing protocols are available, developing the weight and attraction functions is not challenging. Then, if the BPA algorithm can demonstrate a strong performance (compared to the algorithms examined in this thesis), this algorithm will also be included into the candidate task allocation solutions in FMRS.

Optimising the SBPO Algorithm

In Section 4.2.3, the SBPO algorithm is proposed to provide a search-based priority ordering algorithm that is fully compatible with the new schedulability tests developed in this thesis. As a newly-developed algorithm with complicated approaches (e.g., the extended response time calculation approach and the facilities to minimise the pessimism due to the compromises made for compatibility), we mainly focus on the functionality of this algorithm and aim to deliver a practicable SBPO algorithm in this work.

However, this algorithm could be further optimised to achieve a better performance. For instance, similar to the OPA and RPA algorithms, the current version of SBPO orders the processors by their indexes and starts the priority assignment from the first processor (i.e., P_0). However, as the response times of all the unexamined tasks are assumed to be their deadlines, the calculations of the remaining slacks of the tasks in the first processor is more pessimistic (i.e., less accurate) than that of the remaining slacks of tasks in other processors. Thus, there could exists a relationship between the performance of this algorithm and ordering of processors. In future, we aim to investigate such a relationship and to propose a more efficient processor ordering for a better performance.

In addition, as suggested in [87], the response time calculation is extended to $5 \times D$ (i.e., $\eta = 5$) in this algorithm. Although this setting is proved to be effective in Section 4.2.4, there could exist other settings for this parameter that can lead to more accurate remaining slack calculations and is worth investigation.

Optimising the GA parameter Settings for FMRS

In Chapter 5, the GA-based FMRS framework is proposed and the settings of the GA parameters are decided mainly based on the research and suggestions in [87], where a set of experiments are conducted to investigate the efficiency of various GA parameter settings when addressing a task allocation issue in real-time systems with networks. As demonstrated in Chapter 6, the current settings of the GA parameters in FMRS are effective. However, whether there exist better (or optimal) GA parameter settings that can lead to higher efficiency of the proposed framework remains unknown. Thus, an evaluation that investigates the efficiency of various GA parameter settings for FMRS is desirable and could lead to further performance improvement.

Extending FMRS for Improved Robustness and Scalability

As described in Chapter 5, the current version of the FMRS framework aims to provide feasible (i.e., schedulable) resource sharing, task allocation and priority ordering solutions to fully-partitioned systems with shared resources, where the genetic algorithm is finished as long as a feasible solution is found for the given system. However, as discussed in Section 5.4, this framework can be easily extended to further improve the robustness and scalability of a given fully-partitioned system with shared resources, assuming feasible solutions for that system have been found.

To achieve this, the fitness functions should be modified first to also take the remaining slacks of the schedulable tasks into account. For the solutions with a F_D value of 0 (i.e., where each task in the system has met its deadline), the F_D^η function then returns the sum of $R - D$ of all tasks and a smaller F_D^η value represents a better solution. For the solutions with $F_D > 0$, the calculation of F_D^η remains identical to the approach given in Section 5.3.1.

In addition, if feasible solutions are found, the GA-based search will be not be finished immediately. Instead, the framework will focus on these feasible solutions and attempt to evolve better solutions (which can lead to improved robustness and scalability of the given system) within the pre-defined computation expenses (i.e., the maximum generation limit). The detailed approaches for measuring the robustness and scalability of a given system is referred to [43], which proposes a search-based task allocation algorithm for similar purpose via the simulated annealing technique, but without the presence of shared resources.

Heuristic Searching for Priorities and Allocations in FMRS

In the current version of FMRS, the resource sharing solutions are obtained by examining each individual resource via the heuristic-based searching approach while the allocations and priorities of all tasks are assigned by one of the candidate task allocation and priority ordering solutions. However, as demonstrated in Chapter 4, there exists no optimal task allocation and priority ordering solutions in multiprocessor systems with shared resources. Thus, by intuition, adopting the heuristic-based approaches to search for the allocation and priority for each individual task (similar to the algorithms proposed in [87] and [43]) could lead to better results and is worth investigation.

To achieve this, the chromosome encoding should be updated to also consider the allocation and priority of each task in the system. Then, the candidate task allocation and priority ordering algorithms in FMRS can be removed, as the allocation and priority of each task are now obtained via the selection and evolution process. However, these candidate solutions can be used as guidance for producing the first generation with better quality than assigning the allocations and priorities randomly. However, one major concern of this approach is the computation expenses, where there could exist a significant number of possible solutions so that adopting such a framework can be considerably expensive.

Investigating the Efficiency of Simulated Annealing for Multiprocessor Resource Sharing Issues

As discussed in Section 5.2.1, both the genetic algorithm and the simulated annealing technique have been successfully practised in addressing the task allocation and priority ordering issues in real-time systems [42, 43, 87]. However, no discussion towards the comparison between these two algorithms is presented in the above studies. Admittedly, the research towards the efficiency of these heuristic-based searching approaches is out of the research scope of our and the above works. However, it is interesting to investigate whether there exist performance differences between the GA and SA algorithms and to understand the rational of such difference (if exist) when addressing the resource sharing, task allocating and priority ordering issues in real-time systems in a general case. Such a research can provide valuable suggestions when a heuristic-based searching approach is required in future work.

7.3 Closing Remarks

Managing shared resources on multiprocessor platforms often causes considerable schedulability loss due to the prolonged blocking time, and hence, leads to poor schedulability of multiprocessor real-time systems with tasks accessing shared resources. In addition, there exist no optimal resource sharing protocols, task allocation schemes and priority ordering algorithms for multiprocessor systems with shared resources, where different choices of the resource sharing protocols, the task allocation schemes and the priority ordering algorithms can have various impact on the efficiency of multiprocessor resource sharing under different system semantics, resource characteristics and the usage of shared resources.

This thesis contends that, compared to the typical resource control, task allocation and priority ordering approaches (where only one resource sharing protocol is employed to manage all the resources in a system with the existing task allocation and priority ordering algorithms adopted), the schedulability sacrifice due to resource sharing in multiprocessors systems can be reduced by adopting (1) a combination of resource sharing protocols, where each protocol only manage certain resources in the system; (2) a resource-oriented task allocation scheme with full knowledge of the characteristics and usage of the shared resources; and (3) a search-based priority ordering algorithm that is independent from the resource sharing protocols and the corresponding schedulability tests.

In this thesis, a Flexible Multiprocessor Resource Sharing framework for scheduling resource-sharing tasks in fully-partitioned systems under the fixed-priority preemptive scheduling scheme is proposed with new approaches for managing shared resources and assigning allocations as well as priorities to resource-sharing tasks. By giving a set of tasks with detailed resource-usage, this framework aims to provide feasible resource sharing, task allocating and priority ordering solutions via a genetic algorithm based on the new schedulability analysis framework developed in this thesis, which supports the analysis of systems with multiple candidate resource sharing protocols in the FMRS-framework working in collaboration simultaneously .

The evaluation shows that the FMRS framework can effectively reduce the schedulability sacrifice due to multiprocessor resource sharing compared to the typical resource control and the tasks scheduling (i.e., task allocation and priority ordering) approaches for the fully-partitioned platform with the FPPS scheme. As demonstrated by experiments, systems that are deemed to be infeasible under the typical multiprocessor resource sharing and the traditional task scheduling approaches can become schedulable due to the improved efficiency of multiprocessor resource sharing with FMRS adopted.

Appendices

Appendix A

Supporting Heterogeneous and Nested Resource Accesses

In Section 3.2, new schedulability tests for MSRP, PWLP and MrsP were developed, which aim to provide less pessimistic as well as more accurate response time bounding for each task in a given system than that of their original analysis (if they exist) described in Sections 2.5. However, for the ease of presentation, the new schedulability tests are developed based on the assumptions of (1) the homogeneous resource accesses (i.e., the cost for executing a resource is identical for any task in any access) and (2) the non-nested resource accesses (i.e., each task can lock at most one resource at any given time). However, such assumptions can undermine the usability of the newly-proposed schedulability tests due to the restricted resource-accessing model. In this appendix, these limitations are removed and the new schedulability tests are extended to support both the heterogeneous and the nested resource accesses.

A.1 Analysing Heterogeneous Resource Accesses

We first extend the new schedulability tests to support the heterogeneous resource accesses. In the ILP-based analysis [106], each task can have a different execution cost on each resource. However, for a given task, the cost for executing a resource is identical in each access. In this work, a more flexible resource-accessing model is assumed, where the cost of executing a resource can vary in each access of a given task. To facilitate the analysis of such a resource-accessing model, the notation $c_x^k(n)$ is introduced to denote the pure execution cost (without any delay) of τ_x's n$^{\text{th}}$ access to r^k, where $1 \leq n \leq N_x^k$.

In addition, the notion Lcs_x^k is introduced to denote a list of execution costs of the requests issued by τ_x to r^k in one release, where the execution costs are ordering in a non-increasing fashion. For instance, τ_a requests r^1 3 times during each release and the access costs are $c_a^1(1) = 3$, $c_a^1(2) = 5$ and $c_a^1(3) = 7$ units of time respectively, then $Lcs_a^1 = \{7, 5, 3\}^{dList}$. With a duration l and a jitter μ specified, a new notation $Lcs_x^k(l, \mu)$ is introduced, which is a decreasing ordered list of the pure execution costs of the requests to r^k issued by τ_x within the given duration and jitter. Recall the example above, if τ_a is released two times during a given duration 10 and a jitter 5, then $Lcs_a^1(10, 5) = \{7, 7, 5, 5, 3, 3\}^{dList}$.

Equations (A.1) and (A.2) demonstrate the approaches for bounding Lcs_x^k and $Lcs_x^k(l, \mu)$, where $a \,\%\, b$ denotes the remainder of $a \div b$. For the function $Lcs_x^k(l, \mu)$, the execution costs are obtained by the index $n \,\%\, N_x^k + 1$. Consider the same example above, where $N_a^1(10, 5) = 6$, $N_a^1 = 3$ and $1 \leq n \leq 6$, the value of $n \,\%\, N_a^1 + 1$ is $\{2, 3, 1, 2, 3, 1\}$ with n =$\{1, 2, 3, 4, 5, 6\}$ so that $Lcs_a^1(10, 5)$ =$\{5, 7, 3, 5, 7, 3\}$. In addition, as the elements in $Lcs_x^k(l, \mu)$ will be ordered decreasingly at last (i.e., becomes $\{7, 7, 5, 5, 3, 3\}^{dList}$ eventually), such an approach will not undermine the correctness of the analysis but can ease the analytical expression.

$$Lcs_x^k = \{c_x^k(n) | 1 \leq n \leq N_x^k\}^{dList} \tag{A.1}$$

$$Lcs_x^k(l, \mu) = \{c_x^k(n \,\%\, N_x^k + 1) | 1 \leq n \leq N_x^k(l, \mu)\}^{dList} \tag{A.2}$$

In addition, a new notation $Lcsp_m^k(l)$ is introduced, which denotes those costs belonging to the accesses to r^k from the tasks executing on a given processor P_m within a duration l, where

$$Lcsp_m^k(l) = \{Lcs_j^k(l, R_j) | \tau_j \in G(r^k) \wedge P(\tau_j) = P_m\}^{dList} \tag{A.3}$$

For each list described above, $Lcs_x^k(n)$, $Lcs_x^k(l, \mu)(n)$ and $Lcsp_m^k(l)(n)$ returns the value on the nth position of the list. If the list is empty or the nth position does not exist, then a value of 0 will be returned. The list begins (has the greatest value stored) on position 1. For a given list L, its size can be obtained by function $|L|$. Accordingly, $|Lcs_x^k| = N_x^k$ and $|Lcs_x^k(l, \mu)| = N_x^k(l, \mu)$. Table A.1 summarised the above notations introduced for analysing the heterogeneous resource accesses.

Table A.1: Notations for Analysis the Heterogeneous Resource Accesses

$c_x^k(n)$	The pure execution cost of τ_x's nth access to r^k.
$L = \{\}^{dList}$	A given list L with a set of positive values ordered by a non-increasing fashion.
$L(n)$	The nth element from a given list L. A value of 0 is returned if the nth element does not exist.
$\|L\|$	The size of a given list L.
Lcs_x^k	A list of execution costs of τ_x for accessing r^k during one release.
$Lcs_x^k(l, \mu)$	A list of execution costs of τ_x for accessing r^k within a duration l and a jitter μ.
$Lcsp_m^k(l)$	A list of execution costs from tasks on P_m to r^k within a given duration l.

