Distributed Time-Predictable Memory Interconnect for Multi-Core Architectures

Haitong Wang

Doctor of Philosophy

University of York

Computer Science

September 2019

Abstract

Multi-core architectures are increasingly adopted in emerging real-time applications where execution time is required to be bounded in the worst case (i.e., time predictability) and low. Memory access latency is the main part forming the overall execution time. A promising approach towards time predictability is to employ distributed memory interconnects, either *locally arbitrated* interconnects or *globally arbitrated* interconnects, with arbitration schemes, and the pipelined tree-based structure can break the critical path of multiplexing into short steps with small logic size. It scales to a large number of processors that high clock frequency can be synthesised. This research explores timing behaviour of multi-core architectures with shared distributed memory interconnects and improves distributed time-predictable memory interconnects for multi-core architectures. The contributions are mainly threefold. First, the generic analytical flow is proposed for time-predictable behaviour of memory accesses across multi-core architectures with *locally arbitrated* interconnects. It guarantees time predictability and safely bound the worst case without exact memory access profiles. Second, the root queue modification with the root queue management is proposed for multi-core architectures with *locally arbitrated* interconnects that variation of memory access latency is reduced and timing behaviour analysis is facilitated. Third, Meshed Bluetree is proposed as the distributed time-predictable multi-memory interconnect, enabling multiple processors to simultaneously access multiple memory modules.

Acknowledgements

I would like to give my sincere gratitude to my supervisor Prof. Neil C. Audsley for his guidance, patience and encouragement through my journey. I would also like to express special thanks to my research colleagues for their help and support.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for a degree or other qualification at this university or elsewhere. All sources are acknowledged as references. The content of some of the chapters in this thesis has already been published within the following publications.

- H. Wang, N. C. Audsley and W. Chang. Addressing Resource Contention and Timing Predictability for Multi-Core Architectures with Shared Memory Interconnects. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, Australia, 2020, pp. 70-81. [1]

- H. Wang, N. C. Audsley, X. S. Hu and W. Chang. Meshed Bluetree: Time-Predictable Multimemory Interconnect for Multicore Architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3787-3798, Nov. 2020. [2]

Contents

List of Figures

List of Tables

List of Symbols

Symbol	Description
μ_i	a client with index i
N_μ	number of client
B_j	a distributed memory interconnect with index j
N_B	number of distributed memory interconnect
N_β	depth of distributed memory interconnect
β_k	a stage of distributed memory interconnect with index k
D_j	a shared root memory module with index j
N_D	number of shared root memory module
$t(D_j)$	latency of shared root memory module D_j
P_i	memory access path for client μ_i
$P_{(i,j)}$	memory access path from client μ_i to shared root memory module D_j (in multi-memory architecture)
$P(\beta_k)$	local priority of a memory access path at stage β_k
α	Bluetree blocking factor
ω	a memory access
$t(\omega)$	latency of memory access ω
$t_{RQ}(\omega)$	request path latency of memory access ω
$t_{RS}(\omega)$	response path latency of memory access ω

$t^{BC}(\omega)$	best-case latency of memory access ω
$t^{BC}_{RQ}(\omega)$	best-case request path latency of memory access ω
$t^{BC}_{RS}(\omega)$	best-case response path latency of memory access ω
$t^{WC}(\omega)$	worst-case latency of memory access ω
$t^{WC}_{RQ}(\omega)$	worst-case request path latency of memory access ω
$t^{WC}_{RS}(\omega)$	worst-case response path latency of memory access ω
$N^{WC}_{RQ}(\omega)$	maximum blocking number in request path of memory access ω
$N^{WC}_{RQ}(\beta_k)$	maximum blocking number iterative up to stage β_k
$N^{WC}_{\alpha}(\beta_k)$	maximum arbiter blocking number at stage β_k
τ	a sequence of memory requests
$N^{WC}_{RQ}(\tau)$	maximum blocking number in request path of sequence τ
$N^{\mu}_{RQ}(\mu_i)$	outstanding request number from client μ_i
$N^{\mu}_{RQ}(D_j)$	outstanding request number to shared root memory D_j
$T^{\mu}_{RQ}(\mu_i)$	request interval from client μ_i
Q	root queue size
Q_S	minimum size of the root queue for queued service
R	a router network
N_R	depth of router network
N_{mux}	number of Bluetree multiplexer
N_{router}	number of Bluetree router
N_{wire}	number of Bluetree wire

List of Acronyms

Acronym	Description
NoC	Network-on-Chip
TDM	Time Division Multiplexing
BCET	Best-Case Execution Time
WCET	Worst-Case Execution Time
HRT	Hard Real-Time
FRT	Firm Real-Time
SRT	Soft Real-Time
RAM	Random Access Memory
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
DDR DRAM	Double Data Rate synchronous DRAM
FIFO	First-In-First-Out
LRU	Least Recently Used
MRU	Most Recently Used
SPM	Scratchpad Memory
AHB	Advanced High-Performance Bus
AXI	Advanced eXtensible Interface
MoT	Mesh-of-Tree
GAMT	Globally Arbitrated Memory Tree

RQ	Request Path
RS	Response Path
MUX	Multiplexer
DEMUX	Demultiplexer
L	Low Priority
H	High Priority
RTL	Register-Transfer Level
LUT	Look-Up Table

Chapter 1

Introduction

Recently, Moore's law [3][4] is still relevant, while Dennard Scaling [5] has broken down. Moore's law states that the number of transistors per unit area doubles for each technology generation. Dennard Scaling states that the power density stays roughly constant as transistors get smaller, and thus the power consumption stays in proportion with area. Combined Moore's law with Dennard Scaling, computing performance per watt doubles about every two years. However, the breakdown of Dennard Scaling limits the improvement of the computing performance by increasing the processor clock frequency directly. Instead, the current trend is to scale the number of processing cores to achieve high performance.

The conventional method tends to employ multiple processing cores within a single chip. This promotes the multi-core architecture where the multiple processing cores are constructed and interconnected with a shared bus. Compared with the coupling of multiple single core processors, the multi-core processor executes threads concurrently, providing higher performance with less power consumption. However, this introduces the bus contention issue.

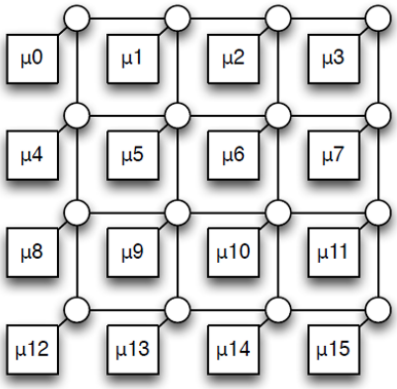

Figure 1.1. Network-on-Chip Architecture

The communications between the processing cores, or between the processing core and the memory module or the peripheral such as I/O peripheral, must be delivered through the shared bus, leading to contention delays. This becomes the bottleneck as the number of processing cores increases. In this case, the multi-core architecture is not scalable.

An alternative method is network-on-chip (NoC) [6][7]. It employs a packet switching communication network to connect the separate processing cores. As shown in Figure 1.1, each processing core connects through a router to the network. Then the communication packets can be delivered across the network with the routing traffic. In this way, NoC allows easy data sharing that a processing core can communicate with its target directly. This promotes the many-core architecture. Compared with the multi-core architecture, NoC provides faster communications with less bus contention. This potentially improves the performance and reduces the power consumption. However, the NoC design burden lies on the router architecture and the communication mechanism [8][9]. It also involves research on topology and layered protocols.

While the processor performance keeps improving, the memory performance remains the system bottleneck. High memory latency leads to expensive pro-

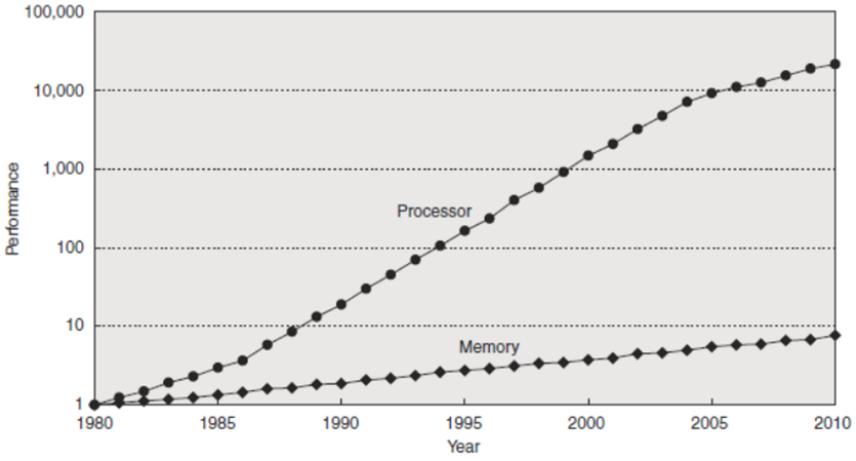

Figure 1.2. Performance Gap between Memory and Processor [10]

cessor stalls, and this directly degrades the overall system performance. The memory wall [11] is introduced to indicate that the performance of the system is decided by the memory speed but no longer by the processor speed. Figure 1.2 shows the processor speed against the memory speed over time. The vertical axis is the logarithmic scale to record the performance gap between the processor and the memory, and the memory performance baseline is 64KB DRAM in 1980. As shown in the graph, the speed of the processor after 1990 can improve more than 60% every year, while the memory can only improve about 7% [10]. The performance gap between the memory and the processor is still widening with a rapid rate.

With the aim to alleviate the memory latency issue, several solutions have been explored, such as the multi-threading. With enough support from both the processor architecture and the OS, the multi-threading allows the switching from a thread to another concurrent thread instead of just waiting. This avoids the processor stalls caused by the memory latency. However, this requires large amounts of resources and introduces overheads. By contrast, the architectural solution is the memory hierarchy design, and the processor may still easily lose performance within the memory hierarchy. Further im-

provement methods mainly rely on the cache to better exploit the locality. Prefetch methods can also be employed with the aim to prepare next data or instructions for the processor ahead of time. As for embedded systems, scratchpad memory is introduced as as alternative for cache. It exploits the application behaviour and manages the memory using explicit instructions.

With a growing number of applications being integrated into modern embedded systems, it places a heavy request on memory subsystem, especially within the multi-core and many-core architecture such as GPU application [12]. As the number of processors to access a single memory module increases, memory access latency inevitably increases. Besides that, multi-core architectures are also increasingly adopted in the emerging real-time applications, such as autonomous vehicles and robotics, where the memory access latency is required to be bounded both in the worst case (i.e., time predictability) and low. With the trend of integrating more applications or employing more processors, the potential contention over memory accesses gets more severe within such architectures. This harms time predictability which is highly undesirable for real-time applications.

Distributed Time-Predictable Memory Interconnect

With the aim to achieve time predictability (or simply predict the timing behaviour of memory accesses), multi-core and many-core architectures typically utilise an arbitration scheme to provide timing guarantees. The potential memory arbitration schemes include time division multiplexing (TDM) scheme, round-robin scheme and priority-based schemes (e.g., static-priority arbitration). The conventional centralised implementation of an arbitration scheme is to deploy a single arbiter, allowing arbitration decisions to be made at the central location. However, as the number of processors grows, the logic

size of the arbiter hardware increases, which limits the maximum synthesisable clock frequency.

A promising approach recently investigated is to employ distributed memory interconnects that the tree-based structure with pipelined stages can break the critical path of the multiplexing into multiple shorter steps with smaller logic size. Although this introduces additional delays in terms of clock cycles, the latency across such interconnect is actually reduced that much higher synthesisable clock frequency is allowed on the distributed hardware. It scales to a large number of processors. The critical path of the distributed interconnect remains constant as it is duplicatedly constructed with the growing number of processors. In addition, pipelining is also supported.

The distributed memory interconnects can be classified as the *locally arbitrated* interconnect and the *globally arbitrated* interconnect. The *locally arbitrated* interconnect is simply constructed upon a distributed binary arbitration tree which multiplexes the memory requests from processors to the shared root memory module through the distributed data paths. By comparison, based on a distributed binary arbitration tree, the *globally arbitrated* interconnect integrates the global scheduling to the distributed data paths.

In general, the *locally arbitrated* interconnect allows the average-case latency to be much lower than the worst case, however making time predictability challenging. By contrast, the *globally arbitrated* interconnect essentially limits the average-case behaviour to be similar to the worst case, facilitating the timing behaviour analysis. However, the processor utilisation within such architecture is potentially reduced, degrading the overall system performance. Besides that, the *globally arbitrated* interconnect requires complex scheduling as well as strict coordination, potentially suffering synchronisation issue. The detailed analysis is shown in following research.

1.1 Research Questions

The focus of this research is to explore the timing behaviour of the multi-core architectures with shared distributed memory interconnects and improve the distributed time-predictable memory interconnect for multi-core architectures.

As multi-core architectures are increasingly being adopted for real-time applications, the execution time of such application is required to be both bounded in the worst case and low. However, the multi-core architecture is typically designed for good average-case performance, and the resource contention within such architecture is inevitable. It potentially causes contention over memory accesses across the multi-core architecture, and this complicates the analysis of memory access latency which is the main part forming the overall program execution time. With the deployment of distributed memory interconnect, the analysis of memory access behaviour across the multi-core architecture further complicates that such architecture appears to be more sensitive to the resource contention due to the introduction of the tree-based structure.

It is to be noted that this research focuses on the memory accesses issued by processors to access the shared memory within the multi-core architecture. In this case, memory access latency is the latency of memory request issued to access the shared memory across the multi-core architecture, including the time consumed across the memory access path, the time consumed for the response of the shared memory, and the time consumed due to the resource contention within the shared memory multi-core architecture.

Based on the above analysis, the following research questions are related and formed. First, it is crucial to guarantee time predictability in multi-core ar-

chitectures for real-time applications. In this research, time predictability requires to statically analyse the timing behaviour of memory accesses across the multi-core architecture and bound the worst-case memory access latency within such architecture. Achieving time predictability within the multi-core architecture is challenging that software components or tasks can contend for the shared hardware resources, such as memory modules, with varying status. Such contention gets more severe with the deployment of distributed memory interconnect due to the tree-based structure. This complicates timing behaviour analysis and harms time predictability, leading to the first research question *Q1* which is summarised as follows.

Q1: Can analytical method predict timing behaviour of memory accesses and bound the worst-case memory access latency in multi-core architectures with shared distributed memory interconnects?

Second, the multi-core architecture inevitably leads to contention over memory accesses. Within the multi-core architectures with shared distributed memory interconnects, the contention to the shared hardware resources, especially the contention to the overlapped data paths across the tree-based interconnect, causes resource sharing issue that memory requests are not fairly served. This potentially causes substantial varying memory access latency. Wide variation of memory access latency leads to wide fluctuation of the overall system performance that the processor can stall with the varying memory response time. In this case, conservative system design has to be considered with pessimistic timing assumptions. This leads to the second research question *Q2* which is summarised as follows.

Q2: Can multi-core architectures with shared distributed memory interconnects be modified at the hardware level to reduce variation of memory access latency?

Third, with the trend of integrating more applications or employing more processors into a system, memory workloads within the multi-core architecture potentially keeps increasing, and the contention over memory accesses aggravates. This increases memory access latency, and high memory access latency degrades the overall system performance. Within the multi-core architecture with the distributed memory interconnect, the architectural bottleneck is either the shared memory resource or the shared tree-based interconnect which connects multiple data paths however overlapped at the tree root. This leads to the third research question *Q3* which is summarised as follows.

Q3: Can multi-core architectures with shared distributed memory interconnects be improved by architectural enhancement for increasing memory workloads?

This research attempts to explore a relevant topic in hardware-software integration. It addresses resource contention and time predictability across the multi-core architectures with shared distributed memory interconnects, contributing towards real-time multi-core systems. It is to be noted that the timing behaviour analysis of memory accesses involves the integration of the shared root memory module into the multi-core architecture. The efficiency of such memory resource directly impacts memory access latency which can be harmed by either varying response time or high response time of this shared memory module. In this case, improvement on independent memory module or memory subsystem is equally necessary.

1.2 Thesis Structure

The reminder of the thesis is structured as follows.

Chapter 2 presents the literature review related to this research. First, it provides the background knowledge and basic concepts of real-time systems. Second, memory or memory subsystem is reviewed with the focus on time predictability and memory latency, including cache, prefetch and scratchpad memory. Third, the shared memory multi-core architecture is reviewed, including memory arbitration schemes and distributed memory interconnects. It also includes a review of state-of-the-art methods to alleviate resource contention within the shared memory multi-core architecture.

Chapter 3 presents the basic architecture and analyses the given research questions. Afterwards, the research hypothesis is summarised based on the problem analysis.

