Extending the Real Time Specification for Java for Cache Coherent NUMA Architectures

Abdul Haseeb Malik

Submitted for the Degree of Doctor of Philosophy

University of York
Department of Computer Science

January 2012

Abstract

The recent past has seen single processing systems becoming obsolete and multiprocessor systems taking over as the architecture of choice. More scalable architectures are being introduced to keep up with the ever increasing computational requirements. Inevitably the software industry has to keep up and provide tools which would allow easy development of parallel applications on these complex architectures. Programming languages unable to support these architectures will become obsolete and new languages will be developed designed to serve the purpose. Existing languages that have the potential of coping with such architectures will be extended.

Java is an object oriented language which provides features like platform independence and concurrency by design. It is supported on a wide range of computational devices starting from high performance computing servers to small hand held mobiles. This makes Java a very popular language, with a large pool of programmers. Therefore, it is not strange to see Java being extended for different multiprocessor and distributed systems.

The Real-Time Specification for Java (RTSJ) was introduced as a high level platform for developing real-time applications. It has been gaining popularity since its inception and is being increasingly used for real time development. The recent version of RTSJ (i.e. RTSJ 1.1) has included some support for multiprocessors. The changes that have been introduced are largely related to the scheduling and dispatching of threads in symmetric multiprocessors. It is assumed that the memory allocations will not affect the timing properties of the system and it will be accessed uniformly. This thesis argues that these changes are not enough to support shared memory multiprocessors on RTSJ; and the following issues need to be addressed:

- In shared memory multiprocessors, uniform memory access (UMA) systems are not scalable and will be replaced by cache coherent non-uniform memory access (cc-NUMA) systems. Locality can largely affect the performance of applications and cause unpredictable delays in real-time applications on cc-NUMA systems. These delays are caused by the distributed memory layout in cc-NUMA and can be reduced by giving applications more control and visibility into the allocation policies of threads and memory.

- The increased computational power of multiprocessors encourages the platform to be shared by multiple applications. In such an environment, real-time applications require temporal guarantees from the platform to meet their timing requirements.

The key contribution of this thesis is to extend the programming model of RTSJ to provide programmers with locality constraints. Existing support in RTSJ for locality is shown to be very limited and a new locality model is presented which enables programmers to develop portable applications. Applications are divided into multiple partitions and temporal guarantees are provided to partitions allowing them to be analyzed in isolation from the rest of the system.

A prototype of the aforementioned model is implemented and tested. A series of experiments are conducted to prove the effectiveness of the model. In particular, the effect of locality is highlighted by performance measuring in a multithreaded environment.

Contents

Abstract iii

Acknowledgements xix

Declaration xxi

1 Introduction 1

1.1 The Real-time Specification for Java . . . 3

1.2 Cache Coherent Non-Uniform Memory Access (cc-NUMA) Architecuture . . . 5

1.3 Support for Multiprocessors in RTSJ . . . 8

1.4 Motivation . . . 9

1.5 Hypothesis . . . 11

1.6 Thesis Aims . . . 11

1.7 Thesis Structure . . . 12

1.8 Summary . . . 13

2 Literature survey 15

2.1 Parallel Programming on Multiprocessors . . . 16

2.1.1 Parallelism . . . 16

2.1.2 Locality . . . 17

2.2 Real Time Systems and Multiprocessors . . . 21

2.3 Java on Multiprocessors . . . 24

2.3.1 Java on Distributed Systems . . . 24

2.3.2 Java on Shared Memory Multiprocessors . . . 30

2.3.3 Discussion . . . 31
2.4 Summary and Requirements . . . 31

3 Existing Support for cc-NUMA Architectures 33

3.1 Linux Support for cc-NUMA . . . 34
3.1.1 Discovering the Architecture . . . 34
3.1.2 Allocating Threads and Objects . . . 36
3.1.3 Supporting Group Budgets . . . 38
3.2 RTSJ and its Support for Multiprocessors . . . 39
3.2.1 The *AffinitySet* Class . . . 43
3.2.2 The Physical Memory Framework . . . 45
3.2.3 Processing Group Parameters (PGPs) . . . 47
3.3 Supporting cc-NUMA Systems . . . 50
3.3.1 Representing the NUMA Architecture . . . 53
3.3.2 Pinning Schedulable Objects to Processors . . . 57
3.3.3 Allocating Objects on Specific Nodes . . . 58
3.4 Limitations . . . 60
3.5 Summary . . . 62

4 Locality Model 65

4.1 Architectural Model . . . 66
4.1.1 Abstractions for Basic Architectural Components . . . 68
4.1.1.1 The Abstract Component Class . . . 68
4.1.1.2 The *Processor* Class . . . 69
4.1.1.3 The Memory Class . . . 71
4.1.1.4 The Device Class . . . 72
4.1.2 Abstractions for Architecture Representation . . . 72
4.1.2.1 Location . . . 72
4.1.2.2 The Locale Class . . . 73
4.1.2.3 The Neighbourhood Class . . . 73
4.1.2.4 The District Class . . . 74
4.1.3 Interface to the Architecture . . . 74

4.1.4 Discussion 75
4.2 Application Model 77
4.2.1 *ExecutionSite*: Abstraction for Controlling Execution 77
4.2.1.1 Threads/Schedulables Instantiation 80
4.2.1.2 Memory Areas Instantiations 80
4.2.1.3 Retrieving Local Heap and Immortal 80
4.2.1.4 Basic Resource Reservation Operations 81
4.2.2 *Place*: A Logical Location 81
4.2.3 *Locality*: Allocating *ExecutionSite* 83
4.2.4 Local ImmortalMemory and Local Heap 85
4.2.5 Conforming to the RTSJ 87
4.2.5.1 RTSJ Rules 88
4.2.5.2 Required Semantics of the Locality Model 89
4.2.5.3 Using the Factory Pattern 90
4.2.6 The Default Execution Model 91
4.2.7 Example 92
4.2.7.1 Static Allocation of Execution Sites 92
4.2.7.2 Implicit Allocation of Execution Sites 95
4.3 Resource Reservations 98
4.3.1 Interface 99
4.3.1.1 External Contract 102
4.3.1.2 Partitioned Reservations 103
4.3.2 Scheduling: The *ReservationScheduler* Class 105
4.3.3 Admission Control 106
4.3.4 Cost Enforcement 108
4.3.5 Discussion 108
4.4 Summary 111

5 Implementation **115**
5.1 Implementation Overview 116
5.1.1 Implementing the AffinitySet Class 117
5.1.2 Access to the Local Memory 118

5.1.3 Extensions Required for the Locality Model 120
5.2 Implementing the Architecture Representation 122
5.3 Implementing the Application Model 127
5.3.1 Creating *Places* 128
5.3.2 ExecutionSite Creation 129
5.3.3 Thread/Schedulables Creation 133
5.3.4 MemoryArea Creation 135
5.4 Implementing Reservations Model 136
5.4.1 Cost Enforcement 138
5.4.2 Priority Assignment 139
5.5 Summary 141

6 Evaluation 143

6.1 Programmability 144
6.2 Portability 147
6.3 Performance 149
6.3.1 The Producer/Consumer Problem 149
6.3.2 The Prime Sieves Example 153
6.3.2.1 Effect of the Locality Model on Performance 155
6.3.2.2 Trade-off between Locality and Load Balancing . . . 157
6.4 Predictability 162
6.4.1 Dispersions 162
6.4.1.1 Locality Model vs. Normal Case 162
6.4.1.2 Local vs. Remote 163
6.4.1.3 Locality vs. Load Balancing 164
6.4.2 Temporal Isolation 165
6.5 Overheads 168
6.5.1 Architecture Representation 168
6.5.2 Application Model 169
6.5.2.1 Creating Places 170
6.5.2.2 Creating an ExecutionSite 171
6.5.2.3 Realtime Thread Creation 172

6.5.2.4 Realtime Thread Startup Latency Without Reservations . . . 173
6.5.2.5 Memory Area Creation . . . 174
6.5.2.6 Allocation Time Test . . . 176
6.5.3 Reservation Model . . . 177
6.5.3.1 Creating ReservationServers . . . 177
6.5.3.2 Realtime Thread Startup Latency using Reservations 178
6.6 Summary . . . 180

7 Conclusions and Future Work 181
7.1 Summary . . . 181
7.2 Contributions . . . 183
7.3 Future Work . . . 184

Appendix 189

A Memory Access Timings on Cache Coherent NUMA Systems 189
A.1 Comparing Local And Remote Memory Accesses . . . 192
A.2 Comparing Access Timings for Different Inter-Connect Speeds . . . 192
A.3 Comparing Access Timings for Different NUMA Distances . . . 195

B The Producer Consumer Example 197
B.1 Statically Allocating ExecutionSites . . . 197
B.2 Allocating ExecutionSites Dynamically by the Runtime . . . 202
B.3 Allocating ExecutionSites with Reservations . . . 206

C Building JRate on 64bit Systems 211

D Schedulability Analysis 213
D.1 Scheduling Model . . . 213
D.2 Top Level Schedulability Test . . . 214
D.3 Local Schedulability Test . . . 214

E The Prime Sieves Example **221**

E.1 Without Using the Locality Model 221

E.2 Using the Locality Model 225

F Overheads of Libcgroup **231**

F.1 Overheads Creating a ReservationServer/Place 231

F.2 Attaching a Thread to a ReservationServer 233

F.3 Summary 234

References **236**

Abbreviations **237**

List of Classes **239**

List of Figures

1.1 The Uniform Memory Access(UMA) Architecture 6
1.2 The Non-Uniform Memory Access(NUMA) Architecture 7
1.3 Time Taken by memcpy() 10

2.1 The bounded delay model for single processors 23
2.2 Java DSM built over existing DSM 28
2.3 Native Java DSMs 28
2.4 Java DSM with built-in DSM 29

3.1 Memory classes in the RTSJ 41
3.2 Schedulable objects in RTSJ 42
3.3 Scheduling in RTSJ 43
3.4 The AffinitySet Class 45
3.5 Capacity on a single processor 48
3.6 Single global capacity on multiple processor 48
3.7 Partitioned equal capacities on multiple processors 49
3.8 Partitioned different capacities on multiple processors 49
3.9 A 4 Node NUMA Architecture based on AMD Opteron 50
3.10 Memory Hierarchy of a NUMA System 51
3.11 Creating an LTPhysicalMemory area 59
3.12 Scope stack for a *ThreadGroup* 62

4.1 Abstractions for Architecture Representation 69
4.2 System Representation at the JVM level 76
4.3 Memory Areas in the Locality Model 85

4.4 Scope stack showing memory access violations 89
4.5 Remote allocation of producer/consumer 94
4.6 Local allocation of producer/consumer 97
4.7 Global Budget 100
4.8 Partitioned Budget 101
4.9 The Locality Model 113

5.1 Simulating global scheduling in Linux 118
5.2 Memory access timings 121
5.3 Sequence diagram: building the Architectural Representation 123
5.4 Graphical Output of lstopo(hwloc) 127
5.5 Implementing the Locality model using the Control Groups 128
5.6 Sequence diagram: initializing the virtual platform 130
5.7 Sequence diagram: creating an execution site using a factory method 132
5.8 Creating and starting thread using the Locality model. 133
5.9 Sequence diagram: Creating a LTPhysicalMemory area 137
5.10 Over-run on a single processor 141

6.1 Architectural Representation of a single processor system 148
6.2 Architectural Representation of a two processor SMP 149
6.3 Execution times for local producer consumer problem using the locality Model 150
6.4 Execution times for remote producer consumer problem using the locality model 153
6.5 Sieve of Eratosthenes 154
6.6 Prime numbers for all N 155
6.7 Number of threads created for all N 155
6.8 Execution times for generating all prime numbers less than N=20000 using the locality model 157
6.9 Execution times for generating all prime numbers less than N=20000 in the normal case 160

6.10 Comparison of execution times using the locality model under different configurations . . . 161
6.11 Architecture representation overheads . . . 169
6.12 Places Creation . . . 170
6.13 ExecutionSites Creation . . . 171
6.14 Real-time Threads Creation . . . 172
6.15 Real-time threads startup latency without reservations . . . 174
6.16 Scoped memory Area Creation Timings . . . 175
6.17 Object Allocation Timings . . . 176
6.18 ReservationServer Creation . . . 178
6.19 Real-time Threads Startup Latency . . . 179

7.1 Execution sites in a real-time Java application . . . 186

A.1 memcopy() timings(in seconds) for different interconnect speeds . . . 194
A.2 Comparing Access Timings for Different NUMA Distances . . . 195

D.1 Scheduling Architecture of Execution Sites on a Processor Based Budget . . . 215
D.2 Bandwidth requirements of $T_1 T_2 T_3$ . . . 219
(a) All tasks' bandwidth calculation . . . 219
(b) α_1, α_2 plane for all tasks' bandwidth . . . 219

F.1 cgroup_init() timings . . . 231
F.2 cgroup_new_cgroup() timings . . . 232
F.3 cgroup_add_controller() timings . . . 232
F.4 cgroup_set_value_string() timings . . . 233
F.5 cgroup_create_cgroup() timings . . . 233
F.6 cgroup_attach_thread() timings . . . 234

List of Tables

3.1 Distances based on bus accesses for Figure 3.9 52

3.2 System Locality Information Table for Figure 3.9 53

3.3 The System Resource Affinity Table (SRAT) for architecture in figure 3.9 showing 16 processors 54

3.4 The System Resource Affinity Table (SRAT) for architecture in figure 3.9 showing memory 54

4.1 The *Component* Class 69

4.2 The *Processor* Class 70

4.3 The *ProcessorType* Interface 70

4.4 The *Cache* Class 71

4.5 The *Memory* Class 71

4.6 The *Device* Class 72

4.7 The abstract *Location* Class 73

4.8 The *Locale* Class 73

4.9 The *Neighbourhood* Class 74

4.10 The *District* Class 74

4.11 The *Platform* Class 75

4.12 The *ExecutionSite* Class 79

4.13 The *Place* Class 82

4.14 The final *Locality* Class 85

4.15 The *HeapPhysicalMemory* Class 87

4.16 Memory assignment rules in the RTSJ 88

4.17 Scope stack inheritance rules in RTSJ 90

4.18 The *PartitionedParameters* Class 102
4.19 The *ExternalContract* Class 102
4.20 The *ClusterContract* Class 103
4.21 The *PartitionedReservation* Class 104
4.22 The *ReservationScheduler* Class 105
4.23 The *ReservationServer* Class 108

5.1 The *LocalMemory* Class 119
5.2 The *ESRealtimeThread* Class 134
5.3 The *ESNoHeapRealtimeThread* Class 134

6.1 Execution times (in microseconds) for local producer/consumer . . . 151
6.2 Execution times (in microseconds)for remote producer/consumer . . . 152
6.3 Execution times (in milliseconds) statistics for the prime sieves in the locality model case 158
6.4 Execution times (in milliseconds) statistics for the prime sieves in the normal case 159
6.5 Analyzing the execution times of all cases 161
6.6 Architecture representation overhead (milliseconds) statistics 169
6.7 Places creation execution times in microseconds 171
6.8 ExecutionSites Creation Execution Times in Microseconds 172
6.9 Real-time Threads Creation Execution Times 173
6.10 Real-time threads startup latency (in microseconds) without reservations statistics 174
6.11 Scoped memory Area Creation Timings (milliseconds) Statistics . . . 176
6.12 Object Allocation Timings (milliseconds) Statistics 177
6.13 Creating ReservationServer Timings (microseconds) 178
6.14 Real-time Threads Startup Latency (in microseconds) Statistics . . . 179

A.1 memcpy() timings (in milliseconds) with N0-N1 interconnect at 200Mhz 193
A.2 memcopy() timings (in seconds) for different interconnect speeds . . . 194
A.3 memcopy() timings (in seconds) with N0-N1 interconnect at 1Ghz . . 196

D.1 Symbols used for schedulability analysis 216
D.2 Workload for the execution site 217
D.3 Candidate interfaces of the execution site 219

F.1 LibCGroup Overhead statistics 235

Acknowledgements

This thesis would have not been possible without the guidance and support of my supervisor, Professor Andy Wellings. I am greatly indebted to him for his time, understanding, patience and support. His invaluable advice and support guided me my ideas into this thesis.

I would like to thank all the members of the RTS Group especially Professor Alan Burns, Dr. Neil Audsley, Dr. Rob Davis, Dr. Yang Chang and Dr. Mohammad AlRahmawy for their help and support. A large part of my work involved implementation work on Linux and I was lucky to have two gurus of Linux, Seyeon Kim and Sitsofe Wheeler. I am grateful for their time and advice which made my job easier.

I would like to thank all my friends, Furqan Aziz, Muhammad Haseeb, Tasawer Khan, Usman Khan, Dr. Ahmad Shahid, Thomas Richardson, Shiyao Lin and Emad Al-Oqayli for giving me strength, self belief and all the enjoyable moments.

My family has been supportive of me throughout my Ph.D. I would like to thank my wife, my son Abdullah and my daughter Hafsah for their patience, sacrifices and support. I also thank my sister, for encouraging me throughout my work. Lastly, but not the least, I would like to thank my father for guiding and supporting me in every possible way, and my mother for her love, support and patience.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2007 - 2012. Except where stated, all of the work contained within this thesis represents the original contribution of the author.

Some parts of Chapter 4 are based on the paper presented in the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010) entitled *A Locality Model for the Real-Time Specification for Java* [Malik et al., 2010].

Chapter 1

Introduction

Java is a platform independent, strongly typed and secure language which has gained considerable popularity since its inception in 1994. It is a high level language which has a syntax very similar to C and C++, however, it discards the complexities of these languages to provide the programmers with a simple and powerful environment. Applications designed for Java follow the *Write Once, Run Anywhere (WORA)* philosophy, hence, applications must be able to run anywhere without prior knowledge of the target hardware and software platform. As a result, Java applications are being used on a range of platforms from large server machines to small embedded devices such as mobiles.

The real-time specification for Java (RTSJ) extends the Java programming language to provide a platform for the programmers to develop and execute applications that have some specific timing constraints. Typically real-time applications monitor events in the real world and then responds to those events in an appropriate manner within a finite period of time. Some real-time systems can tolerate occasional deadlines misses (in soft real-time systems) while in others missing the deadlines is as bad as producing a wrong result (in hard real-time systems). In order to support such systems, a large number of extensions have been made to the standard Java technology especially to the scheduling and the memory models to make the execution of programs more predictable.

Recent years have seen a change of paradigm in the computing world from single processor systems to multiprocessor systems. Single processor systems had reached

the physical limits of speed and a large amount of effort and cost was required for the little gain in performance [Asanovic et al., 2006]. As a result, now single processor systems are becoming obsolete and vendors are replacing them with multi-cores. In 2004, Intel canceled the development of two of their high speed microprocessors to focus on the development of multi-core processors [Flynn, 2004]. Newer architectures are replacing high-clock-speed processors with many simpler and smaller processing elements. The number of cores in future system will be in thousands [Asanovic et al., 2006]. In order to support such parallelism, the memory sub-system also becomes distributed. These architectures will be capable of producing very high performance; however, their programming will be more complex than in the single processor case.

Real-time systems are also increasing in size and complexity and the only architectures that will be able to satisfy the needs of these applications will be multiprocessors. At the same time, it is quite possible that the real-time systems may be sharing the same platform with many other applications. These applications can be of different criticality levels and may have diverse timing requirements. Typically, real-time systems require timing requirements of all the different applications running simultaneously with the real-time application to be known a priori to ensure that timing requirements are met. However, on large systems it becomes nearly impossible to do the global analysis of the system. Hence, architectural complexity and tight timing constraints make the development of real-time systems on multiprocessor architectures extremely difficult. The usual objective of programming platforms for these systems is to hide the complexity of the architectures so that the programmers are not distracted by low-level architectural issues. However, for real-time systems, the programming languages need to provide high level constructs with semantics to use the underlying architecture predictably and efficiently. If achieved, this will lead to programmers being more productive and at the same time predictability and performance of the applications will also increase.

Java's built-in support is adequate to make sure it runs seemlessly on shared memory multiprocessors even if memory access timings become non-uniform. However, for real-time applications, the programmer will want more control over the

allocation policies of threads and objects. The current RTSJ, i.e. RTSJ 1.02, does not support multiprocessors, however, a number of extensions have been proposed for RTSJ 1.1, these are discussed in Section 1.3. The focus of these changes is to provide a set of tools to support the construction of parallel real-time Java systems for symmetric multiprocessor systems (SMPs) having uniform memory access timings. In such systems, there is little to be gained (performance wise) from the programmer having direct control over where data is stored. From a schedulability analysis perspective, the programmer may still require control over where threads are executed, but control of where data is placed has no effect. However, in systems where memory timings change depending in the processor from which it is being accesses, then in such systems, control of data becomes very important because remote accesses can cause unpredictable delays. These delays not only effect the performance of applications but can lead to very pessimistic worst case timing analysis.

In this thesis, we focus on the support that we can provide in RTSJ for soft real-time systems to execute on cache coherent non-uniform memory access (cc-NUMA) systems by providing them resource guarantees. Calculating the worst case execution time (WCET) for such systems is out of the scope of this thesis, however, the effect of non-uniform memory access (NUMA) on the execution times will be analysed.

Section 1.1 gives an overview of the RTSJ, followed by Section 1.2 which discusses different types of shared memory multiprocessors which include uniform and non-uniform memory access systems. Section 1.3 outlines the existing support for multiprocessors in RTSJ. Section 1.4 presents the motivation of supporting cc-NUMA systems in RTSJ followed by the hypothesis in Section 1.5 and the aims of the thesis in Section 1.6. Section 1.7 outlines the basic structure of the thesis. Section 1.8 is the last section which summarizes the whole chapter.

1.1 The Real-time Specification for Java

The correctness of a real-time system depends on two things: the logical results of the computation and the time at which they are produced. Real Time Systems are

defined by [Young, 1982] (as cited in [Burns and Wellings, 2001]) as "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period". Meeting the deadlines does not mean that the system should be super fast; rather it requires the system to be predictable. Real-time Systems can be classified broadly in to hard or soft real-time. Hard real-time systems are very strict in terms of their timing requirements and missing any specified deadline can have catastrophic consequences. Examples of hard real-time systems include flight control system of a combat aircraft, braking system on a car or some industrial process control system. Soft real-time systems on the other hand are much more relaxed in terms of their timing requirements and occasionally deadlines can be missed, however, missing the deadline can lead to degraded quality of service and usually does not have any critical consequences. Examples of soft real-time systems include audio/visual systems which require a specific frame rate to ensure an enjoyable experience.

Since its introduction, Java was designed to be a high performance language with dynamic features such as garbage collection and just in time compilation. In addition there was no support to implement different scheduling policies for thread scheduling within the application. Mostly it relied on the OS scheduler to dispatch threads for execution; which meant that it is quite possible that the highest priority thread might be waiting because a low priority thread cannot be preempted. In order to support real-time systems (RTSs), the Java community has developed the RTSJ which makes a number of extensions to make the execution of the program predictable. Among these changes the most important are the changes that have been introduced to bound the delays of the garbage collector and introduce a new scheduling model to support different scheduling policies.

The Java Language Specification [Gosling et al., 2005] implicitly defines the behaviour of the garbage collector by defining the life cycle of a Java object. Most Java virtual machines (JVMs) implement the lifecycle by using a garbage collector that traverses through all the unreachable objects to reclaim the memory in the heap. In such environments, garbage collector disrupts the normal program execution and causes unpredictable delays, which are not desirable in real-time systems.

The RTSJ introduces new memory areas which are not affected by the garbage collector. These memory areas include the scoped memory areas and the immortal memory area. The scoped memory areas are coarse grained memory areas where objects are not collected individually, however, the entire memory area is reclaimed. Immortal memory areas are never collected, therefore, any object allocated inside here remains forever. These new memory areas provide an alternative to the heap memory area which is the only memory area used for object allocation in Java.

The RTSJ also introduces new threads which are capable of avoiding any interference from the garbage collector. These threads implement a new interface, *Schedulable*, which attach these threads with scheduling parameters. The *Schedulable* interface is implemented by *RealtimeThread* and *AsyncEventHandler* classes and are called schedulable objects or simply schedulables. The scheduling parameters are used to implement different policies and executing schedulables along with the new memory areas allows RTSJ control over the behaviour of the garbage collector.

1.2 Cache Coherent Non-Uniform Memory Access (cc-NUMA) Architecuture

Shared memory multiprocessors (SMMPs) consist of a number of processors connected together with the help of shared memory. Processors communicate with each other with the help of shared variables [Hennessy and Patterson, 2006]. An important characteristic of the SMMP is the presence of a single address space for all processors. The single address space enables the processors to access memory using simple load and store instructions.

SMMPs can further be classified as uniform memory access (UMA) and non uniform memory access (NUMA) depending upon the uniformity of the time taken by different processors to access memory. UMA is a computer architecture defined as "a multiprocessor in which all processors work through a central switching mechanism to reach a shared global memory" [Quinn, 1994].

The multiprocessors which follow UMA architecture are Symmetric Multiprocessors (SMPs) [Hennessy and Patterson, 2006]. SMPs have 2 or more homogeneous

processors which are connected to a shared memory through a common bus. Although there are multiple processors, the memory is uniformly accessible by all of them, hence all data is at the same distance from all the processors as shown in Figure 1.1. The examples of SMPs are SGI Power Challenge, DEC Alpha Server 8400, CDC6600, IBM Power4 and IBM Power 5.

On an SMP, each processor has equal priority to access the shared memory and I/O devices. The shared memory bus (or the shared memory controller) becomes congested during memory accesses as multiple processors use the bus simultaneously. A single OS image runs on top of the SMPs and handles the hardware. As the number of processors increases, bus contention becomes a problem which results in high latency memory accesses. The use of large caches reduces the use of the bus. They provide low latency access to the memory and reduce the bandwidth requirement of the processors. Each processor is able to cache data, and with multiple caches around it is possible that the data is cached at more than one place. This requires that the data in different caches and in the memory is coherent. Updates performed by a processor should become visible to all processors before they use it. This requires the implementation of cache coherent protocols which ensure consistency of data in caches. These protocols have been discussed in detail in [Hennessy and Patterson, 2006].

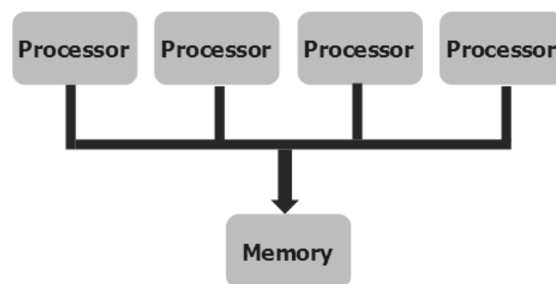


Figure 1.1: The Uniform Memory Access(UMA) Architecture

NUMA is a shared memory multiprocessor which does not support constant time memory access. According to [Quinn, 1994] “in most NUMA architectures, memory is organized hierarchically, so that some portions can be read and written more quickly by some processors than by others”. SMPs do not scale very well because resources are shared with equal priority given to all nodes which causes congestion.

NUMA systems however are somewhat distributed in nature; some memory banks are closer to some processors while farther from others. They are typically created by coupling SMPs together by connecting the memory controllers through a high speed interconnect; as a result NUMA systems scale better than a SMP. While a processor can access all memory areas, still latency of memory accessed depends on where the memory is allocated. Figure 1.2 shows a generic 4 node NUMA system.

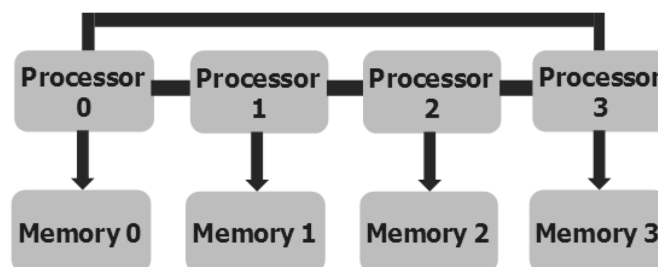


Figure 1.2: The Non-Uniform Memory Access(NUMA) Architecture

Each processor has low latency access to one or more memory banks that are nearer to the processors and higher latency access to the rest. Still latency of the farthest memory banks is very small as compared to distributed systems due to high speed interconnects. Like SMPs, a single OS image is present over the NUMA systems. The important thing to note is that in NUMA architectures, the global address space is provided at the hardware level. This means that all processors can access any memory word with simply load and store instructions. NUMA distance is measured in hops or by latencies.

Cache coherent NUMA (cc-NUMA) is a type of NUMA which allow data to be cached from remote memory. Most of the NUMA machines are cache coherent and like SMPs they also require cache coherence protocols to keep data consistent. Non cache coherent NUMA (ncc-NUMA) machines only allow data from local memory to be cached, though processors are able to access remote memory but they cannot store the data from remote memory in their cache.

Remote accesses have high latency; if the accesses are frequent then data can be migrated or replicated in the local memory [Bolosky et al., 1989]. The SGI 3000 family is a NUMA architecture based on the 64 bit Intel Itanium 2 processor [Woodacre et al., 2003]. This architecture uses 10 bits for specifying nodes, theoretically it

can support up to 1024 nodes or 2048 processors i.e. each node has two processors. These nodes are connected by a high speed interconnect, NUMALink, which provides low latency access to memory on other nodes. For desktops, AMD Opteron and Athlon families are based on the HyperTransport link and are capable of providing support for NUMA systems. The behaviour of an application depends on the number of components (processors, memory and devices) in a system and how they are inter-connected to each other. Because of the wide variety of multi-core architectures, it is necessary to make some assumptions of what is and what is not supported. In this thesis, we consider a reference NUMA architecture. NUMA systems are shared memory multiprocessors where memory is physically distributed among nodes. Nodes can have a number of processor memories directly attached to it. Nodes are connected by a very high speed interconnect which allows fine grained access to remote memory. Usually NUMA systems have the following properties:

- A single address space exists.
- Cache coherency may exist globally or partially.
- Memory access latencies may vary when different processors access the same memory bank.

1.3 Support for Multiprocessors in RTSJ

On multiprocessors, the multi-threading model of Java enables Java applications to use more than one processor thereby effectively speeding up the application (assuming there is 1-1 mapping between Java threads and operating system threads). This allows non real-time applications to execute unchanged on multiprocessors. However, the availability of many processors completely changes the scheduling model for a real-time systems. Therefore, real-time Java applications require multiprocessor scheduling models to be supported by the Java platform.

A number of extensions have been outlined for RTSJ-1.1 [Dibble and Wellings, 2009], which mainly focus on supporting different scheduling models available in state of the art multiprocessor real-time scheduling theory. These changes involve

pinning real-time threads and bound asynchronous event handlers to specific processors, clarification of the logical dispatching model on multiprocessors and the behaviour of processing group parameters on multiprocessors.

Although SMPs have nice properties from a programmability perspective, as previously mentioned they do not scale well due to contention on the shared buses accessing the memory banks. NUMA architectures on the other hand use different busses to access different memory banks, allowing all processing cores to access all the main memory banks; however, the access time will vary from core to core. As a result, performance of a program can be increased by keeping data local to its accessing threads. Therefore, new high level semantics need to be introduced at the language level to support such systems.

1.4 Motivation

Single processor systems will become less frequent while multiprocessors are increasing in popularity. NUMA systems are more scalable multiprocessors and it is likely that this architecture will be used more and more as the platform of choice. Real-time programming will become increasingly difficult with traditional low-level languages on complex multiprocessor architectures.

Real-time Java is a developing real-time language which provides high level abstractions for real-time programming, therefore, this makes real-time Java a suitable language which can be extended for such systems. Supporting NUMA systems means there are performance benefits to be had and the behaviour of the application can be made more deterministic.

Cache coherent NUMA (cc-NUMA) has a memory hierarchy which is physically distributed. This makes it a more scalable architecture than the conventional UMA based SMP system. However, the memory distribution affects the application behaviour on the system because of the introduction of remote memory accesses which have higher latencies. In such systems, there are considerable benefits to be had by allocating related threads and data close to each other.

For example, Figure 1.3 shows the performance benefits that can be had by

keeping data local while using the memcpy() function. This function copies a chunk of memory from one place to another. The figure shows that there is a large performance difference when executing the memcpy() function on local memory when compared to the remote memory. This figure is based on results obtained by executing the TAU benchmark[1] on a small cc-NUMA system. More details can be found in Appendix-A.

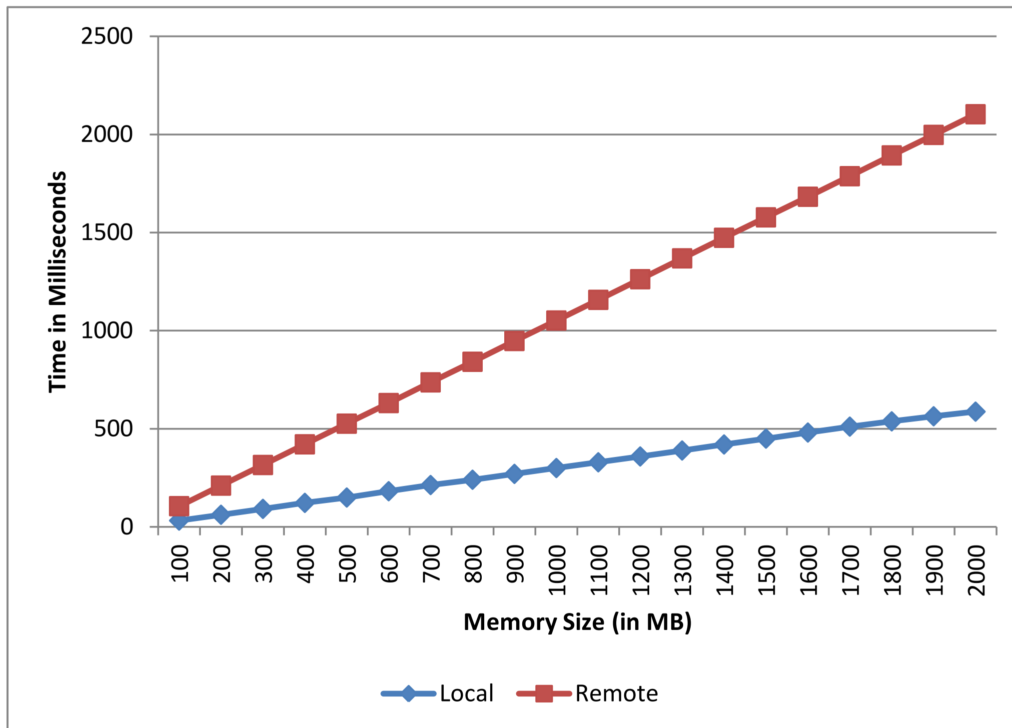


Figure 1.3: Time Taken by memcpy()

Multiprocessors encourage platform sharing among applications, these applications can have different timing requirements and can be of different criticality levels. The Linux OS also supports multiple applications executing at the same time, both real-time and non real-time. Typically for real-time systems, it is required that the system should be globally analyzed and worst case timings of all applications which can be sharing the platform with the real-time application should be taken in consideration. On such a large system, it is nearly impossible to analyze all the different permutations that the applications can have to run simultaneously with the

[1] http://www.nic.uoregon.edu/tau-wiki/Guide:Opteron_NUMA_Analysis

real-time application. Instead, it is more efficient and flexible to isolate the application and provide resource guarantees. Such an approach, removes any dependencies that the application has regarding the hardware or sharing the platform with other applications.

1.5 Hypothesis

The current RTSJ does not have appropriate support mechanism to enable programmers to easily develop portable and deterministic soft real-time applications for NUMA systems. However, it is possible to enhance the ability of RTSJ platforms by providing visibility into the underlying architecture, controlling the allocation policies and finally supporting resource reservation for guaranteeing timeliness and temporal isolation. These extensions will enable programmers to develop more deterministic and portable applications.

1.6 Thesis Aims

This thesis mainly concerns the execution model of real-time Java programs on cc-NUMA shared memory multiprocessors. The major aim of the research is to enhance the programming model of RTSJ to enable programmers to develop portable real-time Java applications on a shared and open environment. To support the hypothesis, the following objectives are considered:

1. To investigate the limitations in existing RTSJ to develop portable real-time Java applications on cc-NUMA architectures.

2. To propose new high level abstractions which make the real-time Java virtual machine (RTJVM) aware of the hardware resources available for the application.

3. To propose new high level abstractions to manage the allocation policies of threads and objects in order to minimize the non-determinism caused by the memory distribution in the NUMA system.

4. To support static and dynamic allocation of applications on hardware.

5. To support temporal guarantees to applications to ensure they meet their timing requirements.

6. To demonstrate that such a model can be implemented by presenting a prototype implementation.

7. To analyze the proposed model to check for improvements in performance and predictability.

Achieving these objectives would facilitate the development of portable real-time Java applications on cc-NUMA architectures.

1.7 Thesis Structure

In accordance with the motivations and objectives of the research, the thesis is organized with seven chapters. Brief descriptions of the remaining chapters are given below:

Chapter 2. Literature Survey: The chapter reviews three different areas. First existing parallel programming languages are reviewed and issues of parallelism and locality are discussed. Next, existing work on real-time systems on multiprocessors is reviewed. An open and shared environment is considered and the support required for soft real time systems is discussed. Java is discussed as the language of choice for parallel programming and how it is used in different contexts of multiprocessors, starting from shared memory multiprocessors to distributed systems. Lastly a set of of requirements are derived which need to be supported towards the goals of the thesis.

Chapter 3. Existing Support for cc-NUMA Architectures: The chapter discusses the extensions made in order to support real-time systems in Java and the support RTSJ has for multiprocessors. The chapter highlights support provided for cc-NUMA systems in Linux and RTSJ. This chapter focusses on the physical memory framework of RTSJ and how it can be used to represent

NUMA nodes which can then be used for explicit memory allocation, this chapter also looks in detail at the AffinitySet class that has been proposed for RTSJ 1.1 and using it to pin schedulable objects to processors. At the end of the chapter, limitations are outlined in the existing RTSJ for the development of portable real-time applications for cc-NUMA systems.

Chapter 4. Locality Model: The chapter presents a locality model for the RTSJ which overcomes the limitations outlined in Chapter 3. New abstractions are presented to represent NUMA architectures along with design patterns which are used to create memory areas and schedulable objects (and Java threads). Extensions are presented to the physical memory model and a new memory area, *PhysicalHeapMemory*, is proposed. A new abstraction *ExecutionSite* to provide locality on a cc-NUMA system. It also discusses open-systems and how contracts can be used to map applications onto NUMA systems. Contracts are discussed in the context of execution sites and open systems based on these execution sites are discussed.

Chapter 5. Implementation: The chapter explains the details of the implementation of the prototype.

Chapter 6. Evaluation: The chapter evaluates the Locality model. A set of experiments are also performed to evaluate the locality model in terms of performance. Overheads of the Locality model and the prototype are presented.

Chapter 7. Conclusions and Future Work: Finally, in the last chapter, conclusions are drawn and recommendations are made for future work.

1.8 Summary

This is the introductory chapter of the thesis which discusses the non-determinism caused as a result of cc-NUMA Systems. This chapter discusses shared memory architectures and gives an overview of the RTSJ along with some of the features it has to support multiprocessors. The chapter further outlines the hypothesis, goals and objectives of this thesis ending with the structure of the rest of the thesis.

Chapter 2

Literature survey

This thesis combines practices from both parallel computing and real-time system communities into the real-time Java programming language to support large scale systems on cc-NUMA architectures.

The goal is to provide *programmability, performance, portability* and *predictability* on shared memory architectures where memories have variable access timings. The chapter highlights issues of parallelism, locality and resource guarantees to achieve the goals of the thesis.

The chapter reviews the existing literature in the following areas:

1. Parallel programming on multiprocessors
2. Real time systems and multiprocessors
3. Java based programming languages on multiprocessors

Cc-NUMA systems are now entering main stream computing due to the scalability issues of traditional SMPs. Real-time systems' transition from single processor to multiprocessors is a very active research area with the emphasis being on scheduling on SMPs. However, cc-NUMA systems have been around for quite some time in the High Performance Computing (HPC) community. The Java platform has also provided a base for extension for parallel programming. In this chapter we discuss multiprocessors in all these three cases. A set of requirements is extracted to support large scale parallel applications on cc-NUMA multiprocessors in the context of RTSJ.

Section 2.1 reviews parallel programming languages especially in the HPC community in terms of parallelism and locality. Section 2.2 discusses real-time systems and multiprocessors. Section 2.3 presents the Java platform in the context of multiprocessors. Finally, section 2.4 summarizes the chapter.

2.1 Parallel Programming on Multiprocessors

The HPC community has vast experience in parallel programming and while other communities are making the inevitable shift to multiprocessors, a number of things can be learnt from the evolution of parallel programming in HPC.

In this section we review different approaches taken in the literature to provide parallelism and locality on NUMA systems.

2.1.1 Parallelism

Parallelism has been the main focus of programming on shared memory multiprocessors. The HPC community focuses on increasing both run-time performances, by maximizing parallelism in applications, and to increase productivity of programmers.

Early work in parallel programming was concentrated towards automatically parallelizing where compilers and runtime systems were used to extract parallelism from sequential programs. The advantage of such an approach is that existing applications need not be modified for the speedup. High Performance Fortran (HPF) [Koelbel et al., 1994] automatically parallelizes operations on matrices and vectors to be performed on separate processors. However, extracting parallelism from sequential programs is very limited and it does not allow the application to fully exploit the parallelism that is on offer at the hardware level.

In order to achieve better performance, parallelism needs to be expressed in applications to exploit the parallelism available at the hardware level.

The POSIX threading library [IEEE, 2008] allows explicit parallelism where threads are created and managed explicitly.

OpenMP [OpenMP, 2008] follows a different strategy where programmers use

directives to tell the compiler the existence of a parallel region. OpenMP is much easier to use as the compiler manages the threads, workload is implicitly divided and threads are implicitly synchronized based on programmer's directives of synchronization.

Java follows a similar approach to the POSIX threading library where threads are created and managed explicitly by the programmer. X10 [Charles et al., 2005] replaces the existing threading mechanism by light weight threads called activities. These activities are managed implicitly in place of Java's existing explicit model.

The RTSJ uses the same threading model as standard Java. RTSJ has integrated it with scheduling parameters which allow implementation of scheduling policies to control the order in which threads will be executed. This thesis uses the same threading model for parallelism and concentrates on providing locality on a cc-NUMA system.

2.1.2 Locality

An application requires efficient mapping onto the hardware architecture to achieve better performance. This performance on a NUMA system depends on the thread and memory allocation policies. Portability of performance is also very important on portable platforms (such as Java), that is to get the best performance regardless of the underlying machine without having to modify the application or the execution environment. Applications which are not supposed to be portable (often in the case of an embedded system) are required to extract the best performance out of the system, however, for portable applications mapping policies should be focussed on achieving portability of performance.

On a cc-NUMA system, related threads and objects need to be placed close together to each other. This reduces the number of remote accesses resulting in a better performance of the application. However, not all mapping strategies follow the same rule. Generally, the following strategies can be adopted during application mapping:

1. In the first approach the mapping is done by the runtime without any knowledge of the application. This approach is commonly used in operating systems

such as Linux where threads and memory are allocated based on load balancing.

Threads are evenly spread across a number of processors and are migrated in case of any load difference between processors. The Linux scheduler tries to balance load in a hierarchical fashion in a cc-NUMA system where it first tries load balancing at the node level and then proceeds to balance the load between the nodes [Aas, 2005].

For the allocation of memory, a number of different policies can be used. Touch policies are very common for memory allocation where the allocation is based on the pattern of data being accessed. For example, the first touch policy allocates the memory local to the processor which accesses the data for the first time. Another policy is the next touch policy which allocates the data on the local memory when it is accessed the second time and is considered to perform better than the first touch policy [Goglin and Furmento, 2009]. Memory interleaving is another policy where data is allocated in a round robin fashion among the nodes. Such a policy does not consider locality among threads and data but rather tries to achieve maximum bandwidth while accessing the data.

These approaches certainly have the advantage that applications are very portable across such platforms and benefit from these policies. However, the performance of the applications is far from the peak performance which can be achieved on the system. In this case, the relationship among threads and between threads and data is ignored which can help in improving the performance of the system.

2. The second approach is based on applications that have been designed to run on specific platforms. Allocations are done explicitly by the programmer requiring significant effort and become extremely difficult and complex in the case of large systems.

   The architecture of the system is analyzed and then thread affinities and memory affinities are used to statically map the application on the architecture.

Following are some of the examples of programming models using this approach:

- The Sequoia programming language [Fatahalian et al., 2006] is designed specifically for embedded systems. It is an extension of C which enables programmers to statically map applications on platforms with non standard memory architectures. It represents the architecture as a virtual tree and explicitly manages data movements between memory banks. It requires programs that can be broken down into a hierarchy of isolated units of execution and then physically mapped onto the architecture.
- The RTSJ provides physical and raw memory access. Objects are created using physical addresses of memory and similarly raw memory can be accessed through their memory addresses (in case of memory mapped I/O) or their port number.

This approach has been designed to extract the maximum performance out of the system; however, applications are not portable. Threads and data can be co-located with devices using this approach.

3. The third approach tries to combine the above two approaches to provide portability and performance. The mapping is based on the negotiations of the application and the architectural platform.

   At the architecture level, the operating system can provide detailed information on the target machine. For example, the information can include number of processors, organization of caches, organization of the memory banks on NUMA machines etc. The application needs to be aware of the processors, memory and devices in a system and how they are connected. A single system image hides all the information regarding the hardware and presents a simple environment for the applications to execute on. Processors and memory devices are usually hidden by the single system image provided by operating systems and the application executes as if it has a single processor and single memory. However, operating systems also provides libraries and APIs which publish the information of the hardware.

At the application level, information can be provided regarding the behaviour of the application. Determining the sharing patterns of threads and data is an important step before the threads/data can be allocated. [Tam et al., 2007] suggest that the sharing patterns of threads should be detected online and threads should be grouped once this pattern is known. The advantage of using this approach is that threads can be dynamically regrouped during different phases of a program based on changing relationships between threads, however, the overheads are very high. Most modern programming languages X10 [Saraswat, 2010], Fortress [Allen et al., 2007], Chapel [Diaconescu and Zima, 2007] depend on the programmers to define the relationship among the components of the application.

Following are some the examples of programming languages that adopt this approach:

- The Fortress [Allen et al., 2007] programming language provides an extensive library of language contructs called *Distributions* which allow the programmers to specify data distribution and locality. Fortress provides *regions* which abstractly describe the architecture of the hardware on which the application is running. Each region contains an execution level where the threads reside and a memory level where the objects reside. The programmer is enabled to place threads in a particular region and objects in most cases are placed local to the thread calling the constructor.
- The X10 language [Saraswat, 2010] has multiple non-overlapping *places*; these are virtual locations which group together multiple activities (threads) and objects. Physical location of places can change depending upon the load balancing policy. Activities and objects remain in the same place for their lifetime and are unable to migrate to other places. Remote access is only possible by spawning asynchronous activities which are used to communicate between two places.
- Chapel [Diaconescu and Zima, 2007] provides *locales* which represent

units of uniform memory access (nodes). There is no difference between local and remote access as there is in X10. Chapel also allows programmers to distribute objects into specific locales and objects are not bound to a single locale for their lifetime.

This approach provides a balance between portability and performance. This is required when the application is portable but still requires threads and data to be co-located. The execution of an application is closely tied to the structure of the hardware. Portable applications need to adapt to target architecture for better performance and efficient use of resources.

As part of this thesis, we focus on providing an environment which will allow the programmers to develop applications with portable performance by supporting a programming model which in similar to the third approach described as above. In addition, the model will also extend support available in the RTSJ to access physical and raw memory on cc-NUMA systems.

2.2 Real Time Systems and Multiprocessors

The paradigm shift towards multiprocessors increases the motivation for sharing the platform and most modern general purpose operating systems allow a number of different applications to run on a platform simultaneously. Executing real-time systems on shared platforms requires the system to behave as if the platform is dedicated to the real-time application where resources are provided to it according to its scheduling parameters and any other application should not be able to cause any disruptions during its execution. In other words the system should be able to provide temporal isolation (or temporal protection) to the real-time application. [Buttazzo et al., 2005] puts it as "the temporal protection property requires that the temporal behaviour of a task is not affected by the temporal behaviour of the other tasks running in the system".

Temporal isolation can be provided to applications by resource reservation mechanisms where resources are guaranteed to the application/component in a timely

manner so that their timing requirements are met. In the case of a multi-threaded application/component resource reservation mechanisms can be applied to the thread level. Essentially multi-threaded applications need a hierarchical scheduler where at the global level the application is competing against other applications for resources. Within the application/component threads compete among themselves for the available resources.

A server is a CPU execution accounting and enforcing mechanism which has a capacity and a replenishment period, it makes sure that an activity or a group of activity does not use more than the capacity and replenishes the capacity periodically. Execution-time servers have mainly been used to service aperiodic activities without affecting periodic and sporadic activities. Aperiodic activities do not have well defined release characteristics; as a result they can have an unbounded demand for processor time while competing with periodic and sporadic activities. Running these aperiodic activities at a lower priority than periodic or sporadic means, they get very little chance to run and they often have poor response times. As a result servers have been used to provide a lower response time for aperiodic tasks.

Alternately, servers have also been used to build compositional real-time embedded systems. The focus of the servers shifts to make sure that components actually get the execution time that they have been guaranteed.

[Deng et al., 1997] were the first ones who used hierarchical scheduling to provide temporal isolation for multithreaded applications. Each application was allocated a dedicated constant utilization server (CUS) with a maximum utilization factor Ui. Later a bandwidth sharing server (BSS) was used to provide temporal isolation in multithreaded applications. Both of these approaches require the global scheduler to be aware of the timing parameters of tasks inside an application. In the case of open systems, such information might not even be present because of the presence of non-real time threads which do not have scheduling parameters (such as WCET and deadlines).

[Mok et al., 2001] present a resource partition model where a resource partition is a periodic sequence of disjoint time intervals. This approach is based on the 2-level hierarchical scheduler where at the first level (application scheduler), tasks

within a task group are scheduled while the second level scheduler (global scheduler) is responsible for assigning partitions to task groups. Both these schedulers are completely isolated at runtime because the global scheduler does not require the timing parameters of the tasks within each task group. The bounded delay model was proposed in [Mok et al., 2001] which generalizes the partition model to accommodate an on-line global scheduler. The bounded delay model can be represented by the interface $(\alpha, \triangle)$, where α is the bandwidth while $\triangle$ is the bound on the bandwidth for which it will be unavailable.The bounded model for single processors can be represented by the Figure 2.1.