A.1.1 MSRP

We start the extension from the new schedulability test of MSRP presented in Section 3.2.1, which provides the basis for analysing PWLP and MrsP systems. As described in Equation (3.1), the blocking incurred by τ_i under MSRP is bounded by E_i, B_i and $I_{i,h}$, where E_i and $I_{i,h}$ are calculated by Equation (3.4) and B_i is determined by Equation (3.7). With the presence of the heterogeneous accesses, these equations should be modified to the capture the worst-case blocking time that a task can incur during each resource access (i.e., in the worst case, the task can be blocked by the request with the highest execution cost on each remote processor).

Recall Equation (3.5), with requests that have an identical execution cost assumed, the spin delay incurred by τ_x for accessing r^k can be effectively bounded by examining whether there exist any requests to r^k that have not been accounted into the blocking time on each remote processor. However, to capture the worst-case scenario with heterogeneous accesses, the blocking time incurred by τ_x for one resource access from a remote processor should be the request with the highest execution cost among the unaccounted requests on that processor, assuming there exists any.

$$e_x^k(l, \mu)(n) = Lcs_x^k(l, \mu)(n) + \sum_{P_m \neq P(\tau_x)} Lcsp_m^k(l)(Nh_x^k(l) + n) \tag{A.4}$$

Equation (A.4) gives the total resource-accessing cost of the nth access to

r^k issued by τ_x within a given duration l and a jitter μ, where $Lcs_x^k(l,\mu)(n)$ denotes the pure execution cost of τ_x's n^{th} access to r^k and $Lcsp_m^k(l)(Nh_x^k(l)+n)$ denotes the worst-case blocking time that τ_x can incur in this access due to the unaccounted request that has the highest execution cost from P_m. Note that the execution costs before the index $Nh_x^k(l)+n$ is already accounted into the blocking time calculation as the analysis starts from the highest priority task. In addition, the execution cost $Lcsp_m^k(l)(Nh_x^k(l)+n)$ is higher than the costs after the index $Nh_x^k(l)+n$ due to the decreasing order. Therefore, the worst-case blocking time of τ_x in the n^{th} access can be obtained.

The above analysing technique is similar with the blocking time calculation given in Equation (3.5), where a resource access can be blocked from a remote processor only if there exist unaccounted requests in that processor. If such an element does not exist in the execution cost list, then a value of 0 is returned. In addition, as with the schedulability tests for homogeneous accesses, this analysis relies on the assumption that the first access to a resource will incur as much spin delay as possible. As described in Section 3.2.1, this assumption will not affect the schedulability results but can ease the analysing process. With the above equations, the direct spin delay E_i and the indirect spin delay $I_{i,h}$ of τ_i can be obtained.

As for the arrival blocking B_i for τ_i, this variable can be safely bounded by the following steps:

1. identify the set of local lower priority tasks (i.e., τ_{ll}) that can cause τ_i to incur arrival blocking i.e., the τ_{ll}s that request the resources in $F^A(\tau_i)$.
2. obtain the total resource-accessing time of the first access issued by each τ_{ll} identified above to each resource in $F^A(\tau_i)$ within the duration of R_i.
3. get the largest value among the above resource-accessing times.

Recall the notation Lcs_x^k, as we ordered the execution costs in a non-increasing order, the first access to a resource issued from a task will always have the highest execution cost. In addition, due to the assumption described above, the first access to a resource will incur the largest amount of spin delay in this analysis. Therefore, the maximum arrival blocking that τ_i can incur is the largest value among the total resource-accessing times of the first access issued by each τ_{ll} to each resource in $F^A(\tau_i)$.

By doing so, the worst-case arrival blocking of τ_i can be bounded, as shown in Equation (A.5), where $Lcs_{ll}^k(1)$ gives the maximum execution cost of τ_{ll} on a given resource r^k that belongs to $F^A(\tau_i)$ and $Lcsp^k_m(R_i)(Nh^k_i(R_i)+N^k_i+1)$ returns the largest execution cost among the requests to r^k on each remote processor P_m that can block this access (i.e., the requests that have not been accounted for in the calculations of E_i and $I_{i,h}$).

$$\hat{e_i} = max\Big\{Lcs^k_{ll}(1) + \sum_{P_m \neq P(\tau_i)} Lcsp^k_m(R_i)(Nh^k_i(R_i) + N^k_i + 1) \\ \Big| r^k \in F^A(\tau_i) \wedge N^k_{ll} > 0\Big\} \tag{A.5}$$

The above presents the approach for analysing the heterogeneous resource accesses in MSRP systems. By replacing Equations (3.5) and (3.7) to Equations (A.4) and (A.5), the new schedulability test of MSRP proposed in Section 3.2.1 can capture the worst-case blocking time with resources that have various execution costs.

A.1.2 PWLP

As described in Section 3.2.2, the approach for bounding the direct and indirect spin delay in PWLP systems is the same with that of the MSRP systems i.e., Equation (A.4) can be directly adopted to PWLP systems for accounting the spin delay. However, the approaches for accounting the arrival blocking and the additional blocking time due to the cancellation mechanism under PWLP should be modified to support the analysis of the heterogeneous resource accesses.

Recall Equation (3.19), under PWLP systems, a task can only be blocked upon its arrival by one critical section as tasks are spinning for a PWLP resource with its base priority. Thus, this equation can be modified to support resource with various execution costs with minor changes, where

$$\hat{e_i} = max\{Lcs^k_{ll}(1) | r^k \in F^A(\tau_i) \wedge N^k_{ll} > 0\} \tag{A.6}$$

For all the local lower priority tasks that can cause τ_i to incur arrival blocking, the task that has the highest execution cost on r^k will be the task that causes τ_i to incur arrival blocking in the worst case, and the amount of arrival blocking is the highest execution cost of that task on r^k i.e., $Lcs^k_{ll}(1)$.

As for the additional blocking caused by the cancellation mechanism, Equation (3.15) should be modified to reflect the worst-case blocking time under heterogeneous resource accesses, as given below.

$$L^k_i = \{\sum_{P_m \neq P(\tau_i)} Lcsp^k_m(R_i)(Nh^k_i(R_i) + N^k_i + n) | 1 \leq n \leq NoP_i\}^{dList} \tag{A.7}$$

Similar with the approach adopted in Equation (A.4), for a given remote processor P_m, the highest execution cost of the unaccounted requests to r^k in that processor can be obtained by $Lcsp_m^k(R_i)(Nh_i^k(R_i) + N_i^k + n)$, where the execution costs before the index $Nh_i^k(R_i) + N_i^k + 1$ (if they exist) has already been accounted into either the spin delay or the arrival blocking.

The above presents the approach for analysing the heterogeneous resource accesses in PWLP systems. With the Equations (A.6) and (A.7) adopted to replace the Equations (3.19) and (3.15) respectively, the schedulability test of PWLP can support the analysis of the heterogeneous resource accesses.

A.1.3 MrsP

As described in Section 3.2.3, MrsP has the identical spin delay accounting approach as that of both MSRP and PWLP. Thus, Equation (A.4) can be directly applied into the MrsP schedulability test for bounding the spin delay. In addition, the arrival blocking under both MSRP and MrsP can be calculated by Equation (A.5) (although with different approaches for determining $F^A(\tau_i)$, see Equations (3.8) and (3.22) for MSRP and MrsP respectively). However, the bounding for the migration cost due to the helping mechanism in Section 3.2.3 should be modified to capture the worst-case scenario with resources that have various execution costs.

Firstly, equation $mt_x^k(l)(n)$ developed in the Theorem 3 in Section 3.2.3 is modified to Equation (A.8) below. Note, the equation in the Theorem 3 can be directly applied here without any changes, as the objective of this function is to identify the remote processors that contain unaccounted requests to the given resource. Such a modification is for the analytical expression consistency in the extended analysis for heterogeneous accesses.

$$mt_x^k(l)(n) \triangleq \{P_m | P_m \neq P(\tau_x) \wedge Lp_m^k(l)(Nh_x^k(l) + n) > 0\} \cup P(\tau_x) \tag{A.8}$$

Then, to reflect various execution costs of one resource, Equation (3.27) is modified with an additional parameter C added (see Equation (A.9)), which takes the execution cost of r^k for a given access.

$$Mig(mt, r^k, C) = C + \sum_{P_m \in mt} \begin{cases} 0, & \text{if } P_m \notin mtp(mt, r^k) \vee \{P_m\} = mt \\ 2 \cdot C_{mig}, & \text{if } \{P_m\} = mtp(mt, r^k) \wedge |mt| > 1 \\ min\{Mhp(mt, r^k, C), Mnp(C)\}, & \text{otherwise} \end{cases} \tag{A.9}$$

Accordingly, Equations (3.25) and (3.26) should be modified to the following ones to cope with the newly introduced parameter C, which replaces the notation c^k in the homogeneous resource accesses case.

$$Mhp(mt, r^k, C) = C_{mig} \cdot \Big(\sum_{P_m \in mtp(mt, r^k)} \big(\sum_{\tau_h \in hpt(r^k, P_m)} \left\lceil \frac{C + Mhp(mt, r^k, C)}{T_h} \right\rceil \big) + 1 \Big) \tag{A.10}$$

$$Mnp(C) = C_{mig} \cdot \big(\left\lceil \frac{C}{C_{np}} \right\rceil + 1 \big) \tag{A.11}$$

With the above equations, Equation (A.4) can be extended to take the migration cost into account, where

$$e^k_x(l, \mu)(n) = \sum_{P_m \neq P(\tau_x)} Mig\Big(mt^k_x(l)(n), r^k, Lcsp^k_m(l)(Nh^k_x(l) + n) \Big) + Mig\Big(mt^k_x(l)(n), r^k, Lcs^k_x(l, \mu)(n) \Big) \tag{A.12}$$

For τ_x's nth access to r^k within the given duration l and the jitter μ, Equation (A.8) (i.e., $mt^k_x(l)(n)$) gives the migration targets for this access (i.e., the processors that contain unaccounted requests to r^k) within the given duration and jitter. Then, with the highest cost among the unaccounted execution costs in a given migration target (i.e., a processor) assigned to function $Mig(mt, r^k, C)$, the worst-case migration cost can be obtained with the presence of heterogeneous resource accesses.

As for the migration cost in the arrival blocking, Equation (3.29) is modified to reflect various execution costs of r^k, where

$$\hat{e_i} = max\Big\{ \sum_{P_m \neq P(\tau_i)} Mig\Big(mt^k_{ll}(Ri)(1), r^k, Lcsp^k_m(R_i)(Nh^k_i(R_i) + N^k_i + 1) \Big) + Mig\Big(mt^k_{ll}(Ri)(1), r^k, Lcs^k_{ll}(1) \Big) \Big| r^k \in F^A(\tau_i) \wedge N^k_{ll} > 0 \Big\} \tag{A.13}$$

Equation (A.13) presents the arrival blocking of τ_i with the migration cost bounded under MrsP with heterogeneous resource accesses, where the function $Mig\big(mt^k_{ll}(Ri)(1), r^k, Lcs^k_{ll}(1)\big)$ gives the execution cost and the migration cost of τ_{ll}'s first access to r^k while $Mig\big(mt^k_{ll}(Ri)(1), r^k, Lcsp^k_m(R_i)(Nh^k_i(R_i)+N^k_i+1)\big)$ returns the cost of a remote request that can block τ_{ll} on a remote processor P_m.

With the above equations adopted, the new schedulability test for MrsP can support the analysis of heterogeneous resource accesses, where Equation (3.28) is modified as

$$e_x^k(l,\mu) = \sum_{n=1}^{N_x^k(l,\mu)} e_x^k(l,\mu)(n) \tag{A.14}$$

and Equation (A.12) is adopted to calculate each $e_x^k(l,\mu)(n)$ in Equation (A.14). As for the arrival blocking, Equation (3.29) is replaced by Equation (A.13) to capture the worst-case scenario with resources that have various execution costs.

Summarising the above, this section presents an extension to the new schedulability tests proposed in Section 3.2 to support the heterogeneous resource accesses. In addition, the analysis of run-time overheads (i.e., the notations CX_1, CX_2, C_{retry}, C^{lock} and C^{unlcok}) can be effectively integrated into the extended schedulability tests by the same approach described in Section 3.2. In next section, the approach for supporting nested resource accesses will be presented.