Chapter 4 analyses timing behaviour of the multi-core architectures with shared distributed memory interconnects. First, it proposes the generic analytical flow to predict the timing behaviour of memory accesses by fully exploring the architectural features and statically bound the worst-case memory access latency. This aims to solve the research question *Q1*. Second, it continues to explore and analyse timing behaviour of the *locally arbitrated* interconnect and the *globally arbitrated* interconnect.

Chapter 5 aims to solve the research question *Q2*. It analyses varying memory access latency across the multi-core architectures with shared distributed memory interconnects and proposes an architectural enhancement to reduce variation of memory access latency. Experimental results from hardware simulations and FPGA implementations evaluate the effectiveness of the proposed work.

Chapter 6 aims to solve the research question *Q3*. It analyses resource contention over the multi-core architectures with increasing memory workloads

and proposes an architectural extension of the tree-based interconnect to enhance the multi-core architecture. Experimental results from FPGA implementations with synthetic memory workloads and real-world benchmarks evaluate the effectiveness of the proposed work.

Chapter 7 draws the concluding remarks and proposes the future work.

Chapter 2

Literature Review

This chapter presents literature review related to this research. Section 2.1 provides background knowledge and basic concepts of real-time systems, including time predictability and worst-case execution time (WCET). Section 2.2 reviews memory and memory subsystem with the focus on time predictability and memory latency. The potential improvement methods are also reviewed, including cache, prefetch and scratchpad memory. Section 2.3 presents the review of shared memory multi-core architectures. It includes time-predictable memory interconnects and critical resource contention within multi-core architectures. Afterwards, Section 2.4 summarises and discusses these contents based on the given research questions.

2.1 Real-Time System

Real-time systems must guarantee the system response within specified time constraints thus to provide accuracy and reliability. The term deadline defines

the time that the system must produce response results. Real-time systems can be classified by the consequence of missing the deadline [13] as hard (HRT) where deadline miss with late delivered system response causes disastrous consequences, firm (FRT) where deadline can be occasionally missed but there is no benefits with the late delivered system response, and soft (SRT) where deadline can be occasionally missed and the system response can be late delivered.

Applications with HRT requirements, such as the flight control system, must guarantee no deadline misses. By contrast, applications with FRT or SRT requirements can have an upper limit on the number of deadline misses. For example, radio applications (in terms of software) with HRT requirements have to guarantee no deadline misses to prevent significant quality degradation. By contrast, video decoding applications with SRT requirements can tolerate occasional deadline misses. There will be the modest reduction of video quality with deadline misses as the consequence. In addition, applications can have both HRT requirements and SRT requirements [13].

As the nature of real-time systems, it is crucial to guarantee time-predictable behaviour. It is to determine the range of time that a task executes for and thus prove that a task can meet the time constraints. The common method is to predict the worst-case execution time (WCET) of the task. Figure 2.1 shows task execution times. The shortest execution time is the best-case execution time (BCET), and the longest is WCET. As for the applications with hard real-time requirements, WCET has to be equal to or less than the deadline. WCET analysis actually bounds the limit of execution times, and the related analysis is both application-dependent and hardware-dependent [14].

WCET can be determined by static analysis. In general, it is to analyse the target model to determine the critical execution path and apply timing factor

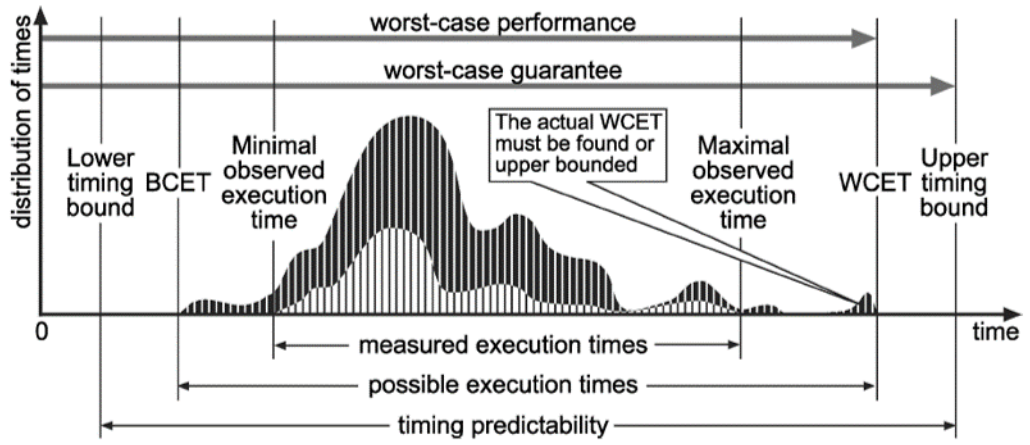

Figure 2.1. Task Execution Time [15]

to the path flow [16]. However, with complicated system architecture, the accurate timing model can be difficult to predict precisely due to the limited predictability of the critical components, such as cache and DDR DRAM. This can also be aggravated by varying state of the system. Therefore, the upper bound on the WCET has to be determined as estimated worst case execution time [16]. In this case, the static analysis leads to pessimistic results.

An alternative is to determine WCET by dynamic time analysis. It is to measure the end-to-end execution times from a number of executions [15]. This actually determines the minimal observed execution time and maximal observed execution time with test case. As shown in Figure 2.1, measurements potentially overestimate BCET and underestimate WCET. Even though the measurement-based method is not that safe for the HRT systems, it is commonly applied in most industry scenarios. The potential measurement can be either to test with the worst-case initial state (determined from execution flow), or to exhaustively test all possible system states.

Based on WCET, task execution is also influenced by the interference or the blocking effects in the system. For example, the execution of the task can be

blocked by other tasks due to the shared resources. This contributes to the worst-case response time, and the worst-case response time of the task has to be guaranteed less than the deadline. Further system design or analysis requires effective scheduling algorithm, such as fixed priority scheduling [17], to order the usage of systematic resources and guarantee the time constraints.

2.2 Memory

Within a system, memory or memory subsystem is to temporarily store the instructions and data that are currently being used or likely to be used by the processor. The hardware module random access memory (RAM) can be both read and written to support the varying instructions and data. The contents in the RAM module can be maintained as long as the power is supplied. It is commonly employed to compose the memory subsystem. RAMs are differentiated by the mechanisms in maintaining their contents. The dynamic RAM (DRAM) uses a single MOS transistor and capacitor to store a single bit data. It has to regularly refresh its content, or the charge leak may lead to the loss of the data. By contrast, the static RAM (SRAM) uses a single flip-flop to store a single bit data. As the SRAM does not have to be refreshed, its access time is close to its cycle time.

Compared with DRAM, SRAM is much faster. However, SRAM is much more expensive than DRAM in terms of the required number of transistors to store data. For example, SRAM with the flip-flop circuit which requires 6 transistors to store a single bit data, while DRAM requires only 1. In this case, a single DRAM module can have much more capacity than a single SRAM module with the same number of transistors. Due to the trade-off between the performance the economical considerations, DRAM is constructed as the

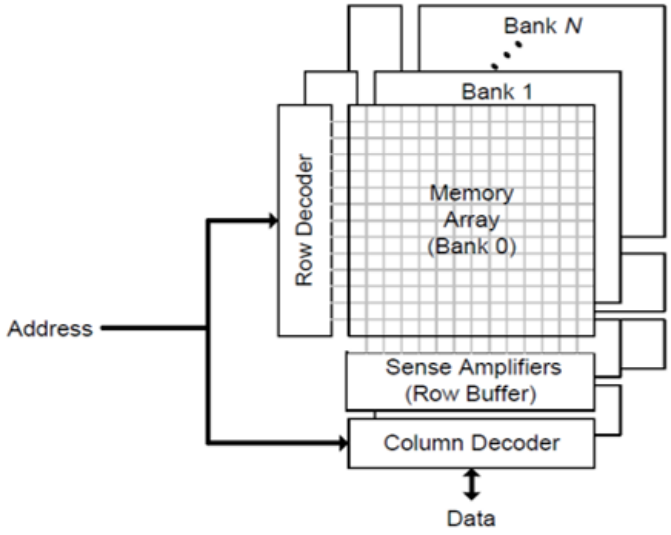

Figure 2.2. DRAM Organisation [18]

primary memory. Modern systems employ the double data rate synchronous DRAM (DDR DRAM) technology. DDR DRAM performs memory access in lower cycle time and provides a higher data transfer rate. For example, DDR3-800E can support 10ns cycle time and 800 megabytes per second transfer rate, with the capacity up to 16 gigabytes.

Figure 2.2 show the modern DRAM organisation. DRAM stores data in a number of banks. Each bank contains a memory array of rows with columns. To access DRAM, the requested bank is first activated. Then the requested row will be selected and loaded into the row buffer which only stores the most recently accessed row. After that, the read or the write can be operated to the columns in the row buffer. Finally, the row in the buffer will be stored back to the memory array. In addition, all the banks will be refreshed regularly to maintain the stored data by the pre-charge.

Due to the latency required to select a row and to pre-charge the row, the internal controller is designed to leave the selected row open for next ac-

cesses [19]. This benefits the performance for the task to access sequential data, however it harms the time predictability. The access time to the primary memory depends on the previous access. It is variable that the same row access requires much less time than the access to a different row. Besides that, the bus turnaround between memory read operation and memory write operation increases the latency. It also requires additional delays if the internal controller issues the pre-charge at the time.

With the analysis above, considering the interference between tasks, DRAM module provides the varying access latency that fluctuates with memory access patterns. This unpredictable timing seriously impacts real-time systems. It is possible to bound the latency of every single memory access with the pessimistic worst-case assumptions. However, this definitely leads to very pessimistic timing results. The efficiency of memory subsystem drops, and execution time increases.

Time-Predictable Memory Access

Chang et al. [20] explores latency variation of DRAM with memory operations caused by access patterns. It also proposes Flexible-LatencY DRAM with the mechanism to balance latency variation across DRAM banks. This also reduces DRAM access latency which suffered from related memory operations. Alternatively, Hassan et al. [21] proposes Reduced Latency DRAM. This type DRAM employs the SRAM-like non-multiplexed address mode instead of the row and column selection. Besides that, the pre-charge is handled by the automatic hardware mechanism rather than the internal controller. Compared with the conventional DRAM, Reduced Latency DRAM provides much lower access latency [22]. It also facilitates the design of time-predictable memory controller.

On the other hand, Akesson et al. [23] develops predictable memory access pattern. It defines the memory accesses into read group, write group, and refresh group. The read or write group only contains the read or write burst to the corresponding banks in sequence. Then the memory groups can be scheduled in pipelined manner referring to the memory operations. Besides that, an amount of additional delay is also imposed between accesses to the same bank. This eliminates the interference among memory accesses, and guarantees a maximum latency bound. Based on this idea, Akesson et al. [23] also develops the predictable memory controller Predator.

Similarly, Paolieri et al. [24] proposes the design of analysable memory controller. It exhaustively analyses the upper bound of each task. Then this controller schedules HRT task considering the worst-case interference and non-HRT task with low priority to eliminate the interference. It actually allocates memory accesses with enough bandwidth to guarantee the time bound. In addition, Goossens et al. [25] develops memory controller to schedule concurrent memory requests by exploiting memory bank parallelism to reduce conflicts. Based on TDM scheme, the bandwidth is shared according to task criticality level at the cost of non-critical task performance.

Memory Latency

Memory latency becomes the major bottleneck in system performance. It potentially leads to expensive processor stalls, and this effect has been aggravated by modern digital products, especially those computationally intensive applications with increasing demand to access high bandwidth data.

To reduce the memory write latency is not a fundamental problem. Methods such as write buffer can be employed to reduce the latency. The processor

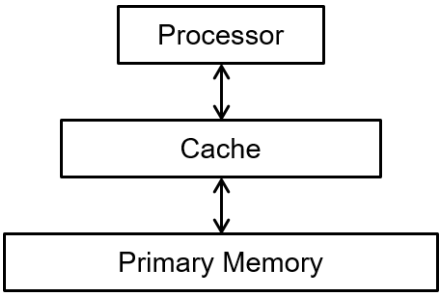

Figure 2.3. Memory Hierarchy

writes the data with address into the buffer. Then the memory write process is completed from the perspective of the processor. It can continue execution with no stalls. By contrast, to reduce the memory read latency is the real challenge. The processor has to wait for the requested data ready before continuing any execution.

The potential solution is multi-threading. With enough support from processor architecture and OS, multi-threading allows the processor switching from a thread to another concurrent thread instead of just waiting. This avoids the expensive processor stalls caused by the memory latency. However, multi-threading requires a large amount of resources and introduces overheads. It may not suit embedded systems.

Alternatively, the design of memory hierarchy is employed to maximise the overall system performance at an acceptable cost. Figure 2.3 shows the general memory hierarchy, with fast, small and expensive cache module at the top, and slow and large primary memory at the bottom. However, the processor still easily loses performance within the memory hierarchy. Further improvement mainly relies on the cache methods to better exploit the locality. In addition, the prefetch methods can also be employed to prepare next data or instructions for processor ahead of time. As for embedded systems, scratchpad memory is introduced as as alternative for cache. The intuition is

to exploit the application behaviour and manage the memory using explicit instructions.

2.2.1 Cache

Cache is constructed using the fast and expansive SRAM in small size. It stores the copies of likely to be used contents from the primary memory, delivering data and instructions to the processor with much faster access time.

Programs follow principle of locality either temporally or spatially. The temporal locality is that if a data location is accessed, it tends to be accessed again very soon. The spatial locality is that if a data location is accessed, its nearby locations tend to be accessed in the near feature. Therefore, the processor tends to access data and instructions from cache more frequently than from the primary memory. This bridges the performance gap between the primary memory and the processor.

The cache operation is controlled by its automatic hardware mechanism. It stores data in a number of blocks or lines. A single cache block can be divided into the valid bit, the tag field and the data field. The valid bit, with the value either 0 or 1, explicitly indicates whether the data is valid or not. The tag field is used to identify the current data. The data field stores a single word or more.

To read data from cache, the processor requests the data with address. The address (physical address) consists of the tag field, the index field and the offset field. The index field is used to locate a specific block from the cache. The tag field is used to be compared with the cache block tag field. If there

is a match with the valid flag, the located cache block is desired. Then the word can be selected from the cache block field using the address offset field.

Cache organisation can be classified according to the schemes in placing cache blocks. If a cache block can only be placed at an exact location, the cache is direct-mapped. If a cache block can be placed at any location, the cache is fully-associative. Otherwise, the cache is set-associative, and the cache block can be placed at a set number of locations. For example, two-way set-associative cache provides two locations to place a single cache block. This also introduces the degree of associativity. The direct-mapped cache is one-way set-associative, and the fully-associative cache is n-way set-associative where n is the total number of its cache blocks.

To write data to cache, the processor supplies the data with address. Then the cache checks for the matching cache block with valid bit 0. It stores the data into this cache block and sets valid bit to 1. However, if the cache block is occupied with the valid flag, the situation differs. As for direct-mapped cache, there is no choice but to discard the original data in the block and replace with the new data.

By contrast, there are several replacement policies for fully-associative and set-associative cache when there is no spare block in the cache or in the set. For example, the random policy randomly discards a cache block thus to store the new data. The first-in-first-out (FIFO) policy is to loop through possible cache blocks, and the data stored earliest will be discarded first. An alternative is the least recently used (LRU) policy. It tracks relative order that cache blocks are accessed. The block which has not been accessed for the longest time will be discarded first. In addition, there are also the bit pseudo LRU policy with an additional most recently used (MRU) bit for each cache block, and the tree-based pseudo LRU policy with binary decision [26].

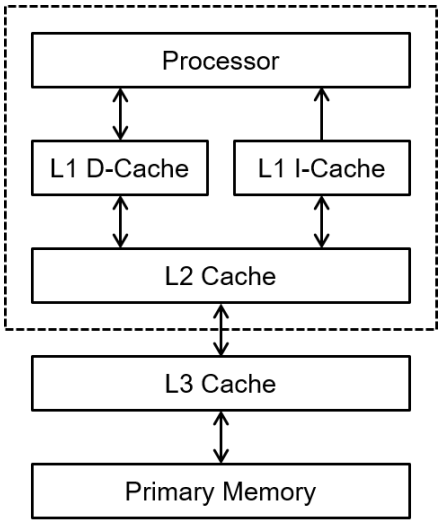

Figure 2.4. Cache Hierarchy

In general, processor issues requests to cache. If desired data is in the cache, it provides good runtime efficiency. If not, cache miss occurs. The processor has to wait for the data transfer delays from primary memory to cache as miss penalty. The miss rate expresses the ratio of cache accesses results in a miss. With high cache miss rate, memory subsystem performance degrades. The compulsory miss occurs in cold starting with an empty cache. The capacity miss occurs when cache is not large enough to store all useful data. The conflict miss occurs in direct-mapped cache or set-associative cache when a useful cache block is discarded but later desired. With the analysis above, both cache organisations and cache replacement polices impact the memory performance [27]. Further improvement is to optimise cache utilisation by better exploiting the principle of locality.