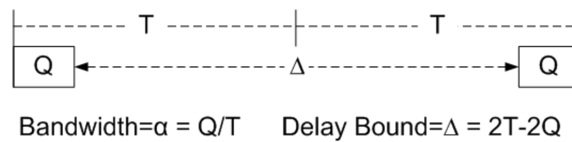


Figure 2.1: The bounded delay model for single processors

These approaches were defined for single processor systems, in case of multiprocessors, the presence of multiple processors increases the complexity of providing resource reservation when applications/components want concurrency on these platforms essentially allowing them to run in parallel on different processors.

The following resource models can be used for resource reservations on multiprocessors:

- The Periodic Model (P, Θ, m')–The multiprocessor periodic resource model (MPR) was presented in [Shin et al., 2008] which was based on the idea of having virtual cluster based scheduling. In this model, any application was allocated a number of periodic tasks or servers. These servers were statically allocated on processors having the period P and they were assigned a cumulative budget Θ. Application level tasks were then scheduled globally on top of the servers. The MPR model was specified by an interface (P, Θ, m'), where m' is the maximum level of parallelism.

- Bandwidth ω – Another interface based on the total bandwidth ω of applications was presented in [Leontyev and Anderson, 2009]. The idea was to allocate

the bandwidth ω onto $\lfloor \omega \rfloor$ processors and then globally schedule the remaining bandwidth on the remaining processors using a periodic server. This interface was designed for soft real time systems and did not have any notion of period.

- Bounded delay multipartition (BDM) – The multi supply function extended Bounded Delay Model for multiprocessors [Bini et al., 2009b] where all the processors had the same $(\alpha, \triangle)$ repeated for m processors. This multi supply function was next generalized in the parallel supply function [Bini et al., 2009a] which allowed different processors to contribute differently to the budget. Based on the parallel supply function, interfaces are derived in [Lipari and Bini, 2010] to reduce the pessimism.

2.3 Java on Multiprocessors

Java has been used in a number of distributed and high performance systems where high level APIs have been provided to support parallel and distributed applications. In this section, we will focus on the different Java based extensions that have targeted distributed systems and shared memory multiprocessors.

2.3.1 Java on Distributed Systems

On a distributed system, Java threads and Java objects are distributed between the nodes of the distributed multiprocessor which do not share a single address space. This, however, should not restrict Java threads from accessing all objects irrespective of the location of threads and objects. This means when a thread running on one node accesses an object on another node then either the code of the thread should be transferred to the remote node or the data from the remote node should be retrieved.

Java provides an API, the Java RMI (remote method invocation), which allows the JVM to invoke methods of objects running on another JVM. Java RMI's stub and skeleton architecture gives the impression that the method is being executed locally while the reality is that it executes on a remote node and returns the results. This allows the programmer to distribute objects on various machines, and invoke methods on objects on a remote machine. Java RMI provides lease-based

distributed garbage collection. Remote objects are considered alive when they have been referenced within a certain period of time i.e. lease period. When the lease expires and it is not renewed, the objects become available for garbage collection. The problem with Java RMI is that it requires the programmer to manage and distribute the computation explicitly. This makes RMI programming difficult because the programmer has to put extra effort to program interfaces and classes which do not help him in solving his actual problem.

Java distributed shared memory systems (DSMs) provide the Java platform over a distributed system by hiding its distribution and providing a single system image (SSI). Java DSMs provide a higher level of abstraction than Java RMI. While Java RMI requires the programmer to explicitly distribute computation load among the nodes, Java DSMs on the other hand automatically maps threads to different processors and provide a shared heap where the objects can be placed.

The following highlights design and implementation issues of Java DSMs:

1. In a Java DSM the following changes are required to different memory areas:

   (a) Implementing the heap – The heap is a memory area which is shared by all threads. In order to extend the heap to Java DSMs the objects should be accessible by all the threads irrespective of the location of the thread or the physical location of the object. One way of doing this is by implementing the heap over an existing DSM.

   (b) Implementing the method area – The method area stores the code, which is shared by all threads. In order to extend it to multiple processors, it should be copied to all nodes of the cluster. Multijav first looks for class bytecode in its local memory, then asks the root node to send the desired class bytecode.

   (c) Implementing the stack – The stack is a JVM memory area which is private to a thread. The stack contains frames which are pushed when a method is invoked and is popped when a method returns. Normally in the case of a Java DSM there is no need to make any changes to implement the stack. But when a thread is distributed over a number

of nodes as it is in the case of method shipping in cJVM [Aridor et al., 1999]. Then the stack is also distributed along with the thread. Here the thread is required to have a global id. Even after the distribution the stack is required to be traversal and it should give the correct value on calling the Thread.currentThread().

2. The following scenarios can be envisaged to provide access to a remote object:

   (a) Method Shipping – Method shipping is used when remote accesses are fine grained and rare, then it is possible to redirect the code to the home node of the object. The thread is not migrated but only the method executes remotely on the object and then returns back to the home node. This approach uses mechanisms such as remote procedure call (RPC) such as Java RMI. However, this is at a higher level than RMI and the communication between thread and remote object is handled by the runtime. This approach has been followed in cJVM [Aridor et al., 1999].

   (b) Object replication – Object replication is used when remote accesses are coarse grained and frequent, and multiple nodes try to access the object simultaneously. The object replication approach allows multiple nodes to simultaneously read the objects. The original copy of the object remains on the home node, which is updated consistently in case of any write and all other copies are invalidated. This approach is used by JESSICA (Java Enabled Single System Image Computing Architecture) [Ma et al., 2000], Jackal [Veldema et al., 2001], JavaSplit [Factor et al., 2006], JESSICA2 [Zhu et al., 2002] and Hyperion [Antoniu et al., 2001].

   (c) Thread migration – Thread migration is usually used to balance the load, but it can also be used to provide access to remote object. It is different from method shipping in a way that in method shipping the control of execution returns backs to the original node after the execution of the method on the remote node. In order to perform thread migration, all the associated data also has to be migrated along with the thread, which makes it a costly approach. Migrating the thread is not viable every time

it has to access a remote object.

(d) Object migration – Object migration only benefits in the case when remote accesses are frequent from a single node, then the object can be migrated to the remote node. However, in a shared memory environment (in Java) objects can be referenced by more than one thread, therefore, it is not viable to migrate the object every time it is referenced.

While all of the above enable threads to access remote objects, the top two are the only ones that are actually used for providing distributed access in Java DSMs. The bottom two are used for load balancing; they can be used for providing remote access to objects.

3. Java DSMs can be implemented using the following approaches:

   (a) Java DSMs built over existing DSMs: In a Java DSM which has been built on already existing DSM subsystem as shown in Figure 2.2, efficient channeling of information from a JVM to the DSM subsystem is very difficult and causes performance lapses. Page based DSMs such as Treadmarks [Amza et al., 1996] which have been used in JESSICA [Ma et al., 2000] and Java/DSM [Yu and Cox, 1997] causes false sharing. The implementations use the API provided by the DSM subsystem for memory and thread management.

   (b) Native Java DSMs: Java DSMs that have been based on interpreter JVMs(JESSICA, Java/DSM) as shown in Figure 2.3, are unable to provide the performance which can be provided by native machine code. As a result a number of Java DSMs i.e. Jackal and Hyperion opted to translate Java code into native code for achieving performance in the long run. Such approaches do not use the JVM and do not benefit from the security and portability provided by the JVM.

   (c) Java DSMs with built-in DSM: In this approach the JVM is modified to provide a distributed shared heap at the JVM level as shown in Figure 2.4. This approach has been followed by cJVM and JESSICA2. cJVM is an

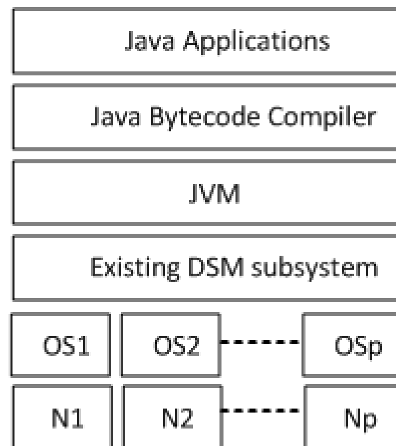


Figure 2.2: Java DSM built over existing DSM

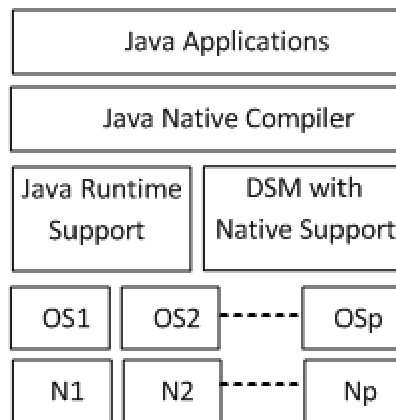


Figure 2.3: Native Java DSMs

interpreter based Java DSM which uses the master/proxy model to give an illusion of distributed shared heap. JESSICA2 uses JIT execution to improve performance.

4. In order to utilize the hardware efficiently, the computational workload requires to be distributed among the nodes. JESSICA and JESSICA2 provide transparent thread migration at runtime. While other Java DSMs such as cJVM use the load balancing function to place the thread on a node but once it is placed on a node then it remains on that node during its lifetime.

5. Most Java DSMs provide locality. Objects are created local to the thread. In case of object replication, a local copy of the object is created on every node where it is accessed. Only in method shipping we get remote accesses which

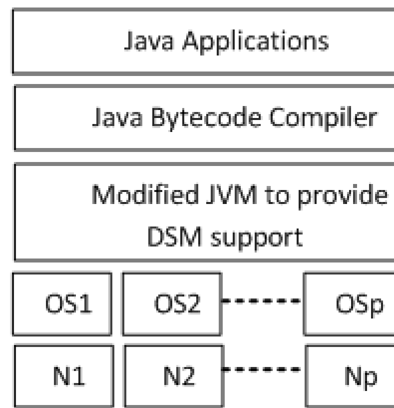


Figure 2.4: Java DSM with built-in DSM

can affect the performance of the application badly.

6. The size and structure of the unit of coherence is an important factor in determining the performance of a Java DSM. Structured unit such as an object provide better locality of reference and less false sharing as compared to a page, which is an unstructured contiguous array of memory locations which can contain unrelated data. Coarse grained units of coherence increase the network traffic and cause more false sharing as compared to fine grained units. The cost of accessing and updating a large memory chunk is considerably more than a small one but it can provide better locality of reference.

7. Memory consistency is an important part of Java DSM. Communication among the nodes is slow and often performance decreases in systems which have communicate very frequently. Sequential consistency models cause more overhead than relaxed consistency models. Therefore most of the Java DSMs have used relaxed memory models in order to provide better performance. For example, JavaSplit uses home based lazy release consistency (HLRC) [Zhou et al., 1996]. This protocol allows multiple writers. It allows object replication of the object at remote nodes and uses invalidation for memory consistency.

8. Local objects have very low maintenance costs as compared to shared objects. Majority of the objects created are local and very few of them are shared. Dealing with them separately can result in improved performance. JavaSplit [Factor et al., 2006] distinguishes between local and shared objects on a

node at runtime. Local objects do have low synchronization and no communication costs as compared to shared objects and the majority of the objects are local objects. This way JavaSplit is able to reduce much of the synchronization and communication overhead. Only objects accessed by more than one thread are termed as shared nodes. The shared objects are given a global thread id and can be replicated by the DSM subsystem on other nodes.

2.3.2 Java on Shared Memory Multiprocessors

The Java programming language has built in support for multi-threading. This allows the Java threads to execute in parallel over a multiprocessor. Java threads are limited to shared memory architectures and are not able to benefit from more scalable architectures found in distributed systems (without using DSM). However, on shared memory multiprocessors, existing multithreaded applications can execute without any problem with the appropriate speedup values. Concurrency utilities introduced in Java 1.5 specification provide extra features (thread pools, blocking queues etc.) to make concurrent programming easier.

Java has also been extended to use annotations which are similar to the directives used in the OpenMP specification. Annotations can be used to instruct compiler to create parallelism which can be used by the runtime to create Java threads. Implementation like JaMP [Klemm et al., 2007] and JOMP [Kambites et al., 2001] use this model. These approaches provide a higher level model than Java threads and are gaining more popularity due to the ease with which the programmers can write parallel code.

On a cc-NUMA system, [Tikir and Hollingsworth, 2005a] provide a distributed heap over a cc-NUMA system. The garbage collector is optimized to migrate objects to provide locality which improves the performance of a parallel benchmark by upto 41%. [McIlroy, 2010] have used annotations to group together threads into teams of threads for co-allocating threads which improves the performance of the application. The JVM can also be used in heterogeneous environments.

2.3.3 Discussion

With new architectures there has been a debate of either creating new parallel languages from scratch which suit these architectures or to extend existing languages because it is not easy to convince programmers to learn new languages.

Parallel programming can be complex for programmers, especially when they migrate from a sequential to a parallel environment where they have to explicitly express parallelism.

Java is a popular high level language which is the reason why so many attempts have been made to extend it for parallel architectures. Extensions that have been provided for the Java language mainly focus on the increase of performance. However, even in the parallel environment, the first priority is programmability, the ability of the programmer to create parallel programmes e.g. the main objective of providing a single system image in a Java DSM is to improve programmability.

Real-time programming language design is difficult in its own capacity. Integrating parallel and real time programming can become very complex. This is the reason why we have selected real-time Java to provide a high level platform for parallel programming. This thesis is limited to supporting soft real-time systems on cc-NUMA systems. However, the work presented should be extendible for more complex heterogeneous architectures.

2.4 Summary and Requirements

This chapter has focused on two separate areas of related work: how parallel languages have evolved to cope with a variety of multiprocessor architectures, and the mechanisms that real-time systems provide to support the scheduling of large scale real-time systems. Based on this review, we can extract out the following requirements for real-time programming of large non-uniform memory architectures within the context of Java-based systems:

1. Locality – to improve performance on a cc-NUMA system, the system needs to provide the ability to allocate threads/objects on certain processors. To

provide portable performance, the ability to co-locate threads/objects needs to be provided.

2. Architecture aware JVM – to support portable performance code, it is necessary to have a JVM which is aware of the architecture.

3. Resource Reservations – to support real time requirements in a shared environment, it is necessary to provide resource reservations.

Chapter 3

Existing Support for cc-NUMA Architectures

In Chapter 2, we reviewed the mechanisms provided by parallel programming languages to exploit locality and considered the impact of allowing real-time systems to share platforms with other applications. Based on the review, the following requirements were derived to support soft real-time programming on cc-NUMA architectures:

- the OS should be able to discover the architecture
- the OS and and the programming language should be able to co-locate tasks and objects
- the OS and the programming language should provide resource reservations
- the OS and the programming language should be able to site a thread/object on a physical proximity domain

This chapter will review existing support provided in RTSJ on Linux. Throughout this thesis we have used the Linux OS as an example of a main stream operating system that has kept track of advances in computer architectures and has responded quickly to provide appropriate support for those advances. Consequently in Section 3.1, we review the support that Linux provides for cc-NUMA systems, and whether it is capable of supporting real-time applications on these architectures.

The RTSJ, as of version 1.1, provides more support for multiprocessor embedded system. Although targeted at SMP architectures, it provides a set of mechanisms that can be used to access the underlying processor and memory architecture. In Section 3.2, we present an overview of the RTSJ and its support for multiprocessors.

Section 3.3 presents a reference cc-NUMA architecture and the existing support in Linux and RTSJ is evaluated to against the requirements derived in Chapter 2.

Section 3.4 outlines the limitations of the existing features and finally the chapter is summarized in Section 3.5.

3.1 Linux Support for cc-NUMA

Linux is an open source general purpose operating system which is being used on a range of platforms. It was never designed to support real-time systems and it had very high response times. The Linux kernel has the backing of a very active community. Early versions of the Linux kernel were non-preemptible i.e. any task running inside a kernel or a driver needed to finish before anything else was allowed to execute [Dietrich and Walker, 2005]. Linux 2.0 supported symmetric multiprocessors by using a single spin lock (the big kernel lock) which allowed only one task to be running kernel code at one time. This was a serious bottleneck for multiprocessors as the processing was still sequential. Later, locks were distributed based on critical regions rather than having a single lock for the whole kernel. However, any task inside the critical region was still non-preemptible and response times were still very high. The realtime-preempt patch has been included in the mainline in the Linux kernel 2.6.18 as a configuration option; it tries to minimize the amount of non-preemptible code.

3.1.1 Discovering the Architecture

Making the application NUMA aware requires the OS to be able to detect all processors, memory and devices and inform the application about the topology of the system. Operating systems in general have struggled to provide the appropriate application-level support for multiprocessor. The main hurdle is the abstraction

layer at the hardware level which is provided to hide the complexity of the underlying architecture. This abstraction layer not only hides the complexity but it also removes the transparency which is essential to develop predictable systems.

In order to provide some transparency, the Advanced Configuration of Power Interface (ACPI) specification 3.0 [Corporation et al., 2005] has been developed to establish industry common interfaces. The goal is to enable robust operating system (OS)-directed motherboard device configuration and power management of both devices and entire systems. The ACPI has optional support for NUMA architectures. It defines the concept of a *system locality* (or *proximity domain*) and provides information about the "distances" between them. Each proximity domain is given an integer identifier. An operating system can assume that two devices in the same proximity domain are tightly coupled, but it cannot make any assumption on the distances between the domains given only their ids. Instead, two tables are provided: the system resource affinity table (SRAT) and the system locality distance information table (SLIT). The former allows the domain of a device to be obtained and the latter the relative distance between the domains to be determined. The OS can choose to optimize its behavior based on the information in these tables.

The Linux kernel parses these tables on boot up and presents this information at the application level in the form of APIs and libraries. Libraries in different operating systems can differ in support and format, e.g. Linux uses the sysfs to publish the information and libraries such as Libnuma are build on top of the file system. Other operating systems have other low level system calls to discover the underlying hardware resources.

Based on the existing support in operating systems, a portable framework has been developed by the MPI community to introduce portable interfaces for obtaining this information [Broquedis et al., 2010]. The hwloc framework gathers information about processors, caches, memory, nodes in a NUMA system etc. on different operating systems such as Linux, Windows, Solaris, AIX, Darwin etc. and provides a high level abstraction to these resources at the application level. For more complex multi-core heterogeneous systems, the multi-core association is developing the multi-core resource API (MRAPI) which will allow the system to query different attributes of

processors, memory and devices (Metadata Primitives in the MRAPI) [Holt, 2009].

3.1.2 Allocating Threads and Objects

The Linux operating system has evolved quickly to support different allocation policies for threads and objects. The following section highlights these policies:

1. Thread binding – On multiprocessors, the OS is supposed to provide support for different scheduling policies from fully partitioned to global scheduling. In Linux, each processor has its own run queue, which enables dispatching threads individually on a fixed processor. Once a thread is allocated to a processor it stays in its run queue until it is either moved to another processor by changing its affinity or through load balancing. Although Linux does not have a single queue for multiple processors for true global scheduling, however, it uses load balancing on real time tasks to make sure that the highest priority threads present in all the run queues are running at any particular instant of time [1]. When there is more than one real time thread in a run-queue, the lower priority real time thread is migrated to a CPU on which it can run. This migration happens at the following events:

    (a) A real time thread becomes schedulable however it finds a higher priority thread already running in the run-queue on which it has been allocated. In such a case it is *pushed* to another run queue where it can run.

    (b) When a CPU finished any task, instead of executing a lower priority thread it *pulls* a higher priority task which is waiting in another queue to get dispatched.

    Linux allows the programmers to pin task to particular processors by providing the system call *sched_setaffinity().* The thread in such a case will not migrate to any processor outside the affinity set of the thread and the migration respects its affinities. More information on this can be found in Section 5.1

[1] http://kerneltrap.org/Linux/Balancing_Real_Time_Threads

2. Memory placement – On NUMA systems, a number of different optimizations can be made regarding memory placement. In terms of real-time systems it is very important to be to able determine where the memory will be present when it is accessed. Therefore, it requires the OS to be able to place the memory on a memory node and then ensure that the memory will remain there for the lifetime of the application. In the default case, Linux enforces a first touch policy for memory which essentially means that without any memory binding policy, it will try to allocate the memory on the same node as the thread which has touched it first. Linux provides a NUMA API [Kleen, 2004] which includes new system calls at the kernel level to support different scheduling policies and different memory allocation policies. At the programmer level, a library is provided which enables the programmer to place memory on a specific node and execute a thread on a specific CPU [Lameter, 2006]. In terms of memory allocations, the following policies can be defined:

    - *mbind_default* – Allocate memory on the local node (the node that the thread requesting allocation currently is running on).
    - *mbind_preferred* – Try to allocate on a particular node first. If this fails, allocate anywhere.
    - *mbind_bind* – Allocate on a specific set of nodes. Allocation will fail if the nodes are unable to provide the memory.
    - *mbind_interleave* – Spread out memory allocations over all available nodes.

    The physical space is very limited, therefore, it is very common for operating systems to swap some of the pages out to the disk and then swap them back in to the memory when required. The memory allocation latencies suddenly increase when the system runs out of physical memory and existing pages have to be swapped out in order to accommodate new pages in the virtual address space. The dynamic use of the physical memory means that it is not necessary that the virtual address will map onto the same physical address. Hence, physical addresses backing the virtual memory can change or even disappear (in case it has been swapped out). In case the memory gets swapped

out, it is not guaranteed that it will be placed on the same node when it is swapped in. The reason for this can be that the memory has been accessed by a thread on a node which is on a node other than the one from which it was swapped out from. From the real-time perspective, this behaviour is completely unacceptable, especially on a NUMA system, where the problem is exacerbated because a page allocated on one node is swapped back in on another node which the real-time application is unaware of. Therefore, it is required that the memory that will be accessed by the real-time application should always remain at the same place as it was. This is possible by locking the memory so it cannot be swapped out. Linux provides a system call *mlock()* which does exactly that. On a system with a large physical memory, the swapping space can be completely disabled to make sure that the memory will not be swapped out.

3.1.3 Supporting Group Budgets

The paradigm shift to multiprocessors has motivated the sharing of the platform by applications having different timing requirements. Real-time applications on such systems require resource guarantees from the platform because a global analysis of the whole system is not viable. In order to provide guarantees, the platform needs to support group budgets. Group budgets have traditionally been used to support aperiodic activities as discussed in [Wellings and Kim, 2008]. In Linux, control groups[2] provide a mechanism to partition groups of tasks and then manage resources allocated to each partition. A control group (cgroup) is a file system, where tasks can be added to it and then be arranged hierarchically. By default all tasks belong to the root cgroup. Each resource (CPU, memory etc.) has an associated controller which limits resources to tasks based on the cgroup they belong to. Building the mainline kernel by enabling the CONFIG_RT_GROUP_SCHED option, provides the support for real-time group scheduling. The real-time group scheduling can be used along with the cgroup filesystem to provide group budget

[2] http://www.mjmwired.net/kernel/Documentation/cgroups.txt

for a set of tasks (process/ threads). The real time group scheduling allows explicit allocation of CPU bandwidth to task groups[3]. The bandwidth can be set by the following two parameters:

1. cpu.rt_runtime_us allows to set the budget (Q in microseconds) of the task group
2. cpu.rt_period_us allows to set the period (T in microseconds) of the task group In every period T, the group is allocated the budget Q.

Once the budget expires, the group is blocked until the arrival of the next period. In Linux, the scheduler tick function is called periodically after a time period, T which is called by a high resolution timer. This function maintains the different CPUs accounting parameters and checks for budget exhaustion. On each scheduling event (at the scheduler_tick(), task deactivation, task pre-emption) the collective execution times of all tasks are compared to the budget value until it exceeds the budget at which point all tasks are descheduled.

A sporadic server patch has been presented in [Faggioli et al., 2010] which changes the real-time group scheduling of Linux. The changes mainly involve the behaviour of threads and instead of blocking, threads are re-entered in a lower priority queue once the budget has expired. It also makes changes to how the budget is replenished after each period. Group scheduling with sporadic servers not only supports temporal isolation but it is also very efficient in terms of CPU utilization. [Checconi et al., 2009] presents a hierarchical multiprocessor reservations model for Linux which is based on the multi-supply function [Bini et al., 2009b]. In addition, SCHED_EDF is being developed which extends the existing real time group scheduling to incorporate EDF.

3.2 RTSJ and its Support for Multiprocessors

Java is a high level programming language which has always favoured programmer productivity and portability. It provides features like automatic memory manage-

[3]http://www.mjmwired.net/kernel/Documentation/scheduler/sched-rt-group.txt

ment and Just-In-Time (JIT) compilation to make programming easier; however, these features add unpredictability to the platform. From the real-time perspective, garbage collection is a main source of unpredictability. In most JVMs, the garbage collector starts at unpredictable moments and stops the flow of the program to reclaim memory. This results in unpredictable interruptions and delays which are not acceptable in real-time systems.

The Real-Time Specification for Java (RTSJ) extends the Java platform to enable programmers to write high level, portable code with real-time properties. It introduces new memory areas (in addition to the Java heap) and extends the Java threading model along with new scheduling policies. It does not make any changes to the Java syntax and provide backward compatibility to Java applications allowing environments compatible with RTSJ to execute existing Java applications.

The following summarizes important enhancements made by the RTSJ to support real-time programming in Java:

- Memory areas – Standard Java only provides the heap memory area for object allocation to the programmers. However, the heap memory area is garbage collected which can cause unpredictable delays. In order to minimize the effect of garbage collection, RTSJ introduces two new types of memory areas: the immortal memory and the scoped memory. Figure 3.1 provides the class hierarchy of the memory areas in RTSJ which shows classes representing the immortal memory area and scoped memory area.

  The immortal memory area is represented by an *ImmortalMemory*, which is a singleton class. Only one immortal memory area exists in a RTSJ application and behaves much like the Java heap memory. The difference exists in the lifetime of objects on both memory areas. Objects allocated on the heap are reclaimed by the garbage collector if they are no longer used, however, objects allocated on the immortal memory area are never reclaimed even if they have do not have any live references to them. Programmers can use the immortal memory area for object allocation and in addition, all static objects are allocated in the immortal memory area by default. The immortal memory area needs to be used carefully because any object, once allocated, exists till

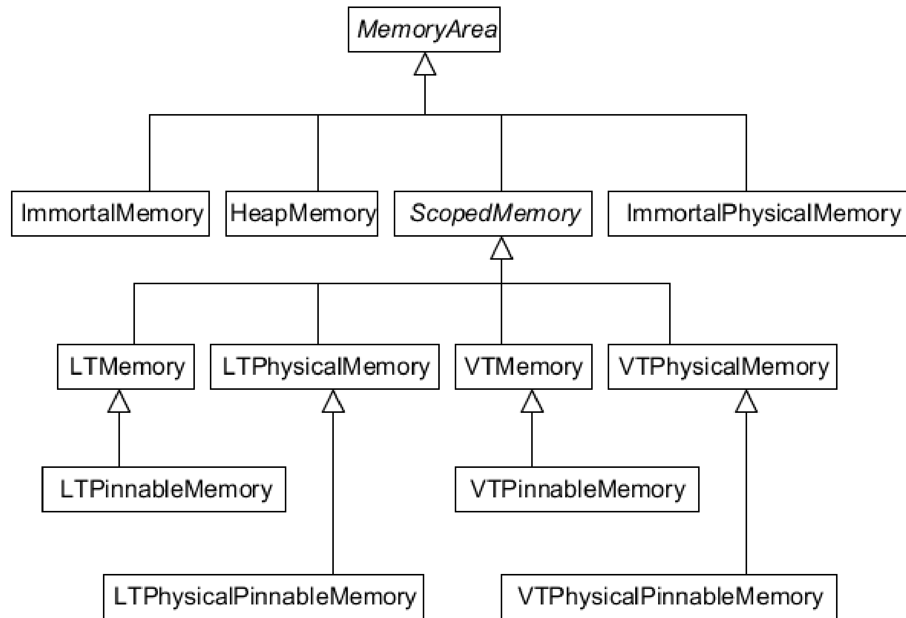


Figure 3.1: Memory classes in the RTSJ

the end of the application causing a potential memory leak.

Scoped memory areas are represented by the *ScopedMemory* class. Scoped memory areas are created by the programmers and contain objects with well defined life times. Two different types of scoped memory areas can be created by the programmers: *LTMemory* (linear time memory) and *VTMemory* (variable time memory). The time taken to allocate objects in LTMemory is linear depending on the size of the object. VTMemory on the other hand can employ optimizations for faster object allocations. The scoped memory regions use reference counting to determine how many schedulable objects are active within the region and when the count turns zero, than the memory is available for reuse.

The RTSJ also provides classes to use physical memory. The physical memory areas allow programmers to allocate objects on physical memory directly. The physical memory classes are discussed in detail in Section 3.2.2.

- Schedulable objects – RTSJ introduces schedulable objects which implements the *Schedulable* interface as shown in the Figure 3.2. The schedulable interface

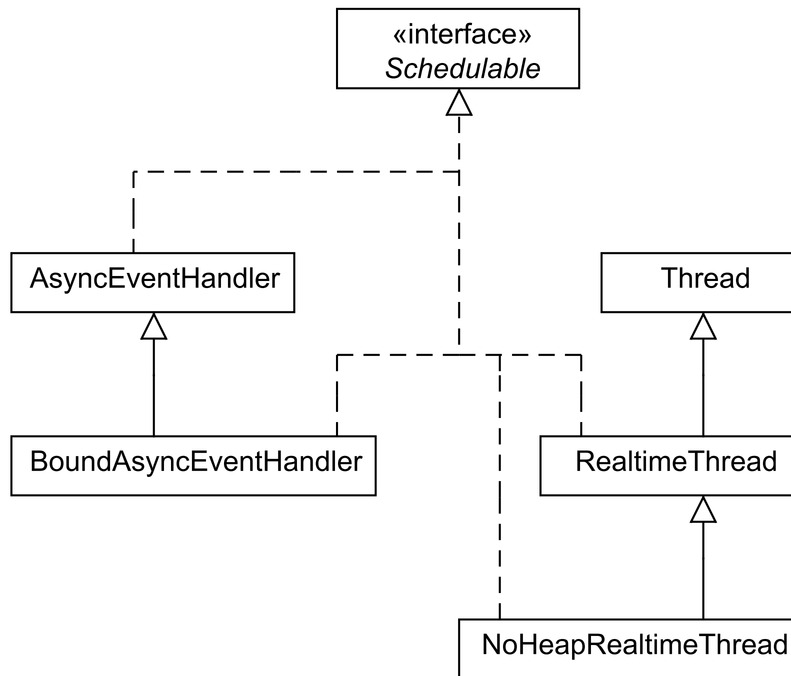


Figure 3.2: Schedulable objects in RTSJ

is implemented by the *RealtimeThread* and the *AsycnEventHandler*. The RealtimeThread can freely move between different memory areas and can access all three types of memory areas: heap, immortal memory and scoped memory areas. The *RealtimeThread* is sub-classed by the *NoHeapRealtimeThread* which can only use the immortal and scoped memory areas. NoHeapRealTimeThreads are threads that typically have higher priority than the garbage collector and cannot use the Java heap or cannot reference any object on the heap.

- Scheduling – Figure 3.3 shows new classes introduced in the RTSJ to implement scheduling policies. An abstract class of *Scheduler* is provided for the feasibility analysis which checks if a given set of schedulables is feasible based on a scheduling algorithm. RTSJ provides a *PriortyScheduler* which provides preemptive fixed priority scheduling. It makes sure that the schedulable object with the maximum priority is running at any point of time. The behaviour of the schedulable objects (or schedulables) can be specified by their *ReleaseParameters*, *SchedulingParameters* and the *MemoryParameters*. The ReleaseParameters specifies timing characteristics of the schedulable objects

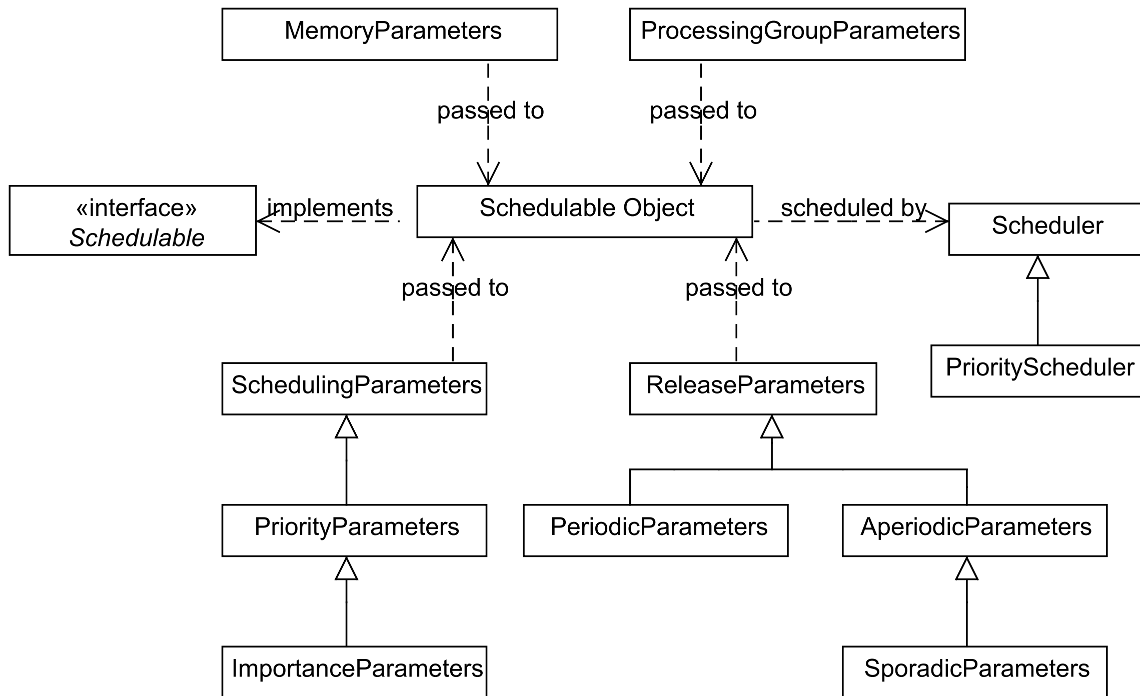


Figure 3.3: Scheduling in RTSJ

e.g. the starting time of the schedulable and the type of schedulable i.e. *periodic, sporadic or aperiodic* etc. The SchedulingParameters define the eligibility criteria for selection (e.g. priority) of schedulable when the scheduler wants to dispatch a new schedulable. *ProcessingGroupParameters* provide a mechanism for cost enforcement for a group of schedulables.

The rest of this section discusses the RTSJ features that can be used to support cc-NUMA systems. A number of new features have been accepted as part of the JSR-282 where RTSJ was being extended for multiprocessors with the focus particularly on SMPs. Here we consider some of the features that are a part of RTSJ 1.1 which can be used to support cc-NUMA systems.

3.2.1 The *AffinitySet* Class

On a multiprocessor, any thread has the choice to run on a number of different processors. Global scheduling and fully partitioned scheduling are the two main scheduling policies that exist for thread scheduling on multiprocessors. In addition, a number of hybrid approaches also exist which essentially combine the two. In Java,

the *int availableProcessors()* method in *java.lang.runtime* provides the number of available processors but no more support is provided in the API in this respect. Therefore, in order to support these scheduling policies, a new *AffinitySet* class (as shown in Figure 3.4) has been proposed for RTSJ 1.1. This new class associates real-time threads and bound asynchronous event handlers to particular processors[4]. It is assumed that each processor has its own logical queue and a schedulable object in the ready state may be present on one or more queues in the system depending on its affinity. Processor affinities could have been added to the existing release parameters, however, in order to avoid major changes to the existing API, a new class has been specified. AffinitySets in RTSJ can be divided in two different categories

- An array of pre-defined *AffinitySets* can be provided by the implementation which can enforce a particular scheduling arrangement. A pre-defined affinity set can contain one or more processors which can be used with methods such as *getHeapSoDefault, getJavaThreadDefault, getNoHeapSoDefault, getPGroupDefault* to set defaults for Java threads and schedulable objects.

- AffinitySets can also be generated by the application by using the *generate()* method. This allows the application to generate affinity sets at runtime which are suited for that platform without compromising the portability of the application. As it is possible that the OS will not allow global scheduling, no application generated affinity sets can contain more than one processor.

The AffinitySet class enables schedulables to be scheduled by a range of multiprocessor scheduling policies (i.e. global, fully partitioned or mixed) if the policy is supported by the OS. From the perspective of a NUMA system, global scheduling on the entire multiprocessor might not be desirable because of the physical clustering of processors (with respect to memory). However, scheduling threads globally on a NUMA node can be desirable for an application. In such a case, the pre-defined affinity sets can be used to represent processors on a NUMA node leading to schedulables being pinned down to a particular node.

[4]http://rtsj.pter.org/jsr282/Affinity/index.html

AffinitySet
generate (java.util.BitSet bitSet): AffinitySet
get (BoundAsyncEventHandler handler): AffinitySet
get (ProcessingGroupParameters pgp): AffinitySet
get (java.lang.Thread thread): AffinitySet
getAvailableProcessors (): java.util.BitSet
getAvailableProcessors (java.util.BitSet dest): java.util.BitSet
getCurrentProcessor (): int
getHeapSoDefault (): AffinitySet
getJavaThreadDefault (): AffinitySet
getNoHeapSoDefault (): AffinitySet
getPGroupDefault (): AffinitySet
getPredefinedSetCount (): int
getPredefinedSets (): AffinitySet []
getProcessors (): java.util.BitSet
getProcessors (java.util.BitSet dest): java.util.BitSet
isProcessorInSet (int processorNumber): boolean
isSetAffinitySupported (): boolean
set (AffinitySet set, BoundAsyncEventHandler handler): void
set (AffinitySet set, ProcessingGroupParameters pgp): void
set (AffinitySet set, java.lang.Thread thread): void

Figure 3.4: The AffinitySet Class

All threads in RTSJ are associated with an affinity set which is either set explicitly, or it is inherited from the parent or a default value has been set. The default affinity sets returned by *getHeapSoDefault, getJavaThreadDefault, getNoHeapSoDefault, getPGroupDefault* are used only if a thread cannot inherit affinity from the parent, as in the following cases:

1. Java threads do not inherit affinity from schedulable objects.

2. Schedulable objects do not inherit affinity from Java threads.

3.2.2 The Physical Memory Framework

One of the most important extension to the Java platform made by the RTSJ is the introduction of new memory regions (scoped and immortal memory areas). The purpose of these memory areas is to prevent any kind of delays that can be caused by the garbage collector. In addition, the specification also defines new classes which enables the programmer the capability to access the physical memory and I/O devices. Following are some of the scenarios where accessing the physical memory

directly can be useful:

- Embedded systems are low in resources and have real time requirements. They have very limited amount of memory that needs to be used efficiently, therefore, they require low level access to the connected devices in order to use them more efficiently.
- Demand paging is known to affect determinism strongly. Allocating directly on the physical memory avoids the demand paging mechanism of the kernel, as the physical memory already has been allocated.
- In addition, accessing the physical memory directly can help to avoid congested shared busses and if there is a specialized device or memory e.g. if we have very fast memory then we will want to allocate the most frequently used data on it.

The RTSJ supports the creation of immortal and scoped memory areas directly on the physical memory by providing ImmortalPhysicalMemory, LTPhysicalMemory and VTPhysicalMemory classes. These memory areas maintain their characteristics with the only difference that their backing store[5] is created directly on the physical memory by either specifying the location or the type of memory where they are to be created. The RTSJ allows provides a *PhysicalMemoryTypeFilter* and a *PhysicalMemoryManager* to enable programmer access to the physical address space.

Normally programmers have no idea about the underlying memory architecture because the operating system hides all the complexity and shields the programmer from any hardware related issues. This can result in inefficient usage of the hardware resources which results in the higher memory access times. The overall effect of this can be degradation in performance or, in case of real-time applications, non-deterministic behaviour from the application. This can be avoided by making the

[5]The backing store is the memory which is represented by the MemoryArea object which would otherwise not be visible to the Java code. The backing store can be created using malloc() etc. It is important to note that each Java object in RTSJ will be associated with two memory areas: the memory area where the the object is allocated and the backing store which the memory represented by it.

application and the JVM aware of the underlying hardware resources. Therefore, a representation of the hardware is required which can be presented to the programmers. The *PhysicalMemoryTypeFilter* can be used to represent the different types of memory that are available to the application for use. According to the RTSJ, any memory device that has a special characteristic should have an associated filter to allow access to this memory. For a NUMA system, each node has a separate non-overlapping range of memory addresses. The *PhysicalMemoryTypeFilter* can be used to represent the memory of a node which can be used by the application to understand the architecture of the system.

3.2.3 Processing Group Parameters (PGPs)

Processing group parameters is an optional feature of the RTSJ which groups together a number of schedulable objects and associates a *cost* for them for every period P. Implementations that support processing group parameters are required to make sure that the cost is not exceeded by the schedulables that are associated with that processing group parameter. Processing group parameters require a group budget to be allocated for them by the operating system. On multiprocessors, processing group parameters can be implemented by the following approaches [Wellings and Kim, 2008].

1. The processing group parameter can be allocated a budget on a single processor. Although the processing group parameter might be able to execute on other processors, it is only provided a guarantee on that processor. The processing group parameter if it wants to use the budget will only be able to use a single processor at any instant of time wasting all the computation power that could otherwise be used. This approach can be seen in Figure 3.5.

2. The second case is when the processing parameter provides a single global budget on multiple processors. All the threads in the processing group will share the budget irrespective of the processor on which they are executing. The budget can be used very efficiently, however, it is very difficult to implement (see Figure 3.6).

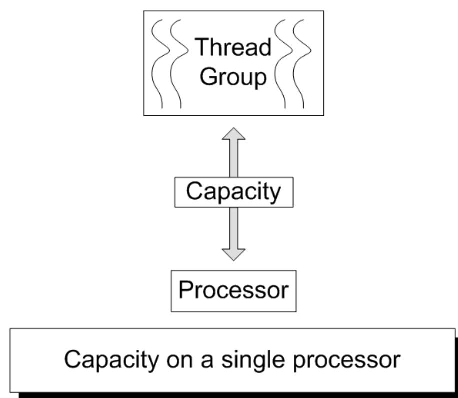


Figure 3.5: Capacity on a single processor

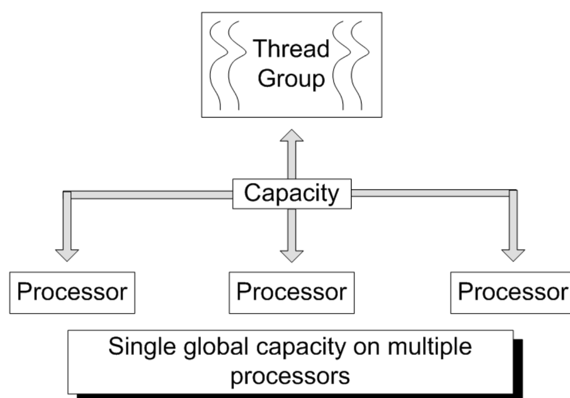


Figure 3.6: Single global capacity on multiple processor

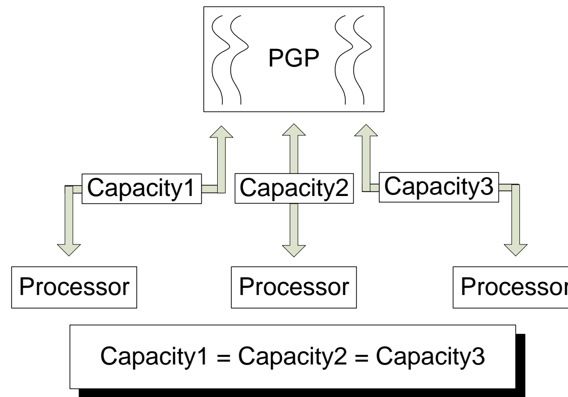


Figure 3.7: Partitioned equal capacities on multiple processors

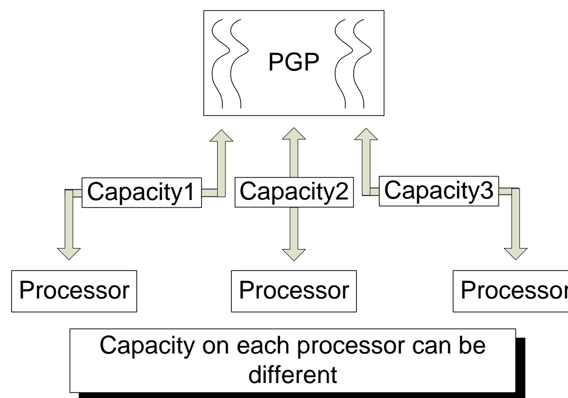


Figure 3.8: Partitioned different capacities on multiple processors

3. The processing group parameter is allocated a number of budgets which are fixed to specific processors. All the processors provide the same amount of budget. This is very similar to the first approach where the budget was only restricted to one processor (see Figure 3.7).

4. The processing group parameter is provided budgets on specific processor like in the previous case, however, here the budget that each processor provides is different (see Figure 3.8).

Due to the implementation challenges in detecting budget overruns in the case of global scheduling in multiprocessors, it has been proposed for RTSJ-1.1[6] that schedulables associated with a processing group parameters will only be able to execute on the same processor.

[6] http://www.rtsj.org/docs/V1.1AlphaProg.html

3.3 Supporting cc-NUMA Systems

In this section, a 4 node cc-NUMA system is presented as a case study to analyze the existing support in the OS and the RTSJ. This system is a 16 core system based on 4 AMD Opteron chips as shown in Figure 3.9. Some features of this system are discussed below:

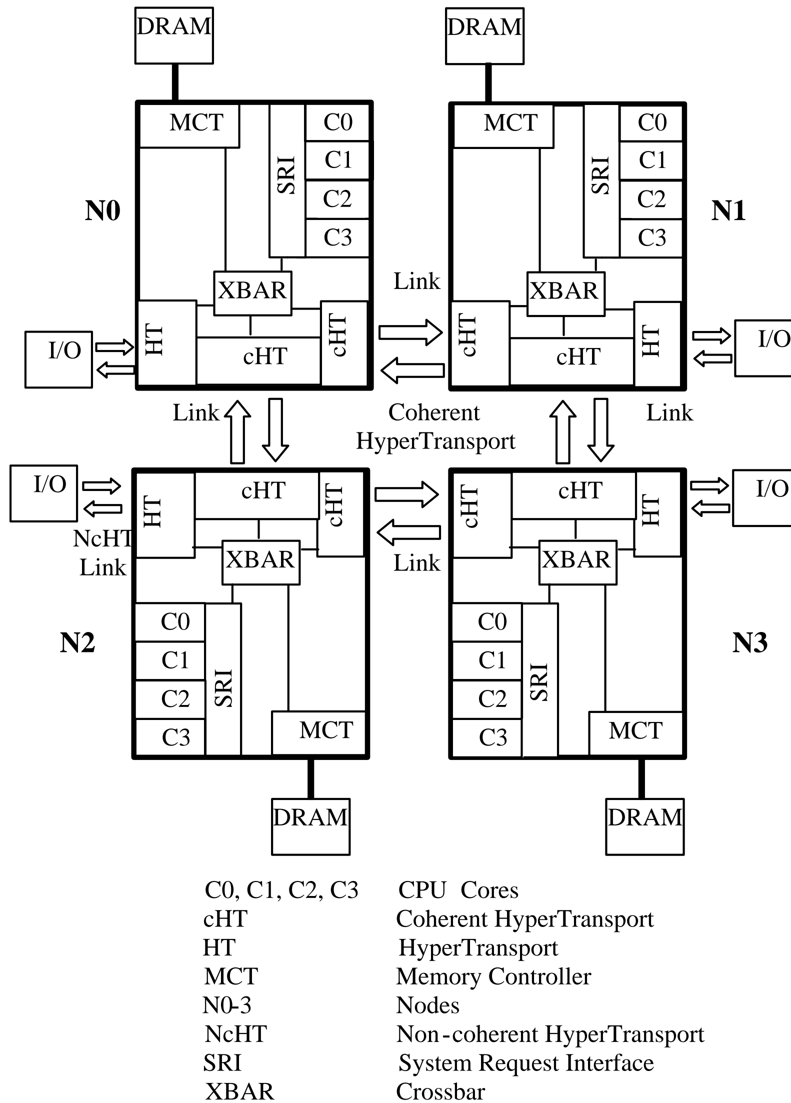


Figure 3.9: A 4 Node NUMA Architecture based on AMD Opteron

1. The system has four sockets, each socket filled by a quad-core AMD Opteron chip. All four AMD Opterons are connected via the coherent HyperTransport (cHT). The cHT is a high speed point to point link between the Opterons. In

the rest of the thesis, each core will be referred as a processor and the Opteron chip will be referred as a node.

2. The system has 4 nodes and a total of 16GB of DRAM memory. Each node has an integrated memory controller and is connected to 4GB of DRAM which is local to all the processors on that node. At boot time, all nodes locate their local memory and map it onto a single physical global address space [AMD, 2003]. Usually the higher order bits of the physical address determine to which node the address belongs. The physical address space (PAS) is not necessarily contiguous i.e. there can be holes in it.

3. The cHT is responsible for keeping the caches coherent at node level. Generally cache coherence is not scalable, therefore, cache coherency for large NUMA systems may be available partially.

4. The latency of memory access depends on the location from where the memory is being accessed. It is not necessary that all the nodes are directly connected to each other; however there must be an indirect path that exists between the nodes through other nodes. Figure 3.10 shows the memory hierarchy of a cc-NUMA system. Although all processors will have the same view of memory, the access time will depend on the location of the data in the hierarchy.

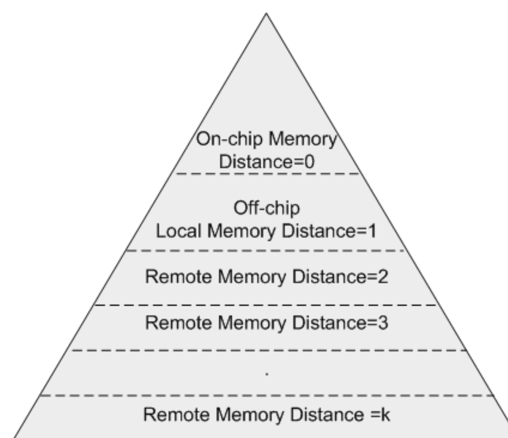


Figure 3.10: Memory Hierarchy of a NUMA System

5. In Figure 3.10, distance is measured as the number of hops from the processor accessing the data to the memory bank where the data resides. In addition

to different levels of memory, there are also multiple levels of caches (L1, L2 etc.) which affect the access timings. Interconnects on a chip are very fast when compared to an off-chip memory access. Therefore, if allowed by the cache coherency protocols, data fetched from caches connected to the other processors can be faster than an off-chip memory access. The distances for Figure 3.9 are given in Table 3.1.