A.2 Analysing Nested Resource Accesses

As described in Section 2.5.9, nested resource accessed are usually controlled by either group locks or ordered locks, where group locks can decrease the degree of parallelism while ordered locks impose restrictions to the resource-accessing model (i.e., the accesses to nested resources must comply with a pre-defined order). As described in [106], supporting the analysis of the ordered locks can cause severe analytical challenges and requires huge modifications to the newly-proposed schedulability tests. In contrast, although group locks serialise the access to nested resources and can decrease the parallelism, this locking approach is schedulability test friendly and is adopted to a large number of resource sharing protocols (e.g., FLMP and M-BWI). In this work, we support the nested resource accesses via group locks. The topic towards supporting the nested resource accesses via the ordered locks (which is currently under investigating with preliminary results presented in [49]) is postponed to future work.

With group locks assumed, a group lock is employed to control a set of resources that are accessed in a nested fashion, where the access to any of the resources managed by a group lock can be granted only if that lock is acquired. For instance, if r^1, r^2 and r^3 are managed by a given group lock and $r^1 \rightarrow r^2 \rightarrow r^3$ (where $\rightarrow$ describes the nested level), accesses to any of these resources should obtain the lock a priori. Thus, from the viewpoint of schedulability test, the resources that are managed by a same lock can be viewed as one resource with various execution costs.

Consider the same example given above with $c^1 = 1$, $c^2 = 2$ and $c^3 = 3$, the nested accesses to either r^2 and r^3 can be viewed as accessing r^1 with an execution cost of 3 and 6 respectively. For instance, during one release, if τ_a accesses r^1 only (without accessing r^2 and r^3) first and then accesses r^2 and r^3 sequentially in a nested fashion, such resource accesses can be modelled as $c^1_a(1) = 1$, $c^1_a(2) = 3$ and $c^1_a(3) = 6$ in one release. In addition, if τ_b access r^2 and r^3 in a non-nested way, then $c^1_b(1) = 2$ and $c^1_b(2) = 3$.

Therefore, with such an approach, the nested resource-accessing model with the group locks adopted can naturally fit into the newly-developed schedulability tests for heterogeneous resource accesses in Section A.1 with only a few modifications required, where the nested resources are removed from the resource list and the functions $F(\tau_i)$, $G(r^k)$ and $F^A(\tau_i)$ now only consider the outer-most resources e.g., function $F(\tau_i)$ now only returns the resources that τ_i can access directly.

In addition, such an approach is also valid even if these resources have varied execution costs. Now we assume that for τ_a, $L^1_a = \{1, 5, 9\}$, $L^2_a = \{2, 6\}$ and $L^3_a = \{3\}$ while $L^2_b = \{2\}$ and $L^3_b = \{3\}$ for τ_b. Thus, with the above resource-accessing behaviours, $L^1_a = \{1, (5 + 2), (9 + 6 + 3)\}$ and $L^1_b = \{2, 3\}$. Accordingly, the extended schedulability tests in Section A.1 can still be adopted directly to analysing such resource accesses, which contain both the heterogeneous and nested resource accesses.

A.3 Summary

Summarising the above, in this appendix, the schedulability tests for MSRP, PWLP and MrsP proposed in Section 3.2 are firstly extended to support the analysis for the heterogeneous resource accesses. Then, we demonstrated the approach to support nested resources via group locks and the techniques for analysing such resource accesses, which can be achieved with minor modifications to the newly-developed analysis in Section A.1. Based on the description above, we have demonstrated that the extended schedulability tests can be directly adopted to analyse the systems with the presence of both the heterogeneous and the nested resource accesses.

Appendix B

Case Study: Determining the Overheads for Litmus$^{\mathrm{RT}}$

As described in Section 3.2, the notations CX_1 and CX_2 are introduced to provide an overall approach to bound the run-time overheads from the underlying operating system. However, to precisely bound these two variables, a real-world operating system must be provided as the scheduling structure varies with different operating systems, which can lead to different behaviours and costs of context switches. In addition, the overheads in C^k, the cost of migrations C_{mig} and the overheads of the cancellation mechanism C_{retry} can be bounded only if the implementations of these protocols are provided.

In this section, a case study is presented to illustrate the approach for bounding the overheads incurred by Litmus$^{\mathrm{RT}}$ and the locking protocols. Similar works have been proposed to collect the run-time overheads of locking protocols under real-world systems [33, 94]. However, these works mainly focus on the cost of the locking protocols themselves and do not consider the overheads from the underlying operating systems. In this work, the run-time cost from both the operating system and the candidate locking protocols will be measured. To measure such costs, the candidate locking protocols are implemented in Litmus first. The description and discussion towards implementing the candidate resource sharing protocols are not given here and are referred to Appendixes D and C. With the run-time overheads measured, these values are assumed in the newly-proposed schedulability tests and are used in the experiments presented in Section 3.4.3 for investigating the impact of run-time overheads to the schedulability results of the newly-developed schedulability tests in Section 3.2.

B.1 Litmus$^{\mathrm{RT}}$

Litmus$^{\mathrm{RT}}$ [19, 30] is a real-time patch for Linux and is developed to provide an experimental platform for investigating various scheduling schemes and synchronisation algorithms in real-time systems. Litmus is mainly built up with four components: a core infrastructure, scheduler plugins, a user-space API, user-space library and tools. The core infrastructure provides a way to connect to the scheduling algorithm of the Linux kernel and overrides the Linux scheduling decisions with the scheduling routine provided in Litmus. In Litmus, various scheduling schemes are supported, including the FPS and EDF with fully partitioned, global and clustered scheduling. As this thesis focuses on the preemptive fixed-priority scheduling with fully-partitioned systems, the P-FP scheduler in Litmus is assumed and the protocols are implemented in this scheduling scheme. In addition, a set of user-space tools are provided to facilitate the research towards this system, such as the feather-trace tool for tracing and measuring the run-time overheads of the major events occurred in scheduling and locking via a set of timestamps.

B.2 Costs of the Major Scheduling Events in Litmus$^{\mathrm{RT}}$

To precisely bound the cost of the context switch under Litmus, the scheduling routine in Litmus is firstly explained. Recall Figure 3.1, the major scheduling events described in this figure can be directly mapped to functions in Litmus (or the underlying Linux system), as shown in Figure B.1.

Under Litmus, the clock that monitors the release time of tasks in event A is realised by a high-resolution timer. Once a release is due, the corresponding handler will be fired and the function `hrtimer_wakeup()` will be invoked. Then, the function `try_to_wake_up()` is invoked to insert the ready-to-released task into the corresponding ready queue via calling function `enqueue_task()`. In Litmus, the task is inserted into the ready-queue of the task's host processor via function `task_wake_up()` provided by the Litmus infrastructure. The cost of this whole procedure is denoted as $C^{litmus}_{release}$ and is collected via the timestamp `TS_RELEASE` provided by the Litmus time-collecting facility.

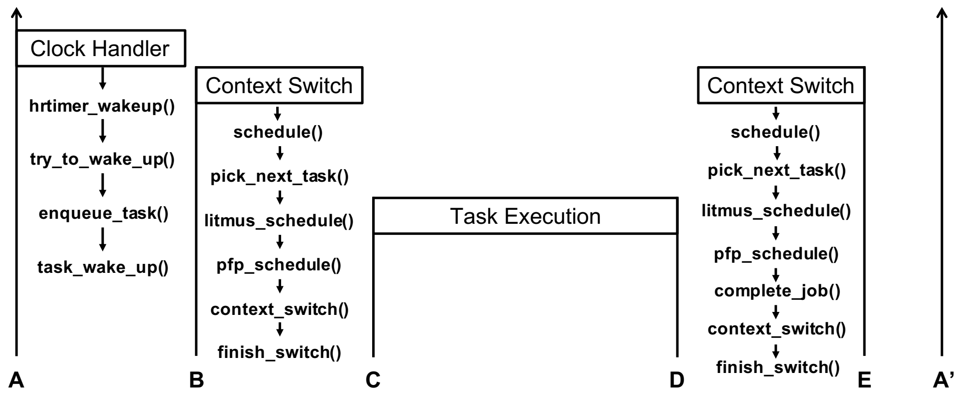


Figure B.1: Major Scheduling Events and Related Functions in Litmus$^{\mathrm{RT}}$.

As for event B, once a task is eligible to execute, the `schedule()` function in Linux is invoked to schedule this task to execute. This is achieved via invoking a set of functions sequentially. Firstly, function `pick_next_task()` is called to get the task that is eligible to execute. Under Litmus, this function is revised so that it is hooked by the Litmus infrastructure, where function `litmus_schedule()` is called. As the P-FP scheduling is assumed, the function `pfp_schedule()` is invoked and the ready-to-execute task can be picked up, which is the highest priority task in the ready queue. Then, function `context_switch()` is called to switch away the previously-executing task (if it exists), where the function `finish_switch()` in Litmus is executed finally to handle the potential migration required by the previously-scheduled task due to resource sharing (with the P-FP scheduler assumed).

The overheads incurred by the above functions are collected via timestamps `TS_SCHED`, `TS_SCHED2`, `TS_CXS` and `TS_PLUGIN_SCHED`. As described in [19], the cost of `schedule()` is measured in two parts by `TS_SCHED` and `TS_SCHED2` respectively. This is because the Linux scheduler has to execute certain post scheduling code after the function `context_switch()` has being invoked inside `schedule()`.

To bound the overheads caused by these functions in context switches, new notations are introduced, where C^{linux}_{sched} denotes the overheads of the first part in `schedule()` before `pfp_schedule()` is invoked, C^{linux}_{post} denotes the cost of the second part in `schedule` (i.e., after `context_switch()` is executed), C^{pfp}_{sched} denotes the overheads incurred by Litmus with P-FP scheduling adopted, C^{linux}_{switch} represents the cost of function `context_switch()`, including the cost of `finish_switch()` in the P-FP scheduler. Accordingly, the cost of one complete context switch (i.e., C_{cxs}) in Litmus can be bounded, where

$$C_{cxs} = C^{linux}_{sched} + C^{pfp}_{sched} + C^{linux}_{switch} + C^{linux}_{post} \tag{B.1}$$

With the overheads incurred by events A and B bounded, CX_1 (the overheads a task can incur from the underlying system before it can execute) is formed, as shown in Equation (B.2).

$$CX_1 = C^{pfp}_{release} + C_{cxs} \tag{B.2}$$

The event E requires the similar functions with that of event B. However, as the current release is finished, the next release time of the task should be re-computed, and then the task should be put to sleep. This is achieved via function `complete_job()` in the Litmus infrastructure, where $C^{litmus}_{complete}$ is adopted to denote such cost.

Table B.1: Overheads in Litmus (and Linux) Scheduling Routine

Variables	Worst-case Cost
C^{linux}_{sched}	845 ns
C^{linux}_{switch}	965 ns
C^{linux}_{post}	736 ns
$C^{litmus}_{release}$	1383 ns
$C^{litmus}_{complete}$	411 ns

With the cost of events A, B and E formed, the total overheads from Litmus (and the underlying Linux) scheduling during the entire lifetime of a task's release (i.e., CX_2) can also be bounded via summing up the costs from all the scheduling events given in Figure B.1, as given in Equation (B.3), where the cost of event E equals to the cost of event B plus $C^{litmus}_{complete}$.

$$CX_2 = C^{litmus}_{release} + 2 \cdot C_{cxs} + C^{litmus}_{complete} \tag{B.3}$$

Table B.1 gives the worst-case cost for each of the variables given above, which are collected from 100,000 executions via the feather-trace tool. The overheads are collected on a Intel Core$^{\text{TM}}$ i7-6700K with a base frequency of 4.0 GHz. During evaluation, hyper-threading on each core is disabled; core 0 is preserved to handle interrupts; core 1, 2, 3 are isolated from the system for overheads collection and the network is disabled.

As for the bounding of C^{pfp}_{sched}, it is obtained via examining the overheads of each major event in P-FP scheduler, as given below. Figure B.2 illustrates the scheduling sequence of the original P-FP scheduler in Litmus.