Cache Optimisation

The architectural optimisation is to comprehensively combine the above aspects. It is to employ multiple levels of caches to structure the cache hierarchy,

such as Intel i7 processor with 3 levels as shown in Figure 2.4. For example, the first level employs separate direct-mapped caches in parallel. Although there can be conflict misses due to the lack of associativity, direct-mapped cache provides fastest access as it requires merely the time to access SRAM module. By contrast, a next level is with higher associativity. It is slower and larger to void misses, considering miss penalty between on-chip and off-chip. The cache coherence can be tracked by the snooping or the directory-based scheme.

Alternatively, compiler can improve the performance of the application program by well exploiting the memory hierarchy features with no hardware modifications. It is the most effective method to minimise cache misses and develop data reuse possibility [28]. For example, loop transformation can optimise the data access thus to improve the temporal locality. Then the iteration executions within a loop nest will be optimised but not affecting the execution results. It also provides good data parallelism. The potential methods include loop permutation to exchange the execution order of inner loops in a loop nest, loop fusion to merge separate loops, and loop fission to reversely split a single loop. In addition, loop tiling [16] is to split a loop to iterate over several smaller tiles of data with the array size to fit cache. It significantly improves data efficiency that the data can be more repeatedly reused rather than repeatedly reloaded.

On the other hand, data transformation optimises data layout thus to improve the spatial locality. It actually reorganises data to better fit into cache to reduce conflict misses. For example, inter array padding [29] is to insert unused pads between arrays thus to separate the base addresses, and the pad size is dependent on the array size. This can void cross interference. In addition, data copying [30] is to copy non-contiguous data into consecutive locations. Although it requires necessary processor overheads, data copying

can avoid self interference. This can improve blocked algorithm and guarantee data reuse in tiles. Panda et al. [31] addresses dynamic memory allocation with the integration of memory architecture and processor register. It models memory access and optimises data organisation.

Although cache improves the average-case performance for general purpose applications, it provides unpredictable access time. The execution time even varies for multiple runs of the same program. WCET does not benefit from the employment of cache due to the pessimistic worst-case assumptions. The potential solution is cache partitioning. It divides the cache in partitions, and then assigns specific partitions to tasks exclusively. This reduces interference between tasks. For example, Liedtke et al. [32] reduces cache contention with OS support. It utilises page colouring to partition set-associative cache among applications, and each application is reserved with a partition of cache.

Alternatively, contention locking is to flag specific contents as locked to prevent these contents from being replaced. For example, Aparicio et al. [33] develops instruction locking based on integer linear programming. It analyses a single task and interference among tasks to dynamically load and lock the most relevant cache blocks. Arnaud et al. [34] divides a task into partitions. It exclusively assigns and locks each task partition to a cache partition. Similarly, Vera et al. [35] develops a complier framework for data cache. It eliminates inter-task interference with cache partitioning and ensures predictable intra-task interference with cache locking. This alleviates cache unpredictability and reduces the worst-case memory access time.

In addition, embedded systems also employ SRAM module directly for special purposes, such as FIFO array. This introduces software-managed cache. It is to disable the automatic hardware caching mechanism and explicitly manage the cache operation by software. For example, Hallnor et al. [36] develops in-

direct index cache. With the aim to avoid tag search, it uses the pointer with a simple hash table to locate cache block. This allows the the fully-associative memory structure to be managed by software with less time. Miller et al. [37] develops a software system that allows SRAM module to be automatically managed as cache. Even though the software managed cache requires additional complexity with overheads, it guarantees time predictability that the memory access is with no cache miss.

2.2.2 Prefetch

Prefetch is to prepare next data or instructions ready before they are requested by processor thus to hide long memory latency. Software prefetch is supported by the compiler to appropriately insert explicit instructions at runtime. This requires the application program knowledge. A similar prefetch method is to pre-execute a piece of specific program thus to support the primary program. Obviously, it is difficult to apply for general applications. By contrast, hardware prefetch is to observe and predict future data or instructions, and then fetch the corresponding blocks sufficiently far ahead of the execution. It requires extra memory module, such as buffers, to store the prefetched blocks without polluting cache contents.

If the prefetched block matches the memory request, it significantly reduces the memory access latency. This can benefit BCET. However, the prefetched blocks can be useless, or sometimes the prefetched contents can fail to be ready in time. In this case, prefetch provides unpredictable behaviour. WCET does not benefit from the employment of prefetch due to the pessimistic worst case assumptions. Besides that, prefetch has to overlap with the demand memory operation. As a speculative execution utilising spare memory bandwidth, it is necessary to keep the demand memory access with no stalls. If

prefetch interferes with memory execution, it instead degrades the memory performance. The following section presents potential prefetch schemes.

Locality-based Prefetch

The simplest prefetch scheme is tagged prefetch [38]. It is to prefetch the next block when the current block is accessed. By contrast, Jouppi et al. [39] proposes stream prefetch. It is to prefetch the next consecutive blocks if the current blocks are accessed in sequence. For example, if blocks with addresses $n-2$, $n-1$ and n are accessed, the blocks with address $n+1$ and $n+2$ will be prefetched. If a prefetched block is accessed, a new one will be prefetched. Stream prefetch performs well for instruction. It is ideal for program which follows purely sequential access pattern. However, programs trend to access multiple data arrays rather than only a single data array even in a single process. These interleaved data arrays interference with steam prefetch, polluting prefetch buffer contents. Palacharla et al. [40] develops multiple stream prefetch buffers in parallel. It allows interleaved data arrays to be prefetched concurrently for programs with regular access patterns.

Based on this, Fu et al. [41] proposes stride prefetch. It is to prefetch the next consecutive blocks with an arbitrary stride. The stride is the number of successive array elements, and it can be detected by a branch prediction table and a lookahead program counter [42]. The table tracks the access pattern which is dynamically updated by the branch prediction relying on the processor state and branch history. The counter increments the lookahead amount for the consecutive data array, and sets to 1 for instruction. Prefetch addresses are then decided by adding the lookahead mount to the last addresses. Alternatively, the stride can be decided using block addresses [43]. A reference prediction table is employed to record last accessed addresses.

Then the stride will be decided by calculating the distance between the addresses. When a single blocked is loaded, the first future block address will be decided as $n + d$ where n is the current block address and d is the stride.

Pointer-based Prefetch

The steam prefetch and the stride prefetch rely on good data locality, and the performance is limited to blocks with closeby addresses. However, it is difficult to deal with data on irregular accesses, such as large jumps in addresses. Such data access pattern is to follow pointers in a linked data structure. It is a common way to build a large data structure, such as database. Nodes in the data structure are linked together through pointers. A pointer at a node points to another node, and the new address is decided by calculations on the current node address. In this way, a linked data structure can be considered as a chain of nodes which are dependent of each other.

A prefetch scheme is to straightforward utilise the natural pointers which already exist at nodes [44]. It is to identify the data structure and discover the way to traverse the data structure by the complier. Then the prefetch instructions will be inserted where the node addresses are available and prefetch all possible future nodes. From the perspective of the compiler, it is actually to prefetch the elements to be executed at the start of an iteration. In addition, Roth et al. [45] employs additional artificial pointers at nodes to link non-successive nodes. For example, root jumping is to deploy root pointers between the roots of different data structures which benefits iterations.

Similarly, content-directed prefetch [46] discovers potential pointers by hardware and fetches all possible future blocks. The basic logic is that the address of a block with the pointer will possibly be used to calculate the addresses

Miss Address	Next Probable Address		
Addr 1	1st P Addr	2nd P Addr	...
Addr 2	1st P Addr	2nd P Addr	...
...	1st P Addr	2nd P Addr	...
Addr N	1st P Addr	2nd P Addr	...

Figure 2.5. Markov Prediction Table [47]

of future blocks which are pointed to. These future addresses are definitely within the range of the same data structure and share the common base addresses. The effectiveness of content-directed prefetch is affected by the prediction accuracy, and the compiler can help to decide the most beneficial blocks at runtime.

History-based Prefetch

As for more irregular accesses with no particular pattern, the prefetch scheme is to utilise the access history. For example, Joseph et al. [47] employs Markov prediction. It utilises miss history to obtain correlations between addresses, provides possible future addresses and prioritises these blocks. Figure 2.5 shows the correlation table that two temporally successive miss addresses can be paired with each other. The first address of a pair is the parent, and the latter is the child. As a parent address may have any number of child addresses, the parent is considered as the key to select the child afterwards. When a particular address is accessed, its correlative addresses are decided by checking the history with the table. Then the blocks will be prefetched in sequential queue according to their weight. Markov prediction performs well for programs with repeated irregular accesses, but the prediction is only made with limited history of current parent addresses.

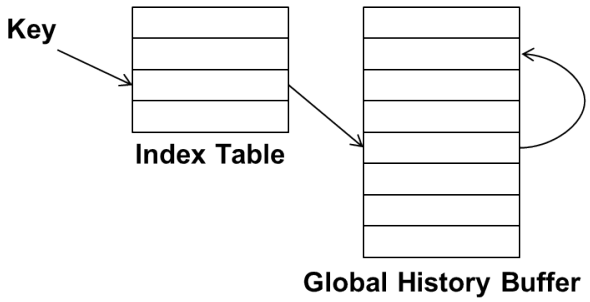

Figure 2.6. Global History Buffer [48]

By contrast, Nesbit et al. [48] develops global history buffer to improve the accuracy of the correlation. Rather than storing fixed amount of history for each address, global history buffer only stores the history for the most recent miss addresses. As shown in Figure 2.6, history addresses are stored indirectly. The index table is accessed by a key, and it indexes global history buffer through a hash map. The buffer is to store the most recent miss addresses. The link pointer is also employed to link each buffer block with the same key from the index table. In this way, these related addresses are chained. The key can be selected as a piece of related information which is useful in these links, such as a miss address or the program counter. This allows better predictions with more complete knowledge of the addresses.

In addition, Srinath et al. [49] proposes feedback-directed prefetch to dynamically utilise the access history. It monitors the last prefetch performance thus to adjust the current prefetch execution. For example, it can tack whether the prefetched blocks are useful. If the ratio of the useful blocks to the total blocks is high, the number of blocks to prefetch for a next time can be increased. If low, the number can be reduced or even stops with an unacceptably high number of useless blocks. The feedback-directed method can support any prefetch scheme above. It contributes to hybrid prefetch scheme. For example, stride prefetch can be improved to adjust its prediction region

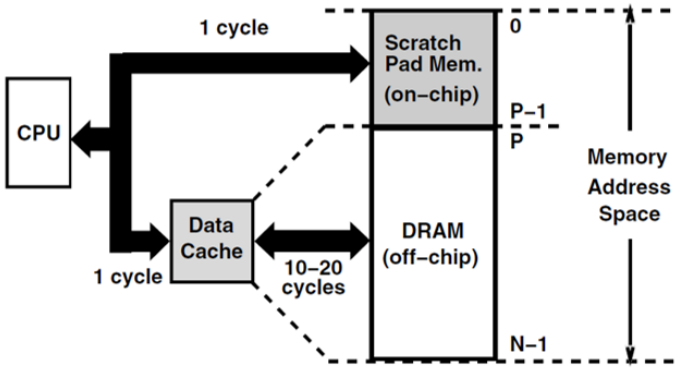

Figure 2.7. Scratchpad Memory Address Configuration [51]

at runtime. If necessary, it can predict addresses at $n + 2d$, $n + 3d$ or even $n + 4d$ ahead of program execution rather than just $n + d$.

2.2.3 Scratchpad Memory

Scratchpad Memory (SPM) is introduced to embedded systems as a design alternative for cache [50]. It is small high speed on-chip SRAM operated by software. Similar to L1 cache, SPM resides on-chip locally to the processor. It is to temporarily store data or instructions for processor with fast access. Compared with cache, SPM only contains data field, with no tag field or the tag comparator to acknowledge hit or miss. SPM operation is in full control by software.

Figure 2.7 shows the common address configuration of the memory architecture with SPM. The on-chip SPM occupies a distinct part of memory address space, with the rest occupied by the off-chip primary memory. Data or instructions can only be transferred to or off SPM with explicit instructions. DMA-based data transfer can also be employed to avoid processor involvement [52]. To access SPM, the desired block is selected directly with no

check or comparison. This guarantees a single clock cycle access with no miss, providing predictable memory access latency. Compared with unpredictable cache, SPM is more popular for real-time embedded systems. The contents in SPM have to be managed effectively which requires the analysis of the application program and the support from the compiler. The following section presents potential SPM allocation schemes.

Scratchpad Memory Allocation Scheme

In general, data and instructions can be statically partitioned between the on-chip SPM and the off-chip primary memory. As for the program data, the most frequently accessed data blocks can be allocated to SPM [53]. It relies on the compiler to recognise the access frequency of the data in the application and partition these data blocks with the consideration of SPM size. Then less accessed data blocks can be allocated to the primary memory. This increases data reuse possibility in SPM. However, there may be conflicts between data blocks. The data partitioning with access frequency actually changes the entire data layout. It can cause the non-conflicting data blocks to conflict with each other, even leading to unpredictable results.

By contrast, Panda et al. [54] allocates the most conflicting data to SPM. Rather than only partitions of data, it analyses entire arrays. Large arrays which fail to fit SPM size will be allocated to primary memory, and the remaining arrays will be allocated to SPM as many as possible with conflict factor. The conflict factor is based on both the array access frequency and the possibility to conflict with other arrays. SPM allocation priority will also be assigned to the array with higher conflict factor and smaller size. This introduces a comprehensive analysis of data features affecting partitioning [55], such as variables or constants and life time of variables.

Suhendra et al. [56] improves WCET through SPM allocation. It discovers the worst-case path and analyses the access frequency of the relative elements along the path. Then the most frequently accessed elements will be allocated to SPM thus to reduce WCET. However, it changes the entire data layout, potentially changing the worst-case path. The method can also be improved to measure the maximum potential WCET reduction of an element over all possible execution paths thus to provide global WCET optimisation of element allocation.

Instructions can also be allocated with similar static scheme. For example, Steinke et al. [53] partitions and allocates instructions according to the execution frequency. It relies on the compiler to decompose a function into basic blocks. Then the most frequently executed basic blocks, rather than the complete function, will be identified and allocated to SPM. However, it may require additional jump instructions for the basic blocks in the on-chip SPM to jump to the basic blocks in the off-chip primary memory. This introduces overheads, especially for a large number of basic blocks with relatively small size. Therefore, allocation priority will be assigned to the consecutive basic blocks with no additional modifications.

The static SPM allocation schemes above improves the program execution efficiency. The performance bottleneck seems to be the SPM size. With too small size, SPM can only store very limited amounts of the beneficial contents, which may not even achieve the locality benefits of a cache. Besides that, the static management potentially under-utilises SPM that the contents are allocated only once and remain constant at runtime. This fails to satisfy the varying memory requests of dynamic program behaviours, especially those applications with multiple compute intensive regions. Alternatively, dynamic SPM allocation scheme performs dynamic data transfers between on-chip SPM and off-chip primary memory to respond to dynamic memory requests.

Kandemir et al. [57] dynamically transfers data tiles at runtime with inserted instructions. It relies on the compiler to first recognise the memory data layout and discover the data access patterns. Then the data will be partitioned into small tiles. An ideal data tile is supposed to have high access frequency and with the appropriate size to fit SPM. From the perspective of the compiler, this scheme combines both data transformation and loop transformation to optimise application programs. It is actually to dynamically partition the available memory space between the competing arrays in each loop thus to increase data reuse and reduce data transfer. However, the performance relies on good overall data regularity and regular access patterns, such as scientific applications with large loop nests.

Chen et al. [58] improves data tile transfer for programs with irregular access patterns. In addition to access frequency of array elements, the compiler also analyses the runtime costs of each transfer and the access benefits of each data tile using SPM. Only the most beneficial transfers with relatively low costs will be selected with related control instructions inserted with the knowledge above. If the benefit of a data tile access using SPM is unacceptably low, it may even be accessed directly from the primary memory to avoid unnecessary transfers.