Distances	$Memory(N0)$	$Memory(N1)$	$Memory(N2)$	$Memory(N3)$
Node0(N0)	1	2	2	3
Node1(N1)	2	1	3	2
Node2(N2)	2	3	1	2
Node3(N3)	3	2	2	1

Table 3.1: Distances based on bus accesses for Figure 3.9

The latency of memory accesses depends on the saturation (communication traffic) of resources as well as the distance. [AMD, 2006] discusses in detail how distances and saturation of resources affects the latency.

The ACPI specification [Corporation et al., 2005] defines a System Locality Information Table (SLIT) to describe the distances between *Localities*. Localities are considered to be SMP nodes and all processors, memory and devices belonging to the node are part of the locality. SLIT stores the latency values from one locality to another. The distance from a locality to itself is normalized to 10 and the rest of the values are scaled based on the local value. Table 3.2 shows the distances for the reference architecture generated by Linux-2.6.27. The generated values show diagonal values of 10, and all the rest as 20. However, distance between N0-N3 should not be the same as N0-N1. The reason being for this is that the SLIT has not been setup properly and only shows difference between local and remote memory (and the not the actual difference of latencies between nodes). For the same architecture we do show that a difference in latencies does exist between nodes with distance=1 hop and distance=2 hops (see Appendix A.3).

There are two points worth making about this example architecture:

$Distances$	$Memory(N0)$	$Memory(N1)$	$Memory(N2)$	$Memory(N3)$
N0	10	20	20	20
N1	20	10	20	20
N2	20	20	10	20
N3	20	20	20	10

Table 3.2: System Locality Information Table for Figure 3.9

1. The architecture physically supports a coherent memory model. By this we mean that the individual processors see the same physical memory because the local caches are kept coherent.

2. There can be significant performances increases by ensuring that when cache misses occur, all main memory accesses are local.

3.3.1 Representing the NUMA Architecture

During the booting process, the OS parses the SRAT and develops further APIs to effectively represent this data. The SRAT for the architecture in Figure 3.9 is shown in Table 3.3 and Table 3.4.

Table 3.3 represents information of the processors, the important information is available in the third (Proximity Domain (PD)) and fourth column (APIC) of the table. The APIC (Advanced Programmable Interrupt Controller) is the processor number which corresponds to a PD which shows which node the processor belongs to. The OS parses this table and finds there are 4 proximity domains (nodes) and a total of 16 processors (0-15).

Table 3.4 represents information of the memory present in the system. It shows there are 6 ranges of physical memory with the *Base Address* in the fourth column and the *Address Length* in the fifth column. Each range belongs to a node which is in the third column i.e. PD (Proximity Domain).

The following considers how we can use RTSJ to represent the architecture:

1. Representing the Processors – On a NUMA system, the implementation should be able to support pre-defined affinity sets which contain processors corre-

Subtable Type (ST)	Length	Proximity Domain(PD)	APIC	Flags	Reserved
00	10	00000000	00	00000001	00000000
00	10	00000000	01	00000001	00000000
00	10	00000000	02	00000001	00000000
00	10	00000000	03	00000001	00000000
00	10	00000001	04	00000001	00000000
00	10	00000001	05	00000001	00000000
00	10	00000001	06	00000001	00000000
00	10	00000001	07	00000001	00000000
00	10	00000002	08	00000001	00000000
00	10	00000002	09	00000001	00000000
00	10	00000002	0A	00000001	00000000
00	10	00000002	0B	00000001	00000000
00	10	00000003	0C	00000001	00000000
00	10	00000003	0D	00000001	00000000
00	10	00000003	0E	00000001	00000000
00	10	00000003	0F	00000001	00000000

Table 3.3: The System Resource Affinity Table (SRAT) for architecture in figure 3.9 showing 16 processors

ST	Length	PD	Base Address	Address Length	Flags	Res
01	28	00000000	0000000000000000	00000000000A0000	0..1	00..00
01	28	00000000	0000000000100000	00000000C7F00000	0..1	00..00
01	28	00000000	0000000100000000	0000000038000000	0..1	00..00
01	28	00000001	0000000138000000	0000000100000000	0..1	00..00
01	28	00000002	0000000238000000	0000000100000000	0..1	00..00
01	28	00000003	0000000338000000	0000000100000000	0..1	00..00

Table 3.4: The System Resource Affinity Table (SRAT) for architecture in figure 3.9 showing memory

sponding to the processor distribution in the nodes. For the architecture presented in Figure 3.9, the following four pre-defined affinity sets should be defined:

AffinitySet affinity[0]={0,1,2,3}
AffinitySet affinity[1]={4,5,6,7}
AffinitySet affinity[2]={8,9,10,11}
AffinitySet affinity[3]={12,13,14,15}

The RTSJ provides an interface to get default affinity sets for different schedulable objects default affinity sets for threads (getPGroupDefault()), schedulable objects using the heap (getHeapSoDefault()), NoHeapRealTimeThread (NoHeapSODefault()), processing groups (getPGroupDefault()).

No such interface is provided for NUMA nodes which specifically returns affinity sets of NUMA nodes. Without such an interface programmers can use the *AffinitySet.getPredefinedSets()* to retrieve the affinity sets of the nodes of the NUMA system. However, programmers need to have specific knowledge of the architecture and implementation to correlate which affinity set belongs to a node in the NUMA.

2. Representing the Memory – The memory of a NUMA node is to be represented to allow specific access to a node memory. A *PhysicalMemoryTypeFilter* represents any memory type that has any special characteristics. A filter is like a device driver which is either the responsibility of the device vendor or the JVM to provide the filter for any particular type of memory. A memory belonging to a particular node is different from conventional memory types because a node memory has it's properties only because of the configuration on which it has been setup unlike the memory any other memory type or device which has some inherent property which requires a particular filter in order to access it. Therefore, it is the responsibility of the JVM on a NUMA system which needs to provide filters for all the different nodes of the system depending on how it has been configured. These filter make sure that the individual node

memory is available for explicit allocation of objects. The memory type filter will statically save the memory range of the node assuming that the memory range of each node will remain same during runtime. For the example system, we need to represent memory devices on all nodes. Therefore, we need four filters one for each node to provide access to all the memory in the system. The following code snippet shows four filters being created where NodeMemory is a type that implements the *PhysicalMemoryTypeFilter* and the *PhysicalMemoryName* interfaces:

```
// This code needs to be implemented the JVM.
// If not then the programmer will have to implement
// this for access to individual memory of a node.
long base [][] = new long[nodes][maxRanges];
long sizes [][] = new long[nodes][maxRanges];
base[0][0]= 0; // equal to 0000000000000000
size[0][0]= 655360; // equal to 000000000010000
base[0][1]= 1048576; // equal to 000000000100000
size[0][1]= 3354394624; // equal to 00000000C7F00000
base[0][2]= 4294967296; // equal to 000000100000000
size[0][2]= 939524096; // equal to 0000000038000000
base[1][0]= 5234491392 // equal to 0000000138000000
size[1][0]= 4294967296; // equal to 000000100000000
base[2][0]= 9529458688 // equal to 0000000238000000
size[2][0]= 4294967296; // equal to 000000100000000
base[3][0]= 13824425984 // equal to 0000000338000000
size[3][0]= 4294967296; // equal to 000000100000000
NodeMemory  memory[0]=
  new NodeMemory(base[0],size[0]);
NodeMemory  memory[1]= new NodeMemory(base[1],size[1]);
NodeMemory  memory[2]= new NodeMemory(base[2],size[2]);
NodeMemory  memory[3]= new NodeMemory(base[3],size[3]);

// we need to create four PhysicalMemoryName objects


java.lang.Object nodememory0=new java.lang.Object();
java.lang.Object nodememory1=new java.lang.Object();
java.lang.Object nodememory2=new java.lang.Object();
```

```
java.lang.Object nodememory3=new java.lang.Object();
PhysicalMemoryManager.registerFilter(nodememory0,memory[0] );
PhysicalMemoryManager.registerFilter(nodememory1,memory[1] );
PhysicalMemoryManager.registerFilter(nodememory2,memory[2] );
PhysicalMemoryManager.registerFilter(nodememory3,memory[3] );
```

The problem with this code, in fact with the physical memory model is that memory has to be accessed through physical addresses only and RTSJ requires physical access to all the address space. This might be acceptable in an embedded system where all the physical memory be directly accessible from user space, however, this is an extreme scenario for the stability of any general purpose OS. In Linux on x86 architecture, the */dev/mem* provides access to the physical address space but limits it to only the I/O space. A patch[7] provides access to all physical address space, however, it is supposed to be used for the RTSJ compliance test, accessing the physical memory might cause the system to crash.

3.3.2 Pinning Schedulable Objects to Processors

The *AffinitySet* class discussed in Section 3.2.1 allows programmers to pin schedulable objects to specific processors. Pre-defined affinity sets corresponding to processors of a NUMA node can be used to allocate threads to processors within a NUMA node. The following code shows setting affinity of a SO on a NUMA node:

```
AffinitySet[] set1 = new
  AffinitySet[AffinitySet.getPredefinedAffinitySetCount()];
set1=AffinitySet.getPredefinedAffinitySets();

// If SO is a schedulable object and we know that set1[0] is
// a NUMA node then the affinity to the node can be set as:
AffinitySet.set(set1[0], SO);
```

Setting the affinity to all the processors on a NUMA node will allow the schedulable object to be globally scheduled on these processors, but only if the OS provides

[7]http://lwn.net/Articles/184783/

multiprocessor global scheduling.

3.3.3 Allocating Objects on Specific Nodes

The RTSJ has a number of different memory areas on which an object can be allocated. These memory areas differ from each other based on their lifetimes and how memory is freed from objects that are no longer active. The following considers how locality can be ensured with the different RTSJ memory areas:

- Immortal Memory Area – A single immortal memory instance exists in a RTJVM which is shared by all schedulable objects. This instance is created during the initialization phase of the RTJVM and throughout the life time of the application it may stay on the same memory address depending on how it has been implemented. This does not allow any allocations to be directed on specific nodes, however, a physical immortal memory area can be created on a specific node which can be used to allocate objects on that particular node.

- Scoped Memory – Like the physical immortal memory area, the RTSJ physical memory framework also provides physical scoped memory areas e.g. the *LT-PhysicalMemory* and *VTPhysicalMemory*. These memory areas can be used to allocate objects on scoped memory areas on a specific node in a NUMA system.

Physical memory areas can be created on a node by passing the type of a memory area. Figure 3.11 shows a sequence diagram to create a LTPhysicalMemory, which shows the following steps:

1. The constructor of the LTPhysicalMemory is called as shown in the following snippet:

   ```
   LTPhysicalMemory ltm= new LTPhysicalMemory(nodememory0, size);
   ```

2. The constructor of LTPhysicalMemory then asks the manager create a physical memory area with the name *nodememory0* with the required size.

3. The manager then finds the memory type filter for the memory name and then asks the filter for a physical memory region with the required size. The filter returns the physical memory address that is free with the required size.

4. The manager then asks the filter to find a virtual address best suited for the physical memory. The filter finds it and returns it back.

5. The manager now asks the filter to prepare the memory by initializing the physical memory by allocating it a virtual address. After the initialization the memory area is passed backed to the constructor.

It is important to note that the reference of nodememory0 is not visible to the application through the specification. Either the runtime or the programmer need to implement it. In theory, programmers can create a filter however, it is very difficult for the programmers because filters are very low level which require understanding of the kernel, Java runtime and the memory subsystem.

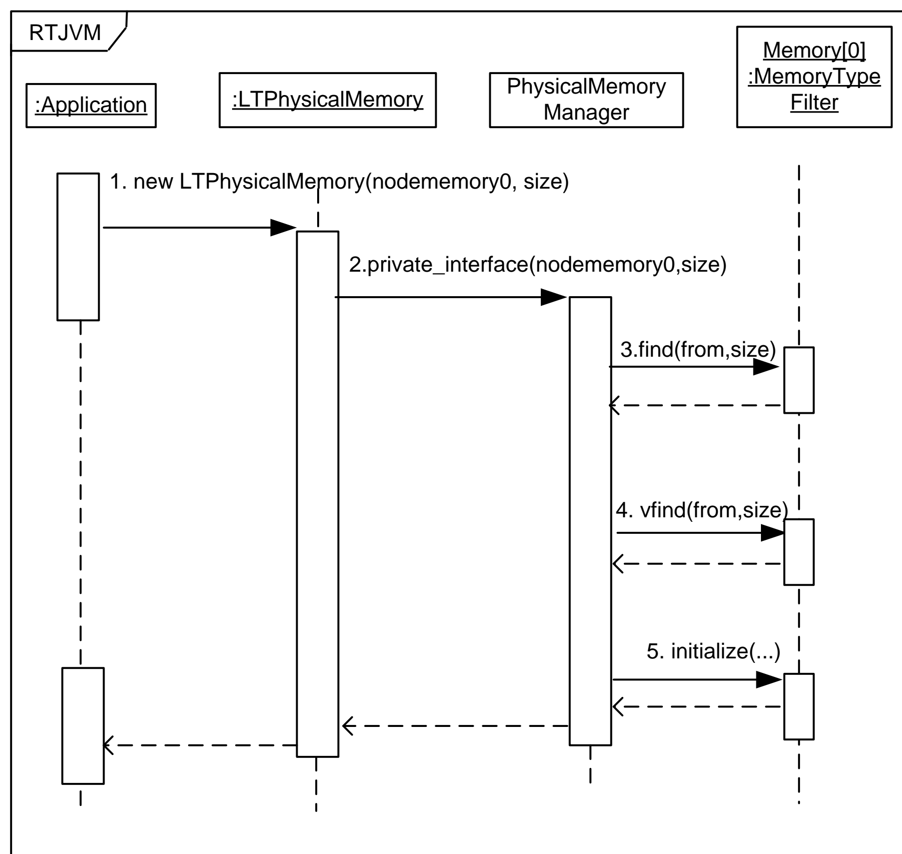


Figure 3.11: Creating an LTPhysicalMemory area

3.4 Limitations

Using existing features in the RTSJ such as the *AffinitySets* and the *PhysicalMemoryTypeFilter* to support open applications on a NUMA system has the following limitations:

- Limited Support – Existing features do not support all the requirements described in Section 2.4. Static allocation is supported where, threads and objects can be allocated individually on separate nodes to provide locality. However, it requires a lot of effort from the programmer and for larger NUMA system or a complex application, it becomes very complex to ensure locality because affinity has to be set for every thread and the programmer has to keep track where other related objects are allocated.

  It can been seen that schedulable objects can be grouped together in processing groups parameters, however, on multiprocessors threads associated with a PGP will execute sequentially undermining the parallelism available to the application. These is no facility to allow multiple processing groups (PG) to be associated with a single schedulable object.

  The RTSJ does not provide any design patterns which will enable dynamic mapping of applications on the platform. Applications have requirements for usage of resources and resources in a NUMA system are clustered together in NUMA nodes. Though resources from different nodes can be used by a thread group, however, to avoid non-deterministic behaviour, a thread group must be mapped onto a single node. This mapping then in essence should be based on the resource requirement of the thread group and the RTJVM needs to provide guarantees to thread groups. No such support exists in the RTSJ at the moment.

- Accessing the Physical Memory Model – The physical memory framework allows programmers to have access to all the physical address space. In an operating system such as Linux, there are concerns over the security of the overall system. Although the RTSJ comes with a security manager which

should make sure that the applications do not misbehave, however, potentially there is a danger of messing up the system because the existing RTSJ does not say which addresses are safe to access. Accessing the physical memory directly from an application can create an arbitration problem when more than one application accesses the physical memory directly. Within a single application, the physical memory manager keeps a record of the addresses that have already been used. No such protection is available to isolate memory from other applications.

- Portability – The architecture of the system is hard coded in the application. Code is not portable across platforms and porting the application will require significant change to the source in order to port for some other architecture.

- Code Locality – No support for the locality of code exists at the moment. Consider the case where the thread is executing on one processor and data is also local to it. However, the method area, which contains the code and the static variables may be on a remote area.

- Local Heap – While existing RTSJ features allow creation of Immortal and Scoped memory areas on NUMA nodes, no such support exists for the heap. Normally the physical location of the heap is not known on a NUMA architecture, however, the RTJVM can be implemented to allocate the heap by distributing it across the NUMA nodes [Tikir and Hollingsworth, 2005b]. In such a configuration the heap will be divided such that all nodes will have a local heap, so objects can be placed locally for all processors. Even if local heaps can be ensured for all processors, while performing GC and compaction the objects may move around. Therefore, the garbage collector should be aware of the boundaries of the NUMA nodes and should respect these boundaries when moving the objects. In addition, the application might want to allocate objects on a specific remote heap. Based on these requirements there is a need to extend the RTSJ physical memory types to include physical heaps. This will not only allow the programmer to partition the heap between the memory banks but it will enable the programmer to target specific heap for

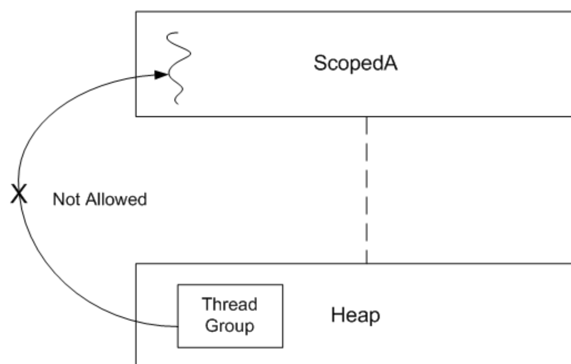


Figure 3.12: Scope stack for a *ThreadGroup*

the allocation of objects.

- Thread Grouping – Thread placement by grouping them together is an easier task for programmers and it is especially beneficial when groups are allowed to be placed dynamically on a node. In Java, grouping of threads is provided by the *ThreadGroup* class. A thread group contains references to threads or thread groups, hence, thread groups are structured as a tree. In the case of RTSJ, thread groups violate the memory assignment rule (see Figure 3.12) e.g. thread groups need to reference any real-time threads which are created on scoped memory areas, however, no references are allowed from a heap or an immortal memory area to a scoped memory area. Therefore, real time threads cannot be a part of any thread group and the *Thread.getThreadGroup()* returns *null*.

  Processing group parameters is another way where a number of threads can be associated to the same object and a group budget can be allocated to the group of threads but again it will make threads sequential on a multiprocessor.

3.5 Summary

This chapter summarized the physical memory model, the affinity set class and the processing group parameters which can be used to support open applications on a cc-NUMA system. However, a number of limitations were made in the existing specification which have been listed at the end of the chapter. The remainder of the

thesis sets out to remove these limitations.

Chapter 4

Locality Model

In the previous chapter, we analyzed existing features of the RTSJ to deploy real-time applications on a cc-NUMA system. It was concluded that the existing features available in the specification are not sufficient, and a number of extensions are required to design and deploy real-time Java applications while maintaining portability and determinism. The following requirements are outlined for this purpose:

- R1 – The existing support for representing the architecture is very limited. The previous chapter showed that effective abstractions are needed to define the architecture which will define individual components in the systems and define the topology of their interconnections.
- R2 – Resources in the NUMA system are grouped together into clusters (or a UMA sub-system). These clusters can only be used effectively if the application is small enough to fit onto one UMA sub-system, otherwise, the application needs to be modular/partitioned (or based on components) such that there are no strong dependencies among these modules. New abstractions are needed which can group together threads and objects and are compliant with the RTSJ.
- R3 – Temporal isolation needs to be provided for thread groups/applications. This will enable them to execute in a timely manner even when the platform is being shared with other scheduling entities (tasks, components and applications).

- R4 – In case of embedded real-time systems, programmers would like to map their application/thread group statically on a cc-NUMA system. Support for such allocations should be provided.

- R5 – For performance portability of applications across platforms, implicitly mapping of thread groups on resources needs to be supported.

- R6 – The physical memory type filter is a good abstraction to provide explicit access to a node memory. However, using the physical memory directly jeopardizes the security of the system and moreover it is not widely available creating potential implementation issues. Therefore, explicit access to node memory needs to be provided by APIs provided by the OS.

Based on these requirements, a locality model is presented in this chapter which defines new abstractions which can be categorized as following:

1. Architectural model – makes the JVM NUMA aware and provides an API for programmers to have more visibility.

2. Application model – allows partitioning and allocation of the application maximizing locality.

3. Resource reservation model – describes how resources can be guaranteed for the thread groups.

The chapter is structured in a number of sections, Section 4.1 discusses the architectural model which is used to represent the architecture. Section 4.2 discusses the application model followed by Section 4.3 discussing the resource reservation model. Section 4.4 is the last section which summarizes the locality model presented in the chapter.

4.1 Architectural Model

The goal of the architectural model is to make the application and the JVM aware of the hardware architecture. This requires the runtime to be able to discover the

hardware components and their topology in the system and then provide an interface for the application.

The memory sub-system in a NUMA sub-system behaves differently at different levels, the following definitions provide an insight into memory properties in NUMA and different sub-systems:

- Processor: A processing element that also has an associated cache. We assume all the processors within a system will have the same instruction set.

- Memory: Memory that is connected to one or more processors through a bus, and is directly addressable by all processors.

- Device: I/O devices which are connected to one or more processors through a bus.

- Multicore Processor: A chip with more than one processing core. These processing cores share a cache.

- Uniform Memory Access (UMA) System: A system that ensures uniform main memory access time for all processors(one or more). Processors usually share a single bus to access memory and devices. Processors, memory and devices are the basic building blocks of a UMA system. Symmetric Multiprocessors (SMPs) are a type of UMA system which have two or more processors.

- Cache Coherent NUMA (cc-NUMA) System: A multiprocessor system where processors have different latencies when accessing the same memory. A single address space exists and caches are kept coherent. Usually a group of UMA subsystems compose a cc-NUMA system.

- NUMA System: A multiprocessor system where processors have different latencies when accessing the same memory, but the caches are not coherent as they were in the case of a cc-NUMA system. Still a single address space exists in NUMA systems. The usual building blocks of a NUMA system are UMA and cc-NUMA subsystems.

As it can be seen that at different levels in the NUMA system there are different characteristics of memory e.g. at the cc-NUMA level we have cache-coherence and a single address space but we do not have uniform access time. Therefore we use different abstractions which will guarantee the type of memory that is provided. These abstractions can be categorized as following:

1. Abstractions that are used for basic components i.e. processors, memory and devices.
2. Abstractions that are used to represent multi-processor systems that are constituted from the basic components.
3. An interface class to provide access to the architecture representation.

The basic hardware components of every system are considered to be processor, memory and devices. These components are then connected by busses and then interconnected to form a multiprocessor system. Figure 4.1 shows the classes that are used to represent the different hardware components of the system. In the rest of this section we will define all these classes in detail. It is assumed that the details of the architecture are available to the application at runtime either in the form of support from the operating system [Broquedis et al., 2010] or as a configuration file provided by the underlying platform to help the RTJVM find all the components of the system.

4.1.1 Abstractions for Basic Architectural Components

A number of new abstractions that represent the basic components of a NUMA systems. These components include the processors, memory and devices. In the model each of these components have been given a separate class that can be shown in Figure 4.1.

4.1.1.1 The Abstract Component Class

The *Component* class represents a basic building block of the NUMA sub-system. It is extended by classes *Processor*, *Memory* and *Device* to be able to create the

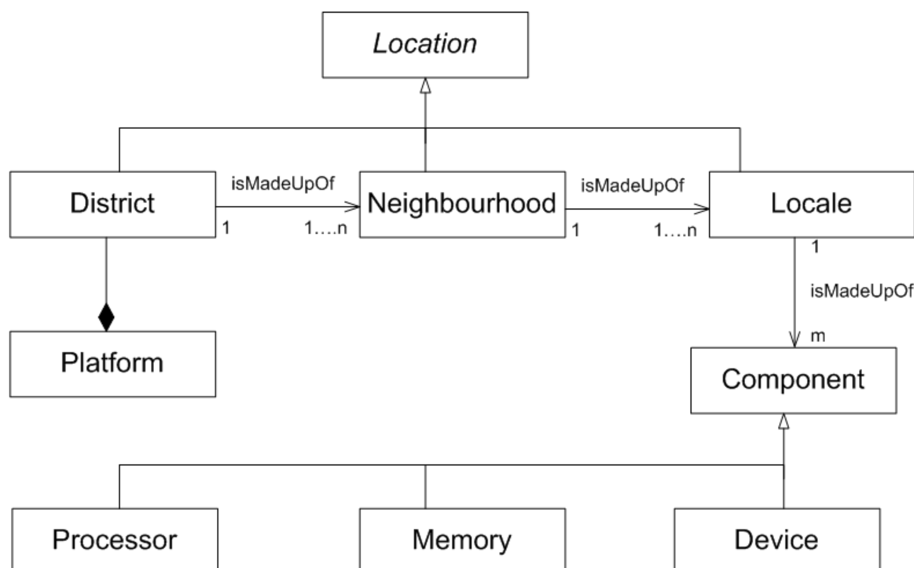


Figure 4.1: Abstractions for Architecture Representation

basic building blocks of the systems.

Class	
public abstract class Component	
Methods	
int	**distance**(Component toComponent) Returns distance from this component to another component.
public Locale	**getLocale**() Returns the Locale of the component.

Table 4.1: The *Component* Class

Each component has a distance to other components, this distance is based on the System Locality Information Table (SLIT). This is a standard table that has been introduced by the ACPI specification and has been discussed in detail in the previous chapter in Section 3.3. It is worth noting here that distance is a package private attribute which can be used only by the runtime for mapping optimizations and is not visible to the programmers.

4.1.1.2 The *Processor* Class

Instances of the *Processor* class represents a logical processor which can have its own cache. The class provides support for different types of processors by introducing

the *ProcessorType*, which is a tagging interface that is used to identify the types of processors and is used in the method getType(). Each processor can have a number of *Cache* objects which can either be shared or private to the processor.

Class	
public class Processor extends Component	
Methods	
public ProcessorType	**getType**() Returns the type of the Processor.
public Cache[]	**getCaches**() Returns an array of the caches belonging to the Processor.
public AffinitySet	**getAffinitySet**() Returns the AffinitySet for this Processor.

Table 4.2: The *Processor* Class

In this thesis, we assume that all the processors will be of the same type. Later this work can be extended for more heterogeneous architectures (see future work in Chapter 7). The ProcessorType interface is provided to enable different processors types to be defined.

Interface
public interface ProcessorType

Table 4.3: The *ProcessorType* Interface

The cache belonging to each processor is represented separately by a *Cache* class. This class is shown in the above table which provides more information on the type, size and level of the cache. It also provides a getProcessors() method which returns all the processors which are attached to the Cache. In case of a shared cache an array of Processors will be returned.

Class	
public class Cache	
Methods	
public int	**getCacheType**() Returns 0 = instruction, 1 = data and 2 = Associative.
public int	**getLevel**() Returns the level of the cache 1 = L1, 2=L2 etc.
public long	**getSize**() Returns the size of the cache in bytes.
public Processor []	**getProcessors**() Returns Processors attached to this cache.

Table 4.4: The *Cache* Class

4.1.1.3 The Memory Class

Instances of the memory class represent a memory device. A memory in RTSJ is any memory where the programmer can create a memory area. This memory area then allows the programmers to allocate objects. The following class is used to instantiate all memory devices which allow the programmer to create a memory area.

Class	
public class Memory extends Component	
Methods	
public PhysicalMemoryName	**getType**() Returns the name of the memory. The name is the same identifier used in the physical memory model.
public long	**getBase**() Returns the starting address of the memory if known.
public long	**getSize**() Returns the size of the memory in bytes.

Table 4.5: The *Memory* Class

The RTSJ already has a physical memory framework (discussed in chapter 3) which registers all memory devices. This class will instantiate all memory devices registered in the physical memory framework and the memory on each node of the NUMA system. The model enables programmers to find local memory for each

processor and vice versa.

4.1.1.4 The Device Class

Instances of the device class represent a device. It integrates with RawMemory classes to provide access to the device. The API can be used to find local processors to each device.

Class	
public class Device extends Component	
Methods	
public RawMemory-Name	**getType**() Returns the name of the device. The name is the same identifier used in the raw memory classes.

Table 4.6: The *Device* Class

4.1.2 Abstractions for Architecture Representation

These abstractions represent a multiprocessor system which is a combination of processors, memory and devices. The purpose of these abstractions is to represent the topology of the system. The model does not use any interconnects or busses to show the relationship between basic hardware components, rather, a hierarchical model is used where the memory at each level provides different characteristics.

4.1.2.1 Location

Location is a new RTSJ programming abstraction that represents a locality in the NUMA system. It is made up of processors, memory and devices which the runtime provides to the programmer.

Class	
public abstract class Location	
Methods	
public Processor[]	**getProcessors**() Returns processors in the current Location.
public Memory[]	**getMemory**() Returns memory in the current Location.
public Device[]	**getDevices**() Returns devices in the current Location.

Table 4.7: The abstract *Location* Class

4.1.2.2 The Locale Class

A locale is a programming abstraction that represents a UMA subsystem. It guarantees the application of having uniform access time when accessing main memory, cache coherence and a single address space.

Class	
public class Locale extends Location	
Methods	
public Neighbourhood	**getNeighbourhood**() Returns the Neighbourhood to which this Locale belongs.
public District	**getDistrict**() Returns the District to which this Locale belongs.
public AffinitySet	**getAffinitySet**() Returns the AffinitySet for this Locale.

Table 4.8: The *Locale* Class

4.1.2.3 The Neighbourhood Class

A neighbourhood is an abstraction that represents a cc-NUMA system. It guarantees the application of having cache coherence as well as a single address space but it does not guarantee uniform access time to all the main memory.

Class	
public class Neighbourhood extends Location	
Methods	
public Locale[]	**getLocales**() Returns an array of Locales in the Neighbourhood.
public District	**getDistrict**() Returns the District to which this Neighbourhood belongs.
public AffinitySet	**getAffinitySet**() Returns the AffinitySet for this Neighbourhood.

Table 4.9: The *Neighbourhood* Class

4.1.2.4 The District Class

A district is an abstraction that represents a NUMA system. It guarantees that a single address space will be present for execution, however it does not make any guarantees for cache coherence or uniform access times. The non-availability of cache coherence makes it unsafe to allow programs to execute within a district. However, districts do enable us to obtain the neighbourhoods of locales which can be used to host RTJVMs and execution sites. A district is composed of one or more neighbourhoods.

Class	
public class District extends Location	
Methods	
public Neighbourhood[]	**getNeighbourhoods**() Returns an array of Neighbourhoods in the District.
public Locale[]	**getLocales**() Returns an array of Locales in the District.

Table 4.10: The *District* Class

4.1.3 Interface to the Architecture

The *Platform* class is a static class which provides an interface to the whole architecture. The architecture is build during the initialization of the RTJVM using the *buildPlatform()* method. This is a package private method and we will use it only

before the application starts. However, for dynamic architectures this method can be used on an event a new hardware component has been added. All methods in the *Platform* class are static and can be used by the application as an interface to the architecture.

Class	
public final class Platform	
Methods	
static void	**buildPlatform**() Builds the architecture representation.
public static District	**getDistrict**() Returns the singleton District reference.
public static Neighbourhood[]	**getNeighbourhoods**() Returns an array of Neighbourhoods in the District.
public static Locale[]	**getLocales**() Returns an array of Locales in the District.
public static Processor[]	**getProcessors**() Returns an array of all processors in the District.
public static Memory[]	**getMemory**() Returns an array of all memory types in the District.
public static Device[]	**getDevices**() Returns an array of all the devices in the District.

Table 4.11: The *Platform* Class

4.1.4 Discussion

In this section, an architecture representation model was presented. The objectives of this model are as following:

- Enabling the programmers to query about the system.
- Making the RTSJ run time platform NUMA-aware.

The architectural model provides a portable way to represent the topology of the NUMA system. It is also compatible with RTSJ physical memory framework allowing programmers and the Raw Memory model. By allowing the retrieval the

memory and devices using the getMemory() and getDevices() methods in the Location classes, the model provides an interface for programmers to physically allocate memory and access device registers. In the rest of the chapter, support is provided to build a runtime which is NUMA aware. Implementation details are provided in Chapter 5.

As an example of the architecture model, the cc-NUMA system that was presented in Section 3.9 is represented. It consists of 4 UMA nodes, each then further contains 4 processors and a memory. The corresponding architecture is built by the JVM which is presented in Figure 4.2.

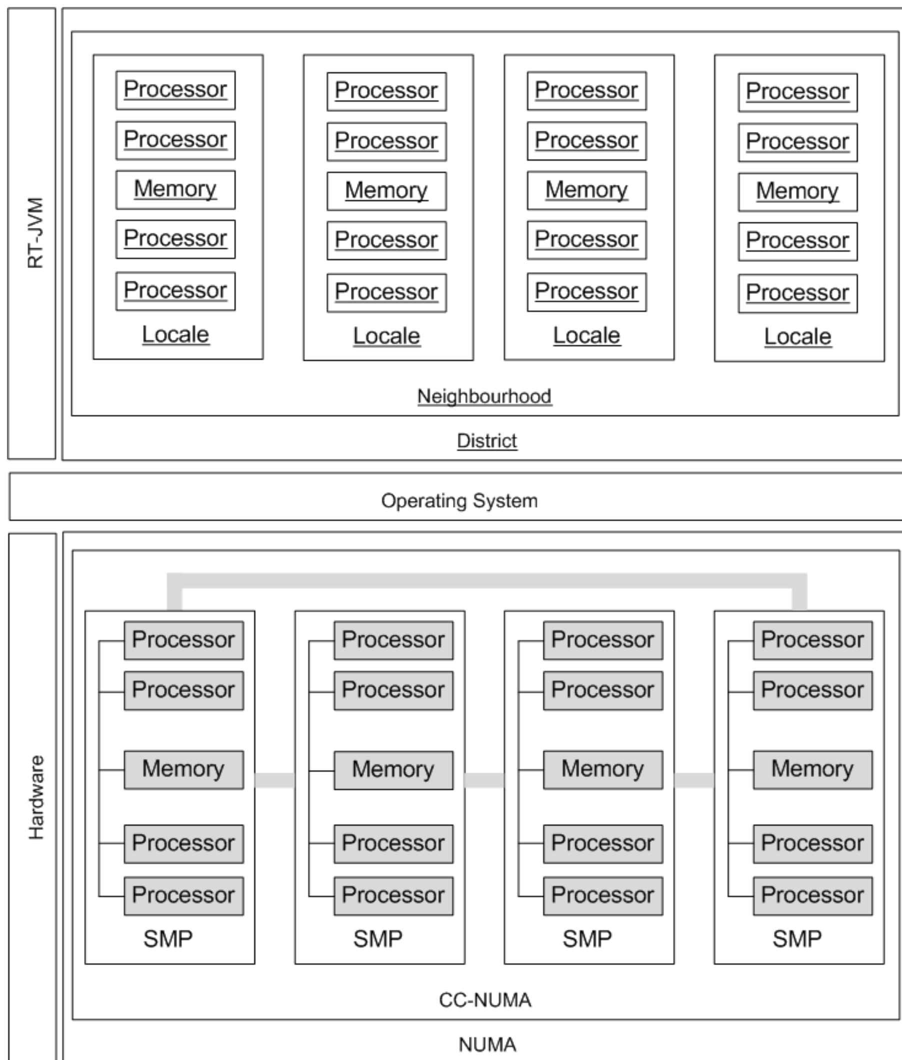


Figure 4.2: System Representation at the JVM level

4.2 Application Model

This section discusses how computations of an application are partitioned into groups of threads. These groups can be then mapped onto a virtual resource hosted on UMA sub-system. The section introduces the following new abstractions:

- ExecutionSite
- Place
- Locality

ExecutionSite is a new abstraction which allows the programmer to create real-time threads and memory areas. The resources for these threads and memory areas will be provided by the *Locality* class which handles the creation and mapping of the ExecutionSite. A Place represents a virtual resource which hosts an ExecutionSite. In addition to the new abstractions, the section also discusses the following:

- Conformance to the RTSJ memory assignment rules
- Local Immortal and Heap memory areas
- Executing existing RTSJ applications

4.2.1 *ExecutionSite*: Abstraction for Controlling Execution

ch4:classES) The execution site is an abstraction which is designed to minimize remote accesses on a NUMA system. It provides factory methods to create threads/schedulables and memory areas. Resources can be guaranteed to an ExecutionSite with PartitionedReservations (see Section 4.3). It is the responsibility of the programmers to identify and distribute closely related threads in an ExecutionSite. The following requirements exist for the mapping of an ExecutionSite:

1. a single address space
2. cache coherence
3. uniform main memory access

The mapping makes sure that they are placed local to each other and avoid any delays caused due to the NUMA architecture. Execution sites support composability at two different levels, first the execution site is composed of a number of schedulable objects and secondly, a number of execution sites combine together to form a real-time Java application. In the model, the only schedulable objects (in RTSJ) that are supported are real-time threads. The model is trivially extendable to asynchronous and bound asynchronous event handlers.

Class	
public class ExecutionSite	
Fields	
SchedulingParameters	priority
Methods	
public Thread	**createJavaThread**(Runnable logic) Creates a Java Thread on this ExecutionSite.
public RealtimeThread	**createRealtimeThread**(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic) Creates a RealtimeThread on this ExecutionSite.
public NoHeapRealtimeThread	**createNoHeapRealtimeThread**(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic) Creates a NoHeapRealtimeThread on this ExecutionSite.
public RealtimeThread	**createRealtimeThreadIfFeasible**(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic) First performs a feasibility analysis for the ExecutionSite using the parameters of the new RTT. Then creates a RealtimeThread in the current ExecutionSite if it is feasible. Returns null if not feasible.
public NoHeapRealtimeThread	**createNoHeapRealtimeThreadIfFeasible** (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic) Same as above but for the NoHeapRealtimeThread.
public MemoryArea	**createMemoryArea**(int MemoryAreaType, long size)

Methods	
	Creates a MemoryArea in the current Location. MemoryAreaType tells which MemoryArea to create. MemoryAreaType = 0 LTMemory, 1 VTMemory, 2 LTPinnableMemory and 3 VT-PinnableMemory.
public MemoryArea	**createMemoryArea**(int MemoryAreaType, long size, Runnable logic) Same as above but also passes logic which will be executed in the newly created MemoryArea.
public AffinitySet	**getAffinitySet**() Returns the affinity set for this ExecutionSite.
public HeapPhysicalMemory	**getHeap**() Returns the physical heap memory for this ExecutionSite.
public ImmortalPhysicalMemory	**getImmortal**() Returns the physical immortal memory for this ExecutionSite.
public PartitionedReservation	**getReservation**() Returns the reservation for this ExecutionSite.
public MemoryParameters	**getMemoryParameters**() Returns the memory parameters for this ExecutionSite.
public boolean	**guarantee**(PartitionedReservation reserve) Returns true if the platform accepts to provide the requested guarantees.
public boolean	**destroy**() Returns true if the platform accepts to remove any guarantees allocated to this execution site.

Table 4.12: The *ExecutionSite* Class

The execution site is a cluster of threads and objects kept together to provide locality. The reason why a new abstraction was proposed instead of simply using the processing group parameter was that the execution site is not only a group of threads but it is virtually a stripped down version of a JVM which is capable of executing a RT-Java program. An execution site is complete with its scheduler,

memory areas and group of threads.

The methods provided in the ExecutionSite class can be divided into following four types:

4.2.1.1 Threads/Schedulables Instantiation

Creating a thread in an execution site means that the thread will be executing on the processors on which the execution site is allocated. The factory method sets the affinity of the thread/schedulable to the affinity of the ExecutionSite and attaches it to the reservations of the ExecutionSite.

4.2.1.2 Memory Areas Instantiations

Execution sites should be able to provide local memory areas to enable threads to allocate objects on the same node where they reside. There are two types of memory areas that are used by the programmers that should be local to the ExecutionSite:

1. Memory areas that are created by the JVM at startup and that remain alive throughout the life of the application. These memory areas include the Heap and Immortal memory area.
2. Memory areas that are created by the programmer. These memory areas include the scoped memory areas.

The execution site provides factory methods to create scoped memory areas. When a scoped memory area is being created using the factory, then the backing store is kept local to the execution site.

4.2.1.3 Retrieving Local Heap and Immortal

The runtime should make sure that a local immortal and local heap memory is created on every node of the cc-NUMA system. The method *ExecutionSite.getHeap()* returns the reference to the local heap and similarly *ExecutionSite.getImmortal()* method returns the local immortal memory area. Each execution site is given a limit on memory usage which cannot be exceeded. Objects allocated inside the local memory area should be globally accessible in the application (only if the accessor

has the reference). The scope stack treats all local heaps and immortals as a single heap or immortal entity. This essentially means that if the object is placed on heap, the memory assignment rules do not care which local heap it exists on. Since objects can be referenced across individual heaps and immortals, there should be no problem of having a single heap and immortal on the scope stack. More details can be found in Section 4.2.4.

4.2.1.4 Basic Resource Reservation Operations

An ExecutionSite can be guaranteed a set of resources. This reservation provides temporal guarantees to the ExecutionSite. Resource reservations have been discussed in detail in Section 4.3.

The method getReservation() can be used to check if the execution site has been guaranteed resources or not. If it returns null, then the platform does not provide any guarantees. In case the programmer wants to change the existing guarantees of the execution site, guarantee(...). This method will return true if the platform does indeed allocate it the required resources and false if the platform is unable to do so.

The destroy() method can be called for releasing any guarantees that have been made by the scheduler for the execution sites.

4.2.2 *Place*: A Logical Location

Place is a virtual resource which is created when the application is instantiated. This class is not visible to the programmers and provides resources required by the ExecutionSite.

After building the representation of the architecture, a Place is created on each Locale. Each Place is given temporal and spatial guarantees on a Locale which are initialized with default values. These guarantees can be renegotiated with the OS at runtime.

Class	
class Place	
Constructor	
Place (Locale local, long heapSize, long immortalSize, RelativeTime[] defaultbudget, RelativeTime[] defaultperiod, int numProcessors)	
Methods	
PhysicalHeap	**getHeap()** Returns the heap which is local to this Place.
PhysicalImmortalMemory	**getImmortal()** Returns the Immortal memory which is local to this Place.
ClusterContract	**getContract()** Returns the contract which is allocated to this Place.
ClusterContract	**setContract(ClusterContract contract)** Sets the contract for this Place.
boolean	**setHeapSize(long size)** Sets the size of the heap memory on this Place.
boolean	**setImmortalSize(long size)** Sets the size of the immortal memory on this Place.
boolean	**setBackingStoreSize(long size)** Sets the size of the backing store for scoped memory areas on this Place.
Locale	**getLocale()** Returns the Locale hosting the Place.
ExecutionSite []	**getExecutionSites()** Returns the ExecutionsSites mapped on this Place.
boolean	**addExecutionSite(ExecutionSite site)** Maps a new ExecutionSite on this Place.

Table 4.13: The *Place* Class

The temporal guarantees are based on a ClusterContract (see section 4.3) which consists of a budget and period on each processor on the node.

It is allocated a local heap and an immortal memory area which can be later used by all ExecutionSites which are hosted on this Place. The *Place* class provides two package private methods getHeap() and getImmortal(). Each Place has a local heap and a local immortal memory area that is shared by all the execution sites

(as shown in Figure 4.3). When the execution sites are mapped on a Place, each execution site provides an object of memory parameters which specify the memory requirements of the execution site.

One or more execution sites can be mapped on to a Place which allow the programmer to create real-time threads and memory areas. The Place keeps a reference to all the execution sites that are mapped onto the locale and it serves as a resource manager for each execution site as it keeps a track of the guarantees that are provided to them.

4.2.3 *Locality*: Allocating *ExecutionSite*

This class acts like an interface between the ExecutionSites and the Locales and provides the necessary mapping between them during the allocation of Execution-Sites. The *initialize(...)* builds the Places on all Locales before the application starts executing.

The execution site can be created by using a number of factory methods in the Locality class, where all the factory methods accept a locale from the programmer. This is used as a hint for the allocation process of each locale. This does not break the portability of the application because programmers cannot create locales themselves and the only reference that they can get is from the *Platform* class or the *Locality* class. Guarantees are only provided if the locale on which the execution site is being mapped has enough resources. In case null is passed as a parameter for the locale then it is upto the runtime to allocate the execution site on an appropriate locale. This selection is made on the basis of available resources on a Place and the requirements of each Execution Site.

Class	
public final class Locality	
Methods	
static boolean	**initialize**(long initialHeapSizes, long initialImmortalSizes, RelativeTime defaultbudget[], RelativeTime defaultPeriod[], int numProcessors) Returns true if virtual resources are build.
public static boolean	**isLocalityModelSupported**()

Methods	
	Returns true if the current runtime supports the Locality model and false otherwise.
public static Locale	**getLocale(ExecutionSite site)** Returns the Locale on which the site is mapped onto.
public static Neighbourhood	**getCurrentNeighbourhood**() Returns the Neighbourhood which is hosting this JVM.
public static ExecutionSite	**createExecutionSites**(Locale smp) Returns an ExecutionSite. Takes smp as a parameter which allows the JVM to map the executionSite on that locale. In case of null, the RTJVM can map the execution site on any of the locales.
public static ExecutionSite	**createExecutionSites**(Locale smp,int numProcessors) Returns an ExecutionSite. Takes smp as a parameter which allows the JVM to map the ExecutionSite on that locale. In case of null, the RTJVM can map the execution site on any of the locales. Also accepts the minimum parallelism required by the ExecutionSite as an numProcessors. This factory tries to create an ExecutionSite with a default budget.
public static ExecutionSite	**createExecutionSites**(Locale smp, RelativeTime[] budget, RelativeTime[] period, int numProcessors, MemoryParameters memoryreq) Returns an ExecutionSite. Takes smp as a parameter which allows the JVM to map the executionSite on that locale. In case of null, the RTJVM can map the execution site on any of the locales. Takes budget, period and numProcessors to request a Reservation. Also requests for memoryreq. Creates the ExecutionSite with or without the guarantees.
public static ExecutionSite	**createExecutionSitesIfFeasible**(Locale smp, RelativeTime[] budget, RelativeTime[] period, int numProcessors, MemoryParameters memoryreq) Same as above but it is created only if feasible.
public static ExecutionSite[]	**getExecutionSites**(Locale loc) Returns execution sites in the *Locale* loc.

Methods	
static Place []	**getPlaces**() Returns all the Places hosted on the platform. Used only by the implementation.
static Place	**getPlace**(Locale loc) Returns the Places hosted on the Locale loc. Used only by the implementation.

Table 4.14: The final *Locality* Class

4.2.4 Local ImmortalMemory and Local Heap

Figure 4.3 shows the arrangement of the memory areas in the Locality model, where each Place has a local heap, local immortal memory area and local backing store for the scoped memory areas. Execution sites mapped onto a particular Locale share the local heap and immortal memory areas provided by the place. the MemoryParameters of each ExecutionSite specify its memory requirements. During the initialization of a Place, the initialize method passes the initial memory size of the local heap and local immortal memory area. If required the size of these local memory areas can be renegotiated at runtime. The implementation can enforce a maximum limit on the size of these memory areas.

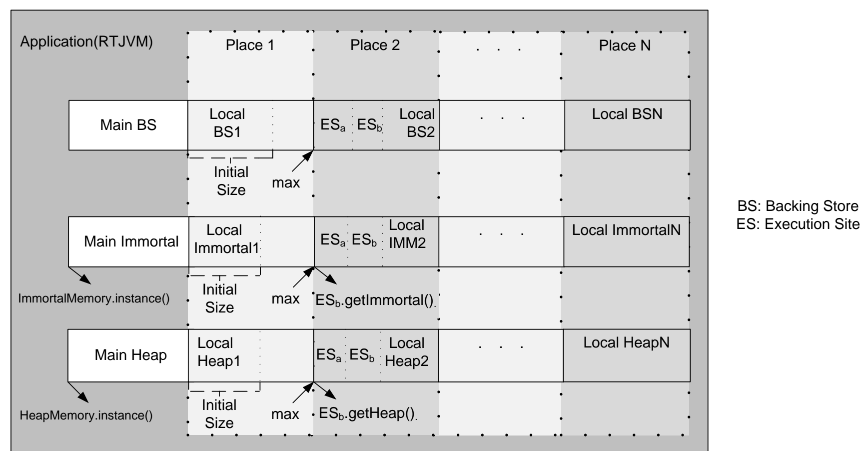


Figure 4.3: Memory Areas in the Locality Model

The problem with this model is that the Heap and Immortal memory area is required to provide local allocation context on different nodes of a NUMA system.

ImmortalMemory and the Heap memory areas are singletons which means only a global memory area exists. For example, the ImmortalMemory area is created during the initialization of the runtime and its reference can be retrieved using the ImmortalMemory.instance() method. Similar is the case with the heap memory area. In order to be able to create local heaps and immortals, the following two strategies can be envisaged:

1. The heap and immortal memory areas are distributed (on all Locales) during the initialization of the JVM.

2. Instances of heap and immortal memory areas are created in addition to default heap and immortal.

The first approach leaves it upto the implementation to provide the local heap and immortal memory area. We, however, have chosen the second approach because of the following two reasons:

1. Local heap and immortals can be referenced individually instead of the global heap.

2. The RTSJ already provides a class to create the physical immortal memory area. This physical immortal memory area extends the singleton immortal physical memory area. There is no such support for a physical heap, however, new architectures with distributed memory systems motivates the need of having an automatically managed memory area which can be physically allocated on a particular memory bank.

In terms of life spans and memory assignment rules in the RTSJ, the heap is very similar to the immortal memory area. The only difference is that garbage collection is enabled for the heap. New multi-core architectures with NUMA memory subsystems motivate the need for a physical heap. The physical heap can be very useful in environments where programmers want to create memory areas on specific nodes and are automatically managed. Therefore, we propose that a physical heap should be added to the RTSJ with similar physical properties to the physical immortal memory area. The following class can be used for the HeapPhysicalMemory

Class
public class HeapPhysicalMemory extends MemoryArea
Constructor
// Constructors similar to the ImmortalPhysicalMemory
Methods
//Methods inherited from the Memory Area class omitted

Table 4.15: The *HeapPhysicalMemory* Class

The main problem of having a physical heap is garbage collection. Garbage collection for real-time Java has been a challenging research area and lies outside the scope of this thesis. A general observation is that because garbage collection is performed on the virtual address space (because it is in user space), therefore we can avoid garbage collections problems by conforming to the following rules:

1. A local heap memory object is created during the initialization of the JVM, when the initialize method of the Place class is called.