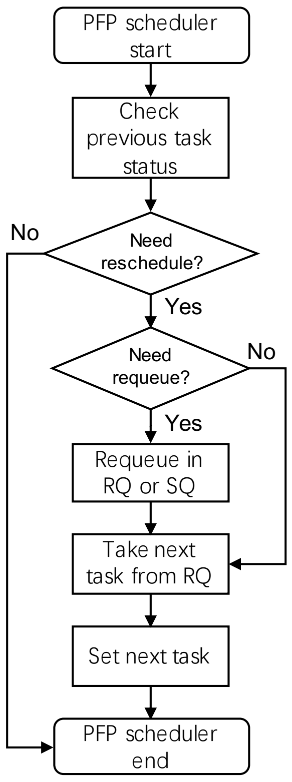


Figure B.2: The Original P-FP Scheduler in Litmus.

Once the P-FP scheduler is invoked, it firstly checks the current status of the previously-scheduled task (i.e., τ_{prev}) to decide wether a re-schedule is required. If not, the scheduler is finished directly and τ_{prev} can keep executing. Otherwise, the scheduler firstly checks whether the task should be re-queued into the run queue (or the sleeping queue when the task's current release is finished). If the task is blocked or requires migration, it will not be re-queued and will be handled in the `finish_switch()` function. Then, the highest priority task is taken from the run queue and is set to be the next running task (if there exist any). With the to-be-scheduled task (i.e., τ_{next}) determined, the scheduler is finished and the result is returned to the Linux infrastructure, where the function `context switch()` is performance to switch τ_{prev} away and to load the cache for τ_{next}.

To facilitate the bounding of C_{mig}, C^{pfp}_{check} demotes the costs for checking τ_{prev}'s status and whether a re-schedule and re-queue are required, the overheads for re-queueing τ_{prev} is denoted as $C^{pfp}_{requeue}$, C^{pfp}_{take} denotes the overheads for taking the highest priority task in the run-queue, and C^{pfp}_{set} represents the overheads for setting the to-be-scheduled task. Accordingly, the worst-case cost of $C^{pfp}_{schedule}$ can be bounded by the above events, as given in Equation (B.4). The overheads of the events in Equation (B.4) are collected on the same machine described before by the feather-trace tool, as reported in Table B.2. Accordingly, with $C^{pfp}_{schedule}$ bounded, the worst-case cost of C_{cxs}, CX_1 and CX_2 can also be calculated, as listed in Table B.3

$$C^{pfp}_{schedule} = C^{pfp}_{check} + C^{pfp}_{requeue} + C^{pfp}_{take} + C^{pfp}_{set} \tag{B.4}$$

Table B.2: Overheads of the P-FP scheduler in Litmus

Variables	Worst-case Cost
C^{pfp}_{check}	492 ns
$C^{pfp}_{requeue}$	603 ns
C^{pfp}_{take}	308 ns
C^{pfp}_{set}	274 ns
$C^{pfp}_{schedule}$	1677 ns

Table B.3: The Scheduling Overheads Incurred by Tasks under Litmus

Variables	Worst-case Cost
C_{cxs}	4223 ns
CX_1	5606 ns
CX_2	10,240 ns

B.3 Run-time Costs Incurred From Locking Protocols

For the cost due to locking protocols, we firstly form the bounding of C_{mig} and C_{retry} in MrsP and PWLP respectively. As both the helping and the cancellation mechanisms requires the modifications in the scheduler, the structure of the P-FP scheduler with these mechanisms integrated should be examined, as given in Figure B.3, where the red blocks indicates the helping facility while the blue blocks denotes the helping mechanism.

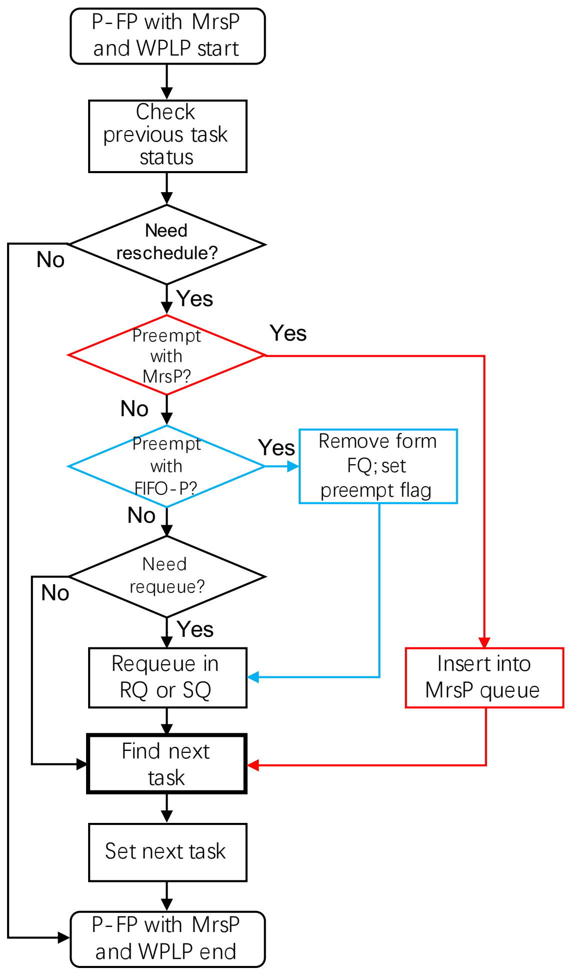


Figure B.3: The Modified P-FP Scheduler with MrsP and PWLP Implemented.

With MrsP and PWLP adopted, the scheduler has to check whether τ_{prev} is preempted while accessing a MrsP or PWLP resource. As for a task that is preempted while waiting for a PWLP resource, before it is re-queued to the run-queue, it will be removed from the corresponding resource-waiting queue and its preemption flag is set to indicate an re-request is required. While such a task is preempted, it must be stuck in the `while` loop in function `pwlp_lock()` (see Appendix C for the implementation details) as it is waiting for the resource. In this `while` loop, the task keeps checking whether the re-request flag is set. If so, the task will go to the beginning of the `pwlp_lock()` and will re-request the resource. To facilitate bounding C_{retry}, notation $C^{PWLP}_{de-queue}$ denotes the cost for removing a preempted task waiting for a PWLP resource from the FIFO queue and setting the re-requesting flag, and $C^{PWLP}_{re-request}$ denotes that cost for re-joining into the resource accessing routine. Thus, the cost due to PWLP's cancellation mechanism C_{retry} can be formed, as given in Equation (B.5). Table B.4 gives the worst-case cost of the cancellation mechanism in PWLP.

$$C_{retry} = C^{PWLP}_{de-queue} + C^{PWLP}_{re-request} \tag{B.5}$$

Table B.4: Overheads by the Cancellation Mechanism in PWLP

Variables	Worst-case Cost
$C^{PWLP}_{de-queue}$	703 ns
$C^{PWLP}_{re-request}$	960 ns
C_{retry}	1663 ns

If τ_{prev} is preempted while accessing a MrsP resource, it will be inserted into a pre-defined slot of a preemption queue introduced to guarantee the correctness of migrations in MrsP (see Appendix D for details). In addition, an executing task that is spinning for a MrsP resource can call `schedule()` explicitly to enters into the helping mechanism if it detects that the current resource holder is preempted. To facilitate bonding C_{mig}, notation C^{MrsP}_{insert} is introduced to denote the cost for queueing a preempted MrsP resource-accessing task into the preemption queue.

With τ_{prev} handled, the scheduler now takes the next task to execute (i.e., the block "find next task"), which can be further extended as Figure B.4. The scheduler will first peak into the run queue and gets the priority of the next task that is eligible to execute. Then it looks into its local preemption queue and checks whether there exists any task that has a higher priority. If yes, the preempted task that is waiting for or holding with a MrsP resource is selected to execute. Otherwise, the scheduler checks that whether the next task in the run-queue is waiting for a MrsP resource that is held by a preempted task. If so, the preempted resource holder is taken from the preemption queue and is scheduled instead of the task in the run queue. If the above conditions are not met, the scheduler will simplify take the next task in the run queue to execute. With MrsP in use, the cost of a migration should be accounted

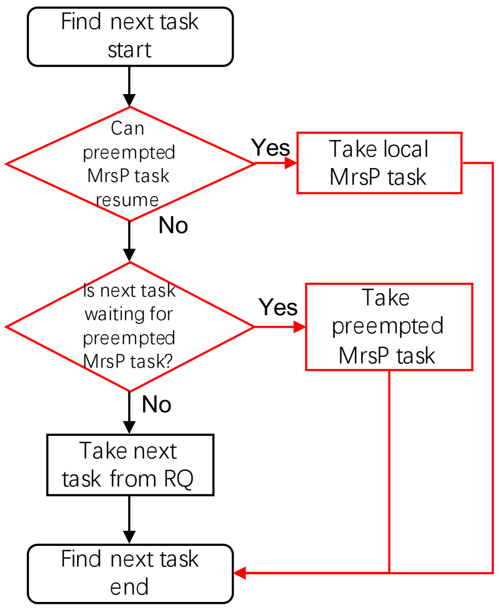


Figure B.4: The Event “Find next task” in the Modified P-FP Scheduler

for from the point that a resource-accessing task is preempted to the point that it is resumed on a remote processor, which is the longest route of the helping mechanism. Let C^{MrsP}_{help} denotes the cost of helping a preempted MrsP resource holder in the P-FP scheduler (i.e., the block “Take preempted MrsP task”).

Thus, once a task accessing a MrsP resource is preempted, it will incur the overheads by C^{linux}_{sched}, C^{pfp}_{check}, C^{MrsP}_{insert} for adding itself into the preemption queue so that a spinning task on a remote processor can detect such an event. Then, the spinning task calls `schedule()` to invoke the scheduler to enters into the helping mechanism, the cost of C^{linux}_{sched} and C^{pfp}_{check} is imposed again. After this, the scheduler re-queues the spinning task and takes the preempted resource holder as the to-be-scheduled task, which impose the overheads of $C^{pfp}_{requeue}$, C^{MrsP}_{help} and C^{pfp}_{set} in function `pfp_schedule()`. Finally, the function `context_switch()` and the second part in `schedule()` are performed to switch away the spinning task and to load the cache for to to-be-scheduled task, which impose the overheads of C^{linux}_{switch} and C^{linux}_{post}. Therefore, C_{mig} can be bounded, where

$$C_{mig} = 2 \cdot C^{linux}_{sched} + 2 \cdot C^{pfp}_{check} + C^{MrsP}_{insert} + C^{pfp}_{requeue} + C^{MrsP}_{help} + C^{pfp}_{set} + C^{linux}_{switch} + C^{linux}_{post} \tag{B.6}$$

Table B.5: Overheads of the Helping Mechanism in MrsP

Variables	Worst-case Cost
C^{MrsP}_{insert}	2347 ns
C^{MrsP}_{help}	745 ns
C_{mig}	8378 ns

Table B.5 firstly reports the overheads of inserting a preempted MrsP resource holder and helping a preempted holder. Then, based on Equation B.6, the worst-case cost of a migration is computed, as given in Table B.5 as well.

Finally, the cost in C^k should also be measured under each candidate locking protocol. In the interest of brevity, the implementation details and the execution sequence of the `lock()` and `unlock()` function for each protocol are not given in the main text of this thesis and is provided in Appendix C. The overheads for locking and releasing a resource under each candidate protocol are collected via the feather-trace tool and are reported in Table B.6.

Table B.6: Overheads of the Locking and Releasing Resources

Variables	Worst-case Cost
C^{lock}_{MSRP}	979 ns
C^{unlock}_{MSRP}	602 ns
C^{lock}_{PWLP}	1255 ns
C^{unlock}_{PWLP}	602 ns
C^{lock}_{MrsP}	1272 ns
C^{unlock}_{MrsP}	1642 ns

The above presents the approach for bounding the worst-case run-time overheads a task can incur in a given operating system (i.e., Litmus in this work) with the candidate locking protocols implemented under a given hardware platform. In this thesis, these values will be assumed in the schedulability tests and will be adopted in experiments.

Appendix C

Implementing MSRP, PWLP and MrsP in Litmus^{RT}

In Appendix B, the run-time overheads of the systems with MSRP, PWLP and MrsP adopted respectively are measured, which include the overheads of the context switches from the underlying operating systems (i.e., Limtus^{RT} with the P-FP scheduler adopted) and the implementation costs of the resource sharing protocols. The measured overheads are then adopted into the experiments conducted in Section 3.4. In this appendix, the implementation details of MSRP, PWLP and MrsP in the P-FP scheduler of Litmus^{RT} are described. The fully functional MSRP, PWLP and MrsP implementations can be accessed via `https://github.com/RTSYork/Litmus_MSRP_PWLP_MrsP`.