The dynamic SPM allocation is also applicable to instructions. For example, Verma et al. [59] selects and dynamically transfers the most frequently executed program parts for SPM. Steinke et al. [60] limits the potential program parts only to loops where a set of instructions continually repeat. Udayakumaran et al. [61] improves to further analyse the costs and the benefits of all potential program parts. Besides that, it also partitions the program into regions by using the most beneficial program points. A potential program points can be the locations where the program has a significant change in locality behaviour, such as at the start of a loop nest or even at the start

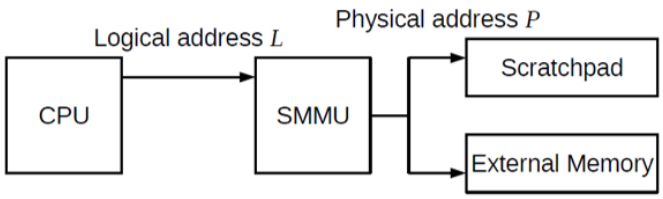

Figure 2.8. Memory Configuration with Scratchpad Memory Management Unit [63]

of a inner loop. This comprehensive analysis guarantees maximum benefits utilising SPM.

Scratchpad Memory Manager

SPM allocation introduces a significant challenge to develop memory-aware explicit management, and the design burden lies on the compiler. Alternatively, Egger et al. [62] develops dedicated SPM manager on hardware which exploits MMU to monitor the logical addresses thus to dynamically control the page transfers for SPM contents. It requires hardware modification to SPM module. The address comparator is combined to decide whether the request can be an access to SPM or not. Based on the compiler knowledge, SPM operation is managed by SPMM at runtime, rather than by the control instructions inserted by the compiler. In this way, SPM manager actually manages SPM module as a global resource.

Similarly, Whitham et al. [64] develops hardware SPM management unit. Figure 2.8 shows memory configuration with SPM management unit which employs a table to track logical addresses of blocks in SPM. A request accesses SPM if the desired address is within the range of addresses in the table. If a new block is transferred to SPM, the table will be updated for future accesses.

It also remains the block logical address constant to the processor, and only the block physical address may be changed through the transfer to or off SPM. This makes access independent of all others for data accesses.

In addition, Whitham et al. [65][66] develops trace SPM to support processor. The trace SPM stores explicitly traces or microcode from post-compiler stage. It is then deployed as a part of processor pipeline to directly control without decoding process. This explores instruction parallelism at the microcode level and statically schedules the frequently executed code blocks to reduce execution time.

2.2.4 Summary

This section reviews the memory subsystem. The primary memory is construed with DRAM providing varying access latency that fluctuates with memory access pattern. The performance gap between the processor and the primary memory is the bottleneck of overall system. The potential solutions to reduce memory latency includes cache, prefetch and SPM.

Cache is construed with SRAM. It exploits the locality benefiting the average-case performance, but WCET does not benefit from the deployment of cache. The optimisation includes to either improve data efficiency or alleviate cache unpredictability to reduce the pessimistic worst case. By contrast, prefetch is based on speculation. It benefits the BCET, but prefetch is unpredictable due to the pessimistic worst-case assumption. Beside that, SPM is constructed with SRAM only storing desired data or instructions with explicit instructions. It requires the accurate program behaviour analysis to develop efficient memory aware management. Although it lacks the generality, SPM is more applied in real-time embedded systems with tight worst-case bound. Ap-

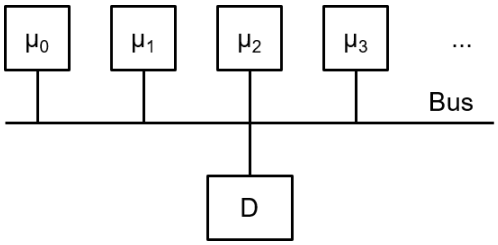

Figure 2.9. Bus-based Multi-Core Architecture

plying HRT constraints requires feasible memory architecture with efficient management scheme for time predictability.

2.3 Shared Memory Multi-Core Architecture

The multi-core architecture is increasingly adopted in modern design. Figure 2.9 shows the conventional bus-based multi-core architecture. It employs a shared bus to connect a number of processors and the shared memory module. The communications between the processors, or between the processors and the memory module must be delivered through the shared bus, such as AHB (Advanced High-Performance Bus) in SoC design [67]. Once a single access occurs, the bus is blocked, which leads to severe contention.

Alternatively, the crossbar design can be deployed to replace the shared bus. For example, Figure 2.10 shows AXI (Advanced eXtensible Interface) interconnect [68]. The design utilises a set of switch boxes, employing dedicated links to connect multiple masters (processors) and multiple slaves (memory modules). This allows multiple those accesses between different master-slave pairs to occur simultaneously. The crossbar interconnect alleviates the contention issue. However, it requires additional hardware resource, and the centralised design limits the maximum synthesisable clock frequency. In this

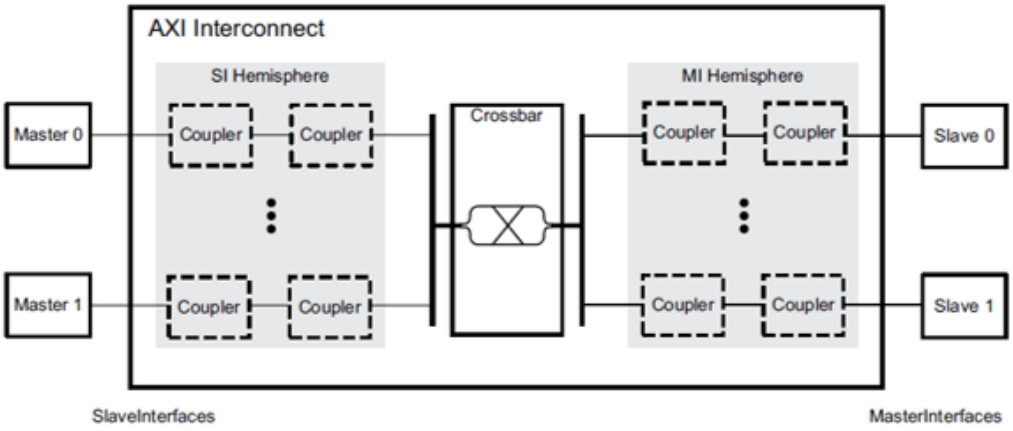

Figure 2.10. AXI Interconnect [68]

case, the multi-core architecture is not scalable with the increasing number of processors.

A promising architecture is network-on-chip (NoC) [6][7]. As shown in Figure 1.1, NoC architecture employs a packet switching communication network. Each processor is connected through a router to the communication network. Then communication packets can be delivered across the network with the routing traffic. In this way, the NoC allows easy data sharing that a processor can communicate with its target directly, and the processor potentially accesses its target with less contention compared with the bus-based multi-core architecture. This promotes many-core architecture.

The design burden of of NoC lies on the router architecture and the communication mechanism [8][9]. For example, Hermes [69] provides the best-effort communication packet delivery across the router network between processors, however it does not provide performance guarantees. By contrast, Ætheral [70] employs TDM scheme in circuit switching across the router network. It can provide the guaranteed service routing that a packet can be delivered within the fixed timing period.

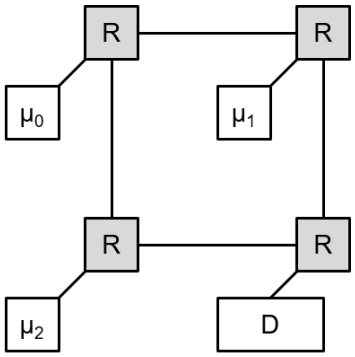

Figure 2.11. Shared Memory Network-on-Chip Architetcure

NoC design also involves the research on the layered protocols and the topology. For example, Figure 1.1 and Figure 2.11 both show the basic NoC architecture based on Manhattan mesh. As for memory or memory subsystem within the NoC architecture, it employs local memory modules to each processor for fast accesses. The shared memory modules are commonly connected to the edge of the router network as shown in Figure 2.11. In this case, such accesses to memory can be operated in a similar to the requests to I/O peripherals.

2.3.1 Memory Arbitration

In the general case, the multi-core or many-core architectures above have more processors than memory modules. This potentially causes contention over memory accesses, which will get more severe with the trend of integrating more processors into the system. The contention to the shared resource leads to interference among applications and harms real-time tasks where time predictability is necessary. With the aim to achieve time predictability (or simply predict the behaviour of memory accesses), the multi-core and many-core architectures typically utilise an arbitration scheme to provide timing guar-

antees within the architecture. The potential memory arbitration schemes include time division multiplexing (TDM) scheme, round-robin scheme and priority-based scheme.

Time Division Multiplexing and Round-Robin

Time division multiplexing (TDM) scheme statically pre-determines a periodical cycle with a number of time slots where a time slot is a time interval that a single memory access can be served. Each processing core is then assigned with a corresponding time slot in the cycle. In this way, the arbitration periodically iterates over the cycle and checks the eligible sates. With strict TDM scheme, the arbitration takes the same time for each slot regardless of whether there is any memory request from the corresponding processing core. This isolates every single memory request, allowing easy timing analysis. However, the shared resource can be idle even with pending requests, potentially wasting bandwidth.

Based on the periodical cycle, round-robin scheme allows better utilisation of the available bandwidth. The arbitration skips any empty slot, and immediately moves to the next eligible one with a pending memory access. Besides that, round-robin scheme serves memory requests at any time, rather than TDM scheme only at the slot boundary. This promotes the work-conserving manner that the shared resource is never idle with pending memory requests. In this case, considering the potential empty time slots that memory requests can be stacked with no service, memory accesses with strict TDM scheme can suffer additional interference without work conservation. By contrast, round-robin scheme is widely used in practical applications even with increased complexity in timing analysis. It can also be extended to guarantee the processor with the minimum service for a number of memory requests,

such as weighted round-robin arbitration scheme [71]and deficit round-robin arbitration scheme [72].

Priority-based Arbitration

Priority-based arbitration scheme allows memory requests with higher priority to be served before others. The basic scheme is static-priority arbitration by which fixed priorities are statically assigned to processors with no changes at runtime. However, the system can be flooded by memory requests with higher priority, while low priority requests are stacked lack of memory service. This potentially leads to starvation. By contrast, dynamic-priority arbitration allows most critical memory requests to be served first. This relies on effective analysis to accurately recognise the program behaviour.

Static-priority arbitration scheme can be extended to provide service to a number of memory requests before others. This promotes frame-based static-priority arbitration, which makes the trade-off between the frame size and the latency. Based on this idea, Akesson et al. [73] proposes credit-controlled static-priority arbitration scheme. It employs rate control with static priority. In this case, the available bandwidth will be regulated and allocated to the processors according to the application behaviour. With the aim to better utilise the available bandwidth, credit-controlled static-priority allows the successive memory requests from a specific processor to be in transfer with the rate control. Besides that, it can be applied in mixed-criticality systems. Similarly, Shah et al. [74] develops priority-based budget scheduling scheme. The above static-priority arbitration scheme with rate control relies on effective analysis to accurately recognise the program behaviour and only benefit specific applications.

2.3.2 Distributed Memory Interconnect

The conventional centralised implementation of an arbitration scheme is to employ a single arbiter, allowing the related arbitration decisions to be made at the central location. However, this requires the design of single-clock-cycle data path, and such design suffers the long-wire issue [75][76]. However, as the number of processors grows, the logic size of the arbiter hardware increases, which limits the maximum synthesisable clock frequency.

A promising approach recently investigated is to employ distributed memory interconnects that the tree-based structure with pipelined stages can break the critical path of the multiplexing into multiple shorter steps with smaller logic size. Although this introduces additional delays in terms of clock cycles, the latency across the interconnect is actually reduced, as much higher synthesisable clock frequency is allowed on the distributed hardware. It scales to a large number of processors. The critical path of the distributed interconnect remains constant as it is dedicatedly constructed with the growing number of processors. In addition, pipelining is also supported.

In general, the distributed memory interconnects can be classified as the *locally arbitrated* interconnect and the *globally arbitrated* interconnect. The *locally arbitrated* interconnect is simply constructed upon a distributed binary arbitration tree which multiplexes the memory requests from processors to the shared root memory module through the distributed data path. By contrast, based on a distributed binary arbitration tree, the *globally arbitrated* interconnect integrates the global scheduling to the distributed data paths. The following section reviews state-of-art research on the design of memory interconnects.

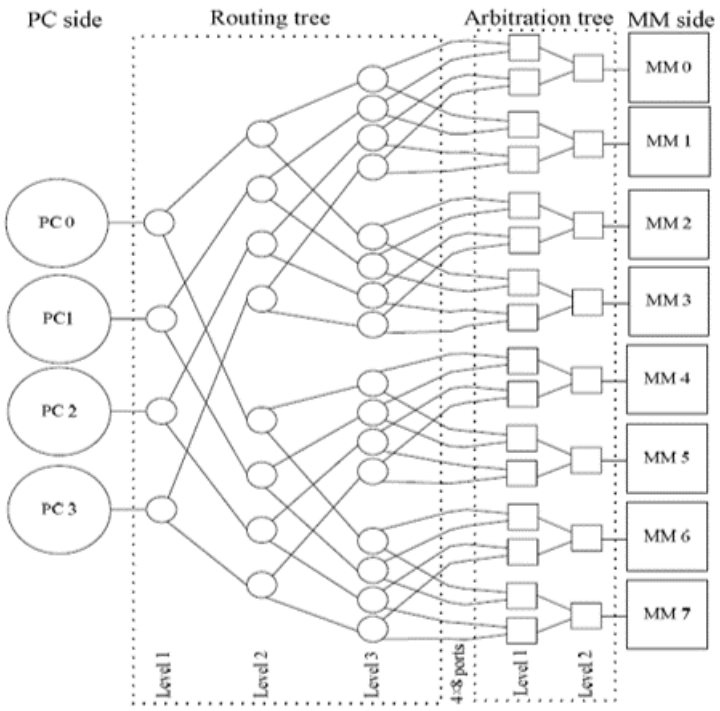

Figure 2.12. Mesh-of-Trees [77]

Mesh-of-Trees

Figure 2.12 shows the design of Mesh-of-Trees (MoT) interconnect [77] which can relay memory requests from multiple clients PCs to multiple memory modules MMs. Based on the mesh-of-trees topology [78][79], MoT is constructed by the coupling of a routing tree and an arbitration tree. The routing tree provides independent routing paths for multiple memory requests from different clients PCs. The arbitration tree multiplexes these memory requests and further routes these memory requests to the corresponding memory modules MMs. As the memory address range can be partitioned across these multiple memory modules MMs, accesses to different memory modules MMs can be processed concurrently. The memory response transfers back to the corresponding clients PCs through the same network with a reverse process. A similar design can be found in the research [80].

In addition, MoT can be developed to support the communications between multiple processors and multi-bank L1 memory [81]. It can allow memory requests to transfer through the network within a single clock cycle. If a memory request is stalled by the arbitration tree due to the contention to a specific memory module MM, this request will be quarantined and allowed to relay in the next clock cycle. In this way, MoT is actually deployed as a set of switches coordinated by a global control signal and operates as the circuit-switched round-robin manner with centralised control, enabling simple timing analysis.

MoT potentially increases system bandwidth by spreading memory addresses over multiple MMs, and the single-clock-cycle design obviously provides low latency in terms of clock cycles. However, the single-clock-cycle data path inevitably requires the long-wire design [82]. This increases the wire delay. With the expanding system configuration (i.e., the number of PCs and MMs), the logic size of the centralised design increases logarithmically, which severely limits the maximum synthesisable clock frequency. Besides that, MoT only provides fair memory accesses with relatively balanced workload patterns on PCs. If the memory workloads are unbalanced, there can be many interfering requests stalled due to the contention to the shared MMs. This actually blocks MoT paths, leading to varying memory access latency.

Arbitration Tree

In contrast to the conventional centralised design above, distributed memory interconnects are also emerging in multi-core and many-core architectures, including the *locally arbitrated* interconnects and the *globally arbitrated* interconnects. Among *locally arbitrated* distributed memory interconnects, single arbitration tree [83] can be developed and constructed with globally syn-

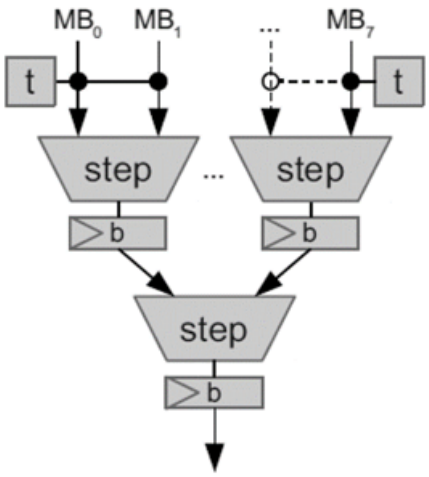

Figure 2.13. Arbitration Tree [83]

chronised timestamps as shown in Figure 2.13. It multiplexes the memory requests from the clients MB to the shared memory module.