2. The execution site is passed a reference of the heap on the locale and no constructors are allowed to create new heap memory areas at runtime.

3. The virtual addresses of the heap are setup as normal.

4. The garbage collector should be aware of the boundaries of the local heaps and respects the boundaries by not moving one local heaps object into another.

These rules are set only in the context of the locality model (they may not be applicable in the case of a general physical heap memory). Irrespective of how the garbage collection is actually implemented, the garbage collector should be compatible with the locality model if it follows this basic set of rules.

4.2.5 Conforming to the RTSJ

In the Locality model, an ExecutionSite, thread/schedulable, and a memory area are created using the factory method. In this subsection, we outline memory assignment rules of the RTSJ, required semantics of the locality model and then implementing these semantics using factory methods.

Memory Area	Heap	Immortal	Scoped
Heap	allowed	allowed	not allowed
Immortal	allowed	allowed	not allowed
Scoped	allowed	allowed	allowed only if reference is to same scope or outer scope

Table 4.16: Memory assignment rules in the RTSJ

4.2.5.1 RTSJ Rules

RTSJ allows more than one memory area where the programmers can allocate objects. However, memory areas behave differently based on their type, e.g. in an immortal memory area, objects are also immortal even if they cannot be referenced anymore. On the other hand, scoped memory areas have a reference count of the number of schedulables using the memory area. The scoped memory area can be reclaimed when the reference count becomes zero even if objects inside it are still being referenced.

In order to provide a consistent memory management model, the RTSJ introduces a set of rules to avoid issues such as dangling pointers. The memory management model in RTSJ enables schedulables to be associated with a MemoryArea by calling its enter() method. When a memory area is entered, it becomes its allocation context and all new allocations are made from the newly entered memory area. Each schedulable object maintains a stack of memory areas it enters which is called its scope stack. The RTSJ defines the following set of rules:

- Accessing Scoped Memory Areas – RTSJ restricts any references from the heap or the immortal memory area to the scoped memory area. In addition, there can be no references to inner scoped memory areas in the case of nested scopes. Table 4.16 shows the complete set of rules. Figure 4.4 shows that *Scope 2* is an inner scoped memory area for *Scope 1* because of their position on the scope stack. Therefore, references are allowed from *Scope 2* to *Scope 1*, however, the opposite are not allowed.

- Single Parent Rule – All scoped memory areas that are being used have only

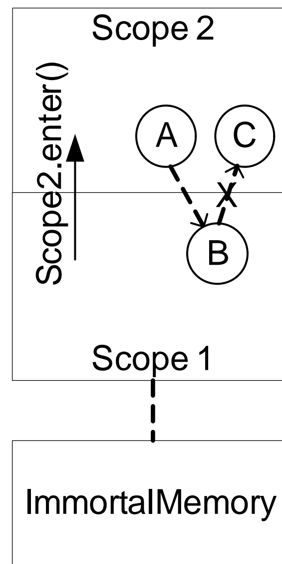


Figure 4.4: Scope stack showing memory access violations

one parent. This parent is either the nearest scoped memory below it on the scope stack or a primodial memory (in case of the outermost scope). The primodial memory area represents the heap, immortal and physical immortal memory areas.

- Heap Access – The NoHeapRealtimeThread instances are not allowed to allocate or reference any object in the heap.

- Scope Inheritance – When a new schedulable is created it inherits the scope from the parent based on rules given in table 4.17.

4.2.5.2 Required Semantics of the Locality Model

The goal of the locality model is to be able to group together tightly coupled threads and objects and allocate them on a particular node. Execution sites, memory areas or threads/schedulables can be created by a schedulable/thread executing in any allocation context. Following are the required semantics:

1. ExecutionSites are placed in the immortal memory area on the required Locale where they are accessible to all schedulables within the application whether they belong to the ExecutionSite or not.

	Current Memory Area	Initial Memory Area	Resulting Stack
1	heap	heap	only heap
2	heap	not heap	heap + initial memory area
3	immortal	immortal	only immortal
4	immortal	not immortal	immortal + initial memory area
5	scopedA	scopedA (or null)	parent's stack upto scopedA
6	scopedA	scopedB	parent's stack upto scopedA + scopedB

Table 4.17: Scope stack inheritance rules in RTSJ

2. Creation of memory areas require that the backing store for the memory area are allocated on the Locale where the ExecutionSite is allocated.

3. Creation of threads/schedulables require that the the affinity should be set and in case the ExecutionSite has any guaranteed reservations, then the schedulables also need to be attached to these reservations. The stack is private to a thread which is created at the time of the creation of the thread. The stack is used quite regularly by a thread therefore it is important that it is also local to the thread.

4.2.5.3 Using the Factory Pattern

The factory method allows the freedom of implement the desired behaviour without changing the existing semantics of RTSJ. Following are the factory methods provided by the Locality model:

1. The factory method used to create execution site checks for the admission control policy. The schedulable object then enters the local immortal memory area of the place where the ExecutionSite will be allocated. All parameters are also copied. The ExecutionSite object is created with or without any resource guarantees and control returns to the original memory area.

2. The factory method used to create memory areas makes sure that the backing store is allocated local to the ExecutionSite. It does not affect the memory assignment rules.

3. The factory method used to create threads/schedulables set the affinities for the threads/schedulables along with any reservations. The stack of the thread is created on the local node when the object of the thread is being created. It is not affected by any of the RTSJ memory assignment rules.

4.2.6 The Default Execution Model

Existing RTSJ applications are supported, but are susceptible to higher latencies caused due to remote accesses. A concurrent Java application contains a number of threads which share a set of objects. There exists a pattern of communication among these threads which determine how closely they are related to each other. On a NUMA system, resources are clustered in cc-NUMA and UMA sub-systems. Each of these groups provide different behaviour of the memory sub-system.

Typically a Java application executes on a set of resources that provide a single address space and cache coherence. In case of a cc-NUMA system, the Java application does not care about the memory access timings if a single address space and a cache coherence is provided. In order to provide backward compatibility with standard Java applications, the following cases can be envisaged for the runtime:

1. The runtime creates a default execution site and places all threads and objects in the default execution site. The execution site can be mapped onto a Place as any other execution site discussed in the locality model.

2. Threads and objects in the application are spread around the cc-NUMA system. For more deterministic timing behaviour, execution sites can be created.

For the case of the Locality model we choose the latter approach because the former limits the parallelism available.

No restrictions have been made on the schedulables to communicate with schedulables in other execution sites, the main reason for this is because the programmers

know that the cost of such an operation will have high overheads, therefore, it is the responsibility of the programmers to avoid such behaviour as much as possible.

4.2.7 Example

An example is presented based on the producer consumer problem showing how the locality model can be used. The objective of which is to create two threads which share an object. One threads writes and the second thread reads the shared object.

The architecture representation is already built before the main method of the Java application is executed. Implementation details of how the architecture is built is presented in Section 5.2.

4.2.7.1 Static Allocation of Execution Sites

The following code shows the creation of two execution sites which are used to create threads/schedulables and memory areas:

```
import javax.realtime.*;

public class ProducerConsumer extends RealtimeThread {

        public void run() {

// The producer and consumer are allocated on two different
// ExecutionSites. Both ExecutionSites are manually allocated
// on different Locales.

// We retrieve the Locales to create the ExecutionSite

        Neighbourhood n1 =
        Locality.getCurrentNeighbourhood();

        System.out.println(n1);
        Locale[] locs = n1.getLocales();

// We create two ExecutionSites on separate Locales

```

```
        ExecutionSite site1 =
        Locality.createExecutionSite(locs[0]);

        ExecutionSite site2 =
        Locality.createExecutionSite(locs[1]);

// We create a Pinnable Memory Area on the first
// ExecutionSite. Then we create the shared object and
// set as a portal.
        long MEMSIZE = 10 * 1024 * 1024;
        final int worksize = 1000;
        int workload = 100;
        final LTPinnableMemory mem =
        site1.createMemoryArea(2, MEMSIZE);
        mem.enter(new Runnable() {
            public void run() {
                mem.pin();
                BufferObject sharedBuffer =
                new BufferObject(worksize);
                mem.setPortal(sharedBuffer);
            }});

// Creating the Producer. The Producer will execute in
// the newly created memory area and will write for
// workload = 100 times in the shared object.
        RealtimeThread pro =
        site1.createRealtimeThread(null, null, null, mem,
                new Producer(mem, workload));

// Creating the Consumer. The Consumer will execute in
// the newly created memory area and will read for
// workload = 100 times in the shared object.
        RealtimeThread con =
        site2.createRealtimeThread(null, null, null, mem,
                new Consumer(mem, workload));
        pro.start();
        con.start();
```

```
    }

// The main function creates a real-time thread because
// a normal Java thread cannot enter a scoped memory area.
// The real-time thread will be created which will setup
// the scoped memory area and create threads in
// ExecutionSites. The created thread will be placed on
// the cc-NUMA by the runtime on any of the processors.

    public static void main(String[] args) {
        ProducerConsumer rt = new ProducerConsumer();
        rt.start();
  }}
```

In the above code, two real-time threads (producer and consumer) are created on two different execution sites as shown in Figure 4.5. The execution sites are allocated on separate Locales. The reference of the Neighbourhood is retrieved where the JVM is hosted. The reference to the Locales are retrieved from the Neighbourhood object. The Locales are passed explicitly in the factory method of the ExecutionSite to force the allocation on specific Locales.

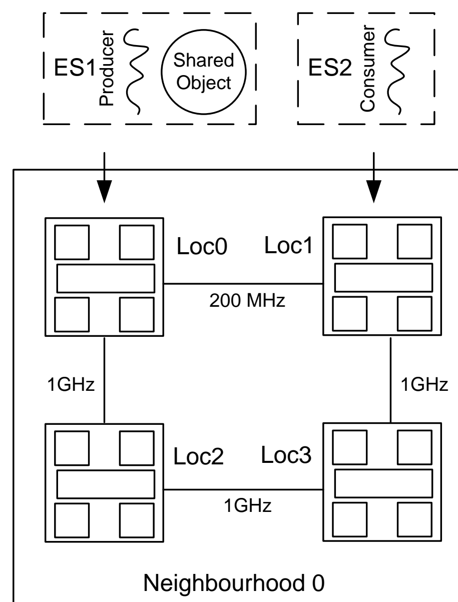


Figure 4.5: Remote allocation of producer/consumer

A scoped memory area *LTPinnableMemory*[1] is created on the first execution site. The shared object is created on this memory area and is set as the portal[2] of the scoped memory area.

Two real-time threads are created using factory methods provided in the ExecutionSite class. The producer thread is created on the first ExecutionSite which is local to the shared object. While the second thread is allocated on the second execution site.

The code shows how threads and objects can be explicitly allocated on particular locations. Such a configuration allows the co-allocation of threads/objects to particular devices or processors. The full code of the program including the producer and consumer classes is provided in Appendix B.1.

4.2.7.2 Implicit Allocation of Execution Sites

Now, the following code shows co-allocation of threads/objects without specifying the physical location:

```
import javax.realtime.*;

public class ProducerConsumer extends RealtimeThread {

        public void run() {

// We create an ExecutionSite which is mapped by the
// runtime.
        ExecutionSite site =
        Locality.createExecutionSite(null);

// We create a Pinnable Memory Area on the
// ExecutionSite. Then we create the shared object and
// set as a portal.
```

[1]In RTSJ, a pinnable scoped memory area is not reclaimed even if all schedulable objects exit the memory area. Normally scoped memory areas are reclaimed once all schedulables exit.

[2]A portal allows an object allocated on a scoped memory area to be shared among schedulables active in the memory area.

```
        long MEMSIZE = 10 * 1024 * 1024;
        final int worksize = 1000;
        int workload = 100;
        final LTPinnableMemory mem =
        site.createMemoryArea(2, MEMSIZE);
        mem.enter(new Runnable() {
            public void run() {
                mem.pin();
                BufferObject sharedBuffer =
                new BufferObject(worksize);
                mem.setPortal(sharedBuffer);
            }});

// Creating the Producer. The Producer will execute in
// the newly created memory area and will write for
// workload = 100 times in the shared object.
        RealtimeThread pro =
        site.createRealtimeThread(null, null, null, mem,
                new Producer(mem, workload));

// Creating the Consumer. The Consumer will execute in
// the newly created memory area and will read for
// workload = 100 times in the shared object.
        RealtimeThread con =
        site.createRealtimeThread(null, null, null, mem,
                new Consumer(mem, workload));

        pro.start();
        con.start();
}

// The main function creates a real-time thread because
// a normal Java thread cannot enter a scoped memory area.
// The real-time thread will be created which will setup
// the scoped memory area and create threads in
// ExecutionSites. The created thread will be placed on
// the cc-NUMA by the runtime on any of the processors.
```

```
    public static void main(String[] args) {

        ProducerConsumer rt = new ProducerConsumer();
        rt.start();
    }
}
```

The code creates one execution site without specifying the Locale. The execution site is allocated by the runtime on any of the available Locales based on the allocation policy[3] that is being used. Both schedulables and shared object are created on the only execution site created as shown in Figure 4.6. This configuration is used when the physical allocation of threads and objects is irrelevant, however, inter thread co-allocation is required. The full code of the program including the producer and consumer classes is provided in Appendix B.2.

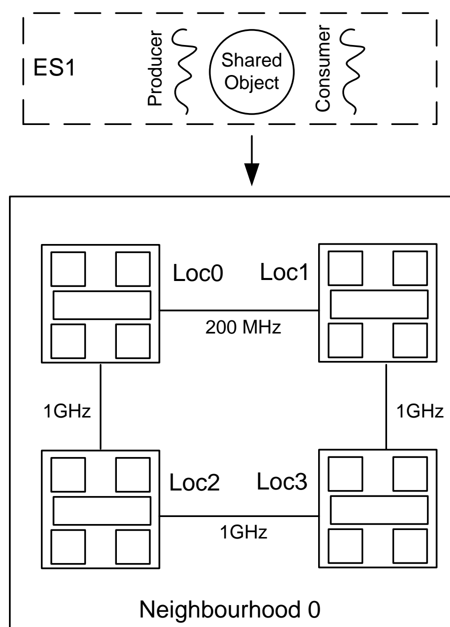


Figure 4.6: Local allocation of producer/consumer

[3]A simple allocation policy is used which is based on the available resources on a *Place*. The objective is not to find the optimal allocation policy rather providing a mechanism where different policies can be used.

4.3 Resource Reservations

The locality model assumes an open system where multiple applications are sharing the same resources. Real-time Java applications based on the locality model consist of a number of execution sites which require guarantees from the system for resource usage. An execution site(ES) is a collection of real-time threads mapped on a cc-NUMA sub-system which has been designed to run on multiple processors at the same time. Each ES has temporal and memory requirements, which the RTJVM should be able to provide. The temporal guarantees for the execution sites should allow it to execute over multiple processors, however, they are restricted to a single locale. In this section, the following four mechanisms will be presented for the locality model:

1. Interface – An interface for the application to specify its requirements.
2. Admission Control – Admission control for the applications based on the evaluation of their requirements.
3. Scheduling – A scheduling policy which provides temporal guarantees.
4. Cost Enforcement – Measuring consumed computation time and ensuring applications do not over run the reservation.

[Mercer et al., 1994] outlines the above mechanisms as a basic requirement for manage resource reservations. Based on these requirements the interface will represent the temporal and memory requirements of an Execution Site. It allows an execution site to request the runtime guaranteed resources. The admission control mechanism allows the runtime to decide if an Execution Site can be guaranteed the required resources. The mechanism should ensure that admitting the execution site should not affect any other guarantees provided by system. Once the execution site is admitted it is up to the scheduler to make sure that the execution site receives the resources that were guaranteed. However, if an execution site tries to overruns the guaranteed resources, a bounding mechanism needs to be in place which will make sure that all overruns are bounded. This will provide isolation to Execution

sites enabling programmers to analyze their execution sites in isolation to the rest of the system.

Details on these mechanisms for the Locality model are discussed in the rest of the section.

4.3.1 Interface

The interface of an execution site describes the computational resources it requires. The requirements of an execution site can be divided as implicit or explicit requirements. The implicit requirements for an execution site include that the execution site can only be mapped onto a locale and use shared caches whenever available. The explicit computational requirements include temporal and memory requirements of the execution site. The explicit requirements are passed as an interface to the runtime to request resources for the Execution Site.

The memory requirements of the Execution Site can be represented by the MemoryParameters class. This class is part of the RTSJ specification and is used both for the purposes of admission control by the scheduler and for the purposes of pacing the garbage collector. Similarly, in the case of an Execution Site, this class can be used for describing the memory requirements and enforcing the limits on memory allocation when using the *new* operation. When a schedulable object exceeds its allocation or allocation rate limit, the error is handled as if the allocation failed because of insufficient memory. The object allocation throws an OutOfMemoryError.

In the case of the Locality model, a partitioned budget is proposed as shown in Figure 4.8. Here the budget is reserved on a per processor basis instead of making it globally available on all processors. The rationale for this is as follows:

The resource reservation model that was presented in [Wellings et al., 2009] used an interface represented by the tuple: {C, T, minPro, maxPro} where C is the minimum available budget within a maximum period, T. The other parameters are minPro and maxPro representing the minimum and maximum number of processors required. This model was based on the assumption that the underlying hardware will be an SMP model. Therefore the budget, C, will be available globally on all processors as shown in Figure 4.7.

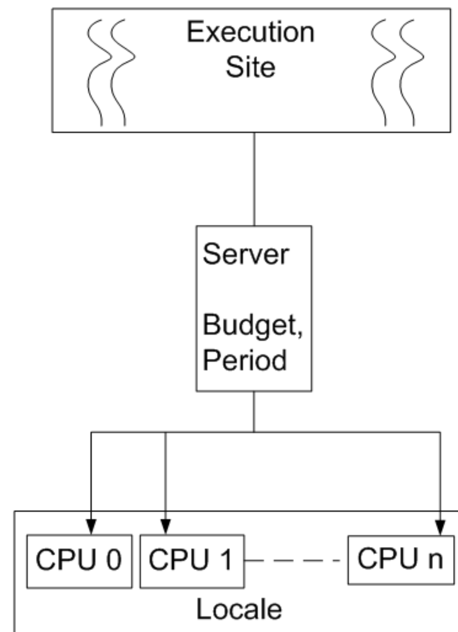


Figure 4.7: Global Budget

In the case of the Locality model, a partitioned budget is proposed as shown in Figure 4.8. Here the budget is reserved on a per processor basis instead of making it globally available on all processors. The rationale for this is as follows:

- Processors in cc-NUMA are arranged in clusters and Execution Sites require a Reservation which is limited to a UMA node instead of the whole cc-NUMA system. Therefore a partitioned approach is needed either on every processor or every node.

- Implementing a cost accounting strategy is very difficult and not very scalable because a global budget requires sharing and synchronization of variables by a large number of threads using the budget. The shared variables will be updated very frequently by all threads executing in parallel. Race conditions can occur unless a lock is provided for mutual exclusion. Such an approach is not feasible for large number of threads.

As a result a partitioned approach has been taken to provide budgets on individual processors. The interface that will be used by the Execution Site is represented by the tuple: $\{\Theta\ [] ,T\ [] , m\}$. Where m is the number of processors and Θ is

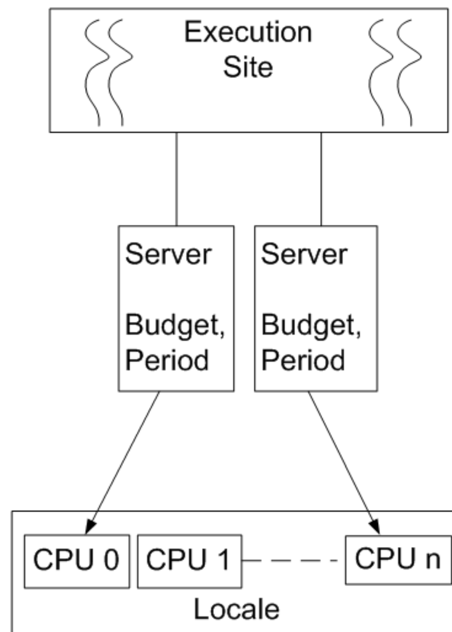


Figure 4.8: Partitioned Budget

the budget on a single processor with period T. The following describes the new reservation model based on partitioned reservations:

- The application has an external contract with the OS which can be broken down into a number of cluster contracts where budget is specified on each processor. A cluster contract is a set of processor budgets provided on an UMA sub-system.

- When an execution site requests for a reservation, it is allocated a partitioned reservation on a set of processors each represented by a PartitionedParameter. The PartitionedParameter is a tuple of {int ProcessorIndex, RelativeTime Budget, RelativeTime Period}.

- The scheduler should be NUMA aware. It schedules Threads and Schedulables globally on the processors where the respective execution site has some budget available.

Class	
public class PartitionedParameters	
Constructor	
public	**PartitionedParameters**(int processor, RelativeTime budget, RelativeTime period)
Methods	
public int	**getProcessorId**() Returns the processor index.
public RelativeTime	**getBudget**() Returns the budget guaranteed on the processor.
public RelativeTime	**getPeriod**() Returns the period on the processor.

Table 4.18: The *PartitionedParameters* Class

4.3.1.1 External Contract

The External Contract provides an interface to the scheduler to manage the contracts on all the Locales. The following provides the ExternalContract class.

Class	
public final class ExternalContract	
Methods	
public static Cluster-Contract	**createClusterContract**(RelativeTime [] budget,RelativeTime [] period, int numProcessors, Locale loc) Creates a new Cluster Contract
public static Cluster-Contract	**getClusterContract**(Locale loc) Returns the cluster contract on a locale
public static boolean	**updateClusterContract**(ClusterContract contract, Locale loc) Updates a cluster contract, returns true if successful and false if not.
public static boolean	**deleteClusterContract**(ClusterContract contract) Deletes a cluster contract and returns true if successful and false if not.

Table 4.19: The *ExternalContract* Class

A ClusterContract is a partitioned budget provided to an application on a particular Locale. It is provided in the following table:

Class	
public class ClusterContract	
Constructor	
ClusterContract(RelativeTime [] budget, RelativeTime [] period, int numProcessors, Locale local)	
Methods	
PartitionedParameters []	**createContract**() Requests the OS to create a Contract on a particular SMP node.
boolean	**setPeriod**(RelativeTime Period, PartitionedParameters res) Changes the period on a particular processor.
boolean	**setBudget**(RelativeTime setBudget, PartitionedParameters res) Changes the budget on a particular processor.
Locale	**getLocale**(RelativeTime Period, PartitionedParameters res) Returns the Locale of the Cluster.

Table 4.20: The *ClusterContract* Class

4.3.1.2 Partitioned Reservations

Each ExecutionSite is allocated a PartitionedReservation. All threads in an ExecutionSite are attached to the reservation. The PartitionedReservation class is given in the following table:

Class	
public class PartitionedReservation	
Constructor	
public **PartitionedReservation**(RelativeTime [] maxPeriod, RelativeTime [] minBudget, int numProcessors)	
Methods	
public PartitionedParameters []	**getPartitionedParameters**()

Methods	
	Returns the parameters which specify how the reservation has been guaranteed. If null, the reservation has not been guaranteed by the scheduler.
public PartitionedParameters []	**createReservation**() Requests the scheduler to create a reservation on a processor with these parameters. No guarantees will be given by the scheduler. If returns null, then it is not feasible. Otherwise returns the affinity set of a processor.
public PartitionedParameters []	**createReservationIfFeasible**() Creates the reservation only if it is guaranteed.
public boolean	**setPeriod**(RelativeTime Period, PartitionedParameters res) Changes the period on a particular processor.
public boolean	**setBudget**(RelativeTime setBudget, PartitionedParameters res) Changes the budget on a particular processor.
public PartitionedParameters []	**changeIfFeasible**() Requests the scheduler to change the reservation only if feasible.
public boolean	**ifFeasible**() Checks if all the schedulables attached to this ExecutionSite are feasible.
public boolean	**addToFeasibility**(Schedulable schedulable) Informs the scheduler that the resource demands of the schedulable should be considered for feasibility analysis.
public boolean	**removeFromFeasibility**(Schedulable schedulable) Removes the schedulable from the feasibility analysis of the reservation.

Table 4.21: The *PartitionedReservation* Class

4.3.2 Scheduling: The *ReservationScheduler* Class

On a cc-NUMA system, a global reservation model cannot be used because the Execution Site requires a Reservation on an SMP cluster and not the whole cc-NUMA system. Therefore, a partitioned approach is used to provide Reservations where CPU time is reserved on every processor. The scheduler creates a PartitionedReservation for the Execution Site which contains $(\Theta_i, T_i)\ \forall i = 1....m$, where m is the number of processors. In order to manage the PartitionedReservation, a ReservationScheduler class has been provided.

The ReservationScheduler extends the scheduler class and implements the methods provided in the scheduler class. In addition, it also adds a number of methods to add the Reservation to the feasibility analysis. It returns an array of PartitionedParameters where each pair of (budget, period) is associated with a particular processor.

Class	
public class ReservationScheduler extends Scheduler	
Methods	
PartitionedParameters []	**addToFeasibility**(PartitionedReservation reserve, Locale loc) Adds the Execution Site to the feasibility analysis and allocates a reservation with PartitionedParameters[] to the execution site
PartitionedParameters []	**addIfFeasible**(PartitionedReservation reserve, Locale loc) Only adds a reservation if it is feasible
PartitionedParameters []	**changeIfFeasibile**(PartitionedReservation reserve, Locale loc) Changes the reservation only if it is feasible
PartitionedReservation []	**getPredefinedReservations** (Locale loc) returns a set of pre-defined reservations.
Other methods of the *Scheduler* class omitted	

Table 4.22: The *ReservationScheduler* Class

4.3.3 Admission Control

The admission control policy determines if a new ExecutionSite can be created on a Locale or not. When an ExecutionSite is created, the following scenarios exist:

- ExecutionSite with no guarantees – The ExecutionSite does not require any guarantees and only wants the threads to be grouped together. The *createExecutionSites (Locale smp)* factory method is used where the Locale smp may or may not be defined. If the Locale is defined, then an ExecutionSite is created on that Locale. Otherwise the ExecutionSite will be placed on any Locale with the most resources available. In any case the ExecutionSite is not refused admission because no guarantees are needed.

- ExecutionSite with default guarantees – The *createExecutionSites (Locale smp, int numProcessors)* requires default guarantees from the Locale smp. If smp is defined, an execution site created with the default or the minimum available budget values. An ExecutionSite is created even if no guarantees are provided. If smp is not defined, the ExecutionSite is placed on a Locale with the maximum available resources.

- ExecutionSite creation irrespective of guarantees –The *createExecutionSites (Locale smp, RelativeTime[] budget, RelativeTime[] period, int numProcessors, MemoryParameters memoryreq)* requires a reservation with $\{budget_i, period_i\}$ for numProcessors along with memreq memory requirements. If smp is defined, then the executionsite is always created on that Locale but if smp does not have the available resources than no guarantees are provided to the Execution Site. If smp is not defined, then the execution site is created on the Locale which is the locale with the maximum amount of resources available which can guarantee the required guarantees.

- ExecutionSite creation only if guaranteed – The *createExecutionSitesIfFeasible (Locale smp, RelativeTime[] budget, RelativeTime[] period, int numProcessors, MemoryParameters memoryreq)* requires a reservation with $\{budget_i, period_i\}$ for numProcessors along with memreq memory requirements. If smp is defined,

then the executionsite is always created on that Locale but if smp does not have the available resources than no execution site is created. If smp is not defined, then the execution site is created on the Locale which is the locale with the maximum amount of resources available which can guarantee the required guarantees. If no Locale can provide the desired resources, then execution site cannot be created.

In terms of admission control for the threads to be created. There are three types of methods that can be used to create Java threads and schedulables on an execution site. These methods are discussed below:

1. createJavaThread (Runnable logic) – This factory creates a Java thread on an execution site. It sets the affinity of the thread to the affinity of the ExecutionSite.

2. createRealtimeThread (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic) – Creates a real-time thread with the affinity set of the execution site. It attaches the thread to the reservation of the ExecutionSite.

   Another method, *createRealtimeThreadIfFeasible*, with the same parameters is provided which checks the feasibility before creating the real-time thread. If the thread is feasible within the budget of the reservation, then it is created and attached to the feasibility analysis.

3. createNoHeapRealtimeThread (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic) – Creates a no heap real-time thread on the execution site and the affinity is set to the affinity of the execution site.

   The *createNoHeapRealtimeThreadIfFeasible* method can be used to perform an online feasibility analysis before creating this thread. If the thread is feasible within the budget of the reservation, then it is created and attached to the reservation.

Different admission control and feasibility analysis policies can be implemented. An example of such the admission control policy and feasibility analysis has been presented in Appendix D.

4.3.4 Cost Enforcement

All execution sites have been guaranteed a budget and they are temporally isolated from each other. In order for the temporal isolation to hold, it is very necessary that the executions sites do not over-run their budget. Therefore, a cost accounting and enforcement mechanism should be in place to ensure execution sites do not over-run their budget.

For each ExecutionSite, servers are created for each processor on which the execution site has been guaranteed the budget. For each partitioned parameter, an execution time server can be represented by the tuple *Budget, Period, Affinity, Priority.* The server cannot execute on more than one processor, therefore, the cardinality of the AffinitySet should be 1. The following shows an execution time server in the ReservationServer class.

Class
class ReservationServer
Methods
ReservationServer(RelativeTime C, RelativeTime T, AffinitySet pro, SchedulingParameters priority)

Table 4.23: The *ReservationServer* Class

4.3.5 Discussion

The Scheduler needs to provide temporal guarantees to the execution site. These guarantees can be provided to an execution site by using a two-level preemptive fixed priority hierarchical scheduling in the locality model. According to this model, each execution site has a local scheduler to schedule all enclosed threads. The hierarchical scheduler provides temporal isolation and reduces the the complexity of the schedulability analysis in the case of a shared platform. Unfortunately, the RTSJ does not support hierarchical scheduling although a number of proposals have

been made [Zerzelidis and Wellings, 2010]. [Wellings et al., 2009] have proposed the following semantics change for the RTSJ:

"Schedulable objects assigned to the same reservation must run in priority order. However, there is no assumption that schedulable objects assigned to different reservations run in priority order.".

This reservations then in effect provide hierarchical scheduling if a reservation has been guaranteed by the scheduler. Therefore, building on this model we assume that hierarchical scheduling is providing reservations that can be guaranteed by the scheduler.

The producer/consumer example presented in Section 4.2.7.2 has been extended to provide guarantees to the ExecutionSite where producer and consumer are created. The ExecutionSite *site* requests some temporal guarantees. Assuming the runtime creates a reservation for *site*, we create the producer and consumer threads using the factory of *site* which attaches them to the reservation allowing them to use the guaranteed budget. The following code snippet shows the creation of an execution site which requests a partitioned reservation:

```
import javax.realtime.*;

public class ProducerConsumer extends RealtimeThread {

        public void run() {

// In order to specify the requirements of the Execution Site,
// we create the budget and the period arrays.
        int numProcessors = 2;
        RelativeTime [] budget = new RelativeTime [numProcessors];
        RelativeTime [] period = new RelativeTime [numProcessors];
        budget[0]= new RelativeTime(500,0);
        period[0]= new RelativeTime(1000,0);
        budget[1]= new RelativeTime(500,0);
        period[1]= new RelativeTime(1000,0);

// We create an ExecutionSite which is mapped by the
// runtime. The mapping is based on the required budget
```

```
// parameters.

   ExecutionSite site = Locality.createExecutionSite(null,
      budget, period, numProcessors, null);

// We create a Pinnable Memory Area on the ExecutionSite.
// Then we create the shared object and set as a portal.
    long MEMSIZE = 10 * 1024 * 1024;
    final int buffersize = 1000;
    int workload = 1000;
    final LTPinnableMemory mem =
      site.createMemoryArea(2, MEMSIZE);
        mem.enter(new Runnable() {
                public void run() {
                        mem.pin();
                        BufferObject sharedBuffer =
            new BufferObject(buffersize);
                        mem.setPortal(sharedBuffer);
                }});

// Creating periodic parameters.

  RelativeTime start = new RelativeTime(0, 0);
  RelativeTime C = new RelativeTime(200, 0);
  RelativeTime D = new RelativeTime(1000, 0);
  RelativeTime T = new RelativeTime(1000, 0);
  PeriodicParameters releaseParams = new
    PeriodicParameters(start,T,C,D,null,null);

// Creating the Producer. The Producer will execute in
// the newly created memory area and will write for
// workload = 1000 times in every period. The thread
// will be added to the reservation and will use the
// guarantees provided to the reservation. In the case
// the budget is exhausted, Producer will have to wait
// for the replenishment of the budget.
   RealtimeThread pro = site.createRealtimeThread(null,
```

```
    releaseParams, null, mem, new Producer(mem, workload));

// Creating the Consumer with the same timing properties.
   RealtimeThread con = site.createRealtimeThread(null,
     releaseParams, null, mem, new Consumer(mem, workload));

        pro.start();
        con.start();
}

// The main function creates a real-time thread because
// a normal Java thread cannot enter a scoped memory area.
// The real-time thread will be created which will setup
// the scoped memory area and create threads in
// ExecutionSites. The created thread will be placed on
// the cc-NUMA by the runtime on any of the processors.

public static void main(String[] args) {
                ProducerConsumer rt = new ProducerConsumer();
                rt.start();
        }}
```

Details of how the partitioned reservations are implemented are presented in Chapter 5 and the complete example including the producer and the consumer classes have been attached in Appendix B.3.

4.4 Summary

This chapter started by presenting a set of requirements to enable developers to design and deploy portable real-time Java applications. In response to these requirements, we summarize the Locality model presented in this chapter to check if the requirements been met.

The locality model provides a number of new abstractions which are aimed to provide locality and visibility into the architecture of the system. The following support has been provided for locality:

- In response to R1 – The architectural model represents the basic components of the NUMA system and describes its topology.
- In response to R2 – The application model enables programmers to partition the application into ExecutionSites which can be allocated onto a virtual set of resources represented by the Place.
- In response to R3 – Each ExecutionSite is provided a PartitionedReservation on a Place which guarantees resources to the ExecutionSite and provides temporal isolation to the Schedulables attaches to it.
- In response to R4 – All devices (including memory and processors) that are present on the system are represented. The references to all such devices can be retrieved from the Platform class. The Locale containing the device can be retrieved by device.getLocale(). Now an execution site can be created on this Locale along with threads and memory banks.
- In response to R5 – The ExecutionSite can be allocated dynamically by the runtime.
- In response to R6 – Implementation of the physical memory type filter is an implementation issue and will be addressed in Section 5.1.2.

Figure 4.9 shows the locality model where each Place has a ClusterContract on a Locale. The combination of all the ClusterContracts form an ExternalContract which is a contract between the OS and the application. The following chapter will discuss the implementation details of the Locality model.

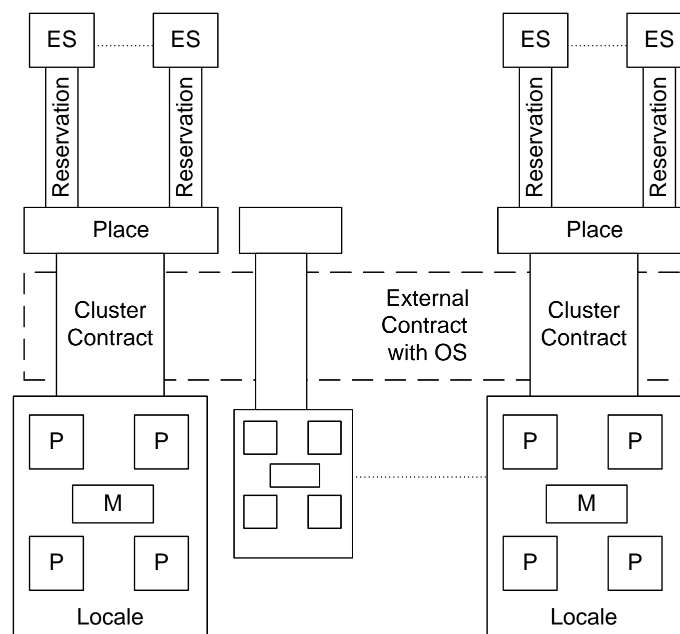


Figure 4.9: The Locality Model

Chapter 5

Implementation

The previous chapter introduced a locality model for the real-time specification for Java. The locality model integrates practices from high performance computing and the real-time systems community to support cc-NUMA architectures in the real-time specification for Java (RTSJ). It relies on programmer's knowledge of the application and allows the programmer to group together related objects and threads. Each group is then reserved resources based on the requirements of the groups.

This chapter discusses a prototype implementation of the locality model which extends an existing open-source RTSJ implementation to incorporate the features introduced by the locality model. In this chapter we will discuss the following:

- Implementation of the Architectural Model
- Implementation of the Application Model
- Implementation of the Resource Reservations Model

All of the above models have been presented in the previous chapter. Section 5.1 provides an overview of the implementation model and discusses the tools that have been used. Section 5.2 then discusses the discovery of the basic hardware components and creating a representation of the NUMA system. Section 5.3 discusses the creation of a virtual platform on the NUMA system, the creation of ExecutionSites, creation of threads and the creation of memory areas. Section 5.4 discusses how resource reservations are allocated. Section 5.5 is the last section which provides a summary of the whole chapter.

5.1 Implementation Overview

The locality model has been implemented using JRate [Corsaro and Schmidt, 2002] as the basic platform. It is fundamentally an extension of the GCJ compiler and libgcj runtime library which also supports native threads required for true parallelism, however, JRate only works with an older version of GCJ (such as GCJ-3.3.x) and has not been ported to more recent versions. JRate was selected because it was the only open source RTJVM which supported true parallelism using native threads.

JRate can be used under two different environments: Linux and Marte OS [Rivas and Gonzlez Harbour, 2001] . While Marte OS provides more hard real-time capabilities, Linux is selected for the support it can provide on a cc-NUMA system.

Following are some of the features of JRate/Linux:

- JRate supports native threads. It uses the native POSIX threading library (NPTL). Each Java thread corresponds to a pthread.

- JRate is compliant with the RTSJ 1.02 and does not support the AffinitySet class. No other support is provided in GCJ which would allow one to bind threads to processors.

- JRate has very limited support for the Physical Memory Areas. It uses the */dev/mem* file to access the physical address space. However, this file can only provide access to the I/O space and access to the physical memory on x86/x64 on Linux cannot be provided.

- Typically Linux requires 64bit in order to support cc-NUMA systems. JRate does not directly build on a 64bit (tested using AMD Opteron) multiprocessor. A number of changes are required to the source and script files build it on such systems which have been listed in Appendix C.

In order to provide locality, it is necessary for the runtime to be able to allocate threads and memory on particular nodes. Therefore the following changes needs to be supported before implementing the locality model:

5.1.1 Implementing the AffinitySet Class

The AffinitySet class has been implemented in JRate. The AffinitySet class is based on the following Linux specific function:

```
sched_setaffinity(pid_t pid, size_t cpusetsize,
                  cpu_set_t *mask)
```

The mask structure contains the processors on which the affinity will be set for the task *pid (or tid)* which is the task identification number on Linux. A number of operations can be performed on the mask to set processor e.g. CPU_ZERO, CPU_SET, CPU_CLR, CPU_ISSET and CPU_COUNT[1]. The sched_setaffinity binds the thread to processors, the scheduler then respects the affinity of a thread by scheduling the threads only on the set of processors which are part of the thread affinity. Other OSs have similar support which can be used e.g. Solaris has Processor_bind.

The AffinitySet class allows the implementation of different scheduling policies on multiprocessors, from fully partitioned to global scheduling. In the JRate/Linux environment,each processor is considered to have a separate run-queue. On each processor the thread with the highest priority is elected for execution. In case of real-time threads,

Figure 5.1 shows a simple experiment showing response times of 64 real-time threads scheduled based on the *push*, *pull mechanism* on the Linux 2.6.27. The experiment is done on a 16-processor system. Threads are created in an increasing priority order and threads are essentially allocated to Linux run-queues. However, still the highest priority tasks complete before lower priority tasks effectively simulating the global scheduling policy on per processor run-queues in Linux.

The AffinitySet class allows the programmers to pin task to particular processors. The thread in such a case will not migrate to any processor outside the affinity set of the thread and the migration respects it's affinities.

[1] for more information http://linux.die.net/man/3/cpu_set

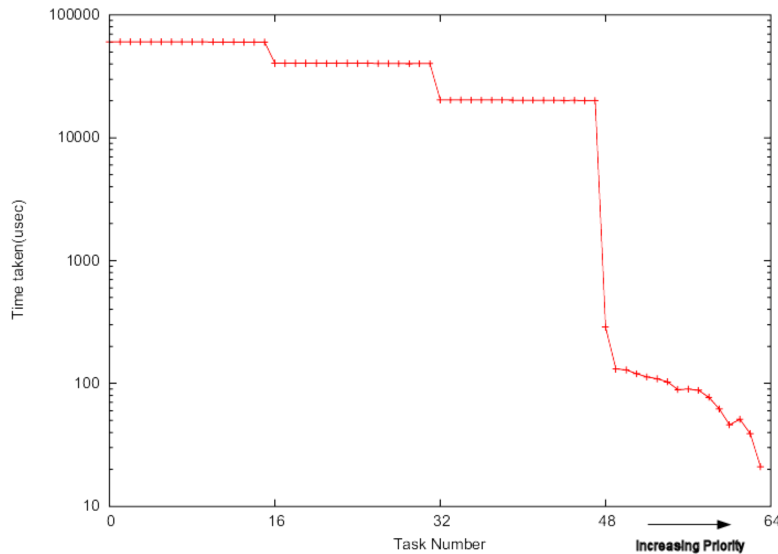


Figure 5.1: Simulating global scheduling in Linux

5.1.2 Access to the Local Memory

As has been explained in Chapter 3, with the physical memory model, it is not safe to use actual physical addresses for the system RAM. The main requirement we have is to be able to allocate memory on a particular node without posing any danger to the stability of the system.

JRate uses the */dev/mem* file with the *mmap(...)* to access the physical address space. The */dev/mem* is limited to provide only provide access to the I/O space on most systems including the x86/x64 architecture.

Our goal is to be able to create a memory area on a specific node and lock it there so that it does not move from there in case of swapping. The following class shows a LocalMemory (in response to requirement R6 presented in Chapter 4) class which implements the PhysicalMemoryFilter and provides access to the memory on a UMA sub-system (Locale):

Class	
public class LocalMemory implements PhysicalMemoryTypeFilter	
Constructor	
public LocalMemory (Locale loc)	
Methods	
public Locale	**getLocale**() Returns the Locale of this.
long	**map**(long size) Allocates a memory area and returns the virtual address.
boolean	**unmap**(long address) Frees the memory and returns true if successful.
//Methods from the PhysicalMemoryTypeFilter Omitted	

Table 5.1: The *LocalMemory* Class

The map(...) method allocates on a particular node and then locks the memory there. The following steps are defined to implement this method:

1. Allocating on a Particular Node – This can be done using different OS libraries e.g. the NUMA API [Kleen, 2004] or the hwloc [Broquedis et al., 2010] library. The NUMA API provides the following function to allocate on a particular node:

   ```
   numa_alloc_onnode(MEMSIZE_IN_BYTES, nodeNUM);
   ```

   This function first tries to allocate memory on a particular node, however, if it cannot allocate memory on that node then it falls back on other nodes. This can however be disabled using numa_set_strict(.)[2] which causes the function to fail if the memory is not available. This function replaces the *malloc* to allocate on a particular node.

   Therefore, the memory type filter for a node memory in RTSJ uses the NUMA API to allocate memory on the node instead of using the physical addresses. Once we get a virtual address we can register the memory with the physical memory manager. The node memory that have been registered with the physical memory manager will not have physical address but rather an associated

[2]http://linux.die.net/man/3/numa_set_strict

type. Strictly speaking, the node memory is not a physical memory area as the memory is not accessed directly and no physical addresses are attached with it, however, since it does represent distinct memory characteristic that can be used just by specifying it's type, which is why we use the physical memory filter for accessing it.

2. Filling in the Memory – All numa_on_alloc(...) (or malloc(...)) returns is a pointer in the virtual address space. No physical memory is allocated to them initially. In order to guarantee that the created memory is actually being backed by physical memory, the memset(...) function is used.

3. Locking the Memory – Figure 5.2[3] shows the latencies of memory allocation in an LTMemory area when we keep on allocating objects. The latencies suddenly increase when the system runs out of physical memory and existing pages have to be swapped out in order to accommodate new pages in the virtual address space. From the real-time perspective, this behaviour is completely unacceptable, especially on a NUMA system, where the problem is exacerbated because a page allocated on one node is swapped back in on another node which the real-time application is unaware of.

   Therefore, we lock the memory using the *mlock()* after it has been allocated.

5.1.3 Extensions Required for the Locality Model

In order to implement the locality model, the following tools can be used:

- Hwloc – Based on the existing support in operating systems, a portable framework has been developed by the MPI community to introduce portable interfaces for obtaining this information [Broquedis et al., 2010]. The hwloc framework gathers information about processors, caches, memory, nodes in a NUMA system etc. on different operating systems such as Linux, Windows, Solaris,

[3]The values have been measured for Timesys Reference Implementation of RTSJ running on a single processor system with 512 MB of RAM.

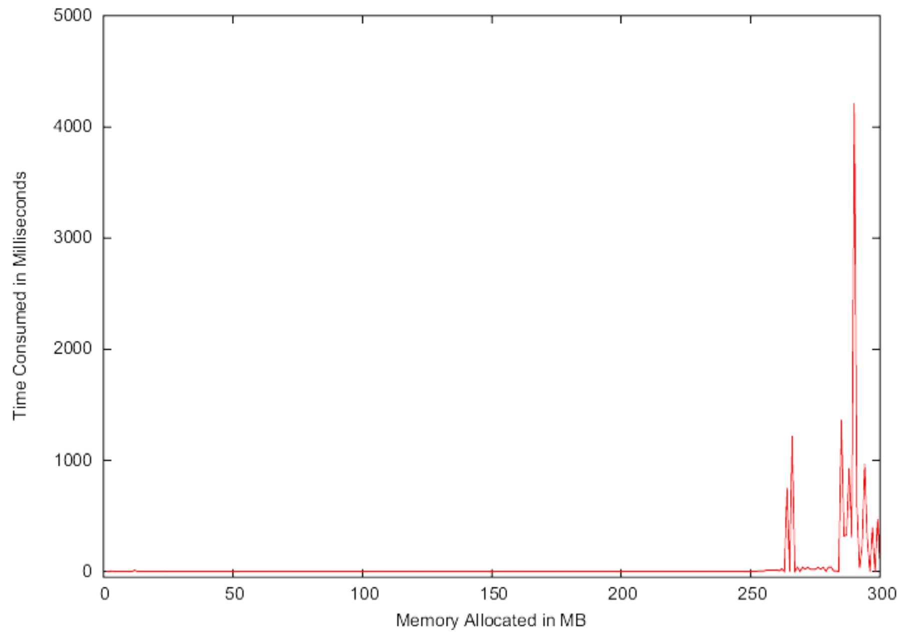


Figure 5.2: Memory access timings

AIX, Darwin etc. and provides a high level abstraction to these resources at the application level.

- Control Groups – Control group (cgroup) is a file system, where tasks can be added to it and then be arranged hierarchically. By default all tasks belong to the root cgroup but they can be added to any CGroup. However, the task can only exist in only cgroup at any particular time. Each resource (CPU, memory etc.) has an associated controller which limits resources to tasks based on the cgroup they belong to. Building the mainline kernel by enabling the CONFIG_RT_GROUP_SCHED option, provides the support for real-time group scheduling. The real-time group scheduling can be used along with the cgroup filesystem to provide group budget for a set of tasks(process/ threads). The real time group scheduling allows explicit allocation of CPU bandwidth to task groups[4]. An API is provided in the libcgroup[5] which can be used to perform basic operations on CGroups.

The rest of the chapter describes the implementation in detail.

[4]http://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

[5]http://libcg.sourceforge.net/

5.2 Implementing the Architecture Representation

This section discusses the implementation of the Architectural representation in the Locality model. In order to represent the architecture, the real-time JVM must have the following capabilities:

1. Discovering the underlying architecture components (processors, memory and devices) and map the logical abstractions to the physical components.
2. Building the NUMA hierarchy in terms of Locales, Neighbourhoods and District and then providing the Platform interface.