Compared to MSRP and PWLP, implementing MrsP is relatively complicated due to the migration-based helping mechanism. In this appendix, we present the implementation details of MRSP and PWLP, which also include the approach for realising the basic facilities in MrsP e.g., the FIFO resource-accessing order. However, the discussion towards the correctness and efficiency of implementing the helping mechanism of MrsP in fully-partitioned systems is postponed to Appendix D.

C.1 MSRP

We start from implementing MSRP under the P-FP scheduler in Litmus^{RT}. The description of the P-FP scheduler is presented in Section B. MSRP is realised via two functions `mrsp_lock()` and `mrsp_unlock()`, which can be invoked from the user space via the interfaces `sys_litmus_lock()` and `sys_` `litmus_unlock()` provided by Litmus^{RT}. Before presenting the implementations of the `lock()` and `unlock()` functions, the data structure of MSRP locks should be described, as given below.

```
struct mrsp_semaphore {
   int lock_id;

   /* a spin lock in kernel */
   spinlock_t lock;

   /* The FIFO resource-accessing queue */
   struct task_list *tasks_queue;
};
```

A MSRP lock contains an id, a kernel-level spin lock `spinlock_t` for protecting the data consistency of the FIFO resource-accessing queue when acquiring and releasing MSRP locks, and a linked list `struct task_list*` for realising the FIFO order. Implementing MSRP is relatively straightforward, which mainly contains the implementations the FIFO resource-accessing order, the non-preemptive resource-accessing priority rule and the spin-waiting approach. Below we present the code of the `mrsp_lock()` function.

```
int msrp_lock(struct litmus_lock* l) {
   struct task_struct* t = current;
   struct msrp_semaphore *sem = msrp_from_lock(l);
   struct task_list *taskPtr = (struct task_list *) kmalloc(
      sizeof(struct task_list), GFP_KERNEL);
   struct task_list *next;
   int err = 0;

   if (t->rt_param.task_params.lock != NULL) {
      err = -EINVAL;
      goto out;
   }

   /* joins into the FIFO resource-accessing queue and priority
       boosting */
   preempt_disable();
   spin_lock_irqsave(&sem->lock, flags);

   add_task(taskPtr, t, &(sem->tasks_queue->next));
   t->rt_param.task_params.priority = 0;

   spin_unlock_irqrestore(&sem->lock, flags);
   preempt_enable();

   /* spinning for the lock */
   while (1) {
      preempt_disable();
      spin_lock_irqsave(&sem->lock, flags);

      next = list_entry((sem->tasks_queue->next.next), struct
          task_list, next);
      if (next->task == t) {
         /* gets the lock */
         t->rt_param.task_params.lock = 1;
         break;
      }

      spin_unlock_irqrestore(&sem->lock, flags);
      preempt_enable();
   }

   out: return err;
}
```

With the `msrp_lock()` function invoked, the calling task and the requested MSRP lock are firstly obtained. Then, the calling thread enters into a non-preemptable critical section to join into the FIFO resource-accessing queue and to raise its priority to 0, which is a priority level preserved by Litmus^{RT} for priority boosting. Under Litmus^{RT}, tasks with a priority of 0 becomes effectively non-preemptive [19]. In addition, the FIFO resource-accessing order is realised via a linked list `struct task_list*`, where a node can be dynamically added into or removed from the list. Once a task requests a MSRP lock, it adds itself at the end of the list via `add_task(taskPtr, t, &(sem->tasks_queue->next))`.

As shown in the code given above, inserting the resource-requesting task into the FIFO queue is conducted in the critical section (i.e., under the protection of `spinlock_t`) with preemptions and interruptions disabled to prevent race conditions when manipulating the queue. In addition, the task becomes preemptable in this critical section so that it will not be preempted after the preemptions are enabled (i.e., the function `preempt_enable()`). After the task exits the critical section, it enters into the while loop and begins spinning (i.e., busy-waiting) until the task becomes the head of the FIFO queue, where `list_entry((sem->tasks_queue->next.next), struct task_list, next)` returns the head of the queue. Once the task is granted with the lock, it stores the reference of the lock into the task structure.

After a task finishes executing with a MSRP resource, the task releases the corresponding MSRP lock via the function `msrp_unlock()`. Below presents the code of the `msrp_unlock()` function.

```
int msrp_unlock(struct litmus_lock* l) {
   int err = 0;
   struct task_struct *t = current;
   struct msrp_semaphore *sem = msrp_from_lock(l);
   struct task_list *my_obj = NULL;

   if (t->rt_param.task_params.lock != l) {
      err = -EINVAL;
      goto out;
   }

   /* set our status to "has nothing to do with the lock". */
   preempt_disable();
   spin_lock_irqsave(&sem->lock, flags);

   t->rt_param.task_params.lock = NULL;
   t->rt_param.task_params.priority = t->original_priority;
   my_obj = task_remove(t, &(sem->tasks_queue->next));

   spin_unlock_irqrestore(&sem->lock, flags);
   preempt_enable();

   kfree(my_obj);

   out: return err;
}
```

With function `msrp_unlock()` invoked, the calling task and the to-be-released MSRP lock are obtained first. Then, the task enters into the non-preemptive critical section to (1) removes the reference of the MSRP lock, (2) restores the task's priority and (3) removes itself (i.e., the head) from the linked list via `task_remove(t, &(sem->tasks_queue->next))`. After this critical section, the task releases the lock and the exits the function `msrp_unlock()`.

C.2 PWLP

As PWLP shares many similar features with MSRP (e.g., the FIFO ordering and busy-waiting), the implementations of these facilities under PWLP are identical with that of in MSRP. However, differentiated with MSRP, tasks under PWLP wait for a PWLP lock with their base priorities and execute non-preemptively only with the lock granted. Therefore, the code `t->rt_param.task_params.priority = 0` is moved into the `while` loop and is executed only if the requesting task becomes the head of the FIFO resource-accessing queue (i.e., the lock holder). In addition, the major difference between MSRP and PWLP is that tasks under PWLP can be preempted while waiting for a PWLP lock. Once being preempted, the task cancels the current resource request and will re-request this resource later on when being resumed by the scheduler (i.e., the cancellation mechanism). Below presents the implementation of the cancellation mechanism.

```
enter:
/* joins into the resource-accessing routine */
t->rt_param.task_params.pwlp_lock = sem;
joins into the FIFO queue...

int goout = 0;
while (!goout) {
   if (t->rt_param.task_params.need_re_request) {
      t->rt_param.task_params.need_re_request = 0;
      goto enter;
   }

   checks the head of the FIFO queue...
   if (next->task == t) {
      t->rt_param.task_params.priority = 0;
      t->rt_param.task_params.lock = 1;
      goout = 1;
   }
}
```

With the function `pwlp_lock()` invoked, the calling task joins into the resource-accessing routine. Firstly, it joins into the FIFO queue and then enters into the `while` loop to wait for the PWLP lock. During spinning, the task could be preempted by a newly-released local higher priority task. Thus, the scheduler is invoked to switch this task away, where the current request of this task is cancelled. The code for cancelling a request is given below.

```
if (prev && prev->rt_param.task_params.pwlp_lock != NULL && preempt) {
   spin_lock(&prev->rt_param.task_params.pwlp_lock->lock);

   prev->rt_param.task_params.need_re_request = 1;
   prev->rt_param.task_params.pwlp_lock = NULL;
   list_del(prev->rt_param.task_params.next);

   spin_unlock(&prev->rt_param.task_params.pwlp_lock->lock);
}
```

If the scheduler identifies that the `prev` task (i.e., the currently scheduled task) is preempted while waiting for a PWLP resource (i.e., `prev->rt_param.task_params.pwlp_lock != NULL && preempt`), it sets the re-requesting flag `need_re_request` for the task and then removes it from the FIFO queue so that the next waiting task (if it exists) can proceed to obtain to lock. Once this task is resumed, it is still waiting in the `while` loop, but with the `need_re_request` flag outstanding. Thus, the function will redirect the task to the `enter` block to start the resource requesting routine again. Once the task is granted with the lock, it becomes effectively non-preemptive with the priority level 0 assigned and then exits this function.

The procedure for releasing a PWLP lock is similar with that of MSRP locks, where the function firstly removes the task from the FIFO queue and then restores the priority of the calling task. For brevity, the implementation details of the `pwlp_unlock()` function is not presented.

C.3 MrsP

As for MrsP, the implementation for realising the FIFO resource-accessing order and the spin-waiting approach is identical with that of MSRP and PWLP, where a linked list is adopted for the FIFO resource-accessing queue and a `while` loop is used to model the busy-waits. However, under MrsP, a priority ceiling facility is adopted, where tasks wait for and execute with a MrsP resource with the ceiling priorities of the resource on their processors.

In this implementation, the ceiling priorities of a MrsP lock is assigned by users before run-time via a pointer `prio_per_cpu` in the data structure of the MrsP lock. In function `mrsp_lock()`, the priority of the calling task is raised if necessary (i.e., if its current priority is less than the ceiling) via the following code. Note that different from the task model adopted in our thesis, under Litmus^{RT}, a lower priority value indicates a higher execution eligibility.

```
int partition = get_partition(t);
int prio = get_priority(t);
int ceiling = sem->prio_per_cpu[partition];

t->rt_param.task_params.priority = ceiling < prio ? ceiling : prio;
```

In addition, a major difference between MrsP with MSRP and PWLP is the migration-based helping mechanism. Supporting such a facility in fully-partitioned systems (where a task is fixed on a processor during its entire lifetime) is complicated and can cause issues that undermine the correctness and efficiency of the protocol. Thus, the details for releasing this helping mechanism are not presented in this appendix and are referred to Appendix D.

Appendix D

Investigating the Correctness and Efficiency of MrsP in Fully Partitioned Systems

As described in Section 2.5.9, MrsP relies on a migration-based helping mechanism, where a preempted resource-holding task can migrate to a processor with a task spinning for that processor. As described, such a migration-based helping mechanism is favourable in theory as high priority tasks can incur less arrival blocking while the resource-accessing tasks can have a less interference from high priority tasks.

However, in practice, the realisation of the helping mechanism can be problematic. The migration targets for a resource-holding task are not constant as remote spinning tasks can also be preempted. Thus, the migration target decision made by the protocol may conflict with the scheduling decisions, and thereby results in incorrect and useless migration behaviours. This issues can cause unpredictable task behaviours with considerable run-time overheads, which directly undermine the efficiency of the protocol. In addition, as described in Section 3.2.3, practising this helping mechanism also has the issue of frequent migrations. To address this issue, a tuneable NP-section is introduced to MrsP to address the frequent migration issue.

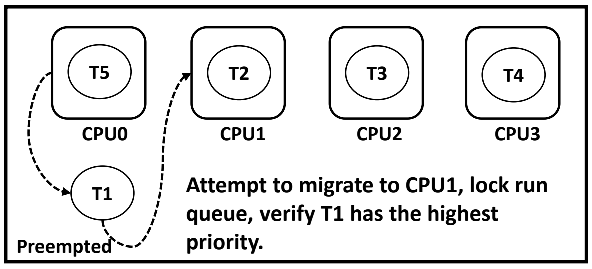
In this appendix, we firstly describe the issue for allowing migrations in fully-partitioned systems. Then, we present the approach that addresses the migration issue and describe the realising approach of the NP-sections. For brevity, we focus on presenting the design of the realising approach for the migration-based helping mechanism rather than presenting the code of the implementation. The implementation details are referred to [108] and `https://github.com/RTSYork/Litmus_MSRP_PWLP_MrsP`.

D.1 False Migrations

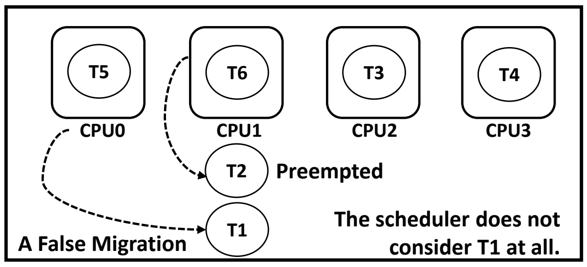
With the generic Linux kernel, task migrations are handled by a set of push and pull operations, as part of the scheduling routine. The push operation is triggered after a scheduling decision to migrate the previous scheduled task (i.e., the task that was executing before this scheduled task) to a remote processor. The pull operation is preformed before a scheduling decision to migrate a remote task to the local processor. According to [19], the fact that both push and pull operations need to manipulate multiple run queues can cause concurrent state changes and it is not possible to have a consistent snapshot without locking all the run queues. Thus, the migration facility in Linux may either trigger superfluous migrations or fail to trigger required migrations due to such race conditions, resulting in unbounded priority inversion. Similar migration failures can occur when adopting MrsP into such a partitioned run queue structure. We identify two major migration problems of MrsP with such push and pull migration operations.