The arbitration node is known as the step in this design, and each step only operates the arbitration of the two local inputs. When a memory request is relayed to the arbitration tree, an individual timestamp is generated and attached to the request by the globally synchronised component t. Then the local step will only allow the request with the lower timestamp to relay. In this way, the local arbiter at each distributed multiplexing stage applies the first-come-first-served scheme. As the memory requests can be stalled by any subsequent step, the local buffer b is employed just after each step. This also avoids the long-wire design and increases the maximum synthesisable clock frequency. The memory response can transfer back to the corresponding clients through a similar tree path that operates the demultiplexing but without the timestamp arbitration.

This design utilises binary arbitration tree to construct the distributed memory interconnect. The pipelined stages break the long wire and allows a higher synthesisable clock frequency to support architectural scalability that

the hardware consumption scales linear to the increasing number of clients. However, the application of this design is very limited, and the interconnect is only feasible to very few platforms. Initially, this arbitration tree is dedicatedly designed for the Microblaze-based system which employs AXI bus [84], thus with small numbers of outstanding memory requests. In this way, the local first-come-first-served scheme can only provide time predictability to such fixed memory access pattern. In AXI single-mode data transfer, there will be only 1 outstanding memory request from a single Microblaze [85]; while in AXI burst mode, the successive burst beats from a single Microblaze will share the same timestamp. Only with these assumptions, there will be very limited interfering requests in the tree network, and the worst-case latency of a request is the wait for the transfer of all the requests with the lower timestamps.

Bluetree

Bluetree [86][87] is initially developed for NoC architecture. It is the external memory tree attached to the NoC architecture, which provides a second network exclusive for the accesses to the shared memory module. This separates the memory traffic from the processor router network and thus prevents memory access from interfering with communication between processors.

As shown in Figure 2.14, Bluetree is constructed by a set of pipelined stages of 2-to-1 multiplexers, connecting the clients μ at the tree leaves to the shared memory module D at the tree root. When a client issues a memory request, this request will be multiplexed and relayed to the shared memory across the distributed tree network. Then the memory response returns to the corresponding client across the bi-directional interconnect. Arbitration occurs to each Bluetree multiplexer in the memory request path to decide which request

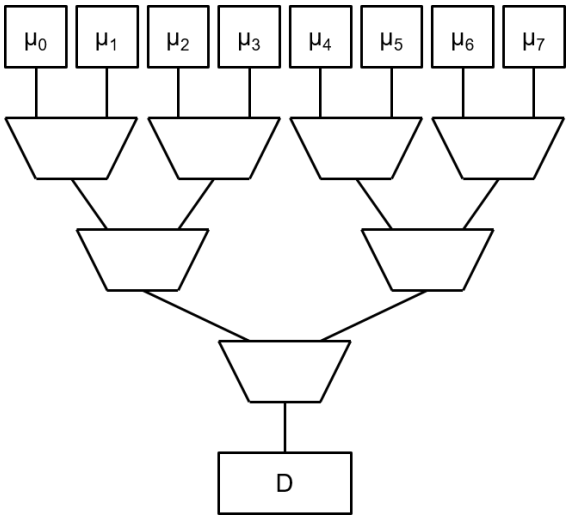

Figure 2.14. Bluetree

from either the local client direction to be relayed to the memory direction, potentially to the next Bluetree multiplexer stages. The Bluetree multiplexer actually employs independent local arbitration scheme to allow the flow of successive memory requests from a local direction. On the other hand, the memory response path is non-blocking (in any Bluetree multiplexer). The Bluetree multiplexer simply decides the route direction of the memory response. In addition, a buffer is designed in each Bluetree multiplexer to break the critical data path of memory accesses.

The *locally arbitrated* Bluetree interconnect does not require full synchronisation, and the Bluetree-based memory architecture allows multiple memory requests to be transferred through the tree network simultaneously. This potentially allows high service bandwidth, which supports multiple memory requests to be relayed in the tree interconnect. However, the contention across the Bluetree memory request path requires complicated analysis on the time-predictable behaviour. Besides that, the local arbitration scheme requires extra logic size to each arbiter at the distributed pipelined stage, which potentially limits the maximum clock frequency in turn.

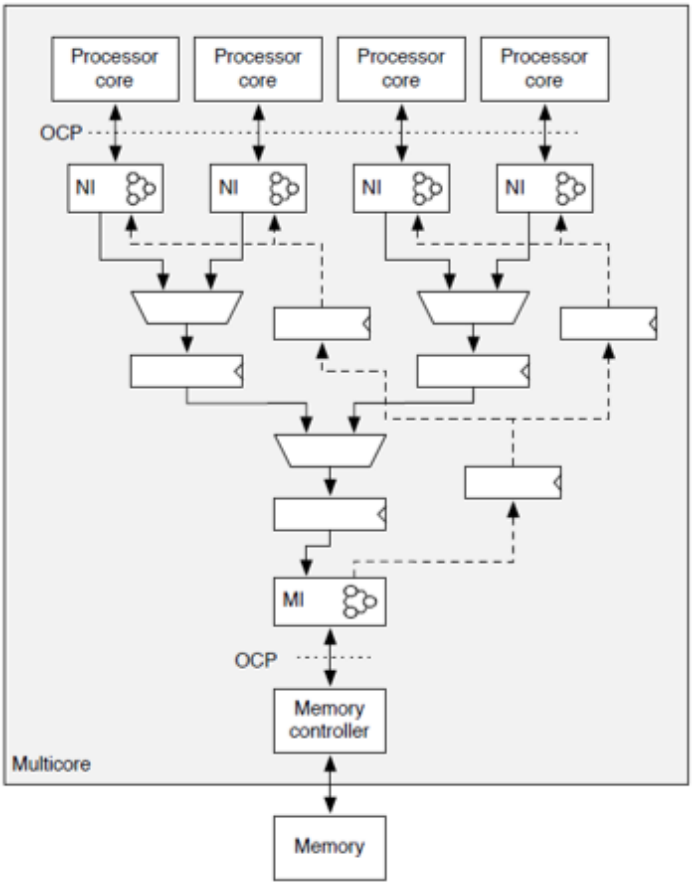

Figure 2.15. TDM Tree [88]

TDM Tree

Compared with the *locally arbitrated* design, the *globally arbitrated* distributed memory interconnect integrates the global scheduling to the distributed data paths. Among *globally arbitrated* interconnects, TDM Tree [88] is built upon the integration of global TDM scheduling components with a distributed tree-based multiplexing network as shown in Figure 2.15.

With TDM Tree interconnect, multiple clients (processor core) are connected through the network interface NI, the distributed tree network and the memory interface MI to the shared memory module. NIs coordinate to perform

the global TDM scheduling scheme, and memory requests can also be stalled at the corresponding local NIs. When a TDM time slot arrives, a memory request from a specific processor is allowed to relay to the tree network. With the global scheduling interval, there is no contention to the shared resources, neither the data paths nor the root memory module. This guarantees no interference exists between memory accesses. The scheduling interval can be decided by the timing behaviour of both the shared memory module and the tree network. As for memory response, the processor core tag can be used to determine the demultiplexing path.

Due to the TDM policy, there is no interfering request in the tree network at a time. In this way, there is no design of arbitration or flow control in the tree network, and it is only deployed with pipelined multiplexing stages for high synthesisable clock frequency. However, the deployment of TDM Tree requires strict synchronisations and complex schedules. The global scheduling interval also limits the potential service bandwidth. Besides that, TDM Tree does not support work conservation. The tree network and the shared memory module can be idle even with many memory requests stalling at local NIs. This potentially leads to a considerable waste of available bandwidth.

Globally Arbitrated Memory Tree

Based on the global scheduling interval, Globally Arbitrated Memory Tree (GAMT) [89][90] extends the distributed multiplexing tree with priority-based rate control schemes, aiming to better utilise the available bandwidth with flexibility. In addition to global TDM scheduling, GAMT also supports frame-based static priority and credit-controlled static priority. The selection of arbitration scheme affects the time-predictable behaviour of memory accesses.

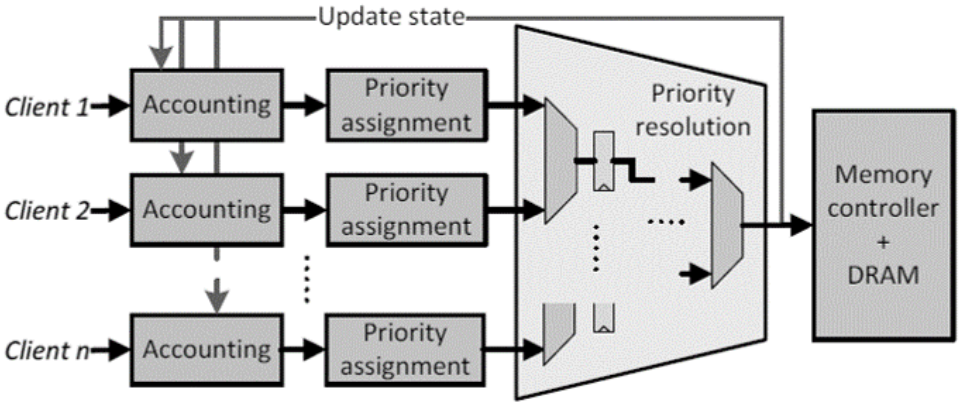

Figure 2.16. GAMT [89]

As shown in Figure 2.16, the accounting component makes the scheduling decision based on the selected arbitration scheme. The accounting logic is to track the eligibility of a client to receive service using a global scheduling interval. Then the priority assignment component is to assign the unique priority to the request according to the accounting logic. The priority resolution component is actually the distributed tree network of pipelined multiplexer stages, and it guarantees the service to the request with the highest priority. In this way, the arbitration is implemented by both the global scheduling and the local distributed multiplexing. Once the request reaches the memory controller as shown in the figure, the feedback is generated to update the eligibility status of the corresponding client with the accounting logic. If two scheduled requests arrive at a multiplexer stage simultaneously, the request with the higher priority will be allowed to relay, leaving the other request to be rescheduled at the next scheduling interval.

GAMT applies the coordination of local distributed arbiters with the global scheduling interval. It can allow successive memory requests from a specific processor to relay to the tree network with an arriving time slot. This potentially provides sufficient flexibility for mixed-criticality systems with diverse

bandwidth and latency requirements. However, the deployment of GAMT requires strict coordination and complex schedules, potentially suffering the synchronisation issue. GAMT can only benefit specific applications, as it is generally hard to model the memory requests on hardware, unlike task scheduling in operating systems. Besides that, GAMT also introduces high logic overhead. Both the accounting component and the priority resolution component in the GAMT require large logic size to check the request priority. This inevitably limits the maximum clock frequency.

2.3.3 Critical Resource Contention

In the general case, multi-core and many-core architectures are typically designed towards average-case performance, with inevitable interference between software components or tasks within the system. The consequent contention to the shared hardware resources can block the flow of memory requests or communication packets. It can also block any subsequent flow, even causing the resource sharing issue. The architectural bottleneck of the multi-core system is the shared memory module, and the contention over memory accesses aggravates with an increasing number of applications integrated or processors employed, potentially leading to high and varying memory access latency.

The intuition to alleviate such contention is to deploy an effective root memory subsystem as presented in Section 2.2. For example, the appropriate memory hierarchy can be employed instead of a single shared module to reduce the average-case memory latency. An alternative method is to better utilise local memory modules at the processors by exploiting the locality of the applications. Following this idea, Dasari et al. [91] analyses memory contention within the multi-core architecture with cache partitioning for precise

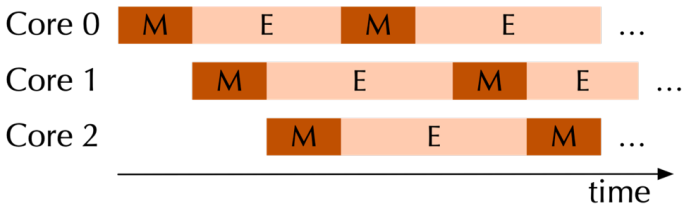

Figure 2.17. Memory Centric Scheduling

timing bounds. In addition, SPM can be employed with explicit instructions, which facilitates timing behaviour analysis and verification within the multi-core architecture. For example, Kelter et al. [92][93] analyses the multi-core architecture with local instruction SPM and data SPM. It bounds the memory access latency considering the dependency that the current access latency impacts subsequent accesses.

Based on a time-predictable hardware platform, the design of real-time system tends to further employ additional partition schemes to achieve temporal isolation. For example, phased execution model can be employed to refactor code into memory phases and execution phases. The memory phase is to store data into local cache or SPM, and the execution phase is for the processor execution only with accesses to local cache or SPM. During the operation, the memory phases between different processors can interference with each other. This causes processor stalls due to the resource contention. Therefore, memory centric scheduling can be utilised to reschedule memory phases and execution phases as shown in Figure 2.17 where M is denoted for memory phase and E for execution phase. This effectively avoids the interference of memory phases between processors.

Following the above idea, PREM [94] is proposed to further improve time predictability. It decomposes tasks to execution phases and memory phases, allowing system bandwidth reservation to each phase. It also reserves enough

budget for the sections of code that cannot be explicitly refactored into memory phases and computation phases. A similar method is periodic memory regulation, such as MemGuard [95]. It reserves memory bandwidth for each core to void inter-core memory access interference.

By contrast, the tree-based architecture appears more sensitive to the contention to the critical resource — the overlapped data path is also shared as well as the root memory module, especially the multi-core architecture with the *locally arbitrated* interconnect. It allows multiple memory requests in transfer simultaneously, leading to contention over either the shared root memory module or the overlapped data paths across the tree-based interconnect. Once the root memory is occupied, the entire request flow is blocked. This complicates the timing behaviour analysis of memory accesses across such architecture.

Garside et al. [96] analyses the worst-case memory access latency across the *locally arbitrated* Bluetree-based architecture regardless of memory workloads. It explores the latency of a memory request over the blocking that this memory request experiences across the given request path. Afterwards, the worst-case blocking condition is proposed that all multiplexers are blocked along the request paths to the memory module. In this case, all multiplexers harm the request flow within this architecture. Results of the worst-case blocking are from simulations employing the above condition. The timing behaviour analysis of multi-core applications employing Bluetree is based on this method [87]. However, this worst-case analysis only considers limited blocking. For example, a buffer is also included within a Bluetree multiplexer to break the critical data path of memory accesses according to the distributed design. If all these buffers across the request path are occupied, potentially by interfering memory requests, this condition will harm the flow of memory requests as well, leading to additional blocking compared with the above analysis. It is possible

that Bluetree in the above research is designed with a different architectural choice.

Compared with the *locally arbitrated* design, the multi-core architecture with *globally arbitrated* interconnect can provide contention-free request paths, avoiding memory access interference. The relevant timing analysis is relatively simple and the worst case potentially reflects the global scheduling cycle. For example, GAMT [89][90] employs additional rate control schemes based on the reserved time slots for clients. It guarantees each client sharing the root memory resource a minimum reserved rate or bandwidth after a maximum latency [90].

In addition to the above research, the impact of the critical resource contention has been widely studied on multi-core and many-core architectures, especially in NoC application. The contention to the shared router blocks the flow of communication packets, leading to high and varying latency across the processor router network. The term hot spot [97][98] or hot module [99] is introduced to describe the components which are with limited bandwidth and in high demand by other components in a system. The following section reviews state-of-art methods to alleviate such critical resource contention.

Resource Reserving

An improvement method to alleviate critical resource contention is to regulate the accesses to such hot module based on resource reserving. For example, Walter et al. [99] proposes wormhole switching with credit-based control scheme to regulate the accesses to a single hot module in the NoC architecture. Each source in the NoC architecture owns a quota that limits the number of packets it can send towards the hot module. When a source

quota is exhausted, it can only resume transmission after being granted an additional credit. The hot module is designed with an allocation controller that receives credit request from a source and sends credit reply back. This aims to distribute the limited bandwidth of the hot module almost equally among sources in the system to achieve resource fairness. Similarly, Hansson et al. [100] proposes channel tree where time slots are reserved for multiple communication channels to achieve contention-free routing in the NoC architecture. This potentially provides isolation among applications.

On the other hand, Shi et al. [101] analyses wormhole switching with priority-based control scheme for NoC communications. It proposes an off-line schedulability analysis and provides the upper bound of packet latency across this architecture, based on delays caused by direct interference from higher priority traffic flows and indirect interference from other higher priority traffic flows, aiming to achieve time predictability of real-time communication in the NoC architecture.