It is assumed that the architecture of the system remains static throughout the lifetime of the application. This is done during the initialization of the JVM before the main method of the Java application is started. Figure 5.3 shows a sequence diagram to build the architecture which shows the following steps:

1. The *buildPlatform(Memory[] mem, Device[] dev)* is called in the Platform class. The memory and devices that have been registered with the PhysicalMemoryManager and the devices which can be used to access RawMemory are passed. Devices cannot be discovered using the hwloc library. Instead only special devices that have been registered with the RTSJ Raw memory factory can be used.
2. The *initializeTopology()* is called to initialize system topology. It further calls a native method, nativeInitializeTopology() which initializes the topology in the hwloc library. The topology in hwloc is arranged in a hierarchical structure which is represented by the *hwloc_topology_t* structure. Once the topology is initialized, the topology structure can be queried for information on the topology.

```
hwloc_topology_t topology;
```

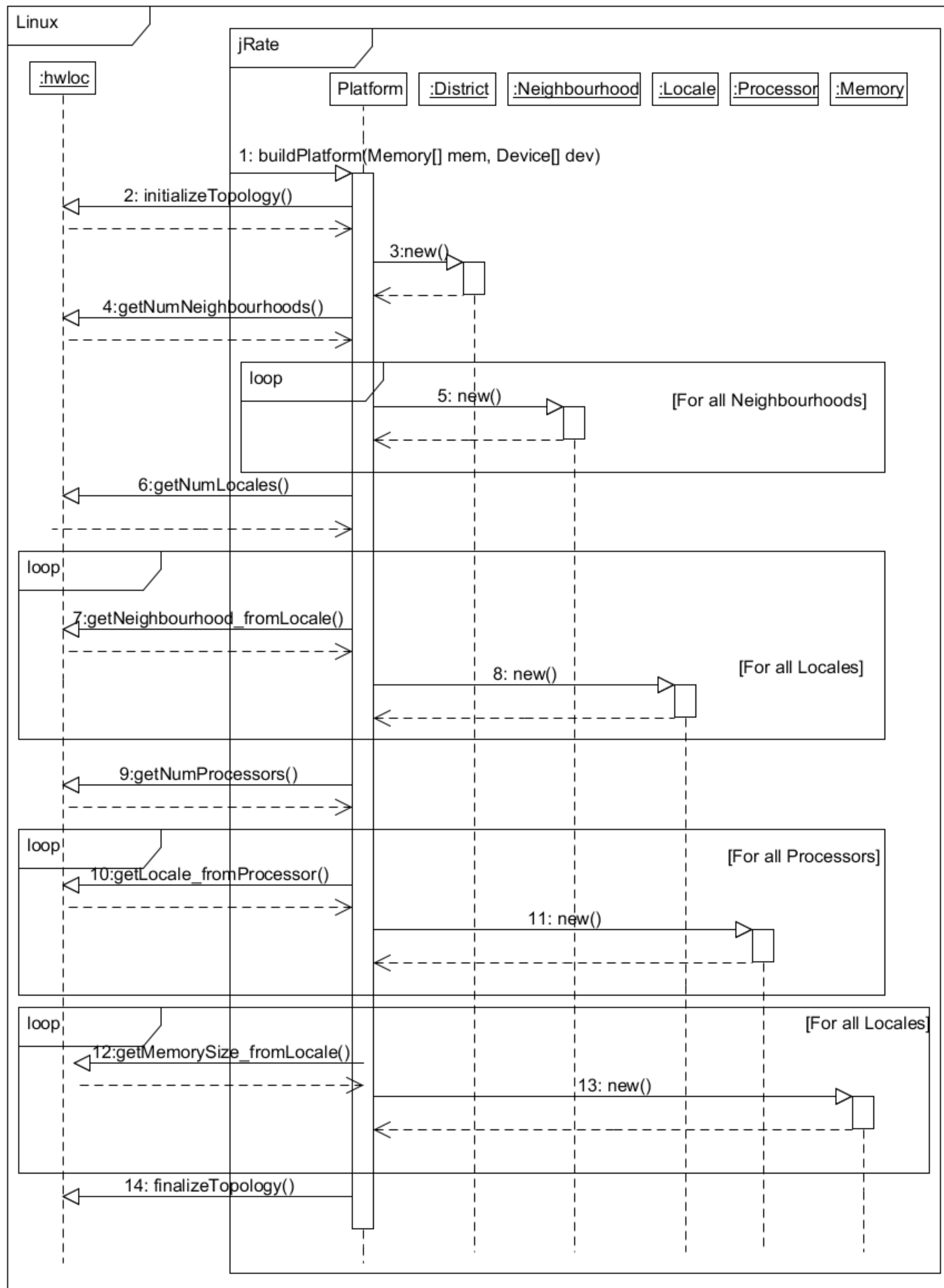


Figure 5.3: Sequence diagram: building the Architectural Representation

```
void javax::realtime::Platform::nativeInitializeTopology()
{
 hwloc_topology_init(&topology);
 hwloc_topology_load(topology)
}
```

A graphical output from the hwloc is shown in the Figure 5.4 for the architecture given in Section 3.2.

3. The singleton district object is created.

4. The native method *getNumNeighbourhoods()* is used to retrieve the number of cc-NUMA sub-systems in the district. In hwloc, a Neighbourhood corresponds to the *HWLOC_OBJ_MACHINE* object which is a set of processor and memory with cache coherence. The following code shows this native method:

```
jint javax::realtime::Platform::getNumNeighbourhoods()
{
return hwloc_get_nbobjs_by_type(topology, HWLOC_OBJ_MACHINE);
}
```

5. We create an instance of Neighbourhood for each cc-NUMA sub-system.

6. The native method, *getNumLocales()*, is called for the number of Locales in the system. In hwloc, a Locale is represented by the *HWLOC_OBJ_NODE* object. The indices of the array correspond to the Id of the locale used by Linux normally from *0 to numLoc -1.* The following code shows this native method:

```
jint javax::realtime::Platform::getNumLocales()
{
return hwloc_get_nbobjs_by_type(topology, HWLOC_OBJ_NODE);
}
```

7. The native method, *getNeighbourhood_fromLocale(...)*, is called to retrieve the Neighbourhood to which this Locale belongs to. The following C++ code returns the index of the Neighbourhood this native method:

```
jint javax::realtime::Platform::getNeighbourhood_fromLocale
  (jint localeNum)
{
hwloc_obj_t obj = hwloc_get_obj_by_type(topology,
  HWLOC_OBJ_NODE, localeNum);
return hwloc_get_ancestor_obj_by_type(topology,
  HWLOC_OBJ_MACHINE, obj)->os_index;
}
```

8. We create an instance of Locale for each UMA sub-system.

9. The native method, *getNumProcessors()*, is called for the number of processors in the system. This method returns the number of processors in the system. The hwloc defines processors using the *HWLOC_OBJ_PU* object. The following C++ code shows this native method:

```
jint javax::realtime::Platform::getNumProcessors()
{
return hwloc_get_nbobjs_by_type(topology, HWLOC_OBJ_PU)
}
```

   Processors in Linux are numbered from *0 to numProc - 1*. From each processor object, we can find the cache information of the processor using the HWLOC_OBJ_CACHE. The cache object can be used to return the size, level of cache and if it is a shared cache.

10. The native method, *getLocale_fromProcessor(...)*, is called to retrieve the Locale to which this processor belongs to. The following C++ code returns the index of the Locale this native method:

```
jint javax::realtime::Platform::getLocale_fromProcessor
  (jint cpu)
{
hwloc_obj_t obj = hwloc_get_obj_by_type(topology,
  HWLOC_OBJ_PU, cpu);
return hwloc_get_ancestor_obj_by_type(topology,
  HWLOC_OBJ_NODE, obj)->os_index;
}
```

11. We create an instance of the *Processor* class for each processor.

12. The native method, *getMemorySize_fromLocale(...)*, is called to retrieve the size of the memory on each Locale. In hwloc, the memory on a Locale is represented using the *hwloc_obj_memory_page_type_s* which contains the size of the memory. Each memory is attached to a node and can be retrieved from the *HWLOC_OBJ_NODE* object.

```
jlong javax::realtime::Platform::getMemorySize_fromLocale(jint
   localeNum)
{
hwloc_obj_t obj = hwloc_get_obj_by_type(topology,
  HWLOC_OBJ_NODE, localeNum);
return obj->memory.local_memory;
}
```

13. A LocalMemoryType is created for each local memory on a Locale (see Section 5.1.2) and an instance of the *Memory* class for each locale.

14. Once all these objects are built, the *finalizeTopology()* sets up the library to be used by the application.The method then calls the finalizeTopology() in the Locale calls to set all the references. The following Java code shows the method:

```
static void finalizeTopology() {
  for (int i = 0; i < getNumLocales(); i++) {
                locatesArray[i].finalizeTopology();
                }
        nativefinalizeTopology();
 }
```

The *finalizeTopology()* also calls *nativeFinalizeTopology*, which is a native method to clean up the hwloc topology. The following C++ code shows the native method:

```
void javax::realtime::Platform::nativeFinalizeTopology()
{
 hwloc_topology_destroy(topology);
}
```

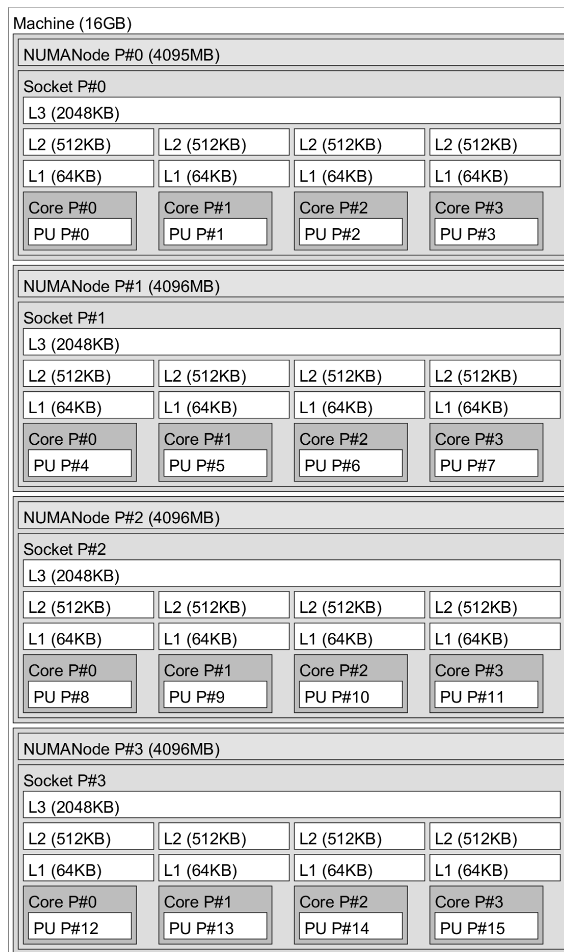


Figure 5.4: Graphical Output of lstopo(hwloc)

The `Platform` class is provided as an interface for the application to access this representation.

5.3 Implementing the Application Model

In this section, the implementation details of the Application model will be presented.

Figure 5.5 shows an RTSJ application based on the Locality model. For each Place, a cgroup is created with the memory controllers and cpuset of the Locale. In the application, there are a number of ExecutionSites. Each ExecutionSite has a number of ReservationServers represented in the figure as RS. The RS cgroup provides the cost enforcement control files for the period and runtime. The Locality model uses a partitioned approach for reservations, Linux provide a parti-

tioned approach by replicating the budget and period values on all processors in the cgroup[6] [Checconi et al., 2009].

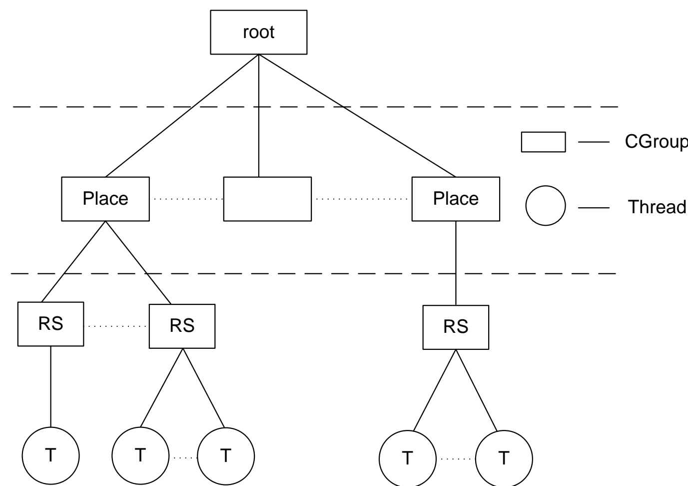


Figure 5.5: Implementing the Locality model using the Control Groups

5.3.1 Creating *Places*

Figure 5.6 shows a sequence diagram to create the virtual platform which provides the essential resources to host the application. Places are created on all available locales during the initialization of the JVM. A *Place* represents a virtual resource on a Locale. It is created on every Locale present in the topology. The architecture needs to be built before the creation of the Places. The following steps are taken:

1. The initialize(...) method is called by the runtime in the Platform class. The parameters passed to this method specify the initial MemoryParameters and the ClusterContract for the Place.

2. A loop is created, in each iteration a Locale is selected for the creation of the Place.

3. A new instance of the Place is created on the Locale.

4. A PhysicalImmortal memory area is created on the Place.

[6] http://lwn.net/Articles/420408/

5. A PhysicalHeap memory area is created on the Place.

6. A native method, createCgroup() is called which creates a cgroup in the Linux kernel. The native method uses the libcgroup library to create the cgroup by using the function cgroup_create_cgroup(...).

7. The factory provided in the ExternalContract is used to create a cluster contract. The ExternalContract class is used an interface to create and manage the ClusterContracts.

8. The factory method then creates a new ClusterContract.

9. For each processor on the ClusterContract, a PartitionedParameters instance is created. This instance shows the cumulative bandwidth being used by the ExecutionSites on the Place. The period is set to in all cases to 1 second for the Place. This value is set based on the limitation of Linux.

5.3.2 ExecutionSite Creation

An ExecutionSite is a group of tasks that have reservation defined for it. Figure 5.7 shows a sequence diagram illustrating the steps to create an ExecutionSite. The ExecutionSite will be shared by a number of Schedulables. The ExecutionSite will use the heap and immortal memory provided by the Place as local memory.

1. An execution sites map onto a locale, the factory methods are provided at the Locality class to allow the RTJVM decide the allocation of an execution site. All these factories accept a locale object which allow the programmer to manually place the execution site.

2. When this object is null, then it is upto the RTJVM to automatically map the execution site on any locale. This mapping can be based on a number of factors which include load on the system, requirements of the execution site and availability of resources on the system. Each ExecutionSite object is placed on an immortal memory area local to the place where it is created.

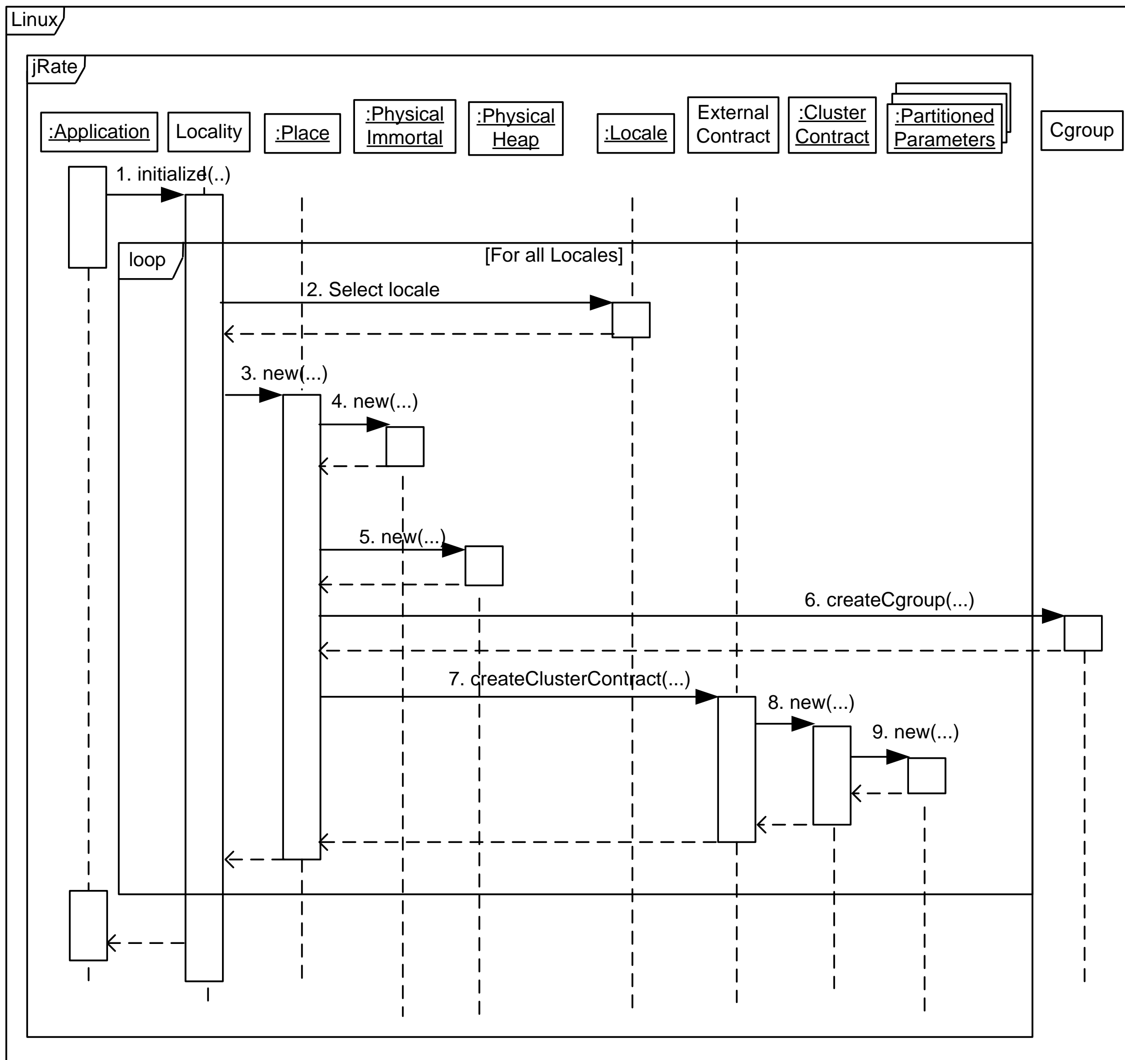


Figure 5.6: Sequence diagram: initializing the virtual platform

3. Checks if the creation of an ExecutionSite on the selected locale is feasible. If the locale was passed as a parameter in step 1, then no guarantees are provided, however, the ExecutionSite is still created on the required Locale. In case the locale is selected in step 2, then another locale is selected. In case of the ExecutionSite not being feasible on all locales, then the selected locale can be used without providing any guarantees.

4. Gets the reference of the Place on the selected Locale.

5. The current allocation context (memory area) is changed to the Immortal memory area of the Place.

6. All parameters of the ExecutionSite are copied to the new memory area using the clone() method.

7. An instance of ExecutionSite is created.

8. The PartitionedReservation is created for the ExecutionSite.

9. If guarantees are to be provided to the ExecutionSite, then the PartitionedReservation constructor calls the addToFeasibility() method in the scheduler for guarantees from the scheduler. An admission control test (such as the one described in Appendix D.2) is executed to check if guarantees can be provided to the ExecutionSite and on which processors. The addToFeasibility(...) method adds the reservation and provides a set of PartitionedParameters which specify the processors on which the reservation have been made.

10. A PartitionedParameters instance is created for each processor specifying the budget guarantees on that processor.

11. A ReservationServer is created for the cost enforcement of each PartitionedReservation.

12. The ReservationServer calls the native method createCgroup() to create and set the parameters of the cgroup. The runtime and period are replicated on

all the CPUs[7] which is consistent with the requirements of the Locality model. Setting the parameters even once is sufficient as a single interface exists for all processors in the cgroup. The native method uses the libcgroup library to create the cgroup by using the function cgroup_create_cgroup(...)

13. Once an executionsite has been created, it is then registered with the Place on which it has been hosted.

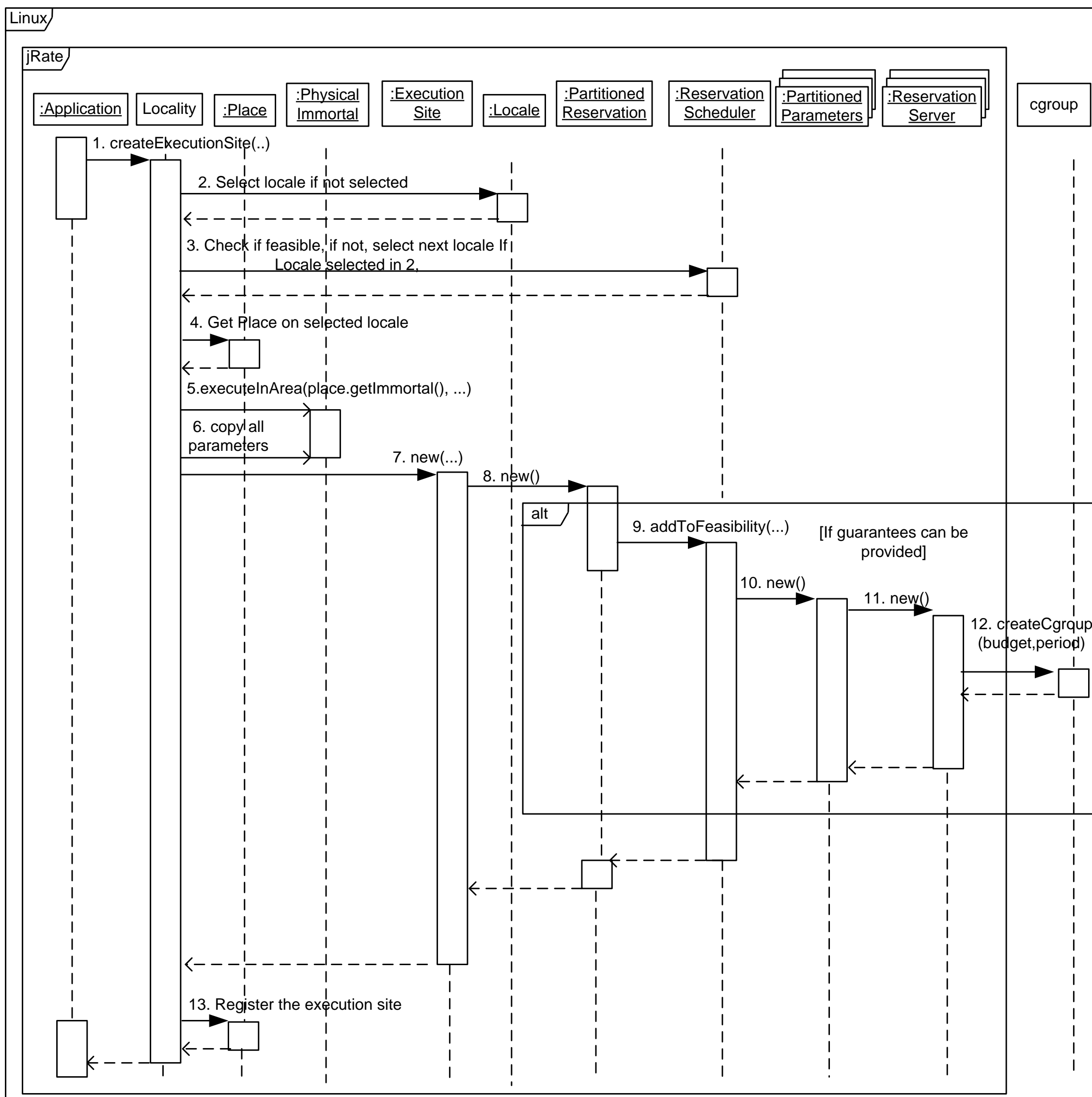


Figure 5.7: Sequence diagram: creating an execution site using a factory method

[7] http://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

5.3.3 Thread/Schedulables Creation

Factories are provided in the ExecutionSite class to create threads and schedulable objects.

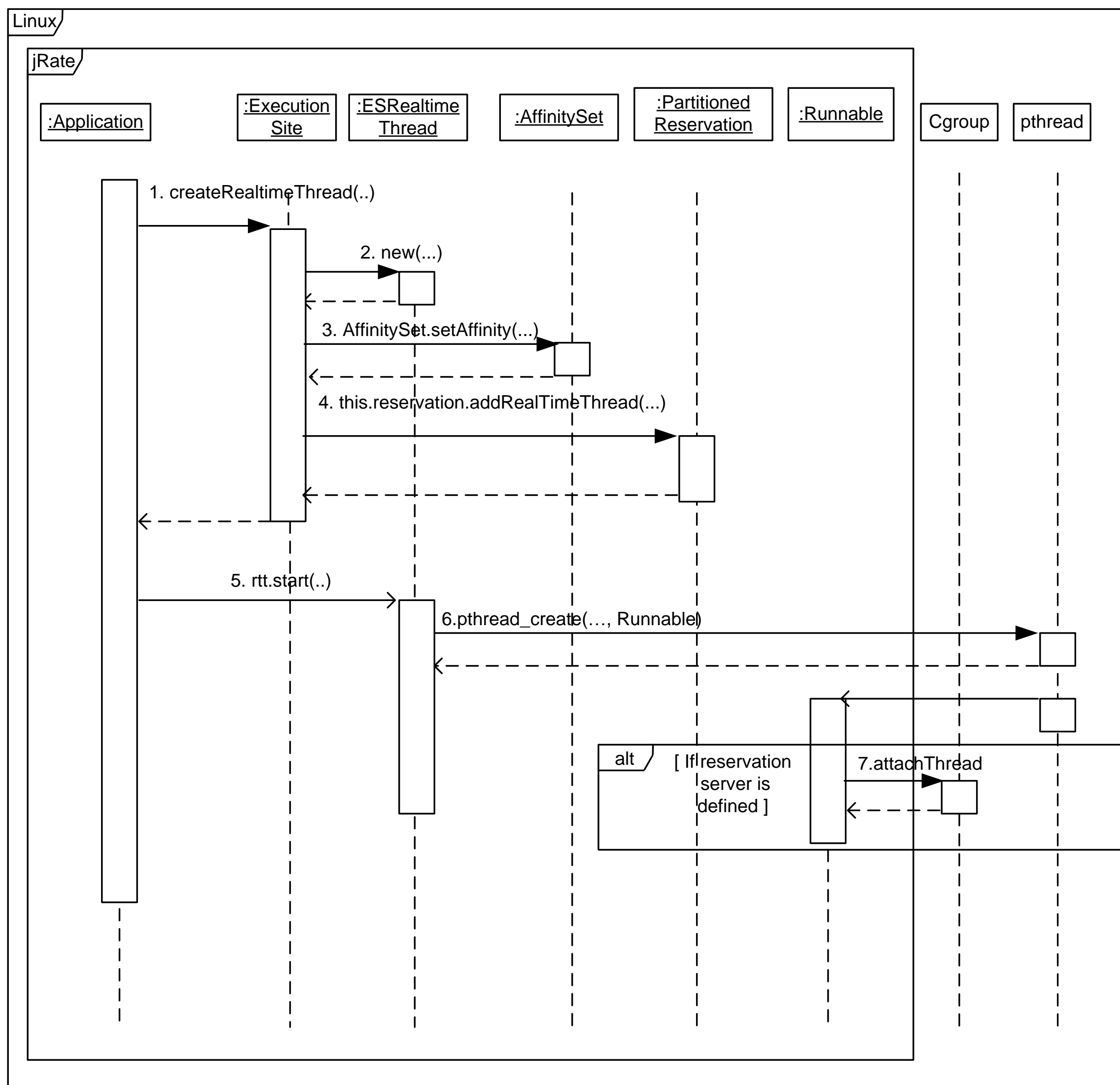


Figure 5.8: Creating and starting thread using the Locality model.

RealtimeThreads and and NoHeapRealtimeThreads are required to have a reference of their ExecutionSite. Therefore the following implementation classes have been used to associate them with ExecutionSites.

Class	
class ESRealtimeThread extends RealtimeThread	
Constructor	
ESRealtimeThread (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic, ExecutionSite ES)	
Methods	
ExecutionSite	**getExecutionSite**() Returns the ExecutionSite for this thread.
boolean	**attachThread**(this.getExecutionSite().getReservation()) Attaches this thread to the reservation of the ExecutionSite.

Table 5.2: The *ESRealtimeThread* Class

Class	
class ESNoHeapRealtimeThread extends NoHeapRealtimeThread	
Constructor	
ESNoHeapRealtimeThread (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, Runnable logic, ExecutionSite ES)	
Methods	
ExecutionSite	**getExecutionSite**() Returns the ExecutionSite for this thread.
boolean	**attachThread**(this.getExecutionSite().getReservation()) Attaches this thread to the reservation of the ExecutionSite.

Table 5.3: The *ESNoHeapRealtimeThread* Class

Figure 5.8 shows a sequence diagram for the creation of a RealtimeThread using the factory method provided in the ExecutionSite. The following steps are shown to create RealtimeThreads:

1. The factory method *createRealtimeThread()* is used to create real-time threads. This is used instead of the constructor normally used for the creation of real time thread. The Runnable logic is passed as a parameter to the factory along with SchedulingParameters, ReleaseParameters, MemoryParameters and MemoryArea.

2. The ESRealtimeThread class extends the RealtimeThread class. We create an instance of this class passing all the parameters to the constructor.

3. The static method of AffinitySet class is used to set the affinity of the real-time thread to set it to the AffinitySet of the ExecutionSite.

4. The addRealtimeThread() for the reservation is called to include the thread to the feasibility analysis. This feasibility analysis makes sure if all threads will meet their timing requirement within the budget allocated to the reservation.

5. The thread is now started.

6. JRate actually creates a pthread when the Java thread is started.

7. If the ReservationServer of the ExecutionSite is defined then the native method attachThread() is used to physically attach the thread to the associate cgroup of the PartitionedParameter. The task ID (tid in Linux) is retrieved and then added to the cgroup of the ExecutionSite using the cgroup_attach_task() function in libcgroup.

5.3.4 MemoryArea Creation

Figure 5.9 shows a sequence diagram to create a scoped memory area on an ExecutionSite. The folowing steps are shown to be undertaken:

1. Factories are provided in the ExecutionSite to create scoped memory areas. The figure shows a factory method requesting the creation of LTPhysicalMemory .

2. The requested memory area is selected which in this case is LTPhysicalMemory because the factory method receives 0 as a parameter which corresponds to LTPhysicalMemory (see Table 4.12).

3. The constructor of the LTPhysicalMemory area is called to tell the constructor to allocate the memory on LocalMemory instance which is local to the ExecutionSite.

4. The constructor calls the map method of the LocalMemory (registered with the PhysicalMemoryManager).

5. The LocalMemory calls a native method, nativeMap() which is responsible of allocating the memory.

6. The native method uses the NUMA API to allocate the memory. Factories are provided in the ExecutionSite to create scoped memory areas. The representing the scoped memory area is placed on the heap memory/immortal memory area (in case of NoHeapRealtimeThread creating a scoped memory).

7. memset(...) is called to make sure all pages are touched forcing the kernel to allocate these pages now instead of waiting until they are used first.

8. Once all the pages are allocated in physical memory. The virtual addresses are locked using the mlock() call.

In case the memory area is being reclaimed, the unmap() method is called in the LocalMemory instance to unlock and free the memory.

5.4 Implementing Reservations Model

The previous section has already discussed the following basic operations of the resource model:

1. Creation of a PartitionedReservation (in Section 5.3.2) which is performed during the creation of the ExecutionSite. PartitionedParameters and ReservationServers if the admission control check is successful.

2. Attaching a thread to a ReservationServer (in Section 5.3.3).

This section discusses the support provided in Linux to provide partitioned reservations. The goal is to provide CPU usage budget on a per processor basis using the support available in Linux. The following issues needs to be considered when implementing the resource reservations:

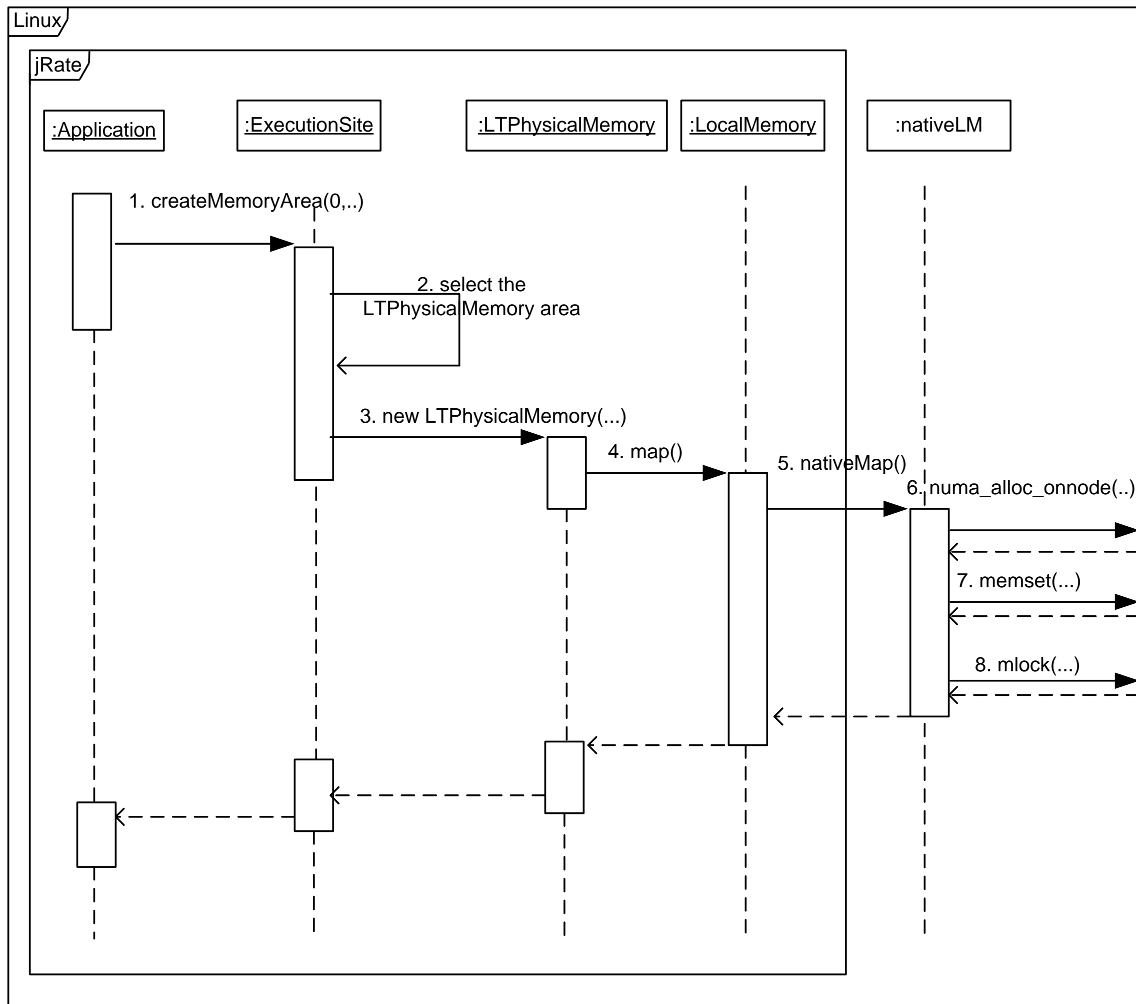


Figure 5.9: Sequence diagram: Creating a LTPhysicalMemory area

- A two level hierarchical scheduler is proposed which essentially provides a partitioned approach at the top level. At the bottom level a global scheduling scheme is proposed.

- In order to be compatible with the RTSJ, the implementation is based on fixed priority scheduling at both levels.

JRate supports native threads which are scheduled by the OS scheduler. Since the Linux scheduler is also based on fixed priority, the RTSJ only sets the priority of threads and leaves the rest to the OS scheduler. Similarly, atwo level hierarchical scheduler needs to be emulated using a single scheduler which flatly schedules all the threads present in the system. Emulating a hierarchical scheduler has been based on the following two mechanisms:

1. Providing a cost accounting and enforcement mechanism on each processor.

2. Setting the priority of threads appropriately.

The following discusses how partitioned reservations can be implemented on Linux using the control groups and are compliant with the RTSJ.

5.4.1 Cost Enforcement

The scheduler creates a number of ReservationServers for each Reservation. A cgroup is created for the set of ReservationServers belonging to the same Reservation. The ReservationServers are created inside the cgroup of the place on which the ExecutionSite is hosted. For the ReservationServers, the following controllers are added to the cgroup:

- cpu – The cpu controller provides control files for the runtime and period of the cgroup.

- cpuset – The cpuset allows to set which processors are assigned to the cgroup. The budget will be provided only on these processors.

The Linux kernel 2.6.27 provides a CPU throttling mechanism[8] which limits the CPU usage time used by a thread or a group of threads (tasks in case of Linux). The time can be explicitly set in the user space by the following two parameters: cpu.rt_runtime_us and the cpu.rt_period_us. The cpu.rt_runtime_us parameter sets the budget(Q in microseconds) of the task group and the cpu.rt_period_us allows to set the period (T in microseconds) of the task group. The budget/period parameters are set for all execution sites which limit the CPU time usage for that group. The budget/period is replicated on all the processors belonging to the execution site. Once the budget on a processor expires, the group is blocked until the arrival of the next period, T.

All threads are attached to their respective groups. The throttling mechanism ensures that the threads do not over-run their budgets.

This, however, only limit the usage of the CPU time and does not enforce any guarantees that the time will be provided.

5.4.2 Priority Assignment

The Linux scheduler always select the real-time task with the highest priority irrespective to which group it belongs as long as there is budget available. In order to provide a hierarchical scheduler to provide guarantees to the execution site, each execution site has a priority.

The following assumption is required for the priorities of the RealtimeThreads: "priorities assigned to threads in one execution site cannot be compared with priorities assigned to threads in other execution sites".

Since all threads are native and are scheduled by a single scheduler, global priorities are set based on priority of the execution site. Threads on each execution site form a range of priorities. The priority ranges of two execution sites do not overlap to make sure that each execution site gets the time that was guaranteed.

All schedulable objects in RTSJ are scheduled at the same level based on preemptive fixed priority, in order to implement hierarchical scheduler based which can

[8]http://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

provide some guarantees, the following observations have been made for implementing hierarchical scheduling:

- Each execution site has a priority (or a priority range) which determines the order in which each of them will get their guarantees.
- Threads inside an execution site should be allocated priorities relative to their execution site. Therefore, each execution site will have in fact a range of priorities and ranges of execution sites should not overlap.
- The priority range is usually very limited which will limit the number of execution sites and threads. However, the restriction of an execution site to be mapped on a single locale effectively distributes the priority ranges into different dispatching domains based on the number of locales.
- The cost accounting and enforcement can be provided by the execution time servers which will not allow any over-runs and once the budget is exhausted, the execution site will have it's priority reduced.
- The execution site will wait for the budget to be replenished while lower priority execution sites execute. After the budget is replenished, the execution site will be allowed to execute again.

When the budget expires on a CPU, the task cannot be descheduled before the next scheduler_tick() arrives. Therefore, the task does overrun the allocated budget although by a small amount.

Figure 5.10 shows that the budget is exhausted at Q_{Ex} but the thread keep on executing until the next tick arrives i.e. t_k at Q_{ov}, which is the instant at which the thread actually stops executing on the ith processor. Therefore, the total over-run is upper bounded by P_{tick} on each processor. On m processors, the overrun is upper bounded by $m * P_{tick}$.

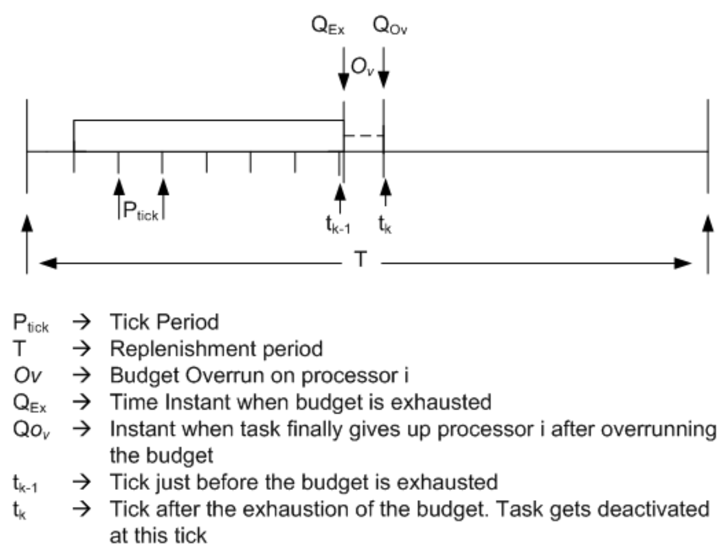


Figure 5.10: Over-run on a single processor

5.5 Summary

In this chapter, an implementation of the Locality model is presented which has been based on JRate. JRate was first extended to support AffinitySets and access to local memory. The architectural model is implemented using the hwloc library. The application model has been based on cgroups where each Place and ReservationServer corresponds to a cgroup. Libcgroup is used to manage the cgroup filesystem in the kernel. The resource reservation model is based on the throttling mechanism which provides budgets on individual processors.

Chapter 6

Evaluation

Chapter 4 introduced a locality model, an extension to the RTSJ's existing semantics to handle cache coherent non uniform memory architectures. Chapter 5 then showed that such a model is implementable in a general purpose OS, Linux. In this chapter the model and its implementation will be evaluated against the following goals:

- Programmability
- Portability
- Performance
- Predictability

The chapter evaluates the architectural representation, application model and the resource reservation model in terms of the above mentioned goals. Section 6.1 discusses the programmability of the locality model reviews the effectiveness of the abstractions to deal with cc-NUMA architectures. Section 6.2 discusses the portability of the locality model. Section 6.3 discusses the performance of the locality model. Section 6.4 discusses predictability in the locality model. Section 6.5 discusses the overheads in the locality model and finally section 6.6 summarizes the chapter.

6.1 Programmability

The locality model has made no change to the syntax or semantics of the RTSJ API. Factory methods are provided instead of making extensive changes to the API.

In order to avoid widespread changes to the RTSJ API, we introduce factory methods to create threads and memory areas. The only change required to create a schedulable object and memory area in the locality model is to use the factory provided instead of using their constructors. This is similar to the approach adopted in RTSJ version 1.1 for affinity sets. Examples have been provided in chapter 4, the following reviews how the locality model changes the programming model using these examples:

- Programmers can still use the existing programming model of RTSJ, however, using new factory methods on cc-NUMA will increase performance and provide a more predictable environment.

- The programmer needs to create an ExecutionSite before creating a schedulable object or a memory area. The Execution site can be created and allocated explicitly as shown in the following snippet:

```
                .
// We create Execution Sites statically on all available
// Locales.
Neighbourhood n1 = Locality.getCurrentNeighbourhood();
Locale[] locs = n1.getLocales();
ExecutionSite[] sites = new ExecutionSite[locs.length];
for (int i = 0; i < locs.length; i++) {
        sites[i] = Locality.createExecutionSite(locs[i]);
}
                .
```

  The ExecutionSite can also be allocated by the runtime as shown in the following snippet:

```
                .
ExecutionSite site = Locality.createExecutionSite(null);
                .
```

In the above cases, no reservations have been made for any temporal or spatial resources. The following code snippet shows creation of an ExecutionSite with reservations and allocated on a Locale by the runtime:

```
                .
// In order to specify the requirements of the Execution Site,
// we create the budget and the period arrays.
int numProcessors = 2;
RelativeTime [] budget = new RelativeTime [numProcessors];
RelativeTime [] period = new RelativeTime [numProcessors];
budget[0]= new RelativeTime(500,0);
period[0]= new RelativeTime(1000,0);
budget[1]= new RelativeTime(500,0);
period[1]= new RelativeTime(1000,0);

// We specify the memory requirements of the ExecutionSite in
// terms of the MemoryParameters.
MemoryParameters mp = new MemoryParameters(8192, 8192, 8192);

// We create an ExecutionSite which is mapped by the
// runtime. The mapping is based on the required budget
// parameters.

ExecutionSite site = Locality.createExecutionSite(null,
      budget, period, numProcessors, mp);
                .
```

- A schedulable object (RealtimeThread) in RTSJ can be created as following:

```
                .
RealtimeThread t1= new RealtimeThread(...,Runnable logic));
                .
```

Alternately, in the locality model the following factory can be used to create a RealtimeThread:

```
RealtimeThread t1 = ES.createRealtimeThread(...,Runnable logic)
```

The factory makes sure that the RealtimeThread has an affinity to the Exe-

cutionSite. The RealtimeThread is also added to the reservation of the ExecutionSite.

- A scoped memory area in RTSJ is created as shown in the figure:

```
                    .
LTMemory mem = new LTMemory(MEMSIZE);
                    .
```

Alternately, the locality model provides a factory method for the creation of a LTPhysicalMemory. The factory makes sure that the memory area is created with its backing store local to the ExecutionSite.

```
                    .
// The first parameter 0 shows that an LTPhysicalMemory
// is to be created.
LTPhysicalMemory mem = ES.createMemoryArea(0,MEMSIZE);
                    .
```

- The model does not provide any restrictions on accessing remote memory areas or external threads accessing memory areas because by doing so, the model will be become very restrictive and complex for programmers. Therefore, it is up to the programmers to make sure that there is very little communication between execution sites.

- The locality model does not make any changes to the threading model of standard Java and RTSJ. Thread creation, work distribution and synchronization are the responsibilities of programmers. While implicit parallelism is being adopted by performance oriented platforms for shared memory multiprocessors [OpenMP, 2008], the model sticks to the explicit model to avoid extensive changes to threads, schedulables and the entire RTSJ specification. In addition, such systems have not yet matured in the real-time systems community and lack the necessary backing of proper scheduling theory.

Therefore, no extra complexity is added for the programmer to deal with in terms of creating threads/schedulables and memory areas in the locality model.

6.2 Portability

Real-time Java applications respects the portability of Java. Most extensions apart from the physical memory and raw memory in RTSJ are portable. The following discusses the portability of applications in the locality model:

- Architectural model – The architectural model provides a portable way representing the architecture, where the runtime creates the representation on runtime.

  The model uses AffinitySet instances throughout the model. While a processor has always been represented using an integer on platforms, but using an integer across platforms can break the portability of the application. Instead the AffinitySet instances is used to represent processors. It cannot be instantiated directly by the programmer, it is rather generated by the AffinitySet class based on the available number of processors.

  The architectural model only includes get methods and does not provide a way for the programmers to instantiate any instances. The runtime builds the platform based on the available resources. The programmers can query the system for available resources and use them instead of hard coding them in the application.

  The architectural representation, however, remains static during the lifetime of an application and requires support from underlying OS to dynamically change based on events related to hardware changes.

  The implementation is based on hwloc which has been provided for a variety of platforms. This adds evidence to the portability of the Architectural model.

- Application model – The application model provides abstractions to group together threads and objects. This supports co-allocation of threads/objects for better performance on any platform. In addition, it also provides support to co-allocate threads and devices by specifying the locale on which the ExecutionSite is going to be mapped. However, even in that case devices/Locales are retrieved from the architecture representation generated at runtime.

- Resource Reservation model – Real-time systems typically have very strict timing requirements and require a dedicated system. We have provided an environment where resources can be guaranteed to real-time systems. Although this environment is not suitable for hard real-time guarantees. However, for soft-real time systems it provides platform independence. The real-time system can execute on any platform which guarantees the required resources.

However, the locality model does make changes to the runtime. Therefore, applications built using the locality model need to be supported by the runtime. This is very similar to how Java provides portability i.e. the application can run on any architecture that is supported by the JVM.

The implementation of Locality Model presented in Chapter 5 has been built and tested on the following three architectures:

1. A single processor system with 512MB of RAM. Figure 6.1 shows the architecture of the system which has 1 District, 1 Neighbourhood, 1 Locale, 1 Processor and 1 Memory.

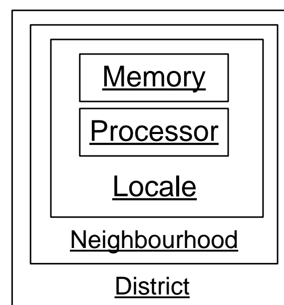


Figure 6.1: Architectural Representation of a single processor system

2. A dual core system with 2GB of RAM. This is an SMP which has 2 processors. Figure 6.2 shows the architecture of the system which has 1 District, 1 Neighbourhood, 1 Locale, 2 Processors and 1 Memory.

3. A cc-NUMA system which has been discussed throughout the thesis which was presented in Figure 3.9 and its architectural model illustrated in Figure 4.2.

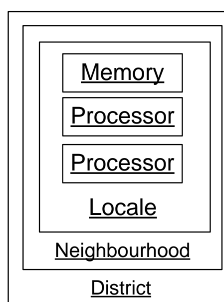


Figure 6.2: Architectural Representation of a two processor SMP

6.3 Performance

The development of benchmarks inevitably lag behind the introduction of new architectures. In our case, the situation is exacerbated because we are envisaging that future real-time embedded systems will become more highly parallel than current systems. There is currently little experience with these systems and how they can be analyzed to determine their timing properties. For these reasons, we illustrate the performance gains, by using execution time as a metric on the producer/comsumer problem and prime number generation using the sieves of Eratosthenes.

The platform used for these experiments is a 16-processor cc-NUMA system based on the AMD Opteron architecture as shown in Figure 3.9. The hypertransport between node0 and node1 is set at 200 MHz for all these experiments (for more details see Appendix A.2). JRate/Linux 2.6.28 is used as the software platform for these tests.

6.3.1 The Producer/Consumer Problem

In this experiment, two threads are created: the producer writes a block of memory and the consumer reads it. The shared object is locked when the producer is writing and the consumer is reading. The locality model will be used to test the effect of locality.

Test Settings The execution time is measured for the following two different cases:

- Local – In this case both threads are created on one ExecutionSite that is allocated to a particular locale. This case has been presented in the example

in Section 4.2.7.1.

- Remote – In the remote case, the producer and consumer are created on two different ExecutionSites that are then statically allocated to different locales. This case has been presented in the example in Section 4.2.7.2.

The experiment is executed using different size of Workload starting from 500 , 1000, ..., 20000. Workload is the amount of data being shared between the producer and consumer. In both cases, we measure the time for both threads (producer and consumer) to finish.

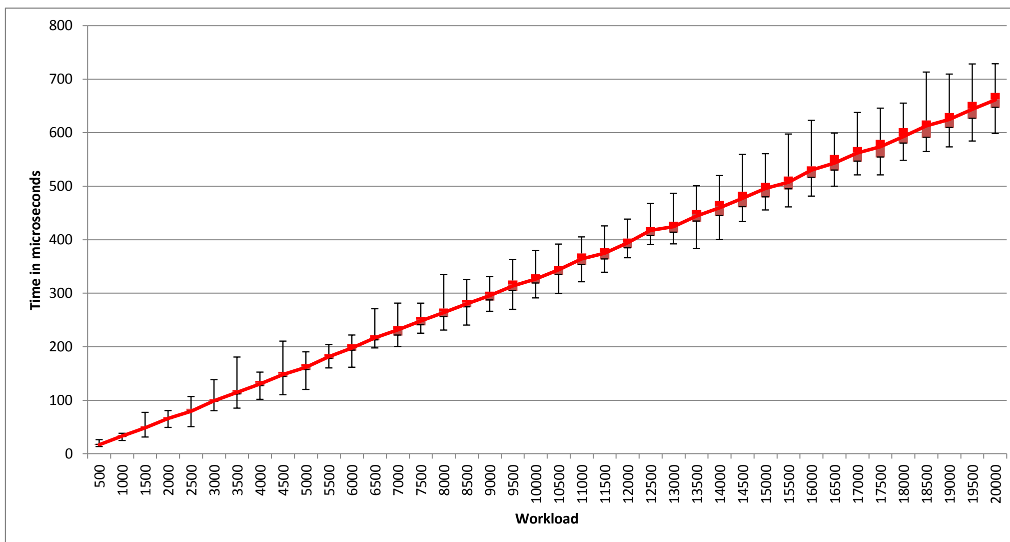


Figure 6.3: Execution times for local producer consumer problem using the locality Model

Results Figure 6.3 shows a boxes and whiskers diagram to show the execution times calculated for the local case and Figure 6.4 shows a similar diagram to show the execution times for the remote case. The statistical analysis of the execution times for the local case is given in Table 6.1. The statistical analysis for the remote case is presented in Table 6.2.