The first migration problem is caused by race conditions between run queues and can happen in both push and pull operations. Once a resource holder is preempted and a migration target is identified, the holder will be placed into the remote run queue. However, before the next scheduling point, a higher priority task can be released immediately so that the migrated task is not considered by the scheduler at all. Such migration can be regarded as a futile attempt as it only provides extra overheads with the need for further migrations rather than offering the task a real chance to execute.

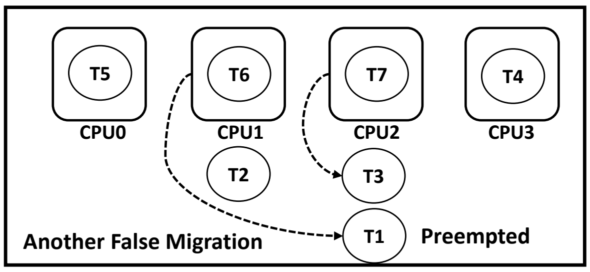
Figure D.1 illustrates this problem with a four core system, where task 1 to 4 request the same resource with low priorities while task 5 to 7 are irrelevant high priority tasks. In Figure D.1a, task 1 (τ_1) is preempted at processor 0 (P_0) while holding the resource so that it migrates to P_1, where τ_2 is spinning for the resource. However, after τ_1 is inserted into the run queue of P_1 (Rq_1), τ_6 is released and is then scheduled to execute. Thus, τ_1 remains in Rq_1 without any chance to execute so that it seeks another processor (Figure D.1b). In Figure D.1c, the same issue occurs when τ_1 migrates to P_2


T5
T2
T3
T4
CPU0
CPU1
CPU2
CPU3
T1
Preempted
Attempt to migrate to CPU1, lock run queue, verify T1 has the highest priority.

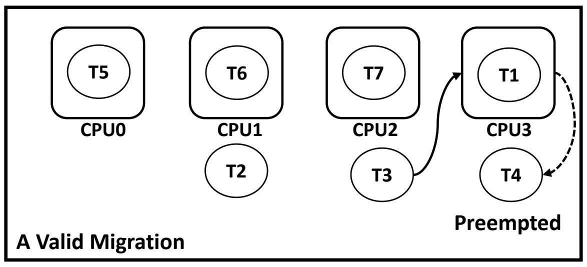
(a) First Attempt



(b) Second Attempt



(c) Third Attempt

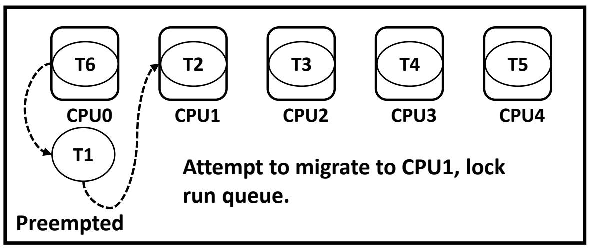


(d) A Success Migration

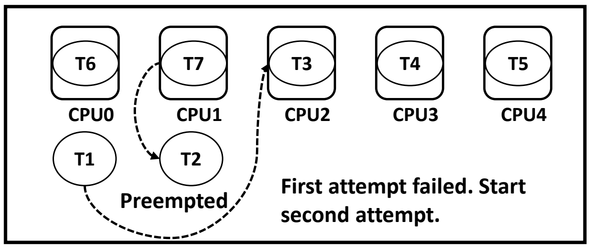
Figure D.1: False Migrations Due to Race Condition.

so that τ_1 is placed in Rq_2 with no chance to execute. Finally, it migrates to P_3 (Figure D.1d), where it preempts the spinning task and executes. In this example, 3 migrations are preformed in order to migrate τ_1 to a valid processor, yet two of them are invalid due to immediate updates of run queues.

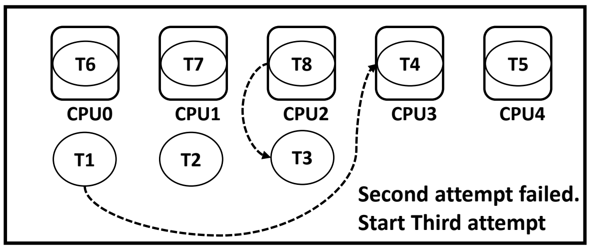
The second issue is caused by the push operation, which is usually con-



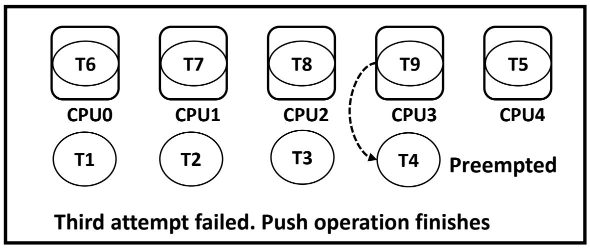
(a) First Push



(b) Second Push



(c) Third Push



(d) Push Operation Fails

Figure D.2: Missing Necessary Migration due to Limited Attempts.

figured with a fixed number of attempts to control overheads. Figure D.2 demonstrates this issue with a system of five processors and 3 push attempts. As shown in Figure D.2a, after τ_1 is preempted, the push operation firstly attempts to migrate τ_1 to P_1. However, due to the release of τ_7 in Figure D.2b, the first attempt fails. In Figure D.2c and D.2d, the second and third attempts fail as well due to the same reason. Thus, the push operation finishes without checking P_4, which is a valid migration target. Such failure can cause a longer resource accessing time of the holder and in consequence, a longer blocking time of all waiting tasks.

Admittedly, a migrated resource holder can be preempted again just after being scheduled, which also requires further migrations. However, false migrations impose extra incorrect behaviours and extra run-time overheads to tasks rather than offering tasks a real chance to execute. In Section D.4 we demonstrate the impact of this issue with experiments.

D.2 The False-migration-free Mechanism

To avoid false migrations, we propose that (1) the helping mechanism should be realised by pull operations only and (2) the migration decisions of the protocol should be made as a part of the scheduling decisions.

With a partitioned run queue structure, the push operation suffers from inescapable race conditions unless obtaining all run-queue locks. As scheduling decisions are made independently on each processor, it is not possible to guarantee that there will not be any release of high priority tasks on the target processor during the migrations by push. In addition, as explained in D.1, necessary migrations can be omitted due to a limited number of attempts. Therefore, push operations should not be adopted for the MrsP implementation to prevent race conditions.

In addition, to prevent race conditions in pull operations, we require that the pull operation needs to be modelled inside the scheduler and as a part of scheduling decisions. During each scheduling point, the pull operation will be triggered if the to-be-scheduled task is spinning for a resource while the resource holder is being preempted on a remote processor. The scheduler then replaces the to-be-scheduled task with the preempted resource holder as the next task to schedule. Thus, the migrated task is always eligible to execute while any newly released high priority tasks need to invoke the scheduler to preempt.

To realise the false-migration-free mechanism, a preemption queue (Pq) and a Pq lock are introduced for each processor. Once a resource-accessing task (either holding or waiting for a resource) is preempted, it will be placed into the Pq of its original processor rather than the Rq of the current processor. Upon a scheduling point, the scheduler looks into its local Pq and Rq and takes the highest priority task to execute. By doing so, the resource accessing task is able to resume on its original processor even though it is preempted on a remote processor. In addition, if the to-be-scheduled task is waiting for a resource while the resource-holding task is preempted (i.e., being placed into Pq), the pull operation removes the task from the Pq and migrates it to the resource-waiting task's processor to execute. To avoid race conditions, the Pq lock must be obtained in order to access that Pq.

By adopting such a facility, we realise the required functionalities defined in the helping mechanism. Meanwhile, we can avoid accessing multiple run queues with the nested access of Rq locks. As the lock of the Pq needs to be acquired inside the scheduler, i.e., after obtaining the Rq lock, deadlocks are prevented because no circular access can be formed. Yet it seems that the cost for a scheduling decision can be increased as the scheduler may need to compete for the Pq locks. However, such competition only occurs if a scheduler is trying to pull a preempted holder (i.e., the to-be-scheduled task is waiting for a resource). Hence, in the viewpoint of cost, there is no difference between spinning for the resource or spinning for a Pq lock to offer help. With the support of the false-migration-free mechanism, we eliminate possible race conditions between processors while migrating so that each migration is a valid migration: the resource holder is guaranteed a chance to execute after migrated. In Section D.4, the evaluation result demonstrates that such a "false-migration-free" implementation is important to the usability of the protocol.

D.3 Realising the NP-sections

As described in Section 3.2.3, to avoid frequent migrations of a resource holder and to improve the efficiency of the helping mechanism, we integrate MrsP with a short non-preemptive section to offer a trade off between the maximum number of migrations a holder can suffer and bounding the resulting blocking time on high priority tasks. Upon each migration, the resource holder is allowed to execute non-preemptively for a short period before it inherits the ceiling priority on the current partition. Accordingly, any newly released high priority tasks have to cope with the cost of one NP section before it can preempt the holder and execute.

With NP sections, a migrated resource-accessing task will be assigned with the priority 0 (the priority preserved by Litmus$^{\mathrm{RT}}$ for priority boosting) so that it can execute effectively non-preemptively. To restore the corresponding ceiling priority of the task after the NP section, one high resolution timer (`hrtimer`) is introduced for each processor. The `hrtimer` will be set each time a resource-accessing task is migrated to its processor. When the timer triggers, it sets the task's priority to the corresponding ceiling priority and invokes the scheduler to check whether a higher priority task is ready to execute. If the holder releases the resource during its NP section, the timer is then cancelled.

D.4 Investigating the Efficiency of the False Migration Free Mechanism

In Section 3.4.3, the performance of the NP-sections are investigated by a set of experiments. In this section, the efficiency of the newly-proposed false-migration-free mechanism is investigated. The experiments are performed by the implementations in [108] on a Intel Core^{TM} i7-6700K with a base frequency of 4.0 GHz. During evaluation, hyper-threading on each core is disabled; core 0 is preserved to handle interrupts; core 1, 2, 3 are isolated from the system and the network is disabled.

To investigate the frequency of false migrations, pressure testing is conducted. The testing program contains three resource requesting tasks on each core as well as three high priority tasks with very short periods (500 μs). Table D.1 gives the total number of migrations triggered by the helping mechanism and the number of false migrations occurred in 100,000 jobs. The test is conducted by a MrsP implementation with generic pull and push operations (MrsP-generic) and the new MrsP implementation (MrsP-new). As shown in the table, the generic implementation has a failure rate of 2.14%. In addition, the number of false migrations is theoretically unbounded and can increase with the increase of parallelism and the number of releases of high priority tasks on each core. However, no false migration occurred in the new MrsP implementation and fewer migrations are triggered as no further migrations are needed to recover from the false ones.

Table D.1: False Migrations in 100,000 executions

Implementation	Total Migrations	False Migrations	Failure Rate
MrsP-Generic	598,107	12,813	2.14%
MrsP-New	428,618	0	0%

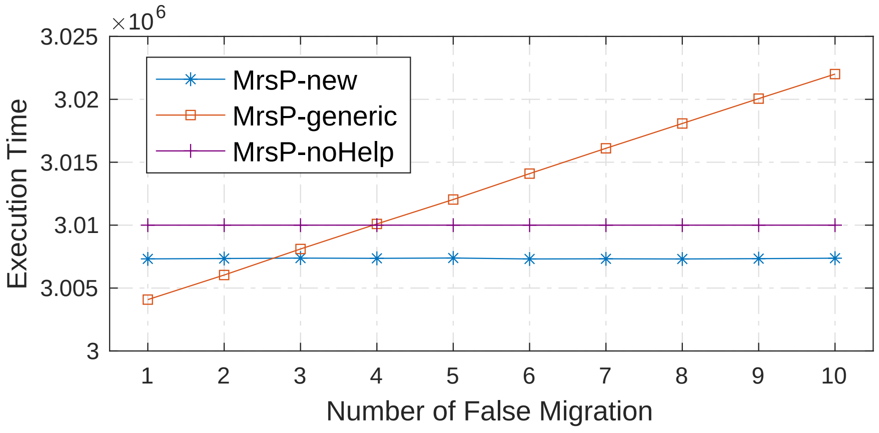


Figure D.3: The Impact of False Migrations on the Critical Section Execution Time

The following experiment demonstrates the impact of false migrations on the execution time. As the false migration is caused by race conditions and is difficult to reproduce on each release, we simulate its affects by preventing the migrated holder from being scheduled. In this test, the length of critical section is 3 ms and the computation time of the preemptor is 10 μs. As shown in Figure D.3, the execution time under MrsP-new (3.007 ms) is not affected by false migrations. As for MrsP-generic, although it has a lower cost for each migration, the execution time is prolonged by false migrations and is higher than that of MrsP-new with more than 2 false migrations. In addition, its execution time exceeds the time with the helping mechanism disabled (MrsP-noHelp) with more than 3 false migrations. Under such situations, MrsP has a poor efficiency and can be outperformed by protocols with a simple ceiling priority facility.