Investing Additional Hardware Resource

An alternative method to alleviate critical resource contention is to invest additional hardware resource. For example, virtual channel with flow control are commonly employed to enhance the shared router in NoC applications. This can alleviate the router contention from multiple communication flows across the NoC architecture and provide flexibility in channel utilisation. Kavaldjiev et al. [102] proposes virtual channel router with simplified dynamic arbitration, aiming to reduce the size of the virtual channel design. Based on this research, Kavaldjiev et al. [103] analyses NoC applications with virtual channel that the traffic guarantee can be provided based on virtual channel allocation. On the other hand, Mello et al.[104] analyses and evaluates the

effectiveness of virtual channel on reducing latency of communication packets over NoC dimensioning. It also discusses that virtual channel can reduce latency variation of communication packets, potentially contributing to insure quality of service of the NoC architecture.

As for the shared memory multi-core architecture with tree-based interconnects, Audsley et al. [105] proposes that multiple memory modules or multiple memory banks can be independently employed at the root of the distributed interconnect. Each memory module or memory bank is potentially designed for each criticality level to support mixed-criticality systems. This provides diverse memory features and potentially increases memory bandwidth. This research also analyses time predictability of the proposed memory architecture. However, it leaves the design burden to the shared root memory controller, and such centralised design at the tree root impacts the maximum clock frequency of the synthesised hardware.

Message Combining

A different method to alleviate critical resource contention is message combining. For example, Pfister et al. [97] introduces the term hot spot and analyses the contention across multi-stage interconnection networks, especially those networks with shared memory resource. It proposes that pairwise memory requests directed at an identical memory location can be combined at a switch node into a single memory request. Afterwards, when the memory reply to the combined memory request reaches a switch node where it was combined, multiple memory replies are decombined for multiple individual requests. Based on this research, Lee et al. [106] demonstrates the limitations of the above pairwise combining as network size increases and proposes k-way combining where up to k messages can be combined at switche nodes.

As for the shared memory multi-core architecture with tree-based interconnects, this method can be modified that memory requests simultaneously arriving at an arbiter can be combined. This aims to reduce the number of memory requests thus to reduce contention over overlapped data paths. Then memory response is decombined to multiple individual ones along the response path. However, this design leaves the burden to the shared root memory module with increased data width. Besides that, it requires an increasing logic size for arbiters at each pipelined stage. This harms the synthesisable clock frequency with an expanding system i.e., an increasing number of processors.

2.3.4 Summary

This section reviews the shared memory multi-core architectures. The memory arbitration is employed to provide the time-predictable memory access behaviour within multi-core architeucres, including TDM, round-robin and priority-based schemes. TDM isolates memory accesses, and round-robin can provide work-conserving manner. By comparison, round-robin has been widely used despite with complicated timing behaviour analysis. In addition, priority-based schemes relies on effective program analysis to benefit specific applications.

The conventional centralised implementation of the arbitration scheme leads to limited maximum synthesisable clock frequency with the increasing number of processors. By contrast, the distributed implementation constructs the tree-based memory interconnect with a number of pipelined stages to break the critical path of the multiplexing into multiple smaller steps with smaller logic size. It allows much higher clock frequency, scaling to a large number of processors. The *locally arbitrated* interconnect is simply constructed upon

a distributed binary arbitration tree which multiplexes the memory requests from processors to the shared root memory module through the distributed data path. Based on this architecture, the *globally arbitrated* interconnect further integrates the global scheduling to these distributed data paths.

Besides that, this section also includes a review of state-of-the-art methods to alleviate critical resource contention within multi-core and many-core architectures.

2.4 Summary and Discussion

This chapter presents the literature review related to this research, and basics of real-time systems is reviewed in Section 2.1.

Section 2.2 reviews memory or memory subsystem with potential improvement methods on guaranteeing time predictability and reducing memory latency. This contributes to an effective shared memory module or shared memory subsystem for the multi-core architecture that the response time of the shared root memory can be bounded with the worst case and low. In addition, such memory can be further improved for increasing workloads with the reviewed methods. However, the integration of the root memory into the multi-core architecture complicates the timing behaviour analysis of memory accesses across such architecture.

Section 2.3 reviews the shared memory multi-core architectures, potentially with shared distributed memory interconnects, and the focus is resource contention. Such architectures can provide the time-predictable hardware platform, and the review also includes potential methods to alleviate critical resource contention. Table 2.1 summaries these methods including resource

Table 2.1. Summary of Methods to Alleviate Critical Resource Contention

Method	**Example**	**Architecture**	**Research Focus**
resource reserving	Walter et al. [99]	NoC	credit-based wormhole switching
	Hansson et al. [100]	NoC	reserving time slots for multiple channels
	Shi et al. [101]	NoC	latency bound of priority-based wormhole switching
investing resource	Kavaldjiev et al. [102]	NoC	reducing size of virtual channel design
	Kavaldjiev et al. [103]	NoC	providing traffic guarantee based on virtual channel allocation
	Mello et al.[104]	NoC	reducing latency with virtual channel over NoC dimensioning
	Audsley et al. [105]	tree-based interconnect	multiple memory at interconnect root
message combining	Pfister et al. [97]	multi-stage interconnect	pairwise combining of memory requests at switch nodes
	Lee et al. [106]	multi-stage interconnect	k-way combining of memory requests at switch nodes

reserving, investing additional hardware resources and message combing, with examples and references. This table also compares these methods in terms of corresponding architectures and brief descriptions of research focus. It is to be noted that only the research by Shi et al. [101] and Audsley et al. [105] involve time predictability on either the NoC architecture or the tree-based interconnect. As the examples of these methods are initially designed for different architectures with different assumptions, the effectiveness of these methods on alleviating critical resource contention across the multi-core architectures with shared distributed memory interconnects requires further analysis in the following research.

Chapter 3

Multi-Core Architectures with Shared Distributed Memory Interconnects

Based on the literature review, this chapter continues to analyse the given research questions. The reminder of this chapter is structured as follows. Section 3.1 analyses the basic multi-core architecture with the shared distributed memory interconnect, including a comparison of the *locally arbitrated* interconnect and the *globally arbitrated* interconnect. Section 3.2 presents problem analysis over the multi-core architectures with shared distributed memory interconnects and suggests potential improvement methods. Afterwards, Section 3.3 summarises the research hypothesis.

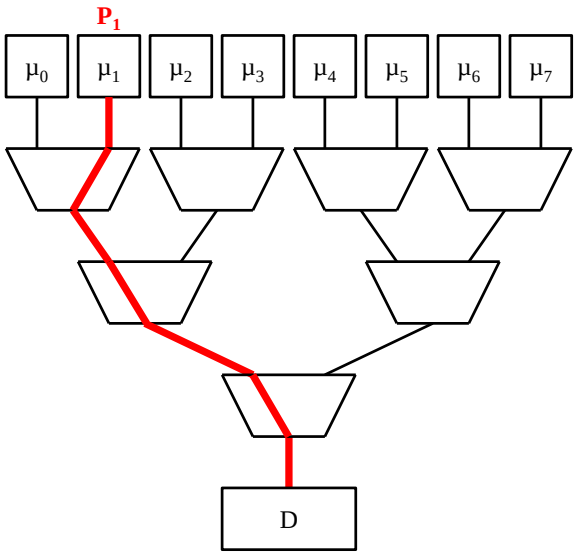

Figure 3.1. 8-Client Basic Architecture

3.1 Basic Architecture

This research focuses on the multi-core architecture with shared distributed memory interconnect. An example of the basic architecture that 8 clients share 1 memory module is shown in Figure 3.1. It consists of 8 clients, the interconnect, and the shared memory module. A client can be a single processing core or a multi-core processor, and denoted by μ_i where i is the client index. Each client has a memory access path P_i, with P_1 for the client μ_1 as highlighted in the figure. The interconnect B employs multiple stages of 2-to-1 multiplexers to construct the tree network, connecting clients at the leaves to the shared memory module D at the tree root. Across this bi-directional network, memory requests issued by the clients are multiplexed and relayed to the shared memory, and memory responses return to the corresponding clients. As the number of clients grows, the tree network scales with more multiplexer stages, which increases the interconnect depth N_β. For example, N_β is equal to 3 in this figure.

In general, multi-core systems are typically designed for average-case performances, with inevitable interference from the software components or tasks. The consequent contention to the shared hardware resources, especially the shared memory resource, can cause blocking within the system, potentially leading to varying and high memory access latency. As shown in Figure 3.1, both the root memory module and the overlapped data paths are shared by all clients. The entire architecture can be affected by any blocking in the overlapped path, especially with blocking closer to the tree root (i.e., the root multiplexer stage β_0 or the shared memory module D). When there is a memory request occupying the memory module, many others stall, just waiting along the shared paths. These pending requests block the entire interconnect and also block subsequent requests.

Locally Arbitrated Interconnect and Globally Arbitrated Interconnect

The distributed memory interconnects can be classified as the *locally arbitrated* interconnect and the *globally arbitrated* interconnect. The *locally arbitrated* interconnect, such as Bluetree [86][87], allows multiple memory requests in transfer simultaneously, however potentially leading to varying blocking behaviour within the architecture. This makes the timing behaviour analysis of memory accesses complicated and challenging, and memory access latency can vary severely that the average case can be much lower than the worst case.

By contrast, the *globally arbitrated* interconnect aims to provide the contention-free data paths for memory accesses. It integrates the global scheduling scheme with the distributed multiplexing stages and budgets a processor with limited memory bandwidth towards temporal isolation. This poten-

tially limits the average-case behaviour to be or to be similar to the worst case, facilitating the timing behaviour analysis of memory accesses.

However, the *globally arbitrated* design fails to alleviate memory workloads. Instead, restrictive reservations reduces the processor utilisation within such architecture, resulting in relatively high average latency. It potentially slows down or even stalls a processor, consequently degrading the overall system performance. For example, TDM Tree [88] strictly shapes memory accesses to the shared resource and thus eliminates contention. In this case, pending memory requests can stall even with an empty interconnect and an idle memory module, wasting available bandwidth. Similarly, GAMT [89][90] employs additional rate control schemes based on the reserved time slots. It only benefits specific applications with successive memory requests.

Besides that, the *globally arbitrated* design requires global clock synchronisation for complex scheduling as well as strict coordination, potentially suffering synchronisation issue. Memory requests can be distributed in time, and they must wait for the strict scheduling interval. If a memory request misses its reserved time slot, it has to wait for a next eligible cycle. In this case, memory access latency increases with proportional to the global cycle.

Table 3.1 summaries the *locally arbitrated* interconnect and the *globally arbitrated* interconnect, with examples and references. This table also compares the *locally arbitrated* interconnect and the *globally arbitrated* interconnect in terms of brief descriptions on memory access latency and time predictability, which is the research focus. Both interconnects can provide the time-predictable hardware platform, and time predictability of the *locally arbitrated* interconnect is more complicated and challenging. The *locally arbitrated* design can lead to severely varying memory access latency. By contrast, the *globally arbitrated* design is with high average memory access latency.

Table 3.1. Summary of Distributed Memory Interconnects

Interconnect	**Example**	**Memory Access Latency**	**Time Predictability**
locally arbitrated	Arbitration Tree [83]	memory access latency varies	yes, but complicated
	Bluetree [86][87]		
globally arbitrated	TDM Tree [88]	high average memory access latency	yes
	GAMT [89][90]		

In the following research, the *locally arbitrated* architecture is defined as the multi-core architecture with the *locally arbitrated* interconnect, and the *globally arbitrated* architecture is the multi-core architecture with the *globally arbitrated* interconnect. According to the previous analysis, the *locally arbitrated* architecture can be deployed for any application however leading to varying blocking behaviour within such architecture. By contrast, the deployment of the *globally arbitrated* architecture relies on the analysis of accurate application behaviour that the memory access pattern, such as the dependency of successive memory accesses, impacts the applicability and the effectiveness of the *globally arbitrated* architecture.

3.2 Problem Analysis

Based on the above analysis, this section continues to analyse resource contention and time predictability across the *locally arbitrated* architecture and the *globally arbitrated* architecture.

3.2.1 Time Predictability

In this research, time predictability requires to statically analyse the timing behaviour of memory accesses across the multi-core architectures and bound the worst-case memory access latency within such architecture. The *locally arbitrated* architecture can allow multiple memory requests in transfer, potentially leading to varying blocking behaviour and thus complicates the timing behaviour analysis. By contrast, the *globally arbitrated* architecture aims to achieve contention-free data paths based on the global scheduling interval, provided that the strict synchronisation can be guaranteed. In this case, the

worst case can be bounded reflecting its global cycle. By comparison, time predictability of the *locally arbitrated* architecture is more challenging, and guaranteeing time predictability of the *locally arbitrated* architecture is the research focus.

In practice, there is often uncertainty with memory access profiles, such as uncertainty on the number of memory requests and memory issuing time instants. In such cases, exact timing analysis is not valid. Bluetree, as an example of the *locally arbitrated* interconnect, has been deployed in multi-core applications [87]. The relevant timing behaviour analysis of the Bluetree-based architecture only considers very limited blocking effect due to the architectural choice [96][87], and the worst-case analysis employs a simulation-based method [96].

Instead, this research proposes the generic analytical flow to predict the memory access behaviour by fully exploring the architectural features of the *locally arbitrated* design and statically bound the worst-case memory access latency across the *locally arbitrated* architecture. The details are shown in Chapter 4 which aims to solve the research question *Q1*. In addition, Chapter 4 continues to explore and analyse the timing behaviour of the *locally arbitrated* architecture and the *globally arbitrated* architecture, with experiments for demonstration.

3.2.2 Varying Memory Access Latency

The multi-core architecture inevitably causes contention over memory accesses, potentially leading to substantial varying memory access latency. Wide variation of memory access impacts the system performance and also affects the system design choice as the memory access latency is the main part form-

ing the overall execution time. The multi-core architecture with the shared distributed memory interconnect appears to be more sensitive to the resource contention due to the tree-based structure that the overlapped data paths are also shared by all clients as well as the root memory module.

The *locally arbitrated* architecture allows multiple memory requests in transfer simultaneously and thus leads to varying memory access latency due to varying blocking behaviour within such architecture. By contrast, the *globally arbitrated* architecture budgets processors based on the global scheduling interval. This aims towards contention-free data paths, potentially limiting the average case to be similar to the worst case. However, memory requests can be more distributed in time. In practice, varying memory workloads may not perfectly satisfy the global scheduling interval, and this potentially leads to substantial varying memory access latency. Based on this analysis, both *locally arbitrated* and *globally arbitrated* architectures potentially suffer variation of memory access latency. By comparison, the *globally arbitrated* architecture is more suitable for specific applications according to the previous analysis. Therefore, the research focus is to reduce variation of memory access latency across the *locally arbitrated* architecture.

Although the deployment of the *locally arbitrated* architecture does not require to model memory requests in applications, it potentially suffers severe varying memory access latency due to the varying blocking behaviour. This is caused by the contention to the critical resource, especially the contention to the overlapped data paths which complicates blocking behaviour analysis. Due to the architectural feature of the tree-based structure, any blocking closer to the tree root blocks the entire interconnect. Requests can be blocked waiting in paths, which also blocks subsequent requests. Besides that, with blocking along the interconnect, new issued requests can overtake and get ahead of pending requests due to local arbitration at distributed stages. In

this case, the sequence of pending requests is broken, and requests are not fairly served. This leads to additional blocking to a portion of pending requests which suffer higher latency than the average case as a consequence. This also complicates timing behaivour analysis which requires to derive detailed status of memory flows and local arbiters at every pipelined stage. In turn, it can lead to conservative system design with enough safety margin to guarantee memory response.

The methods to alleviate critical resource contention has been widely studied on multi-core architectures. A method is to regulate the accesses to critical resource based on resource reserving. However, it relies on effective program analysis to benefit specific applications. This analysis is similar to that of the *globally arbitrated* architecture, and the applicability and the effectiveness very much depend on memory access patterns. A different method is message combining which can potentially combine memory requests thus to reduce the contention to the overlapped data paths within the tree-based interconnect. However, this significantly increases the data width to either the shared root memory controller or the pipelined arbiter stages especially the stages closer to the tree root. It actually tends to move the workloads and leave the burden to the centralised location, i.e., the interconnect root, where the increasing logic size with an expanding system severely harms the synthesisable clock frequency.

Instead, an alternative method is to invest additional hardware resources, such as employing virtual channel with flow control to alleviate the router contention from multiple communication flows in NoC applications. Following this idea, this research proposes the root queue modification with the root queue management to smooth resource sharing and reduce variation of memory access latency across the *locally arbitrated* architecture. In general, it is to employ and utilise an additional hardware queue with queue manage-

ment between the root of the *locally arbitrated* interconnect and the shared memory module. The details are shown in Chapter 5 which aims to solve the research question *Q2*.