Results Analysis The following show the effect of locality on performance:

- Average value – The average value for both the local case and the remote case are shown in Table 6.1 and Table 6.2 respectively for the producer consumer test. The average execution time in the remote case are longer than the values

Workload	Average	STD	Count	Min	90%	95%	99%	Max
500	16.96	1.58	1000	13.51	18.01	18.75	20.05	26.31
1000	32.86	3.13	1000	24.81	35.59	36.96	38.19	38.23
1500	48.26	6.26	1000	31.23	51.89	53.82	56.92	77.34
2000	65.88	4.67	1000	49.21	69.83	71.18	74.92	80.64
2500	79.56	9.37	1000	50.62	85.43	85.85	89.9	106.98
3000	98.68	7.69	1000	80.43	103.9	108.68	111.52	138.5
3500	114.88	12.3	1000	85.22	121.31	125.95	167.11	180.7
4000	130.29	8.97	1000	101.68	137.2	138.55	145.74	152.58
4500	148.17	13.93	1000	110.25	158.44	161.44	207.53	210.44
5000	161.26	12.67	1000	120.25	169.61	178.26	190.31	190.41
5500	181.35	7.75	1000	160.34	188.59	191	198.75	204.21
6000	197.75	10.5	1000	161.69	206.7	213.55	221.7	221.86
6500	216.79	10.08	1000	197.73	227.51	229.98	247.7	270.94
7000	231.44	13.6	1000	200.59	245.93	253.38	264.14	281.49
7500	248.42	13.11	1000	225.28	267.26	269.14	274.9	281.41
8000	264.04	14.78	1000	231.13	278.39	285.48	299.15	335.16
8500	280.19	15.53	1000	240.43	297.11	306.31	314.29	325.56
9000	295.47	13.79	1000	266.21	309.94	319.57	325.64	330.91
9500	313.48	18.85	1000	269.93	332.64	341.38	358.17	362.8
10000	326.68	17.13	1000	291.22	344.22	356.58	370.28	379.71
10500	343.53	16.28	1000	299.63	359.94	374.04	386.69	391.86
11000	364.19	18.36	1000	321.4	391.66	397.62	402.38	405.25
11500	374.85	17.64	1000	339.21	391.11	408.88	423.9	425.8
12000	394.19	15.4	1000	366.29	409.1	426.39	433.87	438.55
12500	417.28	16.21	1000	391.17	440.89	454.29	460.92	467.82
13000	424.53	18.15	1000	392.2	440.84	454.27	479.55	486.81
13500	444.21	22.99	1000	383.37	468.11	484.17	500.07	500.81
14000	459.73	25.14	1000	400.39	487.86	502	517.34	519.93
14500	477.27	22.99	1000	434.15	500.22	516.69	544.43	559.51
15000	496.05	23.9	1000	455.6	532.48	544.81	560.45	560.77
15500	507.36	22.74	1000	461.3	527.74	550.4	561.69	597.53
16000	530.32	27.69	1000	481.52	569.79	588.92	606.08	623.14
16500	542.91	21.5	1000	500.01	564.4	583.54	589.85	599.4
17000	562.25	22.95	1000	521.13	585.03	606.9	631.43	637.8
17500	573.97	26.62	1000	521.15	599.39	626.99	645.57	645.89
18000	592.84	21.28	1000	548.42	617.36	621.27	655.21	655.28
18500	612.87	30.49	1000	564.77	657.49	672.22	711.41	713.32
19000	624.52	25.45	1000	573.49	652.04	674.36	692.37	709.55
19500	643.9	30.42	1000	584.47	689.21	694.96	726.45	728.45
20000	661.18	26.78	1000	598.47	690.08	714.42	722.21	728.85

Table 6.1: Execution times (in microseconds) for local producer/consumer

Workload	Average	STD	Count	Min	90%	95%	99%	Max
500	24.97	2.42	1000	23.1	26.36	31.57	33.23	33.58
1000	46.2	5.81	1000	25.95	51.55	51.81	52.54	63.51
1500	72.43	5.16	1000	66.34	77.6	82.34	88.66	88.68
2000	96.6	6.69	1000	89.59	102.82	103.48	123.37	127.96
2500	120.96	10.54	1000	108.7	131.31	135.93	144.83	187.61
3000	144.32	9.89	1000	129.17	153.38	154.33	167.95	197.67
3500	167.88	13.12	1000	141.8	178.79	181.03	217.66	222.8
4000	184.48	11.82	1000	150.45	202.49	204.41	210.37	224.38
4500	210.37	10.64	1000	190.68	227.54	228.39	231.52	231.89
5000	239.11	14.75	1000	192.7	254.37	256.63	259.86	306.15
5500	266.48	18.47	1000	205.34	283.21	288.45	300.09	375.61
6000	290.31	21.42	1000	258.65	307.68	325.87	377.55	377.73
6500	311.56	26.47	1000	280.49	334.82	390.15	391.79	392.76
7000	332.97	17.6	1000	304.78	355.25	357.5	378.74	394.85
7500	344.14	27.8	1000	281.17	377.75	382.51	385.11	385.72
8000	372.58	27.46	1000	293.12	406.55	410.71	427.99	434.43
8500	400.1	19.23	1000	358.95	428.55	431.33	433.6	438.26
9000	431.39	25.56	1000	349.96	456.08	462.75	496.77	499.94
9500	457.05	40.58	1000	423.16	540.8	551.33	584.45	590.99
10000	479.23	26.88	1000	414.11	511.37	514.25	547.89	552.84
10500	512	66.95	1000	465.59	538.84	713.72	756.36	762.53
11000	526.94	30.77	1000	471.12	560.73	566.77	600.36	616.32
11500	549.6	30.34	1000	488.12	586.19	591.85	610.77	617.91
12000	577.5	39.38	1000	526.75	643.43	653.65	659.89	702.17
12500	590.89	34.98	1000	544.46	647.29	663.52	668.36	670.54
13000	612.69	28.87	1000	560.23	656.57	662.38	664.79	679.91
13500	638.11	33.48	1000	592.84	682.08	690.67	708.3	727.13
14000	667.46	38.68	1000	581.83	710.46	718.8	738.32	740.56
14500	695.79	42.79	1000	640.81	751.73	759.3	802.79	803.92
15000	710.28	38.75	1000	665.97	758.03	763.16	814.17	815.25
15500	746.17	40.21	1000	690.37	791.47	795.72	834.71	836.74
16000	758.64	39.58	1000	703.39	807.98	811.17	823.96	832.74
16500	787.72	45.85	1000	703.88	842.8	848.59	860.98	924.16
17000	791.21	46.53	1000	691.1	853.78	857.57	872.41	913.33
17500	820.72	41.21	1000	753.98	885.51	895.72	902.27	933.34
18000	855.51	43.53	1000	807.63	911.88	923.58	964.89	987.55
18500	864.46	45.92	1000	783.31	931.08	936.86	944.93	949.63
19000	896.75	52.23	1000	828.85	975.34	1005.28	1040.24	1068.18
19500	930.27	68.26	1000	862.36	1061.93	1066.32	1124.75	1125.65
20000	938.22	50.71	1000	880.52	1004.9	1012.64	1131.07	1155.85

Table 6.2: Execution times (in microseconds)for remote producer/consumer

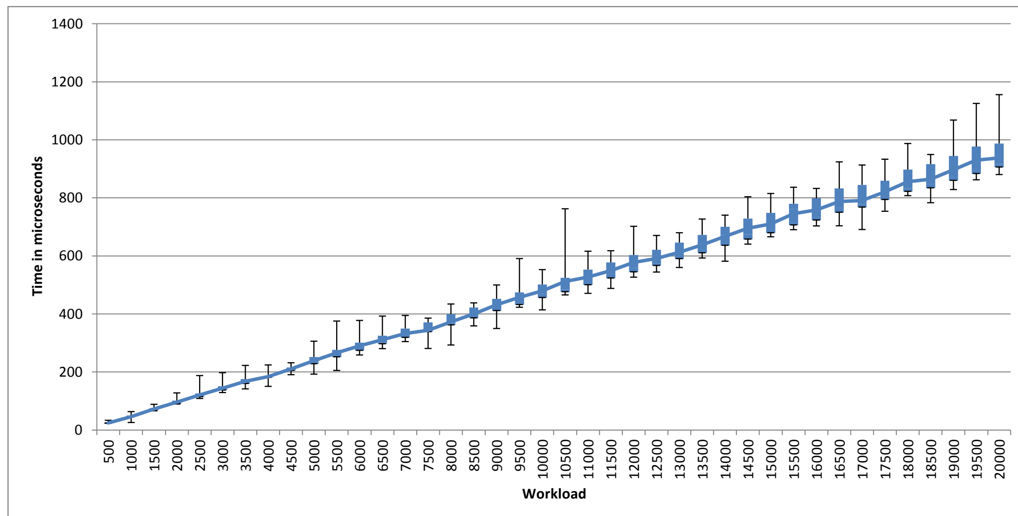


Figure 6.4: Execution times for remote producer consumer problem using the locality model

for the local case. At N=500, the average execution time for the remote case is 24.97 us compared to 16.96 us for the local case. At N=20000, the average in the remote case is 938.22 us compared to locale case's 661.18 us.

- Minimum value – The minimum value for both the local case and the remote case are shown in Table 6.1 and Table 6.2 respectively for the producer consumer test. The minimum time in the remote case are longer than the values for the local case. At N=500, the minimum execution time for the remote case is 23.1 us compared to 13.51 us for the local case. At N=20000, the minimum in the remote case is 880.52 us compared to locale case's 598.47 us.

This experiment highlights the effect of locality on a cc-NUMA system and show that the locality model can be used to increase the performance of an application by keeping threads and objects local to each other.

6.3.2 The Prime Sieves Example

The basic motivation to conduct this experiment is to measure the impact of locality and thread placement in a highly parallel application. Therefore, we choose an already parallel algorithm and we will be measuring the execution times of the same

number of threads in different configurations[1]. We conduct an experiment where we generate prime numbers by the Sieve of Eratosthenes.

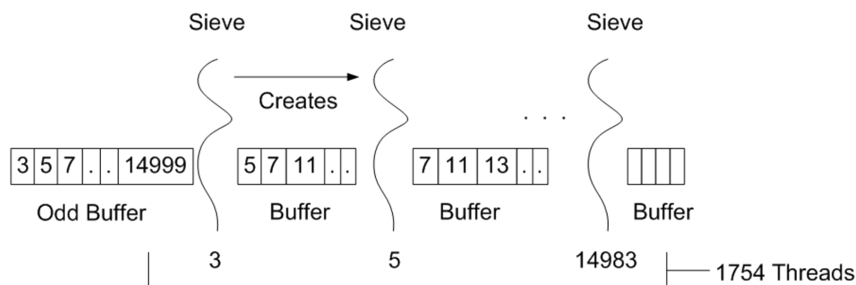


Figure 6.5: Sieve of Eratosthenes

Figure 6.5 shows the generation of primes, where each sieve is a real-time thread which receives a stream of integers; it then forwards this stream to a newly created sieve after some processing. The real-time threads are created dynamically throughout the life time of the application, as each newly created sieve then goes onto create a new sieve. The buffering used to communicate between sieves are the shared data items. Each buffer is only accessed by two sieve threads and therefore the buffer should be kept local to the threads for fast access.

Due to the presence of large instruction caches, the code gets cached in for the first time a cache miss occurs. Any subsequent access to the method code should be a cache hit and hence will not affect the overall timing. Therefore, the performance of the application depends on the following two factors:

1. Distributing the load evenly across the platform.

2. Ensuring locality where ever it is possible.

We create four ExecutionSites, one ExecutionSite on each Locale. Threads are dynamically created using factory methods of the ExecutionSites. The selection of the ExecutionSite for thread creation is based on load balancing and locality.

[1]Note, that this is not the traditional measure of speed-up which is often used to show how an application performance can be improved by the addition of extra processors. Here, we are measuring the effect that locality has in highly parallel applications.

6.3.2.1 Effect of the Locality Model on Performance

This experiment determines the effect of locality model on the performance of the a highly parallel application.

Test Settings In this experiment we generate all prime numbers less than N. Starting from N = 500, and increasing the value of N, to N = 20000 to analyze how the locality model works under different levels of parallelism.

Figure 6.6 shows the primes generated for each value of N e.g. for N = 20000, the maximum prime that is created is 19997.

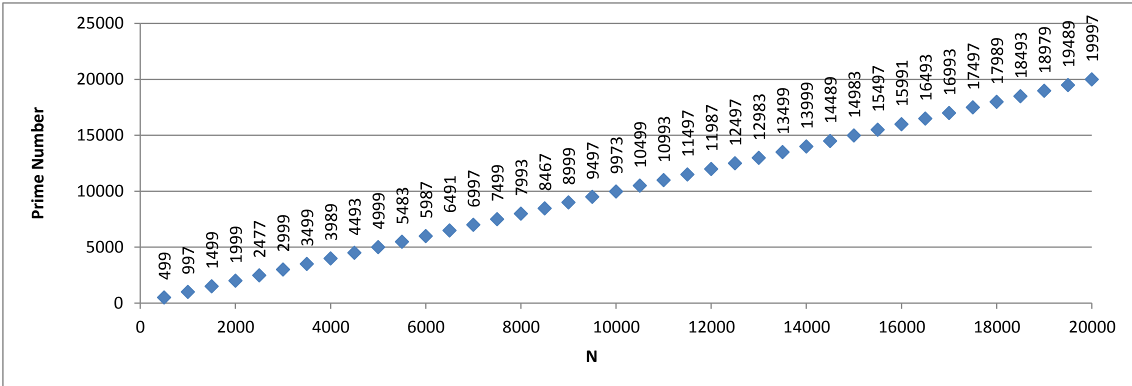


Figure 6.6: Prime numbers for all N

For each new prime a new thread (sieve) is created e.g. in the case of N = 20000, a total of 2262 threads are created (sieves) which can be shown in Figure 6.7 and record the time taken to generate 14983 which is the last prime number in our list.

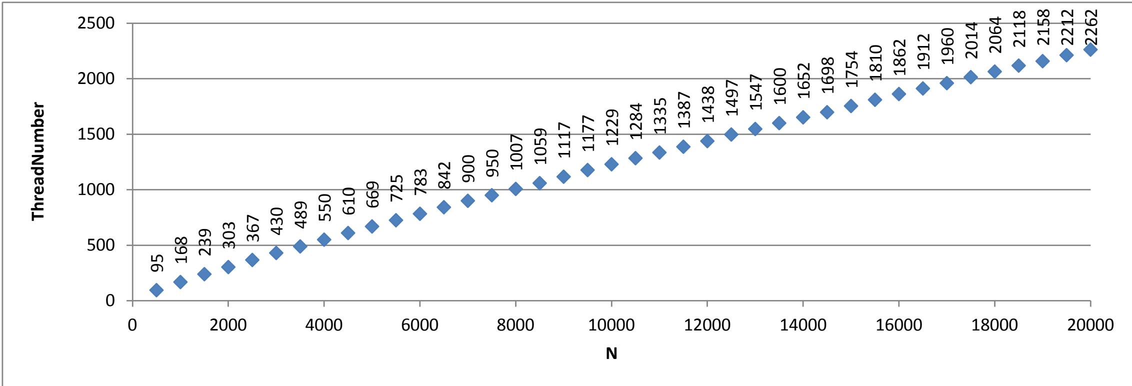


Figure 6.7: Number of threads created for all N

For this experiment, we analyze the following cases:

- Normal – In this case, we do not use the locality model. Prime numbers are generated using standard jRate. The code for this case is presented in Appendix E.1.

- Locality Model – In this case, we generate prime numbers using the locality model. Four execution sites are statically created, one on each locale. We distribute 8 threads in each of the four ExecutionSites and then start again until all threads are allocated. The code for this case is presented in Appendix E.2.

Results Figure 6.8 and Figure 6.9 give a comparison of the time taken to generate the prime numbers by both the cases for all values of N. Detailed statistics for the results have been attached in Table 6.3 (for the normal case) and Table 6.4 (for the locality model).

Results Analysis The following analyzes the results in Figure 6.8 and Figure 6.9:

1. Average – The average execution times for both the locality model and the normal case are shown in Table 6.4 and Table 6.3 respectively. The average values in the normal case are larger than the values for the locality model. At N=500, there is very little difference between the average values of the locality and the normal case with average values 117.1 ms and 142.75 ms respectively. However, the average values for the normal case increase very rapidly and at N=20000, the average execution time in the normal case is 8065.49 ms compared to locality model's 4190.72 ms which is almost half.

2. Minimum – The minimum execution times for both the locality model and the normal case are shown in Table 6.4 and Table 6.3 respectively. The results of the minimum values are similar to the results of the average case. At N=500, the minimum values of the locality and the normal case 111.99 ms and 120.85 ms respectively. However, like the average values, the minimum execution times for the normal case increase very rapidly and at N=20000, the minimum execution time in the normal case is 6822.97 ms compared to locality model's 3951.64 ms.

The normal case does not use the locality model and leave it up to the operating system and the JVM to decide how the application is distributed over the set of processors. This results in the very high execution times. This is because Linux cannot guarantee locality, without any hint from the application, the operating system does not constrain threads and memory to a particular location. Without any affinity provided from the programmer, threads move around on different processors of the system. This results in a very high number of data cache misses and on a cc-NUMA system, the cost of a cache miss is usually very high especially if locality is not guaranteed.

The locality model comparatively performs much better then the normal case.

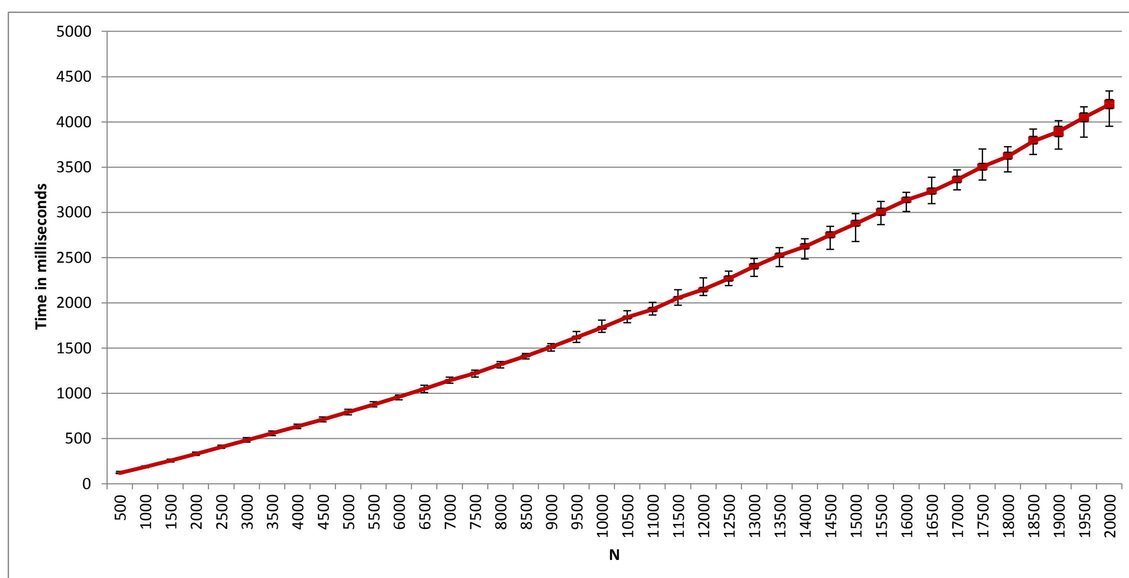


Figure 6.8: Execution times for generating all prime numbers less than N=20000 using the locality model

6.3.2.2 Trade-off between Locality and Load Balancing

In this experiment, the effect of load balancing and locality is measured. We generate all prime numbers less than 15000 and record the time taken to generate 14983 which is the last prime number in this case.

Test Settings The total number of dynamically created real-time threads is 1754 in this experiment and the performance depends on how these threads are allocated along with their objects on the cc-NUMA system. As the threads are being created

N	Average	STD	Count	Min	90%	95%	99%	Max
500	117.1	3.59	1000	111.99	118.83	120.11	135.34	136.28
1000	185.42	4.02	1000	176.3	190.76	191.52	192.25	196.7
1500	256.55	6.68	1000	239.92	265.56	268.19	270.14	270.89
2000	327.53	6.84	1000	312.4	335.33	341.04	343.93	349.46
2500	403.87	6.9	1000	391	411.53	415.81	425.52	425.76
3000	482.12	9.93	1000	459.71	494.27	497.59	504.29	509.34
3500	557.73	8.45	1000	532.04	568.14	569.71	578.78	582.66
4000	632.7	9.06	1000	609.62	641.51	646.11	654.87	658.16
4500	710.73	10	1000	683.94	723.09	724.07	726.26	737.92
5000	792.63	12.29	1000	761.51	806.83	810.82	819.6	822.61
5500	873.63	10.1	1000	849.16	886.32	889.25	899.16	906.42
6000	959.5	11.02	1000	927.49	971.58	973.7	979.25	980.38
6500	1048.58	13.15	1000	1007.61	1064.01	1069	1077.18	1088.66
7000	1143.51	12.14	1000	1111.28	1159.41	1162.72	1171.56	1177.92
7500	1222.38	14.36	1000	1178.57	1239.25	1242.75	1254.1	1255.46
8000	1319.6	14.48	1000	1280.03	1335.72	1339.93	1345.89	1350.47
8500	1412.05	14.48	1000	1379.4	1431.62	1436.75	1439.39	1439.57
9000	1510.51	19.56	1000	1466.2	1536.32	1540.59	1543	1548.21
9500	1616.74	23.17	1000	1562.39	1645.55	1653.11	1680.82	1682.95
10000	1724.59	27.96	1000	1673.41	1759.28	1787.48	1807.33	1808.5
10500	1840.01	29.65	1000	1779.91	1885.64	1891.5	1909.61	1912.22
11000	1926.18	30.95	1000	1864.82	1968.2	1979.82	1999.14	2004.62
11500	2055.2	29.24	1000	1973.01	2093.4	2104.81	2129.37	2144.53
12000	2149.22	38.89	1000	2080.85	2199.05	2222.93	2259.52	2275.78
12500	2267.23	40.69	1000	2190.14	2327.93	2339.51	2345.93	2349.93
13000	2401.7	40.11	1000	2292.15	2452.69	2457.13	2471.33	2489.82
13500	2524.36	37.58	1000	2400.25	2567.69	2583.47	2605.91	2609.67
14000	2622.14	45.3	1000	2484.61	2668.04	2687.04	2706.89	2707.94
14500	2747.34	51.22	1000	2590.99	2817.19	2822.68	2841.04	2845.46
15000	2873.36	50.79	1000	2677.44	2940.07	2956.4	2984.87	2985.81
15500	3004.06	54.18	1000	2864.17	3070.45	3090.64	3106.36	3120.6
16000	3134.9	45.68	1000	3009.01	3190.79	3201.76	3216.53	3221.16
16500	3229.82	58.18	1000	3096.11	3293.96	3315.91	3349.83	3387.89
17000	3362.99	52.75	1000	3248.79	3426.62	3440.23	3462.25	3469.26
17500	3503.82	59.98	1000	3356.92	3572.5	3598.3	3622.68	3700.24
18000	3617.58	60.44	1000	3447.22	3681.39	3703.18	3721.78	3725.58
18500	3786.65	65.81	1000	3640.1	3864.74	3875.95	3918.89	3920.41
19000	3890.75	70.91	1000	3698.92	3974.83	3992.24	4002.95	4013.43
19500	4047.24	70.24	1000	3831.25	4138.31	4162.18	4166.33	4166.5
20000	4190.72	82.5	1000	3951.64	4280.76	4307.53	4341.79	4341.86

Table 6.3: Execution times (in milliseconds) statistics for the prime sieves in the locality model case

N	Average	STD	Count	Min	90%	95%	99%	Max
500	142.75	20.46	1000	120.85	171.29	172.74	174.49	189.57
1000	251.22	42.46	1000	203.45	306.37	308.72	321.86	327.29
1500	370.46	62.42	1000	299.47	455.83	461.49	472.43	480.38
2000	493.21	85.65	1000	395.91	612.87	616.28	618.84	630.54
2500	617.55	110.38	1000	479.38	780.17	791.64	804.1	839.76
3000	781.28	133.49	1000	566.61	954.29	958.37	967.17	986.14
3500	901.41	159.13	1000	709.11	1129.95	1138.71	1145.86	1192.87
4000	1032.75	167.49	1000	815.62	1301.32	1308.95	1338.39	1342.69
4500	1233.42	193.53	1000	982.47	1499.06	1511.68	1516.41	1517.31
5000	1385.36	225.96	1000	1084.53	1703.42	1707.67	1716.06	1724.49
5500	1595.69	237.83	1000	1282.36	1891.81	1904.51	1911.07	1919.45
6000	1693.07	264.93	1000	1307.76	2107.53	2121.9	2144.51	2159.43
6500	1910.91	280.99	1000	1497.18	2327.82	2337.05	2374.38	2382.52
7000	2099.04	312.08	1000	1680.76	2540.2	2564.22	2587.98	2591.29
7500	2220.96	310.16	1000	1788.66	2578.94	2753.9	2781.31	2790.38
8000	2508.23	348.1	1000	2008.77	2983.38	2993.24	3032.02	3070.17
8500	2596.25	364.96	1000	2186.48	3122.97	3189.45	3285.42	3305.15
9000	2845.69	393.43	1000	2219.46	3424.72	3494.52	3507.4	3513.16
9500	3109.55	415.37	1000	2440.17	3665.83	3726.19	3753.57	3754.82
10000	3241.49	450.62	1000	2609.54	3860.21	3933.67	3990.9	4000.95
10500	3554.4	466.39	1000	2876.27	4136.6	4166.84	4190.73	4196.39
11000	3654.08	453.93	1000	2987.49	4299.76	4351.63	4399.55	4403.88
11500	3805.32	501.39	1000	3189.85	4614.06	4646.03	4693.17	4739.31
12000	4030.44	531.53	1000	3325.84	4842.81	4888.7	4922.86	4931.64
12500	4267.78	560.56	1000	3448.62	4996.36	5086.68	5113.15	5151.52
13000	4422.87	570.7	1000	3739.21	5335.87	5372.15	5460.64	5504.7
13500	4838.31	555.75	1000	4027.62	5612.64	5643.08	5709.76	5732.17
14000	4944.5	587.53	1000	4144.67	5809.47	5899.04	5974.47	5983.67
14500	5135.97	596.82	1000	4351.91	6026.05	6101.28	6209.72	6287.86
15000	5504.1	661.74	1000	4523.15	6404.16	6472.55	6504.23	6538.29
15500	5736.84	718.48	1000	4742.31	6725.97	6758	6810.37	6813.94
16000	6015.78	713.36	1000	4824.67	6958.23	7030.1	7085.43	7186.66
16500	6292.75	736.6	1000	5119.25	7256.07	7315.56	7369.05	7413.55
17000	6333.17	745.41	1000	5489.59	7519	7564.92	7651.8	7662.81
17500	6631.89	743.39	1000	5639.37	7780.07	7891.93	8002.36	8016.12
18000	6888.99	756	1000	5789.94	8066.14	8152.84	8206.4	8280.5
18500	7216.11	823.34	1000	6161.51	8428.02	8499.27	8604.83	8617.68
19000	7411.13	863.11	1000	6297.76	8702.13	8737.25	8811.13	8841.82
19500	7760.46	887.66	1000	6563.79	9052.27	9126.35	9198	9202.23
20000	8065.49	880.23	1000	6822.97	9337.62	9431.52	9503.19	9593.61

Table 6.4: Execution times (in milliseconds) statistics for the prime sieves in the normal case

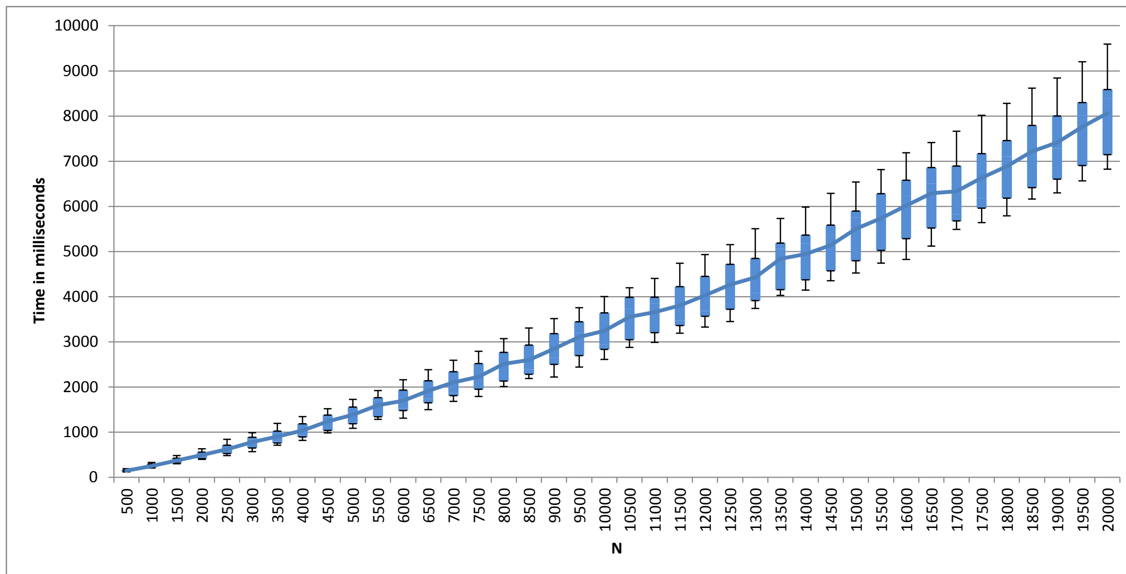


Figure 6.9: Execution times for generating all prime numbers less than N=20000 in the normal case

dynamically, when a thread is created we have to decide in which locale it should be placed among the available 4 locales. Therefore, we distribute newly created threads in the following ways:

- Case 1 – We distribute 1 thread in each of the four locales and then start again until all 1754 threads are allocated. In this case, the dynamic load on the system will be properly balanced but there will be no locality between threads.

- Case 2 – We distribute 8 threads in each of the four locales and then start again until all 1754 threads are allocated. In this case, the dynamic load on the system will be properly balanced but the locality has also improved when compared to case 2.

- Case 3 – As there are 1754 threads, we allocate all these threads in the first ExecutionSite. In this case we have very highly locality but the load on the system is imbalanced.

Results Figure 6.10 gives a comparison of the time taken to generate the prime numbers in all the 4 cases discussed. Each case is executed 1000 times, and the times are shown for each iteration. Table 6.5 provides the statistics of these results.

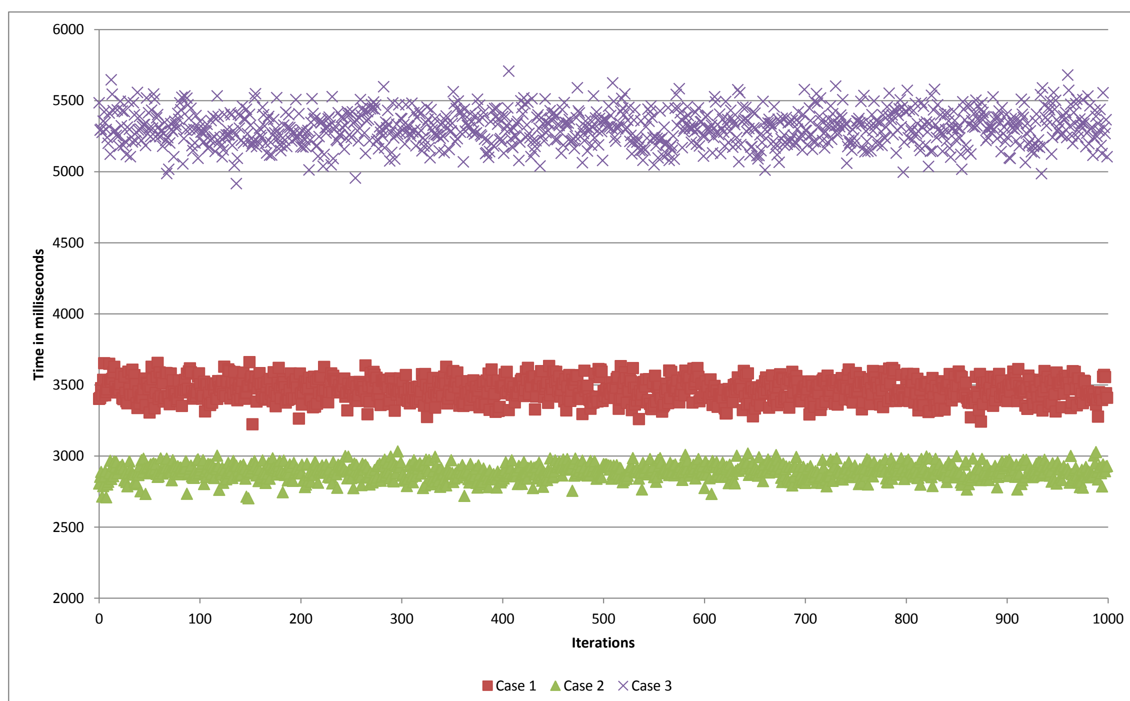


Figure 6.10: Comparison of execution times using the locality model under different configurations

	Average	STDDEV	Min	90%	95%	99%	Max
Case 1	3466.78	74.63	3233.51	3562.79	3586.84	3625.91	3665.54
Case 2	2890.94	51.28	2703.28	2955.26	2972.77	2999.62	3032.96
Case 3	5310.33	115.59	4790.45	5454.63	5485.04	5555.56	5705.63

Table 6.5: Analyzing the execution times of all cases

Results Analysis

The following analyzes the results in Figure 6.10 and Table 6.5:

- In case 2, we have both locality between the threads and load balancing. This produces the best results.
- Case 1 has load balancing but no locality as each thread is allocated on a different ExecutionSite on a different Locale. The difference between case 1 and case 2, shows the effect of locality.
- In case 3, we illustrate the tradeoff between locality and load balancing. Although locality is strong, it cannot compensate for uneven distribution of load.

In summary, using the locality model significantly improves the performance if we have a balance between locality and the load distribution of the system.

6.4 Predictability

The locality model provides guarantees to ExecutionSites. These guarantees can be used to meet the timings requirements of the schedulables inside the ExecutionSite. Appendix D provides a schedulability analysis that is compatible with the locality model. The purpose of providing this schedulability analysis is to show that the the resource reservation model is inline with the real-time scheduling theory for multiprocessors.

This section highlights the effect of the locality model on the predictability of real-time Java applications.

6.4.1 Dispersions

Execution times on a cc-NUMA system vary because of varying memory access timings in the cc-NUMA system. The following shows the effect of locality on these dispersions:

6.4.1.1 Locality Model vs. Normal Case

The prime sieves example that was performed for all N values ranging from 3 to 20000, measured the execution times for the Locality model and the Normal case to compare the difference that makes by allowing the application to handle the allocation policies instead of the OS. The following presents the dispersion values measured in this test:

- Standard deviation – The standard deviation for both the locality model and the normal case are shown in Table 6.4 and Table 6.3 respectively. The standard deviation values in the normal case are larger than the values for the locality model. At N=500, the standard deviation value for the normal case is 20.46 ms compared to 3.59 ms for the locality model. This standard deviation

in the normal case is roughly about 6 times the standard deviation values of the locality model. The standard deviation in the normal case increases very rapidly when compared to the locality model and at N=20000, the standard deviation in the normal case is 880.23 ms compared to locality model's 82.5 ms. Now the standard deviation is roughly 11 times higher in the normal case compared to the locality model.

- Percentiles – The percentiles presented in the Table 6.4 and Table 6.3 show that the average value is much closer to the 90%, 95% and 99% in the locality model when compared to the normal case. For example, at N=500, the locality model has an average of 117.1 ms, 90% is 118.83 ms, 95% is 120.11 ms, 99% is 118.83 ms. For the same value of N, the normal case has an average of 142.75 ms, 90% is 171.29 ms, 95% is 172.74 ms, 99% is 174.49 ms. This trend increases as we continue to increase the values of N.

- Inter-quartile Range – The boxes in Figure 6.8 are much smaller than the boxes in Figure 6.9 which show that the inter-quartile range (IQR) for the locality model is much smaller than the normal case. At N=500, the IQR for the locality model is equal to 2.4 ms whereas for the normal case it is equal to 34.82 ms.

All these statistics show that the dispersion of execution times is less in the case of the locality model when compared to the normal case.

6.4.1.2 Local vs. Remote

The producer/consumer example measured the difference between the local and remote case. It is the important to note that the locality model was used in both cases and in the remote case, the producer was allocated on N0 of the reference architecture along with the shared object and the consumer was allocated on N1. The speed between N0-N1 was set at 200MHz. In the local case both producer consumer were kept local along with the shared object. The following presents the dispersion values measured in this test:

- Standard deviation –The standard deviation for both the local case and the remote case are shown in Table 6.1 and Table 6.2 respectively for the producer consumer test. The standard deviation values in the remote case are roughly about double than the values for the local case. At N=500, the standard deviation value for the remote case is 2.42 us compared to 1.58 us for the local model. This standard deviation in the remote case is roughly about 1.6 times the standard deviation values of the local case. At N=20000, the standard deviation in the remote case is 50.71 us compared to locale case's 26.78 us which is approximately about double.

- Percentiles – The percentiles presented in the Table 6.1 and Table 6.2 show that the average value is much closer to the 90%, 95%and 99% in the local case when compared to the remote case. For example, at N=500, the local case has an average of 16.96 us, 90% is 18.01 us, 95% is 18.75 us, 99% is 20.05 us. For the same value of N, the remote case has an average of 24.97 us, 90% is 26.36 us, 95% is 31.57 us, 99% is 33.23 us. This trend continues as we increase the value of N.

- Inter-quartile Range – The boxes in Figure 6.3 are much smaller than the boxes in Figure 6.4 which show that the inter-quartile range (IQR) for the local case is much smaller than the remote case. At N=500, the IQR for the local case is equal to 0.98 us whereas for the remote case it is equal to 2.28 us.

All these statistics show that the dispersion of execution times is less in the local case when compared to the remote case even though the locality model was being used for both cases.

6.4.1.3 Locality vs. Load Balancing

The prime sieve example was repeated to find how load balancing and locality affect the execution times. For a fixed value of N, i.e. N=15000, we have three cases as described in the Section 6.3.2.2. The following presents the dispersion values measured in this test:

- Standard deviation – The standard deviation for all three cases is presented in the Table 6.5. The standard deviation value for case 1 is 74.63 ms, case 2 is 51.28 ms and case 3 is 115.59 ms.

- Percentiles – The percentiles for all three cases is presented in the Table 6.5. For case 1, the average value is 3466.78 ms, the 90% value is 3562.79 ms, the 95% value is 3625.91 ms and the 99% value is 3665.54 ms. For case 2, the average value is 2890.94 ms, the 90% value is 2972.77 ms, the 95% value is 2999.62 ms and the 99% value is 3032.96 ms. For case 3, the average value is 5310.33 ms, the 90% value is 5485.04 ms, the 95% value is 5555.56 ms and the 99% value is 5705.63 ms. Therefore the percentile values are closest to the average in case 2, followed by case 1 and lastly case 3.

All these statistics show that the dispersion values are least when there is proper load balancing along with locality. The dispersion is the highest when the load is unbalanced on the system.

6.4.2 Temporal Isolation

The locality model provides temporal isolation to threads in an ExecutionSite from any external threads. Let us consider the following example:

```
import javax.realtime.*;

public class test1 extends RealtimeThread {
public void run() {
  Neighbourhood n1 = Locality.getCurrentNeighbourhood();
  Locale[] locs = n1.getLocales();

// Parameters are created for the ExecutionSite showing the
// requirements of the ExecutionSite
  int numProcessors = 1;
  RelativeTime [] budget = new RelativeTime [1];
  RelativeTime [] period = new RelativeTime [1];
  RelativeTime [] budget1 = new RelativeTime [1];
  RelativeTime [] period1 = new RelativeTime [1];
```

```
  budget[0]= new RelativeTime(40,0);
  period[0]= new RelativeTime(100,0);
  budget1[0]= new RelativeTime(40,0);
  period1[0]= new RelativeTime(100,0);
  MemoryParameters mp1 =
    new MemoryParameters(2048, 2048,2048);
  MemoryParameters mp2 =
    new MemoryParameters(2048, 2048,2048);

  ExecutionSite site1 = Locality.createExecutionSite(locs[0],
    budget, period, numProcessors, mp1);
  ExecutionSite site2 = Locality.createExecutionSite(locs[0],
    budget1, period1, numProcessors, mp2);

// PriorityParameters are created to pass as parameters to
// the factory to create threads.
  PriorityParameters pri1=new PriorityParameters(25);
  PriorityParameters pri2=new PriorityParameters(20);

// PeriodicParameters object is created to pass as
// parameters to the factory to set the period of a periodic
// thread.
   RelativeTime start = new RelativeTime(0, 0);
   RelativeTime C = new RelativeTime(20, 0);
   RelativeTime D = new RelativeTime(100, 0);
   RelativeTime T = new RelativeTime(100, 0);
   PeriodicParameters releaseParams = new
     PeriodicParameters(start,T,C,D,null,null);

// Create the higher priority thread on site1. The thread
// is created to occupy the processor but it will be suspended
// as soon as the the budget expires. It will be able to execute
// as soon as the budget is replenished.
  RealtimeThread rtt1 = site1.createRealtimeThread(pri1, null,
    null, null, new misBehaving());

  RealtimeThread rtt2 = site2.createRealtimeThread(pri2,
```

```
    releaseParams, null, null, new periodicThread());


        rtt1.start();
        rtt2.start();
}
public static void main(String[] args) {
                test1 rt = new test1();
                rt.start();
} }
```

```
// This class implements a runnable which executes
// indefinitely in a loop to keep the CPU busy.

public class misBehaving implements Runnable {

public void run() {
  while(true);
} }
```

```
import javax.realtime.*;

public class periodicThread implements Runnable {

public void run() {
 RealtimeThread rTT =
 RealtimeThread.currentRealtimeThread();
 Clock c1 = Clock.getRealtimeClock();;
 for (int i = 0; i < 1000; ++i) {
  rTT.waitForNextPeriod();
  AbsoluteTime at=  c1.getTime();
  System.out.print("\tPeriod:" + i + "\t");
  System.out.println(at.getMilliseconds()+ " ms "
          + at.getNanoseconds() + " ns");
 }      } }
```

This example can be explained as following:

1. Two ExecutioneSites, site1 and site2, are created on Locale 0. Both site1 and site2 require some temporal guarantees which are requested in the form of

budget and period arrays. The guarantees required in this case are only on one processor. We assume that both ExecutionSites are guaranteed budget on the same processor. This will cause maximum intereference between both ExecutionSites.

2. A thread is created on each ExecutionSite, rtt1 and rtt2. rtt1 has a higher priority than rtt2. However, since both threads belong to different ExecutionSites, their priorities cannot be compared.

3. Both threads will execute upto a maximum of their allocated budget in that period. In case any thread exhausts the budget, it will be suspended until the next replenishment period. Therefore, rtt1 which is the misbehaving thread in the example will be suspended once it exhausts the budget allowing rtt2 to execute even though both threads are executing on the same processor.

The Locality Model is capable of providing temporal isolation, however, the implementation does not produce the correct behaviour on Linux. Higher priority threads keep on executing even after exhausting the budget not allowing any CPU time for lower priority ones. Therefore, with the existing support in Linux (throttling mechanism), temporal isolation cannot be provided for real-time applications which require period values that are less than one second.

6.5 Overheads

This section outlines the overheads of the locality model and the overheads of the prototype implementation.

6.5.1 Architecture Representation

The overhead of building the architectural representation is a one time overhead which is generated during the initialization of the JVM. It is independent from the application that will execute on the JVM. This is an overhead which is introduced in the locality model and does not exist in standard RTSJ JVMs.

Test Settings The execution time of the *Platform.buildPlatform(memory, devices)* is measured for the reference cc-NUMA architecture. The prototype implementation has been based on the hwloc library and implementation of the Architecture Representation has been detailed out in Chapter 5.

Results Figure 6.11 shows the time taken to build the representation of the architecture. Table 6.6 lists the statistics for the executions times.

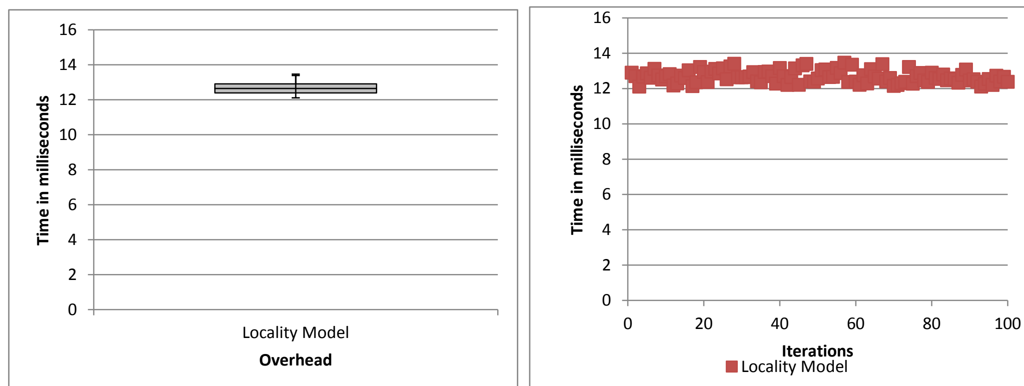


Figure 6.11: Architecture representation overheads

Results Analysis No comparisons can be provided because the overhead in introduced by the locality model. Overheads introduced during the initialization phase of the runtime before the application has actually started executing are usually not very critical because of the nature of real-time applications where tasks are usually periodic and execute for a long period of time. However, this overhead can be very critical for small applications (in terms of execution times).

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	12.68	0.35	100	12.11	13.17	13.32	13.41	13.47

Table 6.6: Architecture representation overhead (milliseconds) statistics

6.5.2 Application Model

In this subsection, the overheads for the basic operations of the locality model are measured and analyzed.

6.5.2.1 Creating Places

Places are created for the reference architecture. A Place is created on each Locale, which includes the creation of a cgroup and creating physical memory areas on each Locale.

Test Settings In this experiment, 4 Places are created and a physical heap and a physical immortal memory area is created each of size 2 MB. The time is measured for the method Locality.initialize(...).

Results Figure 6.12 shows the overhead that is generated as a result of creating Places on a cc-NUMA system. Table 6.7 provides a statistical analysis of the timings.

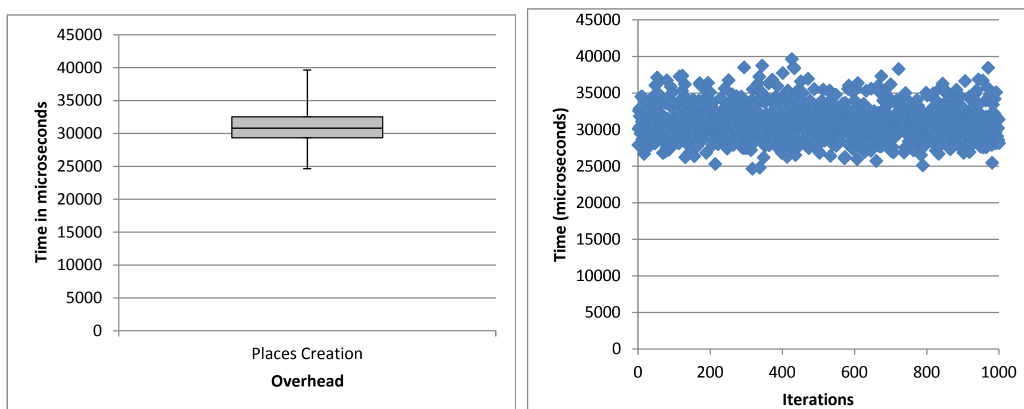


Figure 6.12: Places Creation

Results Analysis This overhead is similar to the overhead of building the architecture representation of the system because both occur before the application has any chance to execute. We consider these overheads non-critical in the sense that they will be generated when the platform is being setup for the application and will not cause any deadline misses or any unpredictable delays when real-time tasks (with hard or soft timing constraints) are executing.

No comparisons can be provided, however, the overheads of creating Places in the prototype implementation can be divided into the following two major costs:

1. The creation of cgroups. The cgroup library has very high latencies that have been presented in Appendix E.1. The sizes of the memory areas to be created. Section 6.5.2.5 shows the the cost to create a memory area in the locality model.

In addition, this overhead also depends on the number of Locales on the system. More Locales means more cgroups to be created and memory areas to be created.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	31043.08	2413.77	1000	24650	34207.7	35325.7	37264.39	39644

Table 6.7: Places creation execution times in microseconds

6.5.2.2 Creating an ExecutionSite

The overhead to create an ExecutionSite is measured in JRate in the locality model. The overhead includes the time taken to perform the admission control mechanism and if the Locale is not defined, the selection of a Locale.

Test Settings We measure the time taken for Locality.createExecutionSite(...).

Results Figure 6.13 gives the timings of creating an ExecutionSite. Table 6.8 presents the statistical analysis of the execution timings measured for the creation of the ExecutionSites.

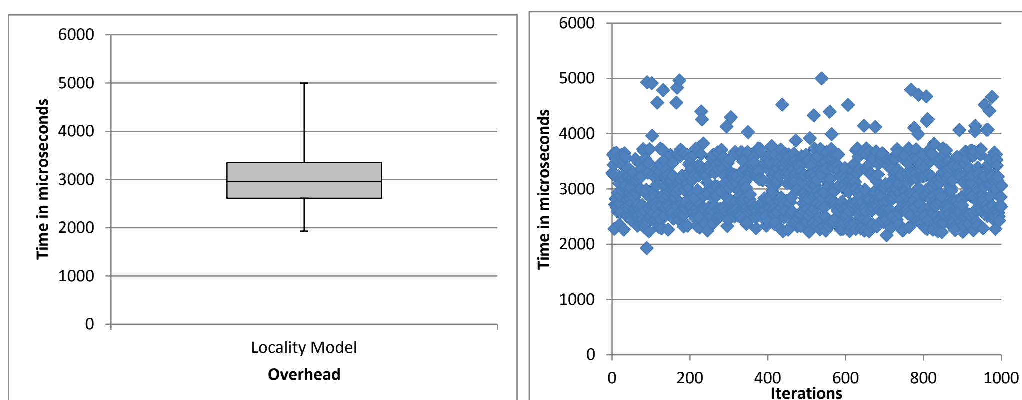


Figure 6.13: ExecutionSites Creation

Results Analysis Table 6.8 provides a statistical analysis of the time to create an ExecutionSite. The overhead includes the admission control for the ExecutionSite on a Locale. The reason for the large dispersion of values is because if admission control fails on one Locale then it starts again on another Locale. A simple admission test is used described in Appendix D.2.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	3012.6	499.07	1000	1930	3622.3	3723.15	4563.03	4999

Table 6.8: ExecutionSites Creation Execution Times in Microseconds

6.5.2.3 Realtime Thread Creation

The overhead to create a RealtimeThread is measured in JRate with and without using the locality model.