D.5 Summary

In this appendix, we conducted an investigation towards the correctness and efficiency of implementing MrsP in fully partitioned systems. We identified the false migration issues due to its migration-based helping mechanism when applied in fully partitioned systems and demonstrated that this issue can cause incorrect migration behaviours, which can impose additional run-time overheads and undermine the efficiency of the protocol. Then, a false-migration-free facility is then introduced to prevent this issue to guarantee the correctness and efficiency of the migration-based helping mechanism. In addition, the realising approach of the NP-sections is also presented. Our evaluation results demonstrate that the false migrations are successfully addressed by the proposed mechanism, which require less migrations when accessing resources so that an improved performance of the protocol can be achieved in practice.

Appendix E

One-Way ANOVA Analysis and Confidence Level

The analysis of variance (ANOVA) [67] is a statistical analysis technique for comparing the statistical significance difference between several data groups. As stated in [75], this statistical technique can isolate the variation that can cause the real differences and the variation that introduces the measurement noise among the total variation in a set of measurements. This analysis is commonly adopted to reveal the impact of certain variables to a given data set [57, 75, 76].

In this thesis, we have investigated the performance of various resource sharing techniques (i.e., MSRP, PWLP, MrsP and the combined techniques), task allocation schemes (the WF, BF, FF, NF, SPA, RCF, RLF-L and RLF-S algorithms) and priority ordering algorithms (the DMPO, OPA-D, RPA-D and SBPO algorithms) under fully-partitioned systems with shared resources under various system settings. For each system setting in a given experiment in this thesis, 1000 systems are generated and are tested to demonstrate the performance of the above algorithms.

In this appendix, the ANOVA analysis is adopted to reveal the statistical significance between the performance among the above algorithms (i.e., to demonstrate that there exists a performance difference between the evaluated algorithms). For each experiment, as there exists only one independent variable (i.e., either the resource control techniques, the task allocation schemes or the priority ordering algorithms), the one-way ANOVA analysis [56] is adopted to investigate whether adopting different algorithms have a significant impact to the schedulability of fully-partitioned systems with shared resources.

The approach for conducting such an analysis is similar with the work proposed in [75]. Firstly, to adopted the ANOVA analysis, a null hypothesis is proposed that no impact (i.e., performance difference) is imposed to the schedulability of fully-partitioned systems with (1) various resource sharing approaches (MSRP, PWLP, MrsP and the approach with multiple protocols), (2) different task allocation schemes (WF, BF, FF, NF, SPA, RCF, RLF-L and RLF-S) and (3) different priority ordering algorithms (DMPO, OPA-D, RPA-D and SBPO) adopted.

As with the settings adopted in [75], the α value is set to 0.05 i.e., a statistical significance level of 95%. In addition, the sample size in this test is set to 1000, which means that each experiment is repeated 1000 times for conducting the ANOVA analysis, where in each run, 1000 systems is examined under each system setting in a given experiment. However, due to the computation expenses concern, the experiments with the GA-based framework is sampled 100 times (which is still sufficient to perform the ANOVA analysis [76]). The notations in the ANOVA analysis is described in Table E.1 (cited from [75]).

Table E.1: Notations of the ANOVA analysis

SS	the sum of squares due to each source.
df	the degrees of freedom associated with each source.
MS	the mean squares, which equals to $\frac{SS}{df}$.
F	the ratio of the variance calculated among the means to the variance within the samples.
$Prob > F$	the computed probability that the null hypothesis can hold.

We firstly present the detailed approach for investigating the performance difference via the one-way ANOVA analysis with $n = 64$ and $n = 96$ in the experiment given in Figure 3.2. The results are given in Tables E.2 and E.3, where we compare the performance difference between each pair of MSRP, PWLP and MrsP under the new schedulability tests (i.e., the blue to green bars in the figure).

With the above results obtained, the F value can be obtained based on the table of F probability distribution for a given level of statistically significance (i.e., $\alpha = 0.05$) [102], where $F_{Protocols}(1999 - 2) = 254.3144$ for both $n = 64$ and $n = 96$. Compared to the F values calculated by the ANOVA test (see Ta-

Table E.2: The One-Way ANOVA Test Results for $n = 64$ in Figure 3.2

Source	SS	df	MS	F	$Prob > F$
MSRP & PWLP	80212.5	1	899390.5	275.12	$1.58{\times}10^{-58}$
Total	497118.8	1999			
PWLP & MrsP	103219.7	1	103219.7	462.29	$2.055{\times}10^{-92}$
Total	549329.1	1999			
MSRP & MrsP	272284.4	1	272284.4	1214.19	$2.9{\times}10^{-208}$
Total	720340.4	1999			

Table E.3: The One-Way ANOVA Test Results for $n = 96$ in Figure 3.2

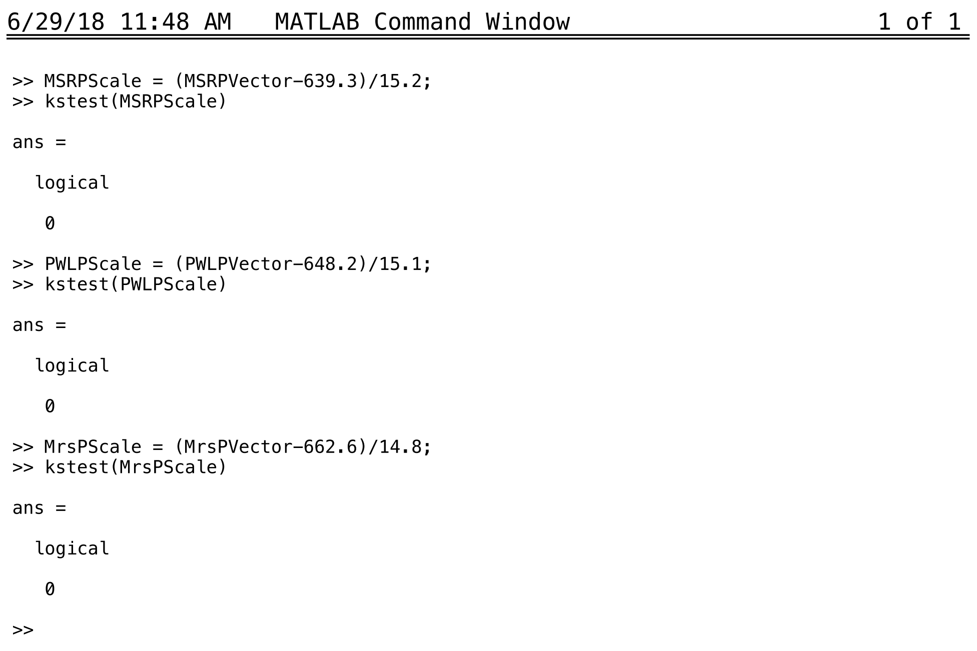
Source	SS	df	MS	F	$Prob > F$
MSRP & PWLP	299390.5	1	299390.5	1413.12	$2.36{\times}10^{-234}$
Total	722695.3	1999			
PWLP & MrsP	1295710.4	1	1295710	5963.23	0
Total	1729842.5	1999			
MSRP & MrsP	349431	1	349431	1588.81	$3.71{\times}10^{-256}$
Total	788856.9	1999			

bles E.2 and E.3), the value of $F_{Protocols}$ obtained in the F distribution table is smaller than the computed F values, which indicates the hypothesis is rejected with a confidence level of at least 95% i.e., there exist significant performance difference between each pair of the studied resource sharing protocols. Therefore, as shown in Figure 3.2, we can draw the conclusion that MrsP has the best performance among the examined protocols with both $n = 64$ and $n = 96$ under the given system setting with a confidence level of 95%.

In addition, one major assumption for adopting the ANOVA test is that the data of the analysed variables must be independent (i.e., normally distributed). The data distribution of the tested variable (i.e., the number of schedulable systems under each protocol) in this appendix is examined via the `kstest()` function in the MATLAB tool (see detailed description of this test in [74]), which returns a decision for the null hypothesis that the data in the given vector comes from a standard normal distribution [74]. Taking the test in Table E.2 as an example, below gives the approach for examining the distribution of the tested data. As the `kstest()` function is for standard normal distribution, each element in the tested vector is scaled by the mean value and the standard deviation [74]. Table E.4 gives the mean value and the standard deviation of the tested data of each protocol. For instance, to examine the distribution of the data vector of MSRP, each element in this vector, say x, should be scaled by $(x - 639.3) \div 15.2$. Then, the `kstest()` function is invoked to examine whether each data vector complies the standard normal distribution (see Figure E.1, the data in each vector can be accessed via `https://github.com/RTSYork/FIFOSpinLockFramework/blob/master/ConfidenceTest.xlsx`). As shown in the figure, the data in each tested vector comes from a normal distribution (i.e., the `kstest()` function returns 0, which fails to reject the null hypothesis above), and hence, the ANOVA test is valid to apply in this thesis to obtain the statistical significance level of the performance of each tested algorithm stated above.

Table E.4: Mean and Standard Devision of the Tested Vector in Table E.2

Protocol	Mean Value	Standard Devision
MSRP	639.3	15.2
PWLP	648.2	15.1
MrsP	662.6	14.8



```
6/29/18 11:48 AM   MATLAB Command Window                       1 of 1

>> MSRPScale = (MSRPVector-639.3)/15.2;
>> kstest(MSRPScale)

ans =

  logical

   0

>> PWLPScale = (PWLPVector-648.2)/15.1;
>> kstest(PWLPScale)

ans =

  logical

   0

>> MrsPScale = (MrsPVector-662.6)/14.8;
>> kstest(MrsPScale)

ans =

  logical

   0

>>
```

Figure E.1: Results from the `kstest()` Functions

With the approach described above, we have investigated the statistical significance of the performance difference of the evaluated algorithms in each experiment presented in this thesis via the MATLAB tool. However, as a large amount of experiments are presented in this thesis, in the interest of brevity, this section only presents the analysis results (the F values) of the evaluated algorithms in two system settings of certain experiments presented in Chapter 3, 4 and 5 as the examples showing that there exist significant performance difference between the evaluated algorithms. The results are summarised in Table E.5, where each F value obtained by the ANOVA analysis is much higher than the value (i.e., 254.3144) obtained from the F distribution tables in [102].

In addition, although we do not present the results between each pair of the tested algorithms in these experiments, the results presented are sufficient to demonstrate a statistical significance level of 95% of the performance difference of the studied resource sharing techniques (i.e., MSRP, PWLP, MrsP and the combined approach), task allocation schemes (i.e., the WF, BF, FF, NF, SPA, RCF, RLF-L and RLF-S algorithms) and the priority ordering algorithms (i.e., the DMPO, OPA-D, RPA-D and SBPO algorithms) (i.e., confirms that there indeed exists performance difference between the evaluated algorithms). For the ease of the presentation, let $L = \{1, 2, 3, 4, 5, 6\}$ denotes $L = \{[1\mu s, 15\mu s]$, $[15\mu s, 50\mu s]$, $[50\mu s, 100\mu s]$, $[100\mu s, 200\mu s]$, $[200\mu s, 300\mu s]$, $[1\mu s, 300\mu s]\}$ in the tables given below.

Table E.5: The F Values Computed by the ANOVA Test.