3.2.3 Increasing Memory Access latency

Memory workloads within the multi-core architectures potentially keeps increasing with the trend of either integrating more applications or employing more processors, and the contention over memory accesses aggravates. The *locally arbitrated* architecture allows multiple memory requests in transfer simultaneously that the contention over either the shared root memory module or the overlapped data paths increases with increasing memory workloads. This leads to increasing memory access latency. By contrast, the *globally arbitrated* architecture avoids contention over memory accesses based on global scheduling interval. However, it does not alleviate memory workloads. For example, with the increasing number of processors, memory access latency increases as well as the global scheduling cycle. In this case, both the *locally arbitrated* architecture and the *globally arbitrated* architecture suffer critical resource contention, potentially leading to increasing memory access latency with increasing memory workloads.

The methods to alleviate critical resource contention has been widely studied on multi-core architectures. The intuition is to deploy more effective root memory subsystem or local memory modules to processors. The effectiveness of such method essentially relies on the analysis of accurate application behaviour thus to exploit data efficiency as well as to predict the memory access behaviour to bound the worst case. A method is to regulate the accesses to critical resource based on resource reserving. The design of real-time systems also tends to achieve temporal isolation. Similar to the analysis of the

globally arbitrated architecture, it aims to provide contention-free behaviour without alleviating memory workloads. A different method is message combining which can potentially combine memory requests to reduce resource contention over the overlapped data paths within the tree-based architectures. However, it fails to alleviate workloads to the shared root memory module.

Instead, an alternative method is to invest additional hardware resources, such as employing virtual channel to alleviate the router contention from multiple communication flows in NoC applications. As for the tree-based structure, Audsley et al. [105] proposes that multiple memory modules or memory banks can be independently employed at the root of the *locally arbitrated* Bluetree-based architecture (potentially through a shared memory controller), aiming to provide diverse memory features to support mixed-criticality systems. This potentially increases memory bandwidth. However, it moves the design burden to the shared memory controller, and the shared tree root remains the architectural bottleneck of the *locally arbitrated* interconnect.

Following the idea of multiple root memory modules being engaged, this research proposes an architectural enhancement that the tree-based distributed memory interconnect can be extended to a multi-memory interconnect based on the mesh-of-trees topology [78][79]. In this way, the new distributed multi-memory interconnect allows multiple processors to simultaneously access multiple memory modules with time-predictable behaviour. This potentially alleviates the contention to a single shared memory module as well as the shared distributed memory interconnect thus to reduce memory access latency in the average case. The details are shown in Chapter 6 which aims to solve the research question *Q3*.

3.3 Research Hypothesis

Based on the above problem analysis, the hypothesis of this research is summarised as follows.

Distributed memory interconnect for multi-core architectures can be improved by architectural enhancement on hardware that the root queue modification with the root queue management reduces variation of memory access latency and the mesh-of-trees extension enables multiple processors to simultaneously access multiple memory modules, whilst guaranteeing the time-predictable behaviour.

The reminder of this research continues to address resource contention and time predictability across the multi-core architectures with shared distributed memory interconnects. This aims to improve the shared memory multi-core architecture with guaranteed time-predictable behaviour, reduced variation of memory access latency and enhanced architectural features for increasing memory workloads, contributing towards real-time multi-core systems.

Chapter 4

Analysing Timing Behaviour of Multi-Core Architectures with Shared Distributed Memory Interconnects

Based on the analysis in previous chapters, this chapter continues to analyse the timing behaviour of the multi-core architectures with shared distributed memory interconnects. The reminder of this chapter is structured as follows. Section 4.1 analyses the resource contention and the blocking effect across the data paths within the *locally arbitrated* architecture and proposes the generic analytical flow to predict the memory access behaviour and statically bound the worst-case memory access latency. This section aims to solve the research question *Q1: Can analytical method predict timing behaviour of memory accesses and bound the worst-case memory access latency in multi-core architectures with shared distributed memory interconnects?* Section 4.2 further

explores and analyses the *locally arbitrated* architecture and the *globally arbitrated* architecture, with experiments to demonstrate the timing behaviour of both architectures. Afterwards, Section 4.3 summarises this chapter and presents discussion.

4.1 Time Predictability of Multi-Core Architectures with Shared Distributed Memory Interconnects

In general, the multi-core architectures are typically designed for good average-case performance that software components or tasks can contend for the shared hardware resources. Within such systems, memory accesses over the distributed tree-based interconnect can cause contention to both the overlapped data paths and the shared root memory module. The challenge is to achieve time predictability which requires to statically analyse the timing behaviour of memory accesses across the multi-core architecture and bound the worst-case memory access latency within such architecture. This is particularly important for real-time applications and will be solved in this section.

The *locally arbitrated* architecture allows multiple memory requests in transfer leading to varying blocking behaviour and thus complicates the timing behaviour analysis. By contrast, the *globally arbitrated* architecture aims to achieve contention-free data paths based on the global scheduling interval, provided that the strict synchronisation can be guaranteed. In this case, the worst case can be bounded reflecting its global cycle. By comparison, time predictability of the *locally arbitrated* architecture is more challenging, and

guaranteeing time predictability of the *locally arbitrated* architecture is the research focus.

Compared with another *locally arbitrated* Arbitration Tree [83], the design of Bluetree is more feasible which has been deployed in multi-core applications [87]. The relevant timing behaviour analysis of the Bluetree-based architecture only considers very limited blocking effect due to the architectural choice [96][87], and the worst-case analysis employs a simulation-based method [96]. Instead, this research proposes the generic analytical flow to predict the memory access behaviour by fully exploring the architectural features of the *locally arbitrated* design and statically bound the worst-case memory access latency across the *locally arbitrated* architecture. The *locally arbitrated* Bluetree is shown as an example in this research.

4.1.1 Bluetree-based Architecture

The Bluetree-based architecture follows Figure 3.1 where the Bluetree interconnect B employs multiple stages of Bluetree multiplexers to connect clients μ_i at tree leaves to the shared memory module D at the tree root. Figure 4.1 shows the design of the Bluetree multiplexer with requests coming from two client directions.

Arbitration occurs in the request path (RQ) to decide which direction of request to be relayed to the memory direction, and potentially next Bluetree multiplexers. The blocking factor α of the internal arbiter is defined such that every α requests from *Client Direction 0* can be blocked by at most a single request from *Client Direction 1* where *Client Direction 0* can be considered as the local high-priority path, and *Client Direction 1* is the local low-priority path. Starvation can be prevented by allowing a request from the low-priority

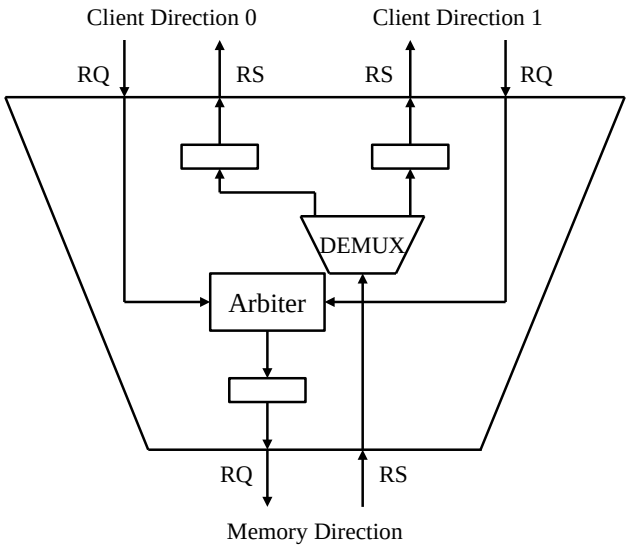

Figure 4.1. Bluetree Multiplexer

path to be relayed for every α requests from the high-priority path. If there is no request from *Client Direction 0*, the arbiter imposes no blocking on *Client Direction 1* with outstanding requests. The implementation of the local arbiter requires an internal blocking counter. When the blocking factor is set as $\alpha = 1$, Bluetree can be considered as the distributed binary tree with local round-robin scheme, providing relatively fair access to the shared memory for all clients.

On the other hand, the response path (RS) is non-blocking. The internal demultiplexer simply decides the route direction of the memory response as shown in Figure 4.1. In addition, a buffer is implemented along each direction as a common pipeline design practice. The Bluetree multiplexer interface is designed to operate in the client-server manner, which allows each local Bluetree multiplexer to function independently, without requiring the operating state knowledge of any other Bluetree multiplexer nearby. The Bluetree interconnect does not require full clock synchronisation.

Figure 4.2. Bluetree Communication Packet Format

The communication packet format across the Bluetree-based architecture is shown in Figure 4.2, including the command field CMD (i.e., the memory command type such as memory read or memory write), the client identifier field CPU_ID, the address field ADDR, and the 32-bit data field DATA.

In a memory request packet, CMD '0' indicates a read request, and CMD '1' indicates a write request. In a memory response packet, CMD '0' indicates a read response, and CMD '1' indicates a write acknowledgement. CPU_ID is required for the packet transfer across the interconnect, and it is used for each distributed multiplexing stage to track or decide the route. When a client issues a request, the corresponding CPU_ID is encoded by the local arbiter at each Bluetree multiplexer to track the route: left shift by 1 bit with '0' for the local high-priority path, or left shift 1 bit with '1' for the local low-priority path. CPU_ID is also used by the demultiplexer along the response path to decide the route back to the corresponding client, decoded by the right shift operation at each local stage. Within a Bluetree-based architecture, 8-bit client identifier field can support a maximum Bluetree depth $N_\beta = 8$.

In the above design, the total bit-width of a memory packet also decides the width of the data bus as well as the Bluetree multiplexers. It is to be noted that this design is reconfigurable and allows flexible extension. For example, a priority field can be employed in the route information for the priority-based arbitration scheme. An extra interface is needed for the conversion of the packet format (e.g., converting the packet format between the Bluetree interconnect and the AXI bus). In addition, the design of the Bluetree interconnect is independent of memory addressing scheme.

4.1.2 Timing Behaviour Analysis

The *locally arbitrated* Bluetree-based architecture is initially designed to provide good average-case performance and guarantee the worst-case memory access latency. In general, the latency t of memory access ω consists of three parts as follows: the request path latency t_{RQ}, the root memory latency $t(D)$, and the response path latency t_{RS}.

$$t(\omega) = t_{RQ}(\omega) + t(D) + t_{RS}(\omega) \tag{4.1}$$

When there is no contention to the memory access ω, i.e., in the best case, it takes 1 clock cycle to cross each pipelined stage, along both the request path and the response path. Therefore, the best-case request path latency $t^{BC}_{RQ}(\omega)$ equals to N_β. According to the Bluetree multiplexer design, the response path is non-blocking. Then the best-case overall latency t^{BC} of the memory access ω can be calculated as follows.

$$\begin{aligned} t^{BC}(\omega) &= t^{BC}_{RQ}(\omega) + t(D) + t_{RS}(\omega) \\ &= 2 \times N_\beta + t(D) \end{aligned} \tag{4.2}$$

The best-case memory access latency $t^{BC}(\omega)$ gives the minimum latency that a memory access experiences across the *locally arbitrated* architecture. It is based on the assumption of no contention, i.e., every pipelined stage is always in the idle status, ready to accept the request and the response without any delay. When there is resource contention to either the data path or the shared root memory, the request may be blocked, which leads to increasing request path latency $t_{RQ}(\omega)$, and consequently increasing total latency $t(\omega)$.

The analysis of blocking effect starts from a single Bluetree multiplexer. Figure 4.3 shows the blocking behaviour of a single Bluetree multiplexer with

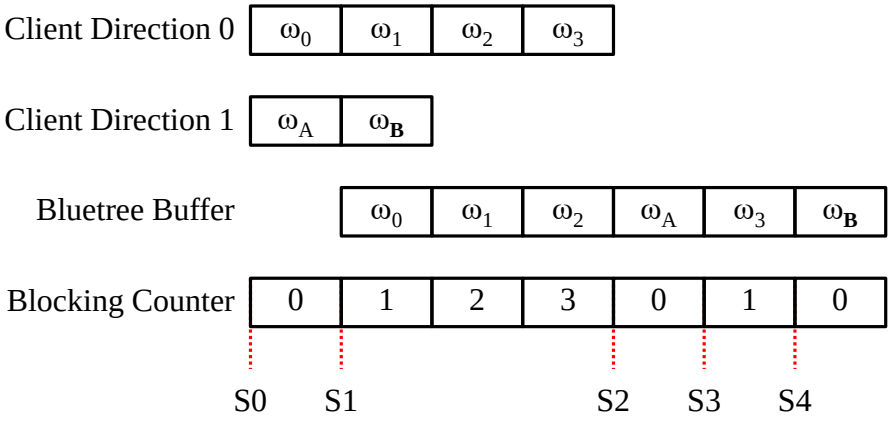

Figure 4.3. Blocking Behaviour of Bluetree Multiplexer

the local arbitration scheme. With the blocking factor α, every α requests from the high-priority path *Client Direction 0* can be blocked by at most a single request from the low-priority path *Client Direction 1*, and every single request from the low-priority path can be blocked by up to α requests from the high-priority path.

As shown in the graph, there are four successive memory requests ω_0, ω_1, ω_2 and ω_3 from *Client Direction 0*, and two successive requests ω_A and ω_B from *Client Direction 1*. ω_0 and ω_A arrive to the Bluetree multiplexer simultaneously, and the only Bluetree buffer stores these requests in sequence according to the local arbitration scheme. The blocking factor α is set as $\alpha = 3$, and the value of the local *blocking counter* also changes with the data transfer which can be split into stages as follows.

S0 The Bluetree buffer is empty and ready to accept a next memory request. The value of the *blocking counter* is 0.

S1 As *Client Direction 0* is the high-priority path, ω_0 is allowed to cross. By contrast, ω_A from the low-priority *Client Direction 1* is blocked. This also blocks ω_B. Simultaneously, the *blocking counter* increments. The similar be-

haviour repeats for next requests ω_1 and ω_2 from *Client Direction 0*, blocking the transfer of ω_A and ω_B from *Client Direction 1*.

S2 When the value of *blocking counter* reaches its maximum limit α that $\alpha = 3$, ω_A from the low-priority *Client Direction 1* is finally allowed to cross to the Bluetree buffer, with ω_3 from the high-priority *Client Direction 0* blocked. Simultaneously, *blocking counter* resets.

S3 The request transfers as S1 process: ω_3 from *Client Direction 0* crosses to the Bluetree buffer, and ω_B from *Client Direction 1* is blocked, with the *blocking counter* incrementing.

S4 As there is no request from *Client Direction 0*, ω_B crosses. Once a request from the low-priority path crosses, the *blocking counter* is reset, regardless of the current value.

Blocking within the entire Bluetree-based architecture can be classified as *inter-path blocking* and *intra-path blocking*. The *inter-path blocking* occurs when a request crosses an arbiter stage and gets blocked by the other local path. Therefore, the *inter-path blocking* is affected by the local arbitration scheme. On the other hand, the *intra-path blocking* occurs when a request is blocked by any other request or response ahead of it, from either the same client or the other clients. Besides that, the interaction between the *inter-path blocking* and *intra-path blocking* needs to be considered. For example, when a request ω_3 experiences *inter-path blocking* from ω_A as shown in Figure 4.3, ω_A overtakes and gets ahead of ω_3, which potentially leads to additional *intra-path blocking* in the overlapped request path.

Based on the above blocking analysis, the memory access across this *locally arbitrated* architecture exhibits predictable behaviour. If exact memory ac-

cess profiles are known, the detailed status of the memory flow and the local arbiter at every pipelined stage can be derived that the accurate timing can be computed. However, it is to be noted that such exact analysis becomes more complicated as the Bluetree depth N_β increases. First, a larger number of pipelined buffers along the data path potentially leads to more *intra-path blocking*. Second, the *inter-path blocking* can increase with the number of arbiters. Third, there is interference between pipelined stages. As the nature of tree-based architectures, if there is any blocking in the stage close to the root, the entire network will be affected. For example, if the Bluetree root stage is blocked, the request flow within this Bluetree-based architecture stalls. Similarly, with more *inter-path blocking* closer to the Bluetree leaf stage, there will be more consequent *intra-path blocking* in the overlapped paths.