Test Settings Execution times are measured for the following:

1. RealtimeThread rtt = new RealtimeThread(..., Runnable logic)

2. RealtimeThread rtt = ES.createRealtimeThread(..., Runnable logic)

Results Figure 6.14 gives a comparison of the timings to create a RealtimeThread with and without using the locality model. Table 6.9 provides a statistical analysis of the time to create a RealtimeThread.

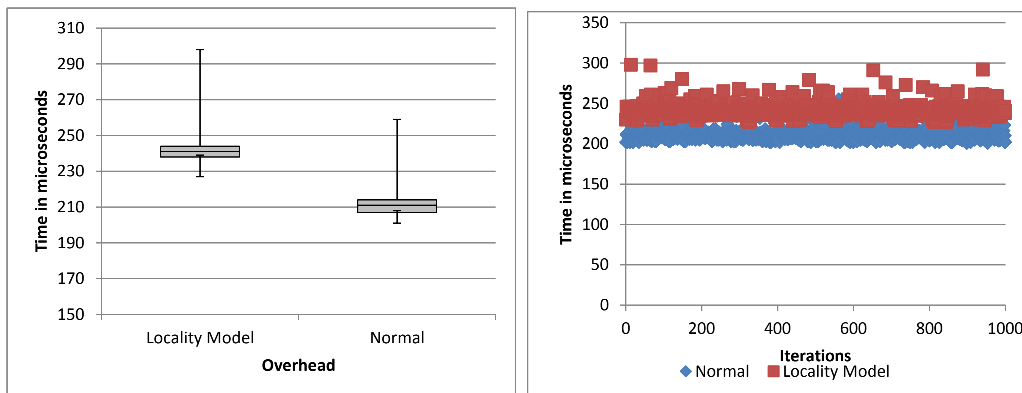


Figure 6.14: Real-time Threads Creation

Results Analysis The following conclusions can be drawn from the measurements presented in Table 6.9 and Figure 6.14.

- Average – The average time taken by the locality model is higher than the normal case.

- Standard deviation – The standard deviation is higher for the locality model but there is very little difference between them.

- Percentiles – There is roughly no difference between the distance of the percentiles from the average for both cases.

The overhead while creating the RealtimeThread is the time used to set the affinity attributes and adding the thread to the feasibility analysis of the reservation. JRate does not actually create a Pthread when this method is called, it is actually created when the start method is called.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	241.65	7.23	1000	227	247	253	268.02	298
Normal	211.53	6.78	1000	201	217	224	234.02	259

Table 6.9: Real-time Threads Creation Execution Times

6.5.2.4 Realtime Thread Startup Latency Without Reservations

In this experiment we measure the time taken to start a thread both in the case of JRate and using the locality model. It is important to note here that the thread created using the locality model belongs to an execution site which has no reservation servers defined. The case where a thread is created on an ExecutionSite with reservation servers has been discussed in Section 6.5.3.2.

Test Settings The following cases are discussed:

1. Normal – A RealtimeThread is created and then we measure the latency to start the thread. This is the time between rtt.start() is called till the thread starts executing.

2. Locality Model – An ExecutionSite is created which has no reservation. A real-time thread is created using the factory, and then it is started. Then time measured for the rtt.start() till the thread starts execution.

Realtime priority is set for the thread is set to make sure that the thread is dispatched without suffering any delays from other real-time threads.

Results Figure 6.15 and Table 6.10 show the startup latencies for real-time threads in both cases.

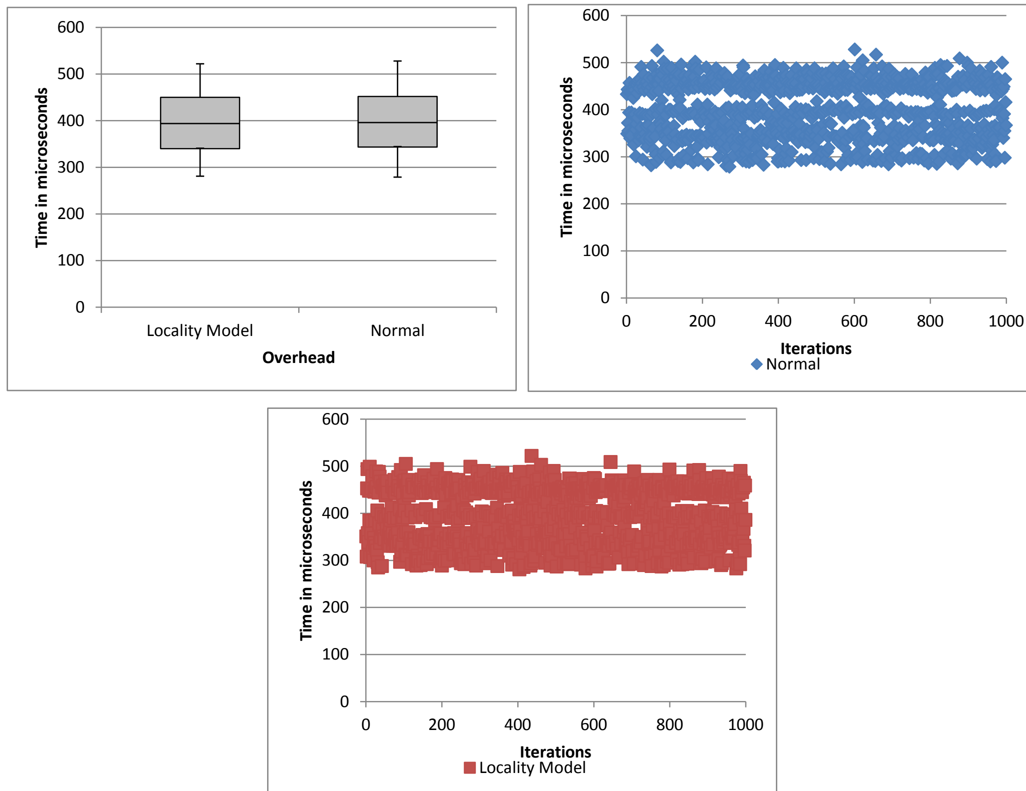


Figure 6.15: Real-time threads startup latency without reservations

Results Analysis There are no overheads in the startup latency when real-time threads are created using Locality Model or Normally using JRate. However, when a thread is started on an execution site with reservation then there is an overhead given in Section 6.5.3.2.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	393.44	59.63	1000	281	462	471	492	522
Normal	397.36	61.1	1000	279	470	478	495.02	528

Table 6.10: Real-time threads startup latency (in microseconds) without reservations statistics

6.5.2.5 Memory Area Creation

The time taken to create the memory area is measured for the locality model and standard JRate.

Test Settings The time is measured for the creation of a MemoryArea which is of

the size of 10 MB . The time is measure for the following method calls:

1. Normal – LTMemory mem = new LTMemory(10*1024*1024)

2. Locality Model – LTPhysicalMemory mem =

   ES.createMemoryArea(0,10*1024*1024);

Results Figure 6.16 shows the time taken by the locality model and the normal case to create the memory area. Table 6.11 provides the statistics analysis of the timings.

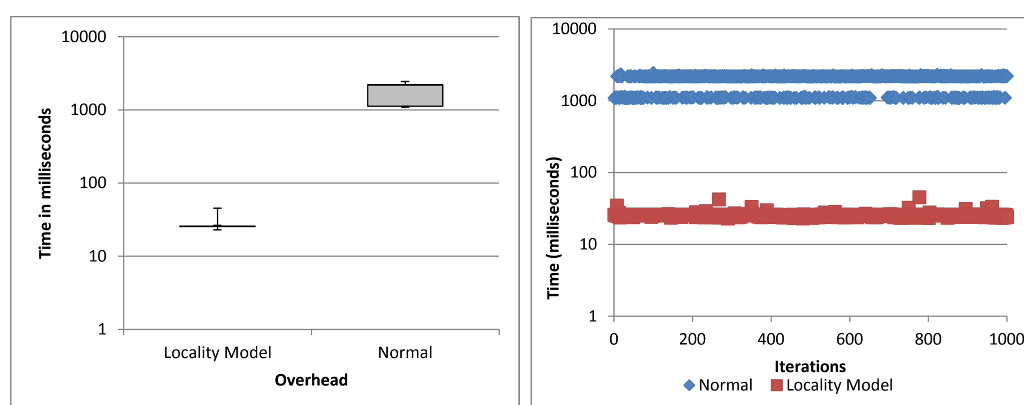


Figure 6.16: Scoped memory Area Creation Timings

Results Analysis The following conclusions can be drawn:

1. Average Measures – The average values show that the memory areas created in the locality model take very little time as compared to the Normal case. One reason for this is locality and the other reason is how both have been implemented. Details of the implementation of the memory areas have been provided in Chapter 5.

2. Dispersion Measures – From the data reported in Table 6.11 and Figure 6.16, it can be seen that the 90%, 95% and 99% values are very close to the average value. Therefore, using the locality model factory to create a memory area is much more predictable on a cc-NUMA system when compared to using the scoped memory area constructor.

Therefore, no overheads are generated to create memory areas, infact the timings are less and more predictable compared to the normal case. The normal case in Figure 6.16 shows two different bands of execution timings. These bands are created because the memory area is created without specifying any node. This results in memory areas being created on nodes having different latencies, hence, forming two different bands.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	25.56	1.17	1000	22.95	25.88	25.97	28.33	45.36
Normal	1899.24	485.7	1000	1090.81	2205.39	2218.2	2271.29	2445.74

Table 6.11: Scoped memory Area Creation Timings (milliseconds) Statistics

6.5.2.6 Allocation Time Test

This test measures the allocation time for objects in an LTMemory area. The results we obtained are presented and analyzed below.

Test Settings The time is measured for the creation of an array of bytes which is of the size 8 MB . The time is measured for memory areas created in the Normal case and the locality model.

Results Figure 6.17 shows the time taken by the locality model and the normal case to create the array of bytes. Table 6.12 provides the statistics analysis of the timings.

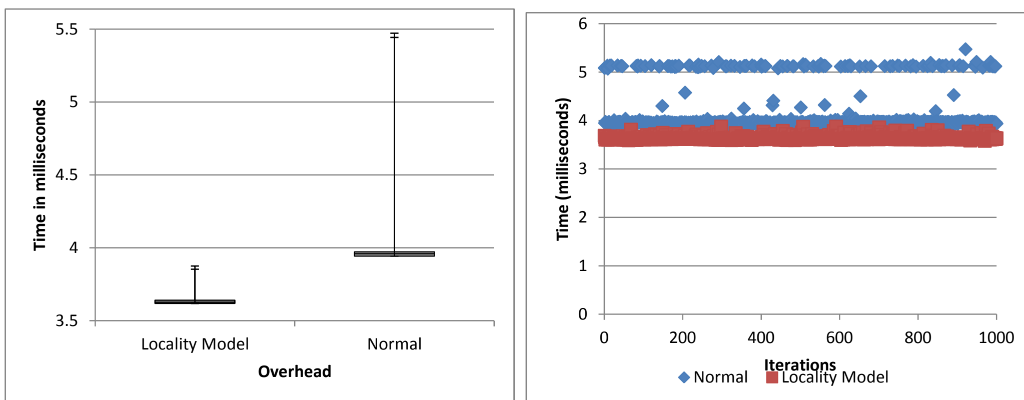


Figure 6.17: Object Allocation Timings

Results Analysis The following conclusions can be drawn:

1. Average Measures – The average values show that the allocation time using the locality model takes less time when compared to the Normal case. Although the difference is little but still the locality model is quicker to allocate the memory area.

2. Dispersion Measures – From the data reported in Table 6.12 and Figure 6.17, it can be seen that the 90%, 95% and 99% values are very close to the average value. Therefore, using the allocation times using the locality model are more predictable on a cc-NUMA system when compared to the Normal case..

Therefore, no overheads are generated to the allocation timings, infact the allocation times are less and more predictable compared to the normal case.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	3.64	0.03	1000	3.58	3.67	3.7	3.79	3.88
Normal	4.1	0.38	1000	3.9	5.11	5.13	5.15	5.47

Table 6.12: Object Allocation Timings (milliseconds) Statistics

6.5.3 Reservation Model

In this subsection we will measure the overheads generated due to the Reservation model. Some of the measures have already been included above when an Execution site was created or a RealtimeThread was created.

6.5.3.1 Creating ReservationServers

ReservationServers are created to provide guarantees. The overheads in the case when no reservation was guaranteed was measured in Section 6.5.2.2. In this case we separately measure the time taken to create a ReservationServer. See Section 5.4 for implementation details.

Test Settings An ExecutionSite is created with some temporal requirements. The scheduler now creates ReservationServer to provide the guarantees. The prototype has been based on cgroups. We measure the time taken to create the reservation server.

Results Figure 6.18 shows the time taken to create a ReservationServer. Table 6.13 provides the statistical analysis of the timings.

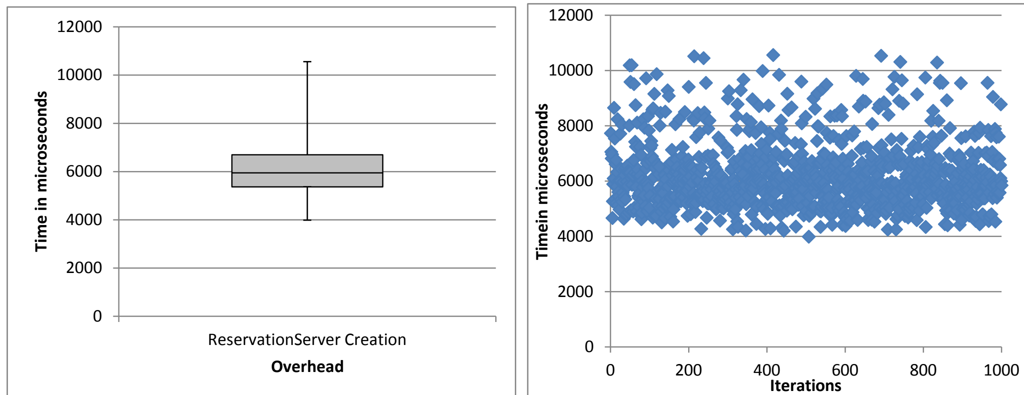


Figure 6.18: ReservationServer Creation

Results Analysis The following conclusions can be drawn:

1. Average Measures – The average values show that creating the reservation server takes a long time. This is because a cgroup is created which corresponds to the ReservationServer of an ExecutionSite.

2. Dispersion Measures – The dispersion measures show a large amount of variance. The standard deviation values are large along with the distances between the the average value and the percentiles (90%, 95% and 99%).

This shows the prototype that has been based of cgroups is not very efficient and it needs to be improved once more support is added in Linux for better resource management.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	6205.38	1221.33	1000	4114	8026.1	8783.3	10004.24	10592

Table 6.13: Creating ReservationServer Timings (microseconds)

6.5.3.2 Realtime Thread Startup Latency using Reservations

The latency is measured to start the thread. This is the same experiment as described in Section 6.5.2.4. However, in this case the thread is being started on an ExecutionSite which has a reservation.

JRate creates the RTT when the start() method is called for a thread. The priority and the affinity attributes are already set for the pthread. However, cgroups are Linux only mechanism and the Linux thread id (tid) is used to attach the thread which can only be done after the thread has actually started. For more implementation details see Section 6.5.2.4.

Results Figure 6.19 shows the time taken to start a thread and attach it to a ReservationServer compared to the Normal case. Table 6.14 provides the statistical analysis of the timings.

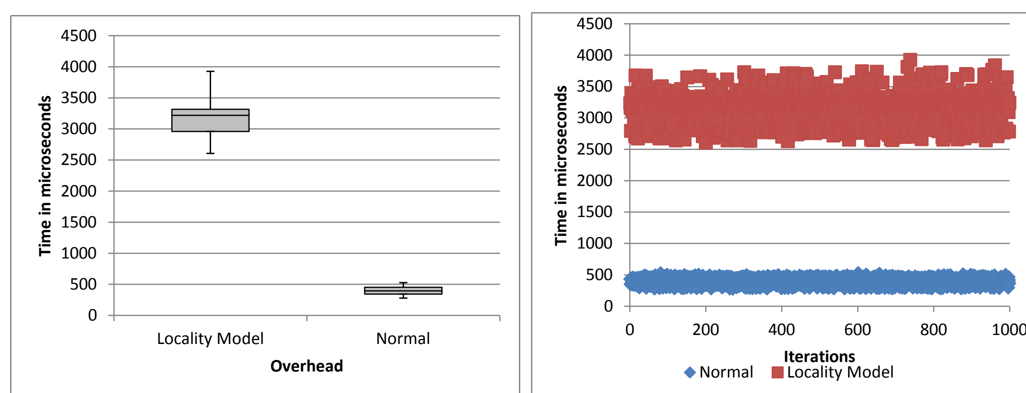


Figure 6.19: Real-time Threads Startup Latency

Results Analysis The following conclusions can be drawn:

1. Average Measures – The average values show that starting the threads takes around 8 times more execution time.

2. Dispersion Measures – The dispersion measures presented in Table 6.14 also show an increase in the amount of variance.

	Avg.	STD	Count	Min	90%	95%	99%	Max
LM	3161.22	245.95	1000	2606	3418	3561.3	3712.2	3926
Normal	397.36	61.1	1000	279	470	478	495.02	528

Table 6.14: Real-time Threads Startup Latency (in microseconds) Statistics

Breaking up this cost of the startup, we find that the major contribution to the overhead is caused by calling cgroup_attach_thread(). The timings for this

call given are given in the Appendix F. Child threads inherit the cgroups. Optimizations are made to reduce the overheads based on the fact that we do not call cgroup_attach_thread().

6.6 Summary

This chapter presents the evaluation of the locality model. It evaluates the locality model in terms of effectively expressing the cc-NUMA architecture to develop portable real-time Java applications. The initial goals of the thesis were to build to a locality model which would combine real-time and parallel computing in real time Java. The main objectives of the thesis were to be able to create a platform which would provide programmability, portability, performance and predictability.

This chapter reviews the locality model in terms of these objectives. It has shown that the locality model provides a high level programming model to write portable application. Experiments show that the locality model increases the performance of a multi-threaded application. It also provides a more predictable platform on a cc-NUMA system.

The locality model has small overheads, in case of a parallel environment, the overheads are not noticeable. On the other hand, the overheads in case of reservations are high because of the libcgroup library. As a result, the implementation is not very efficient when working with cgroups.

Chapter 7

Conclusions and Future Work

This is the last chapter of the thesis which concludes the work that has been done. It also provides an outline for future work. Section 7.1 summarizes the work presented in the thesis. Section 7.2 discusses the contributions of this thesis. Section 7.3 provides the future directions that can be taken after this work.

7.1 Summary

The RTSJ provides a high level real-time programming platform. It is an ideal platform to be extended for new multiprocessor architectures. A basic set of requirements is extracted from existing parallel programming languages and the real-time systems community to be able to develop large scale parallel applications in RTSJ on cc-NUMA architectures. For non-real-time and non-performance-critical Java application, cc-NUMA architecture presents no new problems. The presence of cache coherence means that programmers need not be concerned with where components of their program execute. They can delegate full responsibility to the JVM.

For applications that have predictability and performance requirements, delegation to the JVM will not necessary produce acceptable results, as the JVM is unable to exploit application-specific knowledge. Whilst a JVM may monitor a running application and try to optimize its performance (analogous to JIT compilation), this inevitably undermines predictability.

The existing mechanisms of RTSJ which can be used to support cc-NUMA sys-

tems are analyzed, especially RTSJ's physical memory model and the AffinitySet class. It is argued that although both used together can provide the programmer a way of controlling the allocation policies, however, still it falls short of handling cc-NUMA systems.

To support the programming of cc-NUMA architectures in the RTSJ, the programming model is extended to allow the programmer to express locality constraints. A new locality model is presented which is based on the existing physical memory model along with some extensions and the resource reservations.

The *Architectural representation* defines new abstractions that are defined in the Locality model and provide transparency to the JVM to make it aware of the hardware resources available for the application. Programmers can query the runtime for any information on the topology of the NUMA system. A Platform class is provided as an interface to the architecture representation. The representation is portable because it is built by the runtime based on existing resources. It allows programmers to allocate schedulables/objects local to any available device.

The *Application model* provides an *ExecutionSite* class. It is the main class around which the whole model revolves. The ExecutionSite contains a group of threads which can be scheduled by a local scheduler and can be provided with temporal and spatial guarantees. The runtime provides dynamic allocation of the ExecutionSite. The objective of having an ExecutionSite is to provide portable performance by keeping related objects/schedulables together and guaranteeing them resources. Instances of ExecutionSite class can be independently analyzed because of the guarantees.

The resource reservation model provides resource guarantees to the application. Resource guarantees makes timing requirements of a soft-real time system portable. Any system that can provide the required guarantees can host the real-time system. For timing requirements, the resource reservation model provides budgets on individual processors. This approach has less overhead when compared to a global budget and it scales much better.

A prototype implementation has been developed for the Locality model. The model is evaluated using performance measures. Results show that the Locality

model can be used by programmers to provide locality which considerably improves the performance.

An existing schedulability analysis is used to show that the locality model is compliant with existing scheduling theory for multiprocessors. Using the model on a cc-NUMA system narrows down the range of execution times, hence providing a much more predictable model.

In summary, the Locality model on RTSJ provides a high level platform for creating highly parallel applications on cc-NUMA systems with portable performance.

7.2 Contributions

The following major contributions have been made in this thesis:

1. The RTSJ's existing mechanisms which can be used to support cc-NUMA systems is analyzed, especially RTSJ's physical memory model and the AffinitySet class. It is argued that although both used together can provide the programmer a way of controlling the allocation policies, however, still it falls short of handling cc-NUMA systems. Also it cannot be used for guaranteeing resources on a platform shared by many applications.

2. A new locality model is presented which is based on the existing physical memory model along with some extensions and the resource contracts that have been outlined by [Wellings et al., 2009]. Compared to the existing support provided for cc-NUMA systems in the RTSJ, the locality model provides the following:

    - New abstractions which make the RTJVM aware of the hardware resources available for the application. The primary requirement for introducing new language abstractions is to be able to manage the allocation policies of threads and objects in order to minimize the non determinism caused by the memory distribution in the NUMA system. In other words, tightly coupled threads and objects should be kept local to each other to

avoid remote accesses, which cause added delay. Unrelated threads and objects should be separated out to avoid any competition for resources.

- Applications can be mapped statically based on the input of the programmer or it can be mapped dynamically based on the requirements of the application.
- Temporal guarantees are provided to applications based on resource reservations.

3. A prototype implementation has been developed. An empirical analysis is presented which evaluates the locality model.

7.3 Future Work

The work presented in this thesis combines two different areas: real-time Java and parallel programming. This integration has been getting attention recently due to the influx of multiprocessors which opens a number of research areas for the future. Some of the future work that can be done after this thesis is as following:

- Code Locality – In Java, code is in the form of byte-code. This bytecode is spread across a number of Java class files which can be dynamically loaded by the JVM from the disk, internet or any other location. Loading the class files is the responsibility of a class loader which needs to make sure that the correct class file is loaded which will not jeopardize the security of the Java application and the system as a whole. Once the class file is loaded, it is thought to be present in a logical memory area called the method area. The structure and organization of the information in the method area is not defined by Java rather it is left to the implementation to decide how the method area is implemented. Code locality is as important as data locality, if not more so. So far our model has looked at placement of threads and placement of objects. However, code for a schedulable object is in the method area, therefore it is necessary to make sure that code also remains local to the thread.

In the locality model, it is assumed that each execution site has a local method cache as shown in Figure 7.1. This method cache is a logical unit and can either be explicitly implemented in software or it can be representing the actual hardware instruction cache attached to the processor. The method area in most cases has read only classes which can be replicated across the platform. In systems with large caches, classes are loaded on the first cache miss and remain in the cache unless the system is very heavily loaded with computational processes. In such a case, the hardware cache does act as a cache and there is no need to implement one in software. In cases where there are no hardware caches, an implementation can also implement a method cache on the same principle of a cache to provide local access to code. Method caches have been implemented in the case of hardware based Java processor [Schoeberl, 2004]. Method caches can be implemented in software in the following cases:

- The cache is too small or there is no cache memory and the instructions are being fetched from a remote memory very frequently.
- A high speed scratchpad memory is present.

The case of a scratchpad is interesting because it is also an on-chip high speed memory like the cache but it can be accessed explicitly by the processor. Therefore, all the contents in the scratch pad memory are directly under the control of software, unlike the normal cache where the hardware implicitly controls the cache. [Wellings and Schoeberl, 2009] have used the physical memory model to allocate physical scoped memory areas on scratchpads.

A complete analysis needs be done for the Locality of code and how it affects the timings and performance of an application on architectures that may or may not have large caches.

- Garbage Collection – In real-time systems, garbage collection has been a challenge because of the unpredictable delays it can cause. A HeapPhysicalMemory area is proposed in Chapter 4. The HeapPhysicalMemory requires the garbage collector to support a distributed shared memory heap. The general

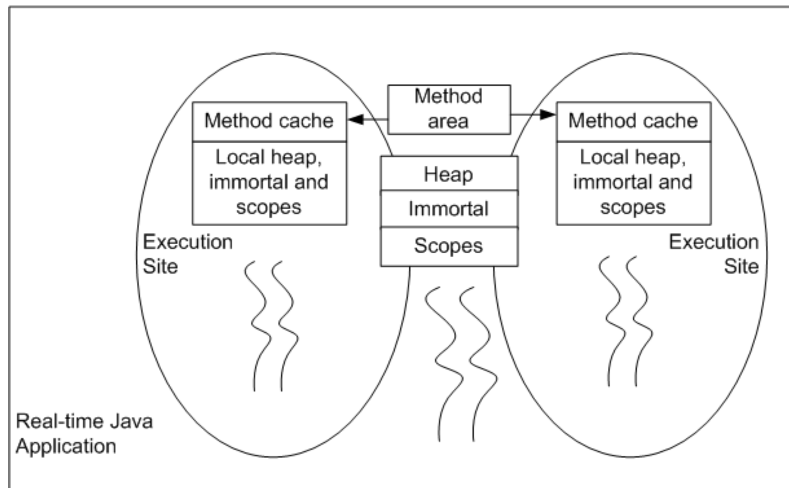


Figure 7.1: Execution sites in a real-time Java application

behaviour of the garbage collector has been described, however, a detailed and more thorough study is required to design and implement garbage collection for such a system.

- Heterogeneous Architectures – The Locality model assumes cache coherence and a single address space. However, modern architectures with extreme levels of parallelism do not provide these because of scalability issues. Therefore, a distributed address space exists which is similar to the case of Java DSMs. However, in this case we have very fast interconnects connecting the nodes. The Locality model has provided a base which can be extended for these architectures.

- Application Mapping – The Locality model has been based on on the premise that the programmers have the knowledge of the relationship between computations and data. Therefore, they can distribute related computations/data into separate components. We do not use any static analysis to find relations among threads/objects, or the relationship between ExecutionSites. It would be interesting to see how mapping can be optimized based on relationships between ExecutionSites.

- Adaptive scheduling – Adaptive scheduling is a very flexible mechanism for soft-real time systems. The scheduling parameters are gradually adapted to

the requirements of the application improving the overall QOS without going through any of the analysis required for real-time systems. It would definitely have been very useful in the Locality model because the current model depends on the cost (average or worst case execution time (WCET)) parameter of the schedulable to check for the feasibility analysis. This cost, however, is not portable because the execution times of a schedulable will be different on different architectures. With adaptive scheduling, cost will be redundant because budget can be fine tuned for the schedulable to ensure all timing requirements are met.

Appendix A

Memory Access Timings on Cache Coherent NUMA Systems

This appendix tries to give an insight into the difference of memory access timing of a cc-NUMA system. The cc-NUMA system that is used to measure these timings is based on the AMD Opteron processor that has been described in Chapter 3. The Reference architecture is a 4 node system where nodes are connected by a hypertransport interconnect. The hypertransport interconnects have fixed speed except for N0-N1 interconnect which can be set speeds from 200 MHz to 1 Ghz.

The TAU benchmark[1] measures the time taken by the memcopy() on a cc-NUMA system. The measurements are calculated by allocating memory on one node and then using memcopy() from all CPUs on the system. The TAU profile code has been slightly changed to use the NUMA API to make the memory allocations which is shown in the following listing:

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <numa.h>
```

[1] http://www.nic.uoregon.edu/tau-wiki/Guide:Opteron_NUMA_Analysis

```
#define MEM_MB 512
#define MEM_SIZE MEM_MB*1024L*1024L
#define ITER 40
#define numNodes 4

double getTime() {
  struct timeval tp;
  static double last_timestamp = 0.0;
  double timestamp;
  gettimeofday (&tp, 0);
  timestamp = (double) tp.tv_sec * 1e6 + tp.tv_usec;
  return timestamp;
}

int getNumCPU() {
  cpu_set_t mask;
  if (sched_getaffinity(0,sizeof(cpu_set_t),&mask)) {
    fprintf (stderr, "Unable␣to␣retrieve␣affinity\n");
    exit(1);
  }
  int nproc = 0;
  for(int i=0; i<CPU_SETSIZE; i++) {
    if( CPU_ISSET(i,&mask) ) {
      nproc++;
    }
  }
  return nproc;
}

void memtest(char *ptr) {
  for (int i=0; i<ITER; i++) {
    memcpy(ptr, ptr+(MEM_SIZE/2),MEM_SIZE/2);
  }
}

void setCPU(int cpu) {
  cpu_set_t mask;
```

```
  CPU_ZERO(&mask);
  CPU_SET(cpu, &mask);
  sched_setaffinity(0, sizeof(cpu_set_t), &mask);
}

void test(int node, int nproc) {
  setCPU(node);
  char *ptr ;
  ptr=(char*)numa_alloc_onnode(MEM_SIZE,node);
  if (!ptr) {
    fprintf (stderr, "failed␣to␣malloc\n");
    exit(1);
  }
  printf ("\nMemory␣allocated␣on␣node␣\%d\n", node);
  // make sure it all gets paged in
  for (long j=0; j<MEM_SIZE; j++) {
    ptr[j] = j;
  }
  for (int i = 0; i < nproc; i++) {
    setCPU(i);
    double start = getTime();
    memtest(ptr);
    double end = getTime();
    printf ("\%d:␣time␣=␣\%G␣seconds\n", i, (end - start)
       / (1000*1000));
  }
  if (ptr)
  {free (ptr);}
}

int main (int argc, char **argv) {
    int nproc = getNumCPU();
  for (int i = 0; i < numNodes; i++) {
    test(i,nproc);
  }
  return 0;
}
```

A.1 Comparing Local And Remote Memory Accesses

Figure 1.3 shows the difference of average timings that *memcpy()* takes when called on local and remote memory. The code of the experiment has been presented in the above listing, however, the experiment has been run for different sizes for memory as shown in Table A.1.

Table A.1 shows the timings that been been presented in Figure 1.3. The configuration for both cases have been discussed as following:

1. Local – The calling thread is set an affinity of CPU 0 which is on node 0 (N0). The allocated memory is also on N0.

2. Remote – The calling thread is set an affinity of CPU 4 which is on node 1(N1). The allocated memory is on N1. The interconnect between N0-N1 is set at 200 MHz for maximum NUMA effect.

A.2 Comparing Access Timings for Different Inter-Connect Speeds

In this section, the average access timings are measured for different speeds of the hypertransport interconnect. The timings are measured for the code presented in the above listing. Memory is allocated on N0 and then it is accessed from all processors (CPU0-CPU15).

Figure A.1 shows that at 200 Mhz, the NUMA effect is much stronger and by increasing the speed of the hypertransport, there is relatively little difference between local and remote accesses. Table A.2 shows the timings that been been presented in Figure A.1.

Memory size (MB)	Average local timings (msec)	Average remote timings (msec)
1	0.63	0.935
100	31.941	105.441
200	61.625	210.181
300	91.653	315.298
400	122.231	420.667
500	149.499	525.987
600	181.718	630.978
700	213.159	736.466
800	240.176	841.406
900	270.036	947.289
1000	299.864	1051.624
1100	329.061	1157.036
1200	358.79	1262.172
1300	389.17	1368.537
1400	419.883	1473.261
1500	449.788	1577.477
1600	480.74	1682.466
1700	510.86	1787.171
1800	537.791	1893.184
1900	563.688	1997.752
2000	587.23	2103.004

Table A.1: memcpy() timings (in milliseconds) with N0-N1 interconnect at 200Mhz

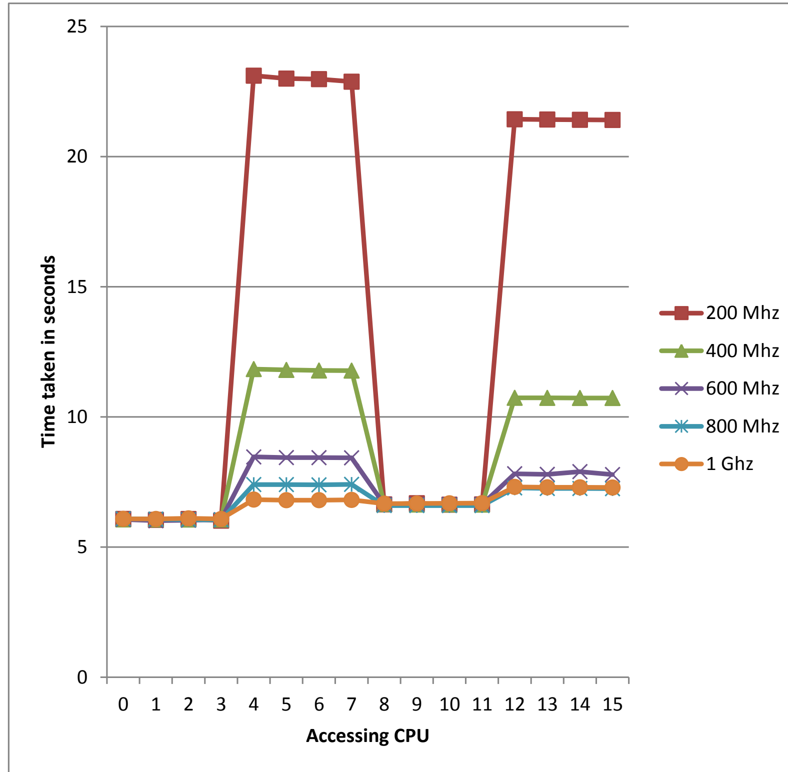


Figure A.1: memcopy() timings(in seconds) for different interconnect speeds

Accessing CPU	200 Mhz	400 Mhz	600 Mhz	800 Mhz	1 Ghz
0	6.0817	6.0649	6.06175	6.0888	6.0847
1	6.0458	6.0639	6.01307	6.0767	6.0811
2	6.0796	6.0592	6.0337	6.0569	6.1045
3	6.0161	6.0594	6.0543	6.0455	6.08005
4	23.1111	11.8398	8.4652	7.4037	6.82737
5	22.9993	11.807	8.4398	7.4026	6.80297
6	22.9789	11.7843	8.4365	7.3926	6.8039
7	22.88	11.7811	8.4307	7.4106	6.81347
8	6.6417	6.6555	6.6412	6.59707	6.6711
9	6.6866	6.6365	6.6404	6.5961	6.6723
10	6.6303	6.6387	6.6413	6.59527	6.6831
11	6.6405	6.6324	6.6391	6.5955	6.6879
12	21.4359	10.7361	7.8188	7.2746	7.3145
13	21.4277	10.7336	7.7956	7.2526	7.2971
14	21.4166	10.7301	7.9001	7.2516	7.2963
15	21.4109	10.7295	7.7872	7.2445	7.2891

Table A.2: memcopy() timings (in seconds) for different interconnect speeds

A.3 Comparing Access Timings for Different NUMA Distances

In this section, the average access timings are measured for different distances in the reference cc-NUMA system. The timings are measured for the code presented in the above listing. Memory is allocated on all nodes (N0-N3)and then it is accessed from all processors (CPU0-CPU15). The hypertransport has been fixed at 1Ghz for this experiment.

Figure A.3 shows that even at 1 GHz (where there is relatively a smaller difference between local and remote as shown in Appendix A.2), there is a difference between access timings for different distances. Table A.3 shows the timings that been been presented in Figure A.3.

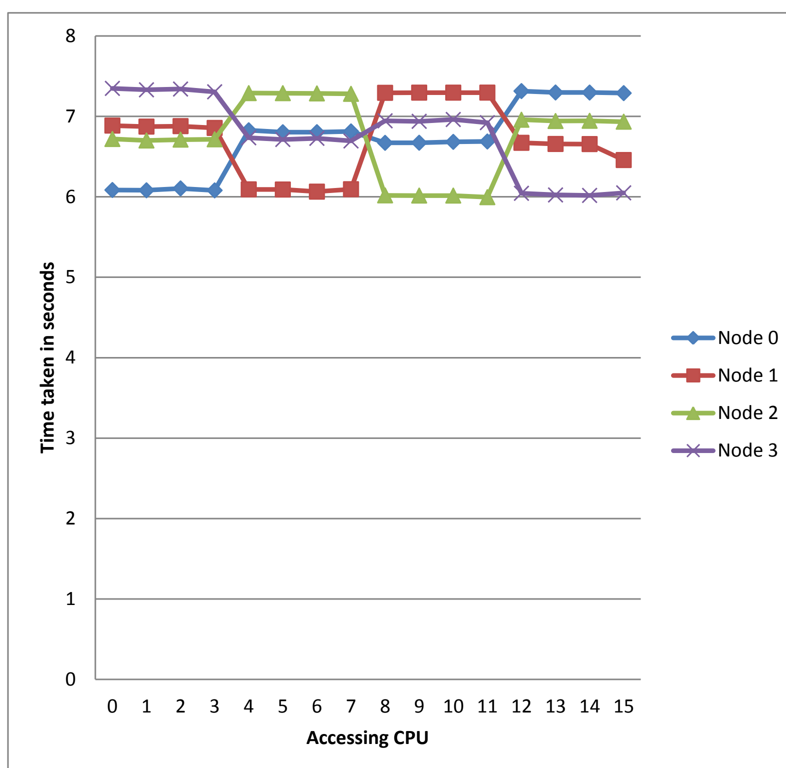


Figure A.2: Comparing Access Timings for Different NUMA Distances

	Node 0	Node 1	Node 2	Node 3
0	6.0847	6.88567	6.71876	7.34818
1	6.0811	6.8728	6.70025	7.33028
2	6.1045	6.87879	6.71103	7.33988
3	6.08005	6.85637	6.71503	7.30562
4	6.82737	6.0927	7.28917	6.73425
5	6.80297	6.09081	7.28875	6.7129
6	6.8039	6.06427	7.28572	6.72728
7	6.81347	6.09453	7.28033	6.69464
8	6.6711	7.29282	6.01796	6.94492
9	6.6723	7.29512	6.01569	6.93868
10	6.6831	7.29513	6.01554	6.96131
11	6.6879	7.29553	5.99562	6.92187
12	7.3145	6.67127	6.95835	6.04381
13	7.2971	6.65693	6.94246	6.02323
14	7.2963	6.65564	6.94517	6.01734
15	7.2891	6.4572	6.93403	6.0493

Table A.3: memcopy() timings (in seconds) with N0-N1 interconnect at 1Ghz

Appendix B

The Producer Consumer Example

This appendix contains the full source for the Producer/Consumer problem that has been presented in Chapter 4. The full source of the example presented in Section 4.2.7.1 is presented in Section B.1. The example presented in Section 4.2.7.2 is presented in Section B.2. The example presented in Section 4.3.5 is presented in B.3.

B.1 Statically Allocating ExecutionSites

```
import javax.realtime.*;
import java.util.LinkedList;


public class ProducerConsumer extends RealtimeThread {

        public void run() {

// The producer and consumer are allocated on two different
// ExecutionSites. Both ExecutionSites are manually allocated
// on different Locales.

// We retrieve the Locales to create the ExecutionSite

        Neighbourhood n1 =
        Locality.getCurrentNeighbourhood();
```

```
        System.out.println(n1);
        Locale[] locs = n1.getLocales();

// We create two ExecutionSites on separate Locales

        ExecutionSite site1 =
        Locality.createExecutionSite(locs[0]);

        ExecutionSite site2 =
        Locality.createExecutionSite(locs[1]);

// We create a Pinnable Memory Area on the first
// ExecutionSite. Then we create the shared object and
// set as a portal.
        long MEMSIZE = 10 * 1024 * 1024;
        final int worksize = 1000;
        int workload = 100;
        final LTPinnableMemory mem =
        site1.createMemoryArea(2, MEMSIZE);
        mem.enter(new Runnable() {
            public void run() {
                mem.pin();
                BufferObject sharedBuffer =
                new BufferObject(worksize);
                mem.setPortal(sharedBuffer);
            }});

// Creating the Producer. The Producer will execute in
// the newly created memory area and will write for
// workload = 100 times in the shared object.
        RealtimeThread pro =
        site1.createRealtimeThread(null, null, null, mem,
                new Producer(mem, workload));

// Creating the Consumer. The Consumer will execute in
// the newly created memory area and will read for
```

```
// workload = 100 times in the shared object.
        RealtimeThread con =
        site2.createRealtimeThread(null, null, null, mem,
                new Consumer(mem, workload));
        pro.start();
        con.start();
    }

// The main function creates a real-time thread because
// a normal Java thread cannot enter a scoped memory area.
// The real-time thread will be created which will setup
// the scoped memory area and create threads in
// ExecutionSites. The created thread will be placed on
// the cc-NUMA by the runtime on any of the processors.

    public static void main(String[] args) {
        ProducerConsumer rt = new ProducerConsumer();
        rt.start();
   }}


class Consumer extends RealtimeThread {

  int worksize;
  int workload;
  final LTPinnableMemory mem;

  public Consumer(LTPinnableMemory mem, int workload) {
        this.mem = mem;
        this.workload = workload;
   }

  public void run() {
        BufferObject sharedBuffer = (BufferObject) mem.getPortal();
        int consume = 0;
        while (consume < workload) {
                synchronized (sharedBuffer) {
```

```
                        try {
                                while (sharedBuffer.producing) {
                                        sharedBuffer.wait();
                                }
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                DataBlock block = sharedBuffer.Consume();
                consume++;
                sharedBuffer.producing = true;
                sharedBuffer.notifyAll();
}}}}

class Producer extends RealtimeThread {

 int worksize;
 int workload;
 final LTPinnableMemory mem;

 public Producer(LTPinnableMemory mem, int workload) {
                this.mem = mem;
                this.workload = workload;
        }

 public void run() {

        BufferObject sharedBuffer = (BufferObject) mem.getPortal();
        int produce = 0;
        while (produce < workload) {
           synchronized (sharedBuffer) {
                  try {
                        while (sharedBuffer.producing == false) {
                                sharedBuffer.wait();
                        }
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
```

```
                    sharedBuffer.Produce(new DataBlock(worksize));
                    produce++;
                    sharedBuffer.producing = false;
                    sharedBuffer.notifyAll();
}}}}

class BufferObject {

        public boolean producing = true;
        int workload;
        LinkedList buf = new LinkedList();

        public BufferObject(int workload) {
                this.workload = workload;
        }

        public int getWorkload() {
                return this.workload;
        }

        public void Produce(DataBlock dataBlock) {
                buf.add(dataBlock);

        }

        public DataBlock Consume() {
                DataBlock block =
        (DataBlock) buf.removeFirst();
                return block;
} }

class DataBlock {

  public DataBlock(int size) {
                byte[] bytes = new byte[size];
} }
```

B.2 Allocating ExecutionSites Dynamically by the Runtime

```
import javax.realtime.*;
import java.util.LinkedList;

public class ProducerConsumer extends RealtimeThread {

        public void run() {

// We create an ExecutionSite which is mapped by the
// runtime.
        ExecutionSite site =
        Locality.createExecutionSite(null);

// We create a Pinnable Memory Area on the
// ExecutionSite. Then we create the shared object and
// set as a portal.
        long MEMSIZE = 10 * 1024 * 1024;
        final int worksize = 1000;
        int workload = 100;
        final LTPinnableMemory mem =
        site.createMemoryArea(2, MEMSIZE);
        mem.enter(new Runnable() {
            public void run() {
                mem.pin();
                BufferObject sharedBuffer =
                new BufferObject(worksize);
                mem.setPortal(sharedBuffer);
            }});

// Creating the Producer. The Producer will execute in
// the newly created memory area and will write for
// workload = 100 times in the shared object.
        RealtimeThread pro =
        site.createRealtimeThread(null, null, null, mem,
                new Producer(mem, workload));
```

```
// Creating the Consumer. The Consumer will execute in
// the newly created memory area and will read for
// workload = 100 times in the shared object.
        RealtimeThread con =
        site.createRealtimeThread(null, null, null, mem,
                new Consumer(mem, workload));

        pro.start();
        con.start();
}

// The main function creates a real-time thread because
// a normal Java thread cannot enter a scoped memory area.
// The real-time thread will be created which will setup
// the scoped memory area and create threads in
// ExecutionSites. The created thread will be placed on
// the cc-NUMA by the runtime on any of the processors.

    public static void main(String[] args) {

        ProducerConsumer rt = new ProducerConsumer();
        rt.start();
    }
}

class Consumer extends RealtimeThread {

  int worksize;
  int workload;
  final LTPinnableMemory mem;

  public Consumer(LTPinnableMemory mem, int workload) {
        this.mem = mem;
        this.workload = workload;
   }
```

```
  public void run() {
        BufferObject sharedBuffer = (BufferObject) mem.getPortal();
        int consume = 0;
        while (consume < workload) {
                synchronized (sharedBuffer) {
                        try {
                                while (sharedBuffer.producing) {
                                        sharedBuffer.wait();
                                }
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                DataBlock block = sharedBuffer.Consume();
                consume++;
                sharedBuffer.producing = true;
                sharedBuffer.notifyAll();
}}}}

class Producer extends RealtimeThread {

 int worksize;
 int workload;
 final LTPinnableMemory mem;

 public Producer(LTPinnableMemory mem, int workload) {
                this.mem = mem;
                this.workload = workload;
        }

 public void run() {

        BufferObject sharedBuffer = (BufferObject) mem.getPortal();
        int produce = 0;
        while (produce < workload) {
           synchronized (sharedBuffer) {
                  try {
                        while (sharedBuffer.producing == false) {
```

```
                                sharedBuffer.wait();
                        }
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  sharedBuffer.Produce(new DataBlock(worksize));
                  produce++;
                  sharedBuffer.producing = false;
                  sharedBuffer.notifyAll();
}}}}

class BufferObject {

        public boolean producing = true;
        int workload;
        LinkedList buf = new LinkedList();

        public BufferObject(int workload) {
                this.workload = workload;
        }

        public int getWorkload() {
                return this.workload;
        }

        public void Produce(DataBlock dataBlock) {
                buf.add(dataBlock);

        }

        public DataBlock Consume() {
                DataBlock block =
        (DataBlock) buf.removeFirst();
                return block;
}  }

class DataBlock {
```

```
  public DataBlock(int size) {
                byte[] bytes = new byte[size];
} }
```

B.3 Allocating ExecutionSites with Reservations

```
import javax.realtime.*;

public class ProducerConsumer extends RealtimeThread {

        public void run() {

// In order to specify the requirements of the Execution Site,
// we create the budget and the period arrays.
        int numProcessors = 2;
        RelativeTime [] budget = new RelativeTime [numProcessors];
        RelativeTime [] period = new RelativeTime [numProcessors];
        budget[0]= new RelativeTime(500,0);
        period[0]= new RelativeTime(1000,0);
        budget[1]= new RelativeTime(500,0);
        period[1]= new RelativeTime(1000,0);

// We create an ExecutionSite which is mapped by the
// runtime. The mapping is based on the required budget
// parameters.

   ExecutionSite site = Locality.createExecutionSite(null,
      budget, period, numProcessors, null);

// We create a Pinnable Memory Area on the ExecutionSite.
// Then we create the shared object and set as a portal.
    long MEMSIZE = 10 * 1024 * 1024;
    final int buffersize = 1000;
    int workload = 1000;
    final LTPinnableMemory mem =
      site.createMemoryArea(2, MEMSIZE);
```

```
        mem.enter(new Runnable() {
                public void run() {
                        mem.pin();
                        BufferObject sharedBuffer =
            new BufferObject(buffersize);
                        mem.setPortal(sharedBuffer);
                }});

// Creating periodic parameters.