Experiments		F Values	Experiments		F Values
Figure 3.3	$M = 8$	17454.48	Figure 3.5	$L = 3$	37428.87
	$M = 12$	60333.57		$L = 5$	187469.8
Figure 3.6	$A = 11$	11974.93	Figure 3.7	$L = 2$	2992.24
	$A = 26$	18010.53		$L = 4$	2984.64
Figure 4.1	$L = 3$	198599.54	Figure 4.4a	$A = 6$	297220.21
	$L = 5$	336446.76		$A = 16$	95161.15
Figure 4.7a	$L = 1$	2838.28	Figure 4.7b	$L = 2$	3444.43
	$L = 6$	1210.17		$L = 5$	36888.78
Figure 6.3	$A = 1$	1839.49	Figure 6.4	$L = 2$	408.35
	$A = 26$	845.43		$L = 4$	925.68
Figure 6.6	$M = 14$	2035.05	Figure 6.8	$L = 3$	569.59
	$M = 18$	9766.1		$L = 5$	3540.27

Bibliography

[1] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, and C. Siemers. An efficient spin-lock based multi-core resource sharing protocol. In *Performance Computing and Communications Conference (IPCCC), 2014 IEEE International*, pages 1–7. IEEE, 2014.

[2] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems (TOCS)*, 15(2):134–165, 1997.

[3] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 337–346. IEEE, 2000.

[4] D. Andre. The evolution of agents that build mental models and create simple plans using genetic programming. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 248–255. Morgan Kaufmann Publishers Inc., 1995.

[5] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte. Towards hierarchical scheduling in AUTOSAR. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–8. IEEE, 2009.

[6] G. Ascia, V. Catania, and M. Palesi. A multi-objective genetic approach to mapping problem on network-on-chip. *J. UCS*, 12(4):370–394, 2006.

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept 1993.

[8] N. C. Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* University of York, Department of Computer Science, 1991.

[9] N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[10] T. Back. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 57–62. IEEE, 1994.

[11] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[12] T. P. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *In International Conf. on Real-Time and Network Systems*. Citeseer, 2005.

[13] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2009.

[14] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[15] A. Biondi, B. B. Brandenburg, and A. Wieder. A blocking bound for nested FIFO spin locks. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 291–302. IEEE, 2016.

[16] K. Bletsas and N. Audsley. Optimal priority assignment in the presence of blocking. *Information processing letters*, 99(3):83–86, 2006.

[17] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 47–56. IEEE, 2007.

[18] B. B. Brandenburg. "SchedCAT: Schedulability test collection and toolkit". `http://www.mpi-sws.org/bbb/projects/schedcat`. Accessed: 2016-11-1.

[19] B. B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011. `https://cs.unc.edu/~anderson/diss/bbbdiss.pdf`.

[20] B. B. Brandenburg and J. H. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 49–60. IEEE, 2010.

[21] B. B. Brandenburg and J. H. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design automation for embedded systems*, 17(2):277–342, 2013.

[22] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 342–353. IEEE, 2008.

[23] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, 1991.

[24] A. Burns, R. I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C= D task splitting scheme. *Real-Time Systems*, 48(1):3–33, 2012.

[25] A. Burns and A. Wellings. *Analysable Real-Time Systems: Programmed in Ada*. CreateSpace Independent Publishing Platform, 2016.

[26] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[27] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol–mrsp. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 282–291. IEEE, 2013.

[28] G. C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29(1):5–26, 2005.

[29] M. V. Butz, K. Sastry, and D. E. Goldberg. Tournament selection: Stable fitness pressure in XCS. In *Genetic and Evolutionary Computation Conference*, pages 1857–1869. Springer, 2003.

[30] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 111–126. IEEE, 2006.

[31] J. Cane and T. Manikas. Genetic algorithms vs simulated annealing: A comparison of approaches for solving circuit partitioning problem. Technical report, University of Pittsburgh, 1996.

[32] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[33] S. Catellani, L. Bonato, S. Huber, and E. Mezzetti. Challenges in the implementation of MrsP. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 179–195. Springer, 2015.

[34] Y. Chu and A. Burns. Flexible hard real-time scheduling for deliberative AI systems. *Real-Time Systems*, 40(3):241–263, 2008.

[35] R. I. Davis and A. Burns. Robust priority assignment for fixed priority real-time systems. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 3–14. IEEE, 2007.

[36] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[37] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *Acm Computing Surveys*, 43(4):1–44, 2011.

[38] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns. A review of priority assignment in real-time systems. *Journal of systems architecture*, 65:64–82, 2016.

[39] K. A. De Jong. *Analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, Ann Arbor, MI, USA, 1975. `https://deepblue.lib.umich.edu/handle/2027.42/4507`.

[40] S. N. Dinh, J. Li, K. Agrawal, C. Gill, and C. Lu. Blocking analysis for spin locks in real-time parallel tasks. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[41] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 377–386. IEEE, 2009.

[42] P. Emberson and I. Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *Real-Time Systems Symposium, 2008*, pages 270–279. IEEE, 2008.

[43] P. Emberson and I. Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *IEEE Transactions on Software Engineering*, 36(5):704–718, 2010.

[44] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 90–99. IEEE, 2010.

[45] N. Fisher, S. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10–pp. IEEE, 2006.

[46] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. AUTOSAR–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, 2009.

[47] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 189–198. IEEE, 2003.

[48] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In

Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE, pages 73–83. IEEE, 2001.

[49] J. Garrido, S. Zhao, A. Burns, and A. Wellings. Supporting nested resources in MrsP. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 73–86. Springer, 2017.

[50] L. George, N. Rivierre, and M. Spuri. *Preemptive and non-preemptive real-time uniprocessor scheduling.* PhD thesis, Inria, 1996. `https://hal.inria.fr/inria-00073732`.

[51] D. E. Goldberg. *The design of innovation: Lessons from and for competent genetic algorithms*, volume 7. Springer Science & Business Media, 2013.

[52] J. R. Haritsa, M. J. Carey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal-The International Journal on Very Large Data Bases*, 2(2):117–152, 1993.

[53] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference, General Track*, pages 217–230, 2001.

[54] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[55] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 111–122. IEEE, 2016.

[56] J. Jones. Stats: One-Way ANOVA. `https://people.richland.edu/james/lecture/m170/ch13-1wy.html`. Accessed: 2018-03-02.

[57] J. Jones. Stats: Two-Way ANOVA. `https://people.richland.edu/james/lecture/m170/ch13-2wy.html`. Accessed: 2018-03-02.

[58] L. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans. Trends in multicore DSP platforms. *IEEE Signal Processing Magazine*, 26(6), 2009.

[59] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Euromicro Conference on Real-Time Systems*, pages 249–258, 2009.

[60] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[61] J. Koljonen and J. T. Alander. Effects of population size and relative elitism on optimization speed and reliability of genetic algorithms. In *Proceedings of the ninth Scandinavian conference on artificial intelligence (SCAI 2006)*, pages 54–60. Citeseer, 2006.

[62] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 469–478. IEEE, 2009.

[63] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209. IEEE, 1990.

[64] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1-4):209, 1989.

[65] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[66] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *Computers IEEE Transactions on*, 44(12):1429–1442, 1995.

[67] D. J. Lilja. *Measuring computer performance: a practitioner's guide.* Cambridge university press, 2005.

[68] S. Lin. *A Flexible Multiprocessor Resource Sharing Framework for Ada.* PhD thesis, University of York, 2013. `http://etheses.whiterose.ac.uk/5668/`.

[69] S. Lin, A. Wellings, and A. Burns. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages.

Concurrency and Computation: Practice and Experience, 25(16):2227–2251, 2013.

[70] G. Lipari, G. Lamastra, and L. Abeni. Task synchronization in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601, 2004.

[71] C. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. *JPL Space Programs Summary, II (1)*, pages 37–60, 1969.

[72] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[74] MathWorks. One-sample Kolmogorov-Smirnov Test. `https://uk.mathworks.com/help/stats/kstest.html`. Accessed: 2018-06-25.

[75] H. Mei. *Real-Time Stream Processing in Embedded Systems.* PhD thesis, University of York, 2018. `http://etheses.whiterose.ac.uk/19750/`.

[76] H. Mei and A. Wellings. Using jetbench to evaluate the efficiency of multiprocessor support for parallel processing. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 47. ACM, 2014.

[77] P. Mesidis and L. S. Indrusiak. Genetic mapping of hard real-time applications onto noc-based mpsocs-a first approach. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–6. IEEE, 2011.

[78] B. L. Miller and D. E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary computation*, 4(2):113–131, 1996.

[79] M. Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[80] T. Nair and K. Sooda. Comparison of genetic algorithm and simulated annealing technique for optimal path selection in network routing. *arXiv preprint arXiv:1001.3920*, 2010.

[81] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. *Principles of Distributed Systems*, pages 253–269, 2010.

[82] G. Ochoa, I. Harvey, and H. Buxton. Optimal mutation rates and selection pressure in genetic algorithms. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, pages 315–322. Morgan Kaufmann Publishers Inc., 2000.

[83] S.-H. Oh and S.-M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 31–36. IEEE, 1998.

[84] G. F. Pfister and V. A. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 100(10):943–948, 1985.

[85] F. Pölzlbauer, I. Bate, and E. Brenner. On extensible networks for embedded systems. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 69–77. IEEE, 2013.

[86] V. Poorjafari, W. L. Yue, and N. Holyoak. A comparison between genetic algorithms and simulated annealing for minimizing transfer waiting time in transit systems. *International Journal of Engineering and Technology*, 8(3):216, 2016.

[87] A. Racu and L. S. Indrusiak. Using genetic algorithms to map hard real-time on noc-based systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8. IEEE, 2012.

[88] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123. IEEE, 1990.

[89] R. Rajkumar. *Synchronization in real-time systems: a priority inheritance approach.* Kluwer Academic Publishers, 1991.

[90] R. Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*, volume 151. Springer Science & Business Media, 2012.

[91] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269. IEEE, 1988.

[92] S. Rezeka. Modelling and sensitivity analysis of an ABS hydraulic modulator. In *American Control Conference, 1994*, volume 1, pages 826–830. IEEE, 1994.

[93] L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization.* IEEE Computer Society, 1990.

[94] J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings. Implementation and evaluation of multiprocessor resource synchronization protocol (MrsP) on litmus$^{\mathrm{RT}}$. *In Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

[95] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conference on Real-Time Systems*, pages 181–190, 2008.

[96] H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings E (Computers and Digital Techniques)*, 137(1):17–30, 1990.

[97] S. S.-F. Smith. Using multiple genetic operators to reduce premature convergence in genetic assembly planning. *Computers in Industry*, 54(1):35–49, 2004.

[98] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *computer*, 27(6):17–26, 1994.

[99] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 433–440. IEEE, 2000.

[100] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 134–143. IEEE, 1997.

[101] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[102] L. A. University of California. F-Distribution Tables. `http://www.socr.ucla.edu/Applets.dir/F_Table.html`. Accessed: 2018-03-02.

[103] Y.-C. Wang and K.-J. Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 246–255. IEEE, 1999.

[104] A. J. Wellings. *Concurrent and real-time programming in Java*. John Wiley, 2004.

[105] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 266–277. ACM, 2007.

[106] A. Wieder and B. B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 45–56. IEEE, 2013.

[107] O. U. P. Zapata and P. M. Alvarez. EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation. *Seccion de Computacion Av. IPN*, 2508, 2005.

[108] S. Zhao. Implementing MrsP on fully partitioned systems. Technical report, The University of York, Department of Computer Science, 2016. `https://github.com/RTSYork/mrsp/blob/master/ImplementatingMrsP.pdf`. Accessed by 17/Feb/2018.

[109] S. Zhao, J. Garrido, A. Burns, and A. Wellings. New schedulability analysis for MrsP. In *Embedded and Real-Time Computing Systems and*

Applications (RTCSA), 2017 IEEE 23rd International Conference on, pages 1–10. IEEE, 2017.

[110] S. Zhao and A. Wellings. Investigating the correctness and efficiency of MrsP in fully partitioned systems. In *The 10th York Doctoral Symposium on Computer Science and Electronics*. The University of York, 2017.

[111] Q. Zhu, Y. Yang, M. Natale, E. Scholte, and A. Sangiovanni-Vincentelli. Optimizing the software architecture for extensibility in hard real-time distributed systems. *IEEE Transactions on Industrial Informatics*, 6(4):621–636, 2010.

[112] A. Zuhily and A. Burns. Optimal (D-J)-monotonic priority assignment. *Information Processing Letters*, 103(6):247–250, 2007.