  RelativeTime start = new RelativeTime(0, 0);
  RelativeTime C = new RelativeTime(200, 0);
  RelativeTime D = new RelativeTime(1000, 0);
  RelativeTime T = new RelativeTime(1000, 0);
  PeriodicParameters releaseParams = new
    PeriodicParameters(start,T,C,D,null,null);

// Creating the Producer. The Producer will execute in
// the newly created memory area and will write for
// workload = 1000 times in every period. The thread
// will be added to the reservation and will use the
// guarantees provided to the reservation. In the case
// the budget is exhausted, Producer will have to wait
// for the replenishment of the budget.
   RealtimeThread pro = site.createRealtimeThread(null,
     releaseParams, null, mem, new Producer(mem, workload));

// Creating the Consumer with the same timing properties.
   RealtimeThread con = site.createRealtimeThread(null,
     releaseParams, null, mem, new Consumer(mem, workload));

        pro.start();
        con.start();
}

// The main function creates a real-time thread because
// a normal Java thread cannot enter a scoped memory area.
```

```
// The real-time thread will be created which will setup
// the scoped memory area and create threads in
// ExecutionSites. The created thread will be placed on
// the cc-NUMA by the runtime on any of the processors.

public static void main(String[] args) {
                ProducerConsumer rt = new ProducerConsumer();
                rt.start();
        }}




// The Consumer thread class
class Consumer extends RealtimeThread {
int worksize;
int workload;
final LTPinnableMemory mem;

public Consumer(LTPinnableMemory mem, int workload) {
  this.mem = mem;
  this.workload = workload;
}

public void run() {
  BufferObject sharedBuffer = (BufferObject) mem.getPortal();
  int consume = 0;
   while(true){
     this.waitForNextPeriod();
     while (consume < workload) {
       synchronized (sharedBuffer) {
         try {
           while (sharedBuffer.producing) {
             sharedBuffer.wait();
             }
          } catch (InterruptedException e) {
            e.printStackTrace();
         }
```

```
    DataBlock block = sharedBuffer.Consume();
    consume++;
    sharedBuffer.producing = true;
    sharedBuffer.notifyAll();
}  }  }  } }

// The producer thread class
class Producer extends RealtimeThread {

int worksize;
int workload;
final LTPinnableMemory mem;

public Producer(LTPinnableMemory mem, int workload) {
  this.mem = mem;
  this.workload = workload;
}

public void run() {
  BufferObject sharedBuffer = (BufferObject) mem.getPortal();
  int produce = 0;
  while(true){
   this.waitForNextPeriod();
   while (produce < workload) {
     synchronized (sharedBuffer) {
        try {
          while (sharedBuffer.producing == false) {
            sharedBuffer.wait();
            }
          } catch (InterruptedException e) {
            e.printStackTrace();
         }
       sharedBuffer.Produce(new DataBlock(worksize));
       produce++;
       sharedBuffer.producing = false;
       sharedBuffer.notifyAll();
} } }   } }
```

```
// The buffer list
class BufferObject {

public boolean producing = true;
int workload;
LinkedList buf = new LinkedList();

public BufferObject(int workload) {
  this.workload = workload;
}

public int getWorkload() {
  return this.workload;
}

public void Produce(DataBlock dataBlock) {
  buf.add(dataBlock);
}
 DataBlock Consume() {
  DataBlock block = (DataBlock) buf.removeFirst();
  return block;
}  }

// The shared object
class DataBlock {

public DataBlock(int size) {
  byte[] bytes = new byte[size];
  }
}
```

Appendix C

Building JRate on 64bit Systems

Jrate is an extension of GNU GCJ. JRate is required to be build from GCC source files which include the gcc core, g++ and gcj components. the On most systems, jRate can be built using with the usual *./configure and GNU make* mechanism. However, on most recent systems, the following steps are required to built it on a recent Linux version on 64-bit hardware:

- JRate is only compatible with GCC-3.3.x and has not been ported to newer versions of GCC. It patches the GCC-3.3.x sources to include the real-time support in the GCJ compiler and runtime. Building jRate requires an already installed version of GCC to start the bootstrap of the GCC-3.3.x which has been patched to include JRate. It is important to note that GCC-4.x and later versions cannot directly bootstrap GCC-3.3.x without actually patching the GCC-3.3.x sources. Therefore, the easiest way is to install GCC-3.3.x and use it to build jRate.

- The last stable version of jRate-3.7.2 is not compatible with 64 bit systems. Updated sources from the svn repository can be downloaded from *http://svn.sourceforge.net/jrate* which can than be built.

- Multilib should be disabled. Passing *disable-multilib* as an argument to the configure does not work, therefore, either the script files need to be changed or an easy way is to change the Makefile explcitly in the end to make sure multilib is disabled.

Appendix D

Schedulability Analysis

The objective of this appendix is to present a schedulability analysis which is compatible with the Locality model. This will prove that the abstractions presented in the locality model are in line with the existing state of the art scheduling theory for multiprocessors. This can also be used to generate new parameters for the ExecutionSite.

The analysis provided does not consider blocking time as it is out of the scope of the thesis.

D.1 Scheduling Model

An application based on the Locality model has one or more execution site. Each execution site consists of schedulable objects. Each schedulable object is characterized by its release parameters which include execution time C_i, a period or a minimum inter-arrival interval T_i, a deadline D_i. In this schedulability analysis, we will only be considering a constrained deadline model where the deadlines of the threads are less than or equal to their deadline. Moreover, deadline monotonic priority ordering (DMPO) is used for the assignment of schedulables.

Each execution site is allocated a *PartitionedReservation* which acts as a virtual multiprocessor system for the ExecutionSite, where each processor is represented by the *ReservationServer* object. The ReservationServer object specifies the budget and period guaranteed on a particular processor that is guaranteed by the top level

scheduler. Figure D.1 shows that for each ExecutionSite a number of Reservation-Servers (SS1, SS2 etc.) are created which provide budget on particular processors.

D.2 Top Level Schedulability Test

A fully partitioned scheduling policy is adopted at the top level. Each *Reservation-Server* object is set an affinity to a particular processor and has an active priority. The priority of the ReservationServer object is assigned based on the rate monotonic priority ordering. The admission control policy uses the Liu and Layland utilization test [Liu and Layland, 1973] to check if a particular partitioned parameter is feasible on a processor or not.

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \quad \text{(D.2.1)}$$

For every ReservationServer object in a PartitionedReservation, the utilization test is applied, the processor is allocated if it is feasible, otherwise the next available processor is checked until the match is found. The admission control policy described in Section 4.3.3 is applied. Due to the limitations of the implementation, all ReservationServers belonging to the same ExecutionSite have the same budget and period. As a result all ReservationServers have the same priority (i.e. of the schedulable which is within the priority range of the ExecutionSite). All ReservationServers belonging to the same ExecutionSite have a collective cgroup which performs the actual cost accounting and enforcement on all processors. For more details see Section 5.4.

D.3 Local Schedulability Test

The execution site can be allocated a fixed budget on different processors using the *ReservationServer*. In this subsection, we will use the bounded delay multi-partition for the schedulability of threads inside an execution site.

According to the parallel supply function (PSF) [Bini et al., 2009a], the execution site is schedulable on a set ReservationServers represented by $\{Y_k\}_{k=1}^{m}$ if

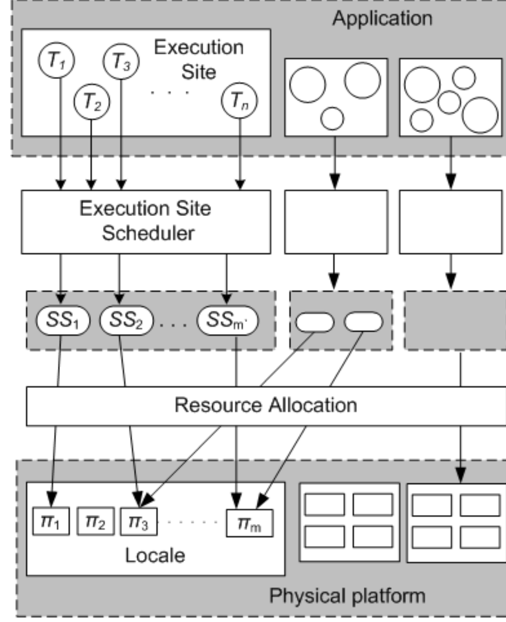


Figure D.1: Scheduling Architecture of Execution Sites on a Processor Based Budget

$$\bigcap_{i=1,\ldots,n} \bigcup_{k=1,\ldots,m} kC_i + W_i \leq Y_k\left(D_i\right) \tag{D.3.2}$$

where n is the number of tasks, m is the number of processors, Y_k is the parallel function on each processor and W_i is the maximum interfering workload defined as the maximum amount of work that can be done by higher priority tasks in an interval $[a, b)$ after which task T_i will miss deadlines.

$$\overline{W_j^{FP}} = \sum_{i=1}^{j-1} \overline{W_{j,i}} \tag{D.3.3}$$

$\overline{W_{j,i}}$ is the interfering workload and i is the set of indices of tasks of priority higher than T_j and can be calculated

$$\overline{W}_{j,i} = N_{j,i}C_i + \min\left\{C_i, D_j + D_i - C_i - N_{j,i}T_i\right\} \tag{D.3.4}$$

where $N_{j,i}$ is the number of releases of a higher priority task

$$N_{j,i} = \left\lfloor \frac{D_j + D_i - C_i}{T_i} \right\rfloor \tag{D.3.5}$$

While all the values in equation D.3.2 can be calculated from the task set except the PSF $\{Y_k\}_{k=1}^{m}$ itself. In order to calculate the PSF for the ExecutionSite, we

$Y_k(t)$	the parallel supply function on the interval t
C_i	computation time of the ith task.
D_i	deadline of the ith task.
T_i	period of the ith task.
W_j	interfering workload experienced by the jth task.
I	the BDM interface
m	number of processors
α_i	availability of ith processor
β_k	bandwidth on the first k processors
$\triangle$	bound on the delay between availability. in case of the worst case platform Π^wc

Table D.1: Symbols used for schedulability analysis

further need to use the bounded delay model (BDM). The symbols used in the following are described in listed in Table D.1. The BDM is defined in definition 3 in [Lipari and Bini, 2010] as following:

An interface I is a bounded-delay multipartitition (BDM) interface I such that $I = (m, \triangle, [\beta_1, ..., \beta_m])$ with $\triangle \geq 0$,

$$\forall k = 1, ..., m \qquad 0 \leq \beta_k - \beta_{k-1} \leq 1 \tag{D.3.6}$$

$$\forall k = 1, ..., m \qquad \beta_k - \beta_{k-1} \geq \beta_{k+1} - \beta_k \tag{D.3.7}$$

Now according to theorem 1 in [Lipari and Bini, 2010], the delay on all processors in the worst case platform is the same, therefore,the PSF $\{Y_k\}_{k=1}^m$ is equal to the following:

$$Y_k^{wc}(t) = \beta_k(t - \triangle)_0 = \sum_{i=1}^k \alpha_i(t - \triangle)_0 \tag{D.3.8}$$

where

$$1 \geq \alpha_1 \geq \alpha_2 \geq .... \geq \alpha_m \tag{D.3.9}$$

therefore, the schedulability condition in equationD.3.2 becomes

$$\bigcap_{i=1,\ldots,n} \bigcup_{k=1,\ldots,m} kC_i + W_i \leq \beta_k(D_i - \triangle)_0 \tag{D.3.10}$$

or

$$\bigcap_{i=1,\ldots,n} \bigcup_{k=1,\ldots,m} \sum_{j=1}^{k} \alpha_j \left(D_i - \triangle\right) \geq kC_i + W_i \tag{D.3.11}$$

Let us consider an execution site which has three real time threads such that T_1 = {2,8,8}, T_2={3,9,9}, T_3={1,10,10} as shown in Table D.3. Priorities are assigned to these tasks based on their deadlines. Therefore, T1 has the highest priority and T3 has the lowest(assuming higher values represent higher priority). We calculate the workload for all three tasks using equation D.3.3 .

	C	T	D	Priority	Interfering Workload
T1	2	8	8	19	0
T2	3	9	9	18	4
T3	1	10	10	17	10

Table D.2: Workload for the execution site

Now let us consider that the ExecutionSite has been allocated 3 Reservation-Servers: SS1 $=\{9, 10\}$, SS2 $= \{9, 10\}$ and SS3 $= \{9, 10\}$. For the bounded delay model, an execution time server which which has a budget of P every Q period of time we have processors $m = 3$, $\alpha_i = \frac{Q_i}{P_i}$ and maximum delay i.e. $\triangle_i = 2P - 2Q$ which is maximum time the processor i will not be available. The case where no processor will be available is $\triangle = 2$. Based on these values we calculate the interface $I = (m, \triangle, [\beta_1, ..., \beta_m])$ that is available to the ExecutionSite:

$$\alpha_1 = 0.9$$
$$\alpha_2 = 0.9$$
$$\alpha_3 = 0.9$$
$$I = [3, 2, (0.9, 1.8, 2.7)]$$

Now we calculate the required interface by the task, where the task set will be schedulable. This calculation is done using equation D.3.11 in the following steps:

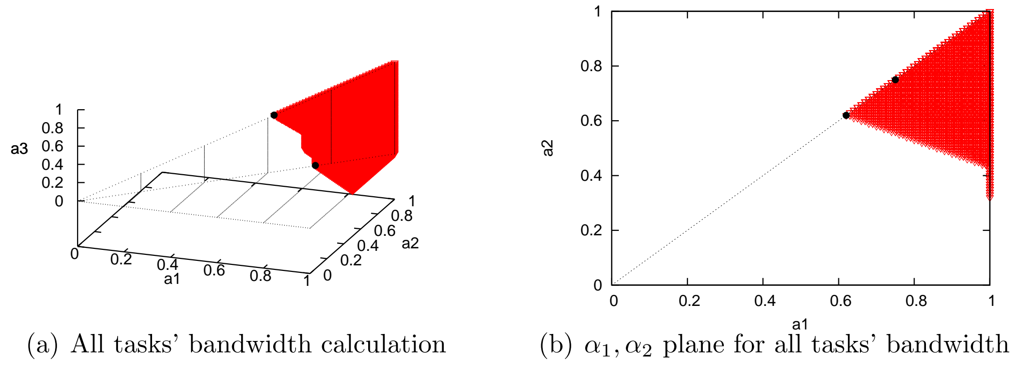
1. The bandwidth requirement of each task T_i is calculated based on the maximum execution time requirements C_i, the maximum workload interference W_i, deadline of the task D_i, number of processors and lastly the delay bound. Therefore, in order to solve the equation D.3.11, we first need to calculate $\frac{kC_i+W_i}{(D_i-\triangle)}$ for all values of k where $k = 1, ..., m$. This is calculated as following:

i=1	k=1	$\alpha_1 \geq 0.33333334$
	k=2	$\alpha_1 + \alpha_2 \geq 0.6666667$
	k=3	$\alpha_1 + \alpha_2 + \alpha_3 \geq 1.0$
i=2	k=1	$\alpha_1 \geq 1.0$
	k=2	$\alpha_1 + \alpha_2 \geq 1.4285715$
	k=3	$\alpha_1 + \alpha_2 + \alpha_3 \geq 1.8571428$
i=3	k=1	$\alpha_1 \geq 1.375$
	k=2	$\alpha_1 + \alpha_2 \geq 1.5$
	k=3	$\alpha_1 + \alpha_2 + \alpha_3 \geq 1.625$

For each task T_i, the bandwidth values on 1, 2, 3 processors have been calculated. The possible solution set for each task consists of all solutions possible for different values of k.

2. After calculating, the bandwidth requirements of each task we need to calculate the bandwidth requirement of the execution site. This can be done by taking the intersection of all the solutions of all the tasks. We end up with a number of critical points which are on the edge of the solution space of $\alpha 1, \alpha 2, \alpha 3$ space and can be seen as black dots in Figure D.2(a) and Figure D.2(b). These points are given in Table D.3 At these points the execution site becomes schedulable while before hitting these points it was not. All the red area is the set points where the execution site is schedulable.

   The interfaces shown in Table D.3 show that on three processors, the execution site will require at least an overall bandwidth of 1.86. On the other hand, it can be seen that the same execution site will require a bandwidth of 1.50 if it is mapped onto only 2 processor with each processor providing 0.62. As



(a) All tasks' bandwidth calculation (b) α_1, α_2 plane for all tasks' bandwidth

Figure D.2: Bandwidth requirements of $T_1T_2T_3$

β_1=α_1	β_2	$\alpha_2 = \beta_2 - \beta_1$	β_3	$\alpha_3 = \beta_3 - \beta_2$	$I = [m, \triangle, (\beta_1, ..., \beta_m)]$
0.62	1.24	0.62	1.86	0.62	[3, 2, (0.62, 1.24, 1.86)]
0.75	1.50	0.75	1.50	0	[3, 3, (0.75,1.50,1.50)]

Table D.3: Candidate interfaces of the execution site

the number of processors increases, the analysis becomes more pessimistic but allocation becomes more flexible as it is easier to map an execution map with its bandwidth spread across a number of processors. Based on these results, the task set in the ExecutionSite is schedulable on the PartitionedReservation provided. The task set would have been schedulable even if two processors were used with availability 0.9.

The BDM can also be used to determine the required Interface for any ExecutionSite. This can be used to request the required reservation for the ExecutionSite.

Appendix E

The Prime Sieves Example

This appendix contains the full source for the primes of Eratosthenes problem that has been used in Chapter 6 for evaluation. SectionE.1 is the source code without using the Locality model. Section E.2 provides the source code using he Locality model.

E.1 Without Using the Locality Model

```
import javax.realtime.*;
import java.util.LinkedList;

public class PrimeNum extends RealtimeThread {
 AbsoluteTime t;
 Clock c;
 final int N;

public PrimeNum(Clock c, AbsoluteTime t, int N) {
  this.c = c;
  this.t = t;
  this.N = N;
 }

 public void run() {

// Generate a buffer containing all odd numbers starting
```

```
// from 3 to less than N.
 Buffers oddBuffer = new Buffers();
 for (int i = 3; i <= N; i = i + 2) {
     IntClass newnum = new IntClass(i);
     oddBuffer.numList.add(newnum);
  }
// In order to terminate the program we pass -1.
// The program will terminate if the prime is -1.
  IntClass newnum = new IntClass(-1);
  oddBuffer.numList.add(newnum);

// A sieve is created.
  Seive Seive1 = new Seive(oddBuffer, 0, c, t);
  RealtimeThread t1 = new RealtimeThread(null, null,
    null, null, null, Seive1);
  t1.start();

 }

 public static void main(String[] args) {

// All primes will be generated less than N.
// We measure time starting now till we find
// the last prime.
  final int N = 15000;
  Clock c1 = Clock.getRealtimeClock();
  AbsoluteTime at = c1.getTime();
  PrimeNum firstThread = new PrimeNum(c1, at, N);
  firstThread.start();
    }
}


class Seive extends Thread {

Buffers numBuffer;
int threadNumbers;
```

```
public Clock c;
public AbsoluteTime at;

public Seive(Buffers numBuffer, int threadNumbers, Clock c,
  AbsoluteTime at) {
  this.numBuffer = numBuffer;
  this.threadNumbers = threadNumbers;
  this.c = c;
  this.at = at;

}

public void run() {
  this.threadNumbers++;
  IntClass primeObject;
  int prime = 0;
  int primeNumber;
// The numBuffer is shared between two sieves.
// The sieve receiving the numbers waits if there are no numbers
// in the buffer.

synchronized (numBuffer) {
  try {
  if (this.numBuffer.numList.isEmpty()) {
    numBuffer.wait();
  }

  primeObject = (IntClass) this.numBuffer.numList.removeFirst();
  primeNumber = primeObject.number;
// This is the first number which is always a prime number.
// All multiples of this number will be filtered out and then
// passed to next sieve.
// If the Number is -1 then the program terminates as this is
// the last sieve.
if (primeNumber != -1) {

  boolean nextSeive = false;
```

```
  while (!nextSeive) {
    if (this.numBuffer.numList.isEmpty()) {
      numBuffer.wait();
  }
  primeObject = (IntClass) this.numBuffer.numList.removeFirst();
  prime = primeObject.number;
  if (!(prime % primeNumber == 0)) {
    nextSeive = true;
   }
  }
  Buffers newnumBuffer = new Buffers();
  synchronized (newnumBuffer) {
  newnumBuffer.numList.add(primeObject);
  newnumBuffer.notify();
 }

// Creates a new sieve.
  Seive S2 = new Seive(newnumBuffer, threadNumbers, c,
   this.at);
  RealtimeThread Seive2 = new RealtimeThread(null, null,
    null, null, null, S2);
  Seive2.start();

// Pass any numbers to the newly created sieve which are not
// multiples of the prime of this sieve.
  if (prime != -1) {
    while (prime != -1) {
      while (this.numBuffer.numList.isEmpty()) {
        numBuffer.wait();
        }
      primeObject = (IntClass) this.numBuffer.numList
        .removeFirst();
      prime = primeObject.number;
      if (prime % primeNumber != 0) {
        synchronized (newnumBuffer) {
          newnumBuffer.numList.add(primeObject);
          newnumBuffer.notify();
```

```
  }}}}}
   else {
// This is executed by the last sieve and displays the time
// taken to generate all the primes.
  AbsoluteTime at2 = c.getTime();
  RelativeTime rt1 = at2.subtract(at);
  System.out.println("Time␣Taken␣in␣Milliseconds="
    + rt1.getMilliseconds() + "." + rt1.getNanoseconds());

  }
}
catch (InterruptedException e) {
  e.printStackTrace();
} } } }



class Buffers {
  LinkedList numList = new LinkedList();
}

class IntClass
{
public int number;
public IntClass(int num) {
 this.number=num;
} }
```

E.2 Using the Locality Model

```
import javax.realtime.*;
import java.util.LinkedList;

public class PrimeNum extends RealtimeThread {
  AbsoluteTime t;
  Clock c;
  int N;
```

```
public PrimeNum(Clock c, AbsoluteTime t, int N) {
    this.c = c;
    this.t = t;
    this.N = N;
 }

public void run() {
  // Generate a buffer containing all odd numbers starting
  // from 3 to less than N.
   Buffers oddBuffer = new Buffers();
   for (int i = 3; i <= N; i = i + 2) {
         IntClass newnum = new IntClass(i);
         oddBuffer.numList.add(newnum);
  }

// We create Execution Sites statically on all available Locales.
   Neighbourhood n1 = Locality.getCurrentNeighbourhood();
   Locale[] locs = n1.getLocales();
   ExecutionSite[] sites = new ExecutionSite[locs.length];
   for (int i = 0; i < locs.length; i++) {
       sites[i] = Locality.createExecutionSite(locs[i]);
   }
// In order to terminate the program we pass -1.
// The program will terminate if the prime is -1.
   IntClass newnum = new IntClass(-1);
   oddBuffer.numList.add(newnum);

// A sieve is created in the first ExecutionSite.
// We pass the index of the ExecutionSite on which the sieve is
// created along with references of the ExecutionSites.
// The real-time thread is passed the local memory area to set the
// allocation context of the new thread.
   int ThreadsinES = 0;
   int ExecutionSiteIndex = 0;
   Seive Seive1 = new Seive(oddBuffer, sites, ExecutionSiteIndex,
           ThreadsinES, c, t);
   RealtimeThread t1 = sites[ExecutionSiteIndex].
```

```
     createRealtimeThread(null, null, null,
     sites[ExecutionSiteIndex].getHeap(), Seive1);
   t1.start();
 }

public static void main(String[] args) {
    Platform.buildPlatform(null, null);
    long initialHeapSizes = 0;
    long initialImmortalSizes = 0;
    RelativeTime[] defaultbudget = null;
    RelativeTime[] defaultPeriod = null;
    int numProcessors = 3;
    Locality.initialize(initialHeapSizes, initialImmortalSizes,
             defaultbudget, defaultPeriod, numProcessors);
    // All primes will be generated less than N.
    // We measure time starting now till we find
    // the last prime.
    int Num = 15000;
    Clock c1 = Clock.getRealtimeClock();
    AbsoluteTime at = c1.getTime();
    PrimeNum firstThread = new PrimeNum(c1, at, Num);
    firstThread.start();
 }
}



class Seive extends Thread {

  Buffers numBuffer;
  int threadNumbers;
  public Clock c;
  public AbsoluteTime at;
  ExecutionSite[] ES;
  int ESindex;

public Seive(Buffers numBuffer, ExecutionSite [] ES,
 int ESindex, int threadNumbers, Clock c,AbsoluteTime at)
```

```
   {
  this.numBuffer = numBuffer;
  this.threadNumbers=threadNumbers;
  this.ES=ES;
  this.ESindex = ESindex;
  this.c = c;
  this.at = at;
 }

public void run () {
 this.threadNumbers++;
 IntClass primeObject;
 int prime=0;
 int primeNumber;

// The numBuffer is shared between two sieves.
// The sieve receiving the numbers waits if there are no numbers
// in the buffer.

  synchronized(numBuffer) {
  try {
  if (this.numBuffer.numList.isEmpty()) {
      numBuffer.wait();
      }

  primeObject = (IntClass) this.numBuffer.numList.removeFirst();
  primeNumber = primeObject.number;

// This is the first number which is always a prime number.
// All multiples of this number will be filtered out and
// then passed to next sieve.
// If the Number is -1 then the program terminates as this
// is the last sieve.
if (primeNumber != -1) {
// If 8 sieves are added to this execution site consecutively
// then move on to the next ExecutionSite on another Locale.
// This is to balance the load among the ExecutionSites.
```

```
  if (this.threadNumbers ==8 ) {
    if (this.ESindex == this.ES.length-1) {
       this.ESindex = 0;
  } else  {
        this.ESindex++;
  }
  this.threadNumbers = 0;
  }
  boolean nextSeive=false;
  while(!nextSeive) {
    if (this.numBuffer.numList.isEmpty()) {
    numBuffer.wait();
                }
  primeObject = (IntClass) this.numBuffer.numList.removeFirst();
  prime = primeObject.number;
   if (!(prime%primeNumber==0))
      {
       nextSeive=true;
   }

  Buffers newnumBuffer =new Buffers();
  synchronized(newnumBuffer) {
        newnumBuffer.numList.add(primeObject);
        newnumBuffer.notify();
 }


// Creates a new sieve on the ExecutionSite
// identified by the ESindex.
  Seive S2=new Seive(newnumBuffer, this.ES,
  this.ESindex, threadNumbers,c,this.at);
  RealtimeThread Seive2=ES[this.ESindex].createRealtimeThread(
   null,null,null,this.ES[this.ESindex].getHeap(),S2);
  Seive2.start();

// Pass any numbers to the newly created sieve which are
```

```
// not multiples of the prime of this sieve.
  if (prime != -1) {
    while (prime!=-1) {
      while (this.numBuffer.numList.isEmpty()) {
         numBuffer.wait();
         }
      primeObject = (IntClass)
        this.numBuffer.numList.removeFirst();
      prime = primeObject.number;
      if (prime%primeNumber!=0) {
        synchronized(newnumBuffer) {
          newnumBuffer.numList.add(primeObject);
          newnumBuffer.notify();
                        }}}}}}
else{
// This is executed by the last sieve and
// displays the time taken to generate all the primes.
  AbsoluteTime at2=c.getTime();
  RelativeTime rt1=at2.subtract(at);
  System.out.println("Time␣Taken␣in␣Milliseconds="
    + rt1.getMilliseconds() + "." + rt1.getNanoseconds());
  }   }
catch (InterruptedException e) {
                e.printStackTrace();
}}}}

class IntClass {

 public int number;
 public IntClass(int num) {
    this.number=num;
   }
}

class Buffers {
  LinkedList numList = new LinkedList();
}
```

Appendix F

Overheads of Libcgroup

This appendix contains the overheads caused by the Libcgroup library.

F.1 Overheads Creating a ReservationServer/Place

The cgroup is created for each Place and ReservationServer. In this section we present overheads of all calls that need to made to create a cgroup.

Initializing the Cgroup Library Figure F.1 shows the overhead of initializing the cgroup library.

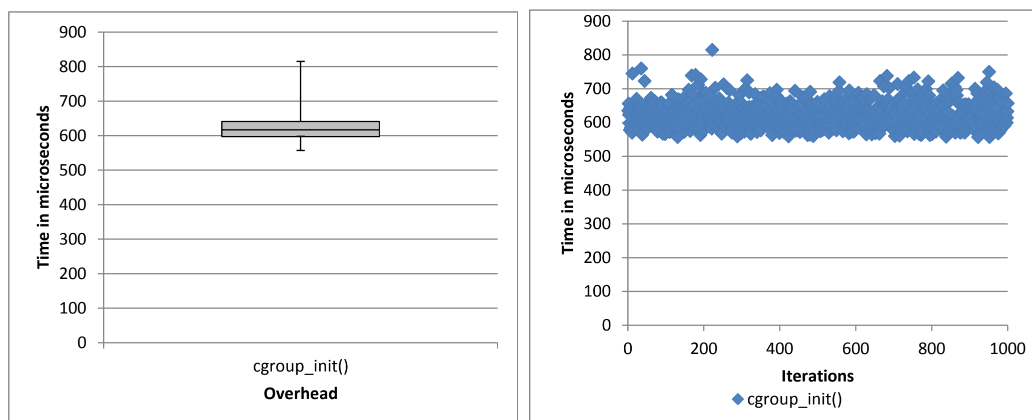


Figure F.1: cgroup_init() timings

Creating the Cgroup Internally Figure F.2 shows the overhead of creating an internal cgroup datastructure.

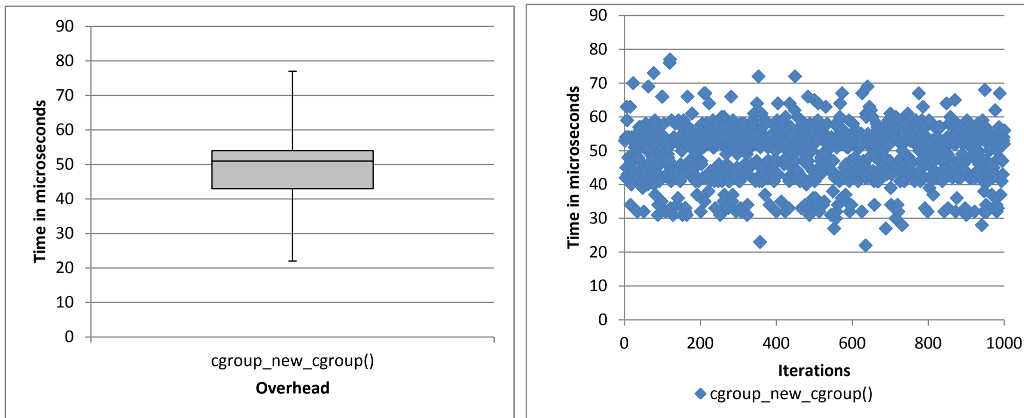


Figure F.2: cgroup_new_cgroup() timings

Adding Controller to a Cgroup Figure F.3 shows the overhead of adding a controller to the cgroup structure in the internal cgroup structure.

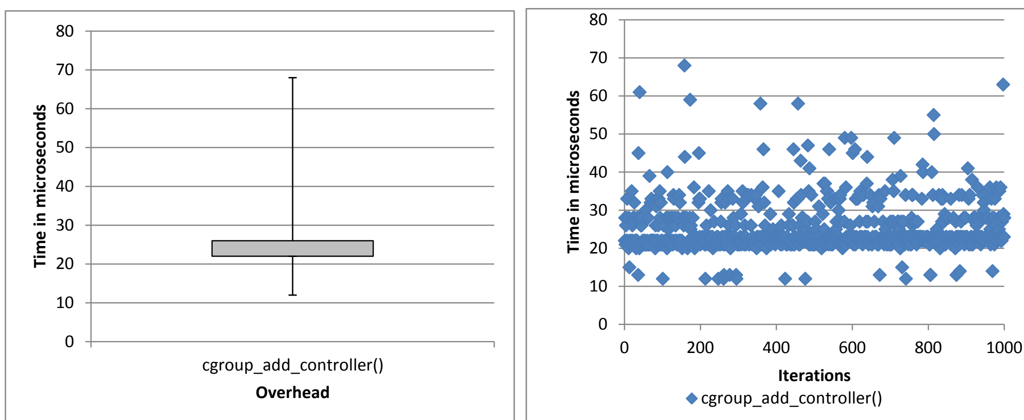


Figure F.3: cgroup_add_controller() timings

Setting Values of a Control File Figure F.4 shows the overhead of setting values of the controller in the internal cgroup structure.

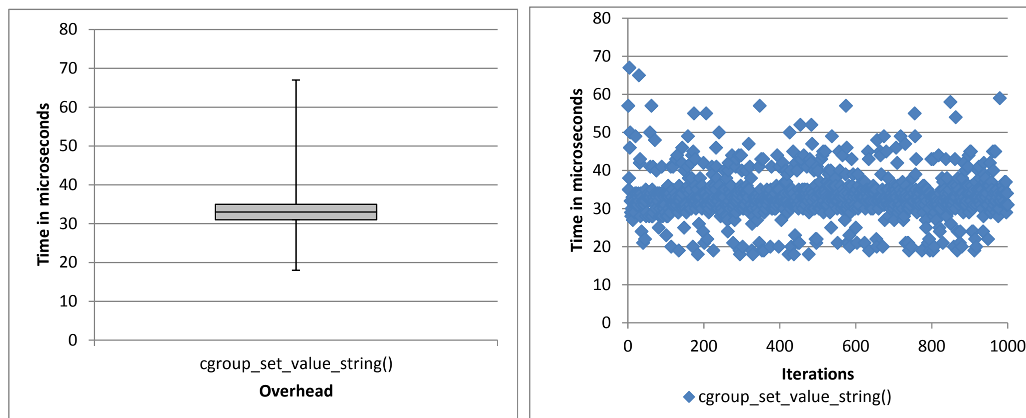


Figure F.4: cgroup_set_value_string() timings

Creating a Cgroup Figure F.5 shows the overhead of physically creating a cgroup in the kernel.

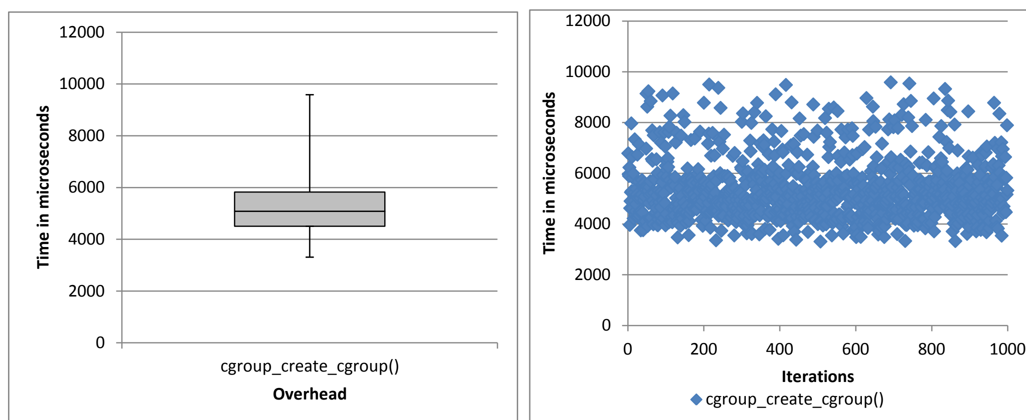


Figure F.5: cgroup_create_cgroup() timings

F.2 Attaching a Thread to a ReservationServer

A thread needs to be attached to a cgroup which corresponds to a collective set of ReservationServers in the Locality model. The startup latency of a real-time thread shows a large overhead when the thread is attached to the cgroup using the cgroup_attach_thread() call.

Attaching a Thread to a CGroup Figure F.6 shows the overhead of attaching a thread to a cgroup.

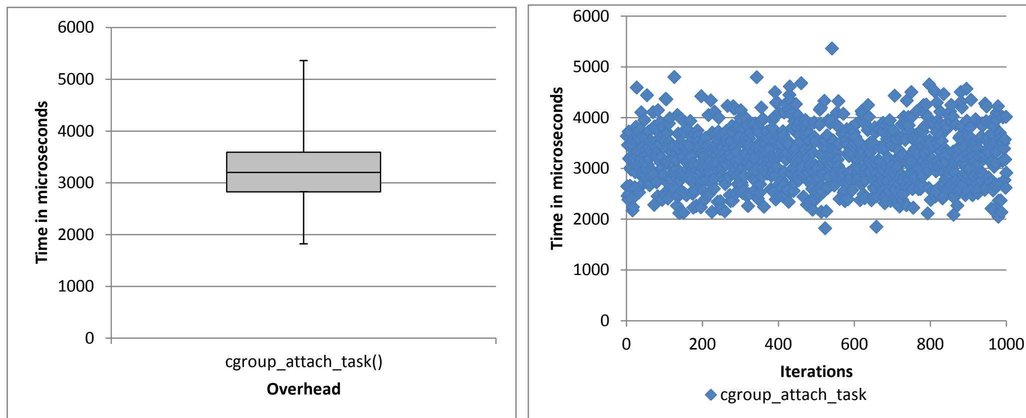


Figure F.6: cgroup_attach_thread() timings

F.3 Summary

The functions can be categorized into the following two categories:

1. that changes the internal cgroup structures
2. that communicate with the Linux kernel

The first type of functions have low overheads, and the second ones have high overheads. Table F.1 shows the summary of all the overheads that have been presented in this appendix. It is clearly visible that cgroup_attach_thread() and cgroup_create_cgroup() communicate with the kernel that is why they have a large overhead and the dispersion in the values is also very high. [Bagnoli, 2010] have used ioctl()[1] instead of using libcgroup to communicate with the kernel and have shown that such the overheads can be reduced by using alternate ways of communicating with the kernel.

[1]http://www.kernel.org/doc/man-pages/online/pages/man2/ioctl.2.html

	Average	STD	Count	Min	0.9	0.95	0.99	Max
cgroup_init()	621.34	35.43	1000	557	668.2	688.1	725.06	815
cgroup_new_cgroup()	48.9	8.09	1000	22	58	60	67	77
cgroup_create_cgroup()	5341.34	1206.65	1000	3310	7163.4	7896.9	9069.9	9586
cgroup_add_controller()	24.48	5.97	1000	12	33	35	49	68
cgroup_set_value_string()	33.25	6.07	1000	18	41	44	55	67
cgroup_attach_task	3219.04	541.66	1000	1822	3930.2	4129.8	4504.02	5363

Table F.1: LibCGroup Overhead statistics

Abbreviations

ACPI Application Programming Interface

API Application Programming Interface

APIC Advanced Programmable Interrupt Controller

BS Backing Store

BDM Bounded Delay Multi-partition

CC-NUMA Cache Coherent Non-Uniform Memory Access

cHT Coherent HyperTransport

DSM Distributed Shared Memory

EDF Earliest Deadline First

ES ExecutionSite

HPC High Performance Computing

HT HyperTransport

JEOPARD Java Environment for Parallel Real-Time Development

JVM Java Virtual Machine

NUMA Non-Uniform Memory Access

OS Operating Systems

PD Proximity Domain

RMI Remote Method Invocation

RTJVM Real-time Java Virtual Machine

RTS Real Time System

RTSJ Real Time Specification for Java

SLIT System Locality Information Table

SMP Symmetric Multiprocessor

SMMP Shared Memory Multiprocessor

SRAT System Resource Affinity Table

SSI Single System Image

UMA Uniform Memory Access

WCET Worst Case Execution Time

List of Classes

Cache	An abstraction for a cache attached to a processor.
Component	An abstract class sub-classed by the Processor, Memory and Device classes.
ClusterContract	Represents temporal guarantees of a Place on an SMP node.
Device	An abstraction for a physical device.
District	An abstraction representing a NUMA system.
ESNoHeapRealtimeThread	A NoHeapRealtimeThread created in an ExecutionSite.
ESRealtimeThread	A RealtimeThread created in an ExecutionSite.
ExecutionSite	An abstraction which provides locality and and temporal isolation by grouping threads and objects.
ExternalContract	A system wide contract between the OS and the RTJVM about the guarantees of resources.
HeapPhysicalMemory	A physical Heap Memory Area.
Locale	An abstraction representing a Uniform Memory Access (UMA) Symmetric Multiprocessor (SMP).

Locality	A final class which maps the ExecutionSites on a Locale.
LocalMemory	Abstraction representing a memory on a node; implements the PhysicalMemorytypeFilter.
Location	An abstraction which represents a multiprocessor. Contains Processors, memories and Devices.
Memory	An abstraction for a physical memory device.
Neighbourhood	An abstraction representing a Cache Coherent Non-Uniform Memory Access (CC-NUMA) system.
PartitionedParameters	Represents the budget guarantees on a particular processor.
PartitionedReservation	The contract between an ExecutionSite and the RTJVM defining the temporal resources guaranteed to that ExecutionSite.
Place	A set of resources on each Locale which are shared by all ExecutionSites allocated on that Locale.
Platform	An interface to the Architectural Model used to query information about the system.
Processor	An abstraction for a processor.
ProcessorType	A Java Interface for defining different types of processors.
ReservationScheduler	A sub-class of the scheduler class which allows admission control of ExecutionSites.
ReservationServer	An abstraction which provides cost enforcement on a processor.

References

[Aas, 2005] Aas, J. (2005). Understanding the Linux 2.6.8.1 CPU Scheduler. SGI, 2005. http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf, accessed on August *22*, 05.

[Allen et al., 2007] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Jr., G. L. S. and Tobin-Hochstadt, S. (2007). The Fortress Language Specification. Technical report Sun Microsystems, Inc.

[AMD, 2003] AMD (2003). BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors. Technical report.

[AMD, 2006] AMD (2006). Performance Guidlines for AMD Athlon 64 and AMD Opteron ccNUMA Multiprocessor System. Technical report.

[Amza et al., 1996] Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W. (1996). TreadMarks: shared memory computing on networks of workstations. Computer *29*, 18 –28.

[Antoniu et al., 2001] Antoniu, G., Bougé, L., Hatcher, P., MacBeth, M., McGuigan, K. and Namyst, R. (2001). The hyperion system: compiling multithreaded java bytecode for distributed execution. Parallel Comput. *27*, 1279–1297.

[Aridor et al., 1999] Aridor, Y., Factor, M. and Teperman, A. (1999). cJVM: A Single System Image of a JVM on a Cluster. In Proceedings of the 1999 International Conference on Parallel Processing ICPP '99 pp. 4–, IEEE Computer Society, Washington, DC, USA.

[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W. and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183 EECS Department, University of California, Berkeley.

[Bagnoli, 2010] Bagnoli, G. (2010). Design and development of a mechanism for low-latency real time audio processing on Linux.

[Bini et al., 2009a] Bini, E., Bertogna, M. and Baruah, S. (2009a). Virtual Multiprocessor Platforms: Specification and Use. In Proceedings of the 2009 30th IEEE Real-Time Systems Symposium RTSS '09 pp. 437–446, IEEE Computer Society, Washington, DC, USA.

[Bini et al., 2009b] Bini, E., Buttazzo, G. and Bertogna, M. (2009b). The Multi Supply Function Abstraction for Multiprocessors. In Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on pp. 294 –302,.

[Bolosky et al., 1989] Bolosky, W., Fitzgerald, R. and Scott, M. (1989). Simple but Effective Techniques for NUMA Memory Menagement. SIGOPS Oper. Syst. Rev. *23*, 19–31.

[Broquedis et al., 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S. and Namyst, R. (2010). hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on pp. 180 –186,.

[Burns and Wellings, 2001] Burns, A. and Wellings, A. J. (2001). Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX. 3rd edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Buttazzo et al., 2005] Buttazzo, G., Lipari, G., Abeni, L. and Caccamo, M. (2005). Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science). Plenum Publishing Co.

[Charles et al., 2005] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. *40*, 519–538.

[Checconi et al., 2009] Checconi, F., Cucinotta, T., Faggioli, D. and Lipari, G. (2009). Hierarchical Multiprocessor CPU Reservations for the Linux Kernel. In Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009), Dublin, Ireland.

[Corporation et al., 2005] Corporation, H.-P., Corporation, I., andPhoenix Technologies Ltd., M. C. and Corporation, T. (2005). Advanced Configuration and Power Interface Specification, Revision 3.0a.

[Corsaro and Schmidt, 2002] Corsaro, A. and Schmidt, D. C. (2002). The Design and Performance of the jRate Real-Time Java Implementation. In The 4th International Symposium on Distributed Objects and Applications, DOA 2002, Lecture Notes In Computer Science; Vol. 2519 pp. 900 – 921,.

[Deng et al., 1997] Deng, Z., Liu, J.-S. and Sun, J. (1997). A scheme for scheduling hard real-time applications in open system environment. In Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on pp. 191 –199,.

[Diaconescu and Zima, 2007] Diaconescu, R. and Zima, H. (2007). An Approach To Data Distributions in Chapel. Int. J. High Perform. Comput. Appl. *21*, 313–335.

[Dibble and Wellings, 2009] Dibble, P. and Wellings, A. (2009). JSR-282 status report. In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems JTRES '09 pp. 179–182, ACM, New York, NY, USA.

[Dietrich and Walker, 2005] Dietrich, S. and Walker, D. (2005). The Evolution of Real-Time Linux.

[Factor et al., 2006] Factor, M., Schuster, A. and Shagin, K. (2006). A platform-independent distributed runtime for standard multithreaded Java. Int. J. Parallel Program. *34*, 113–142.

[Faggioli et al., 2010] Faggioli, D., Bertogna, M. and Checconi, F. (2010). Sporadic Server revisited. In Proceedings of the 2010 ACM Symposium on Applied Computing SAC '10 pp. 340–345, ACM, New York, NY, USA.

[Fatahalian et al., 2006] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J. and Hanrahan, P. (2006). Sequoia: programming the memory hierarchy. In SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing p. 83, ACM, New York, NY, USA.

[Flynn, 2004] Flynn, L. J. (2004). Intel Halts Development Of 2 New Microprocessors.

[Goglin and Furmento, 2009] Goglin, B. and Furmento, N. (2009). Enabling high-performance memory migration for multithreaded applications on LINUX. In Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on pp. 1 –9,.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G. and Bracha, G. (2005). Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). Addison-Wesley Professional.

[Hennessy and Patterson, 2006] Hennessy, J. L. and Patterson, D. A. (2006). Computer Architecture, Fourth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Holt, 2009] Holt, J. (2009). Designing an Industry Standard API to Manage Multicore System Resources.

[IEEE, 2008] IEEE (2008). IEEE Std 1003.1-2008 Portable Operating System Interface (POSIX) Technical Standard, vol. 1,.

[Kambites et al., 2001] Kambites, M. E., Obdrlek, J. and Bull, J. M. (2001). An OpenMP-like interface for parallel programming in Java. Concurrency and Computation: Practice and Experience *13*, 793–814.

[Kleen, 2004] Kleen, A. (2004). An NUMA API for Linux.

[Klemm et al., 2007] Klemm, M., Bezold, M., Veldema, R. and Philippsen, M. (2007). JaMP: an implementation of OpenMP for a Java DSM. Concurrency and Computation: Practice and Experience *19*, 2333–2352.

[Koelbel et al., 1994] Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele, Jr., G. L. and Zosel, M. E. (1994). The high performance Fortran handbook. MIT Press, Cambridge, MA, USA.

[Lameter, 2006] Lameter, C. (2006). Local and Remote Memory: Memory in a Linux/NUMA System.

[Leontyev and Anderson, 2009] Leontyev, H. and Anderson, J. (2009). A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. Real-Time Systems *43*, 60–92.

[Lipari and Bini, 2010] Lipari, G. and Bini, E. (2010). A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation. In Proceedings of the 2010 31st IEEE Real-Time Systems Symposium RTSS '10 pp. 249–258, IEEE Computer Society, Washington, DC, USA.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM *20*, 46–61.

[Ma et al., 2000] Ma, M. J. M., Wang, C.-L. and Lau, F. C. M. (2000). JESSICA: Java-enabled single-system-image computing architecture. J. Parallel Distrib. Comput. *60*, 1194–1222.

[Malik et al., 2010] Malik, A. H., Wellings, A. and Chang, Y. (2010). A locality model for the real-time specification for Java. In Proceedings of the 8th Inter-

national Workshop on Java Technologies for Real-Time and Embedded Systems JTRES '10 pp. 36–45, ACM, New York, NY, USA.

[McIlroy, 2010] McIlroy, R. (2010). Using Program Behaviour to Exploit Heterogeneous Multi-Core Processors. PhD thesis, Glasgow, UK.

[Mercer et al., 1994] Mercer, C., Savage, S. and Tokuda, H. (1994). Processor capacity reserves: operating system support for multimedia applications. In Multimedia Computing and Systems, 1994., Proceedings of the International Conference on pp. 90 –99,.

[Mok et al., 2001] Mok, A., Feng, X. and Chen, D. (2001). Resource partition for real-time systems. In Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE pp. 75 –84,.

[OpenMP, 2008] OpenMP (2008). The OpenMP Application Program Interface.

[Quinn, 1994] Quinn, M. J. (1994). Parallel computing (2nd ed.): theory and practice. McGraw-Hill, Inc., New York, NY, USA.

[Rivas and Gonzlez Harbour, 2001] Rivas, M. and Gonzlez Harbour, M. (2001). MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In Reliable SoftwareTechnologies Ada-Europe 2001, (Craeynest, D. and Strohmeier, A., eds), vol. 2043, of Lecture Notes in Computer Science pp. 305–316. Springer Berlin / Heidelberg. 10.1007/3-540-45136-6_24.

[Saraswat, 2010] Saraswat, V. (2010). Report on the Programming Language X10. Language specification IBM.

[Schoeberl, 2004] Schoeberl, M. (2004). A time predictable instruction cache for a java processor. In In On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), volume 3292 of LNCS pp. 371–382, Springer.

[Shin et al., 2008] Shin, I., Easwaran, A. and Lee, I. (2008). Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors. In Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on pp. 181 –190,.

[Tam et al., 2007] Tam, D., Azimi, R. and Stumm, M. (2007). Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. SIGOPS Oper. Syst. Rev. *41*, 47–58.

[Tikir and Hollingsworth, 2005a] Tikir, M. and Hollingsworth, J. (2005a). NUMA-Aware Java Heaps for Server Applications. In Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International p. 108b,.

[Tikir and Hollingsworth, 2005b] Tikir, M. M. and Hollingsworth, J. K. (2005b). NUMA-Aware Java Heaps for Server Applications. In IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers p. 108.2, IEEE Computer Society, Washington, DC, USA.

[Veldema et al., 2001] Veldema, R., Hofman, R. F. H., Bhoedjang, R. A. F., Jacobs, C. J. H. and Bal, H. E. (2001). Source-level global optimizations for fine-grain distributed shared memory systems. SIGPLAN Not. *36*, 83–92.

[Wellings and Schoeberl, 2009] Wellings, A. and Schoeberl, M. (2009). Thread-Local Scope Caching for Real-time Java. In Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on pp. 275 –282,.

[Wellings et al., 2009] Wellings, A. J., Chang, Y. and Richardson, T. (2009). Enhancing the platform independence of the real-time specification for Java. In JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems pp. 61–69, ACM, New York, NY, USA.

[Wellings and Kim, 2008] Wellings, A. J. and Kim, M. S. (2008). Processing group parameters in the real-time specification for Java. In Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems JTRES '08 pp. 3–9, ACM, New York, NY, USA.

[Woodacre et al., 2003] Woodacre, M., Robb, D., Roe, D. and Feind, K. (2003). The SGI AltixTM3000 Global Shared-Memory Architecture. Technical report.

[Young, 1982] Young, S. (1982). Real-Time languages: design and development. Ellis Horwood Publishers, Chichester, UK.

[Yu and Cox, 1997] Yu, W. and Cox, A. L. (1997). Java/DSM: A Platform for Heterogeneous Computing. Concurrency and Computation: Practice and Experience *9*, 1213–1224.

[Zerzelidis and Wellings, 2010] Zerzelidis, A. and Wellings, A. (2010). A framework for flexible scheduling in the RTSJ. ACM Trans. Embed. Comput. Syst. *10*, 3:1–3:44.

[Zhou et al., 1996] Zhou, Y., Iftode, L. and Li, K. (1996). Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. SIGOPS Oper. Syst. Rev. *30*, 75–88.

[Zhu et al., 2002] Zhu, W., Wang, C.-L. and Lau, F. (2002). JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. In Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on pp. 381 – 388,.