4.1.3 Worst-Case Analysis

In practice, there is often uncertainty with the memory access profiles, such as uncertainty on the number of memory requests and the memory issuing time instants. In such case, the exact timing analysis is not valid. A similar consideration has been adopted in [96]. However, the relevant worst-case analysis only considers very limited blocking effect and employs a simulation-based method. Instead, the reminder of this section proposes the worst-case analysis by fully exploring the architectural features of the *locally arbitrated* design (however with pessimistic results). This analysis can also be extended to other configurations than the Bluetree-based architecture.

Based on the analysis that *inter-path blocking* and *intra-path blocking* blocking only occur along the Bluetree request path, the worst-case assumption is proposed that the Bluetree request path gets flooded by interfering requests — (i) all pipelined buffers across the data path are occupied, and (ii) the

local arbiter always harms the request flow. Therefore, the calculation on the worst-case latency t^{WC} of the memory access ω can be reformed as follows where t^{WC}_{RQ} is the worst-case request path latency. It is to be noted that the root memory latency $t(D)$ is considered as a fixed constant to simplify further analysis, which potentially represents the worst-case latency of a memory module such as DDR DRAM.

$$t^{WC}(\omega) = t^{WC}_{RQ}(\omega) + t(D) + t_{RS}(\omega) \tag{4.3}$$

Each blocking that the request ω experiences in the request path induces an amount of path latency proportional to the root memory latency $t(D)$ within the Bluetree-based architecture. Essentially, the request flow stalls until the memory is idle again to accept the next request. This latency caused by waiting for the root memory masks the path latency across the pipelined stages. Therefore, the maximum blocking number denoted as $N^{WC}_{RQ}(\omega)$, which the request ω experiences across the request path, can be used to calculate the worst-case request path latency $t^{WC}_{RQ}(\omega)$ as follows.

$$t^{WC}_{RQ}(\omega) = N^{WC}_{RQ}(\omega) \times t(D) \tag{4.4}$$

In this way, the calculation of the worst-case latency t^{WC} of the memory access ω across the architecture with the Bluetree depth N_β can be reformed from (4.3) as follows.

$$\begin{aligned} t^{WC}(\omega) &= t^{WC}_{RQ} + t(D) + t_{RS}(\omega) \\ &= N^{WC}_{RQ} \times t(D) + t(D) + N_\beta \\ &= (N^{WC}_{RQ} + 1) \times t(D) + N_\beta \end{aligned} \tag{4.5}$$

The term *priority path* is introduced here to analyse the maximum blocking number, involving both the *inter-path blocking* and the *intra-path blocking*.

Similar to [96], *priority path* in this reasearch is used to track the local priority at each Bluetree stage β_k across the request path where k is the stage index. Referring to the interconnect in Figure 3.1, *priority path* P_1 for the client μ_1 can be $P_1 = \{L, H, H\}$, for example, where L is for the local low-priority and H for the local high-priority. Therefore, the path P_1 within the Bluetree interconnect is across the local low-priority path at the Bluetree stage β_2, the local high-priority path at β_1, and the local high-priority path at the Bluetree root stage β_0, eventually to the memory module D_1. The related local priority can be expressed as $P_1(\beta_2) = L$, $P_1(\beta_1) = H$, and $P_1(\beta_0) = H$.

By tracking the local priority, the calculation of the maximum blocking number $N_{RQ}^{WC}(\omega)$ across the corresponding Bluetree request path is iterative, based on the calculation of the maximum blocking number at each Bluetree stage β_k. Intuitively, the blocking number at any given Bluetree stage β_k is dependent on (i) the amount of blocking that has occurred at previous stages along the request path, and (ii) the amount of blocking that can occur at the current stage, which is dependent on the local blocking factor α. Following this idea, $N_{RQ}^{WC}(\beta_k)$ is defined as the iterative blocking up to and including the Bluetree stage β_k, and the maximum arbiter blocking number $N_{\alpha}^{WC}(\beta_k)$ is to represent the blocking at the Bluetree stage β_k only. The iterative calculation can be expressed as follows where $+1$ indicates that the local Bluetree buffer is also occupied.

$$N_{RQ}^{WC}(\beta_k) = N_{RQ}^{WC}(\beta_{k+1}) + N_{\alpha}^{WC}(\beta_k) + 1 \tag{4.6}$$

The maximum arbiter blocking number $N_{\alpha}^{WC}(\beta_k)$ is locally decided by the blocking factor α at the corresponding Bluetree stage β_k. With the local arbitration scheme discussed earlier, every α requests from the local high-priority path can be blocked by at most a single request from the local low-priority path, and every single request from the local low-priority path can

be blocked by up to α requests from the local high-priority path. Given the iterative blocking $N_{RQ}^{WC}(\beta_{k+1})$, $N_{\alpha}^{WC}(\beta_k)$ can be calculated with the local priority $P_i(\beta_k)$ where +1 is to include the request ω and determine the total amount of requests to cross the local arbiter at this Bluetree stage.

$$N_{\alpha}^{WC}(\beta_k) = \begin{cases} \lceil \frac{(N_{RQ}^{WC}(\beta_{k+1})+1)}{\alpha} \rceil & \text{H} \\ (N_{RQ}^{WC}(\beta_{k+1})+1) \times \alpha & \text{L} \end{cases} \tag{4.7}$$

For example, if there are $N_{RQ}^{WC}(\beta_{k+1}) + 1$ requests at the local high-priority path of a Bluetree stage (the number of requests accumulated across the stages plus the request itself), the maximum blocking from the low-priority path is this number divided by α and then applied a ceiling function.

To summarise the above analysis, the maximum blocking number up to and including any given Bluetree stage β_k can be computed with (4.6) and (4.7). The maximum blocking number that the request ω experiences across the request path $N_{RQ}^{WC}(\omega)$ can be calculated iteratively, starting from the Bluetree leaf stage to the Bluetree root stage β_0 within the interconnect. Finally, the maximum blocking number in the request path $N_{RQ}^{WC}(\omega)$ equals to the maximum blocking number accumulated to the root stage $N_{RQ}^{WC}(\beta_0)$ as follows. Afterwards, the worst-case latency $t^{WC}(\omega)$ can be calculated with (4.5).

$$N_{RQ}^{WC}(\omega) = N_{RQ}^{WC}(\beta_0) \tag{4.8}$$

As examples applying the above method, Table 4.1 shows the maximum blocking number for 8-client Bluetree-based architectures respectively. The row is for Bluetree local blocking factor α, and each local arbiter is set with the same value in the entire interconnect. The column is for the Bluetree path P_i. The table content shows the maximum blocking number N_{RQ}^{WC} for each request path. With N_{RQ}^{WC} given, the worst-case latency can be calculated using (4.5). For example, assuming the root memory latency $t(D) = 20$ in clock

Table 4.1. Maximum Blocking Number in 8-Client Bluetree-based Architecture

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
$\alpha = 1$	30	30	30	30	30	30	30	30
$\alpha = 2$	17	23	28	41	32	44	53	80
$\alpha = 3$	14	24	32	58	38	66	90	170

cycles, the worst-case memory access latency across any path in the 8-client Bluetree-based architecture with the blocking factor $\alpha = 1$ can be calculated as $t^{WC}(\omega) = (N_{RQ}^{WC} + 1) \times t(D) + N_\beta = (30 + 1) \times 20 + 3 = 623$.

As shown in Table 4.1, with increasing Bluetree blocking factor α, the maximum blocking number in the request path $N_{RQ}^{WC}(\omega)$ decreases with more local high-priority tracks. With blocking factor $\alpha = 1$, the maximum blocking number N_{RQ}^{WC} remains the same value for different request paths. According to the design of Bluetree local arbitration, the Bluetree interconnect can be considered as distributed tree stages with local round-robin scheme when $\alpha = 1$. It provides fair accesses to the shared memory for all clients.

Analytical Results and Measured Results

This section compares the analytical worst-case memory access latency and the measured worst-case memory access latency across the 8-client Bluetree-based architecture, with blocking factor $\alpha = 1$, 2 and 3 where each local arbiter is set with the same value in the entire interconnect. The root memory latency is assumed as a constant $t(D) = 20$ in clock cycles. In this case, the analytical results are calculated with maximum blocking number in Table 4.1 following the above example.

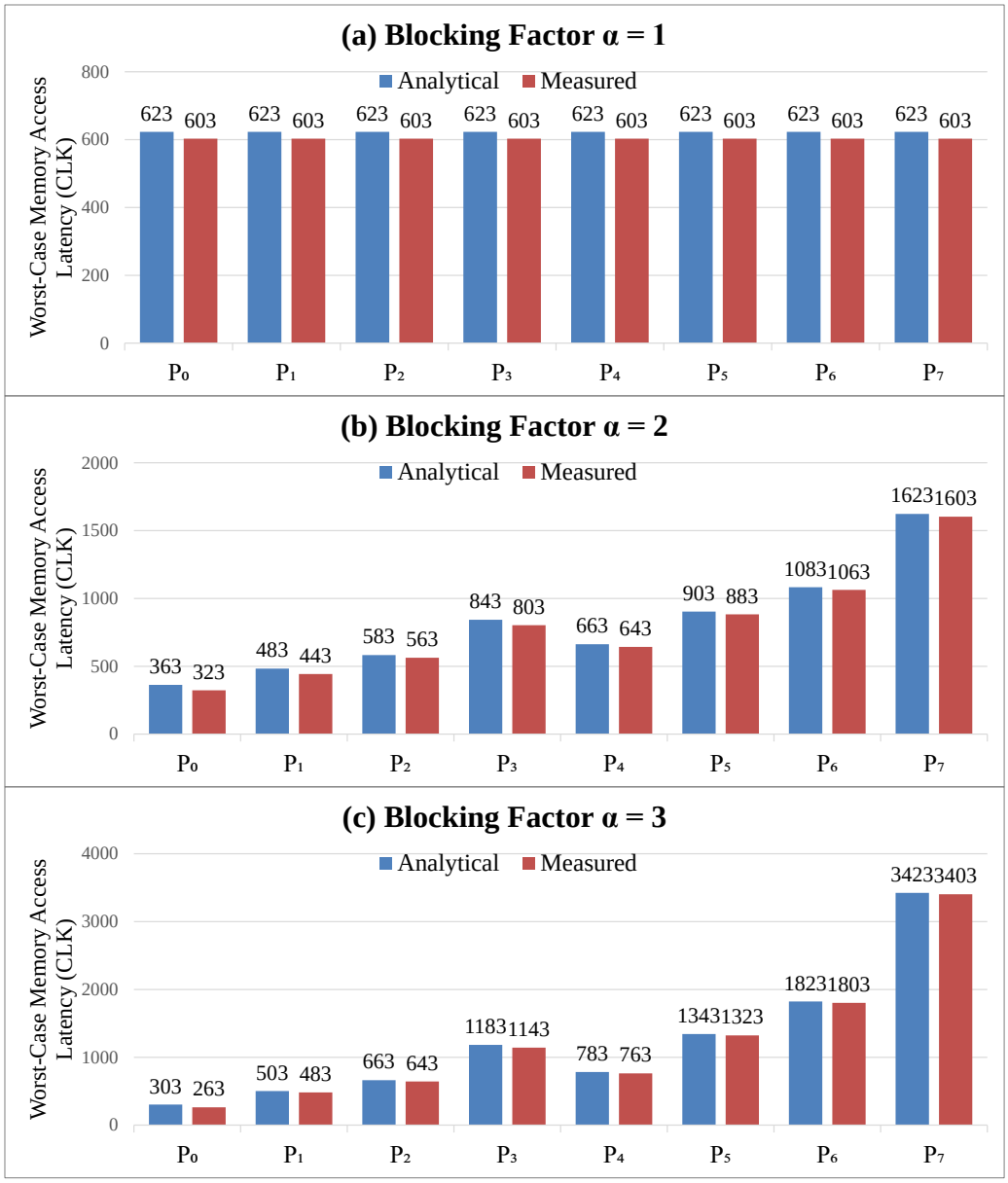

Figure 4.4. Worst-Case Memory Access Latency across Bluetree-based Architecture

The measured results are from hardware simulations. Traffic generators are employed as clients, and each traffic generator keeps pushing memory requests into its memory access path. In this case, the system can be flooded with memory requests (potentially pending), aiming towards that each mem-

ory request experiences its maximum blocking. The root memory module is implemented using Bluespec BRAM package [107] with extra delays as a constant $t(D) = 20$ in clock cycles. The system is implemented using Bluespec System Verilog [107][108], with simulations running on BlueSim simulator [107][108]. This simulation measures memory access latency across the 8-client Bluetree-based architecture that the latency of each memory access is measured.

In this measurement, it is observed that memory access latency gradually increases with the proceeding of the simulation until remaining at a constant value. This maximum measured constant is selected as the measured worst-case memory access latency across the relevant memory access path. Figure 4.4 shows the comparison of analytical worst-case results and measured worst-case results with bar chart. The horizontal axis is for memory access path P_i, and the vertical axis is for the worst-case results in clock cycles. It is observed that the measured results are smaller than the analytical results in each memory access path. Referring to Figure 4.4 (a), the worst-case memory access latency is identical with blocking factor $\alpha = 1$ in each memory access path, both analytical and measured results. In addition, difference between an analytical result and a measured result slightly varies in Bluetree-based architectures with blocking factor $\alpha = 2$ and 3.

Discussion

This proposed method defines the generic analytical flow to predict the memory access behaviour across the *locally arbitrated* architecture and statically bound the worst-case memory access latency. It can be extended to other architectural configurations than the Bluetree design, which requires modification to the analysis of the local arbitration scheme. This worst-case analysis

can produce pessimistic bounds as the results, which potentially leads to conservative system design and resource dimensioning, as the memory access latency is the main part forming the overall program execution time.

It is to be noted that this proposed method focuses on the timing behaviour of single memory request. If a sequence of memory requests is studied as a whole, this method may overestimate the overall latency of this sequence, which potentially leads to a higher maximum arbiter blocking number $N_{\alpha}^{WC}(\beta_k)$ at any distributed stage. In this case, instead of a single request ω, the sequence of requests are assumed as $\tau = \{\omega_1, \omega_2, \omega_3, ...\}$ as a whole. The maximum blocking number that this sequence experiences across the request path $N_{RQ}^{WC}(\tau)$ can be calculated iteratively using with (4.6) and (4.7), starting from the Bluetree leaf stage with a start value τ. Such modification actually calculates the maximum blocking number of the last request in the sequence, and the rest of requests in this sequence is considered as the *intra-path blocking* to the last request at the Bluetree leaf stage. This aims to avoid duplicate accumulation of *inter-path blocking* to memory requests in the sequence.

In summary, the *locally arbitrated* distributed memory interconnect shows time-predictable behaviour. If the exact memory access profiles can be provided, the accurate memory access latency in such architecture can be determined with no pessimism as discussed in the previous analysis, based on the detailed status of the memory flow and the local arbiter at every pipelined stage. With uncertainty on memory access profiles which is often the case in reality, the worst-case analysis proposed in this section has to be employed for real-time applications even with pessimistic results. The worst-case bound provided can also be tightened in the future work, e.g., by restricting the demand from processors with limit, and the discussion on the tightness also requires sufficiently representative memory workloads to be fair.

4.2 Timing Behaviour of Multi-Core Architectures with Shared Distributed Memory Interconnects

Distributed time-predictable memory interconnects are designed for multi-core architectures to support real-time applications. The above worst-case analysis actually shows the behaviour when the *locally arbitrated* Bluetree architecture is flooded by memory requests, thus with pessimistic results. This section further explores and analyses the timing behaviour of the *locally arbitrated* architecture and the *globally arbitrated* architecture in more general cases, with experiments to demonstrate the timing behaviour of both architectures.

Memory Workloads

Due to architectural features of multi-core architectures with shared distributed memory interconnects, multiple memory requests have to share the overlapped interconnect as well as the root memory module. Taking the *locally arbitrated* architecture as an example, simultaneous memory requests in transfer cause contention, and thus memory access latency is increased. With increasing memory workloads, more available system bandwidth is consumed. If the requested bandwidth keeps increasing, the system will saturate at some point, without delivering any additional bandwidth. In this case, any further memory request will only have to wait for the service of the system. This saturation phenomenon commonly occurs with shared resource [109]. As shown in Section 4.1, the saturation point of the *locally arbitrated* Bluetree-based architecture is determined by the static worst-case analysis. It clearly

bounds the maximum request number in a specific Bluetree path. However, memory workloads of the relevant worst-case assumption is independent of the response time that a client just keeps pushing requests into the system regardless of memory response.

First, the number of memory requests issued to the system is limited, either by the characteristics of the application software, or by the architecture of a processor (i.e., maximum number of outstanding memory requests before the processor stalls). Second, the workload pattern is dependent on the memory response. With such workload pattern, blocking still occurs due to the contention to the shared resource, and memory access latency increases. However, a client has to slow down the release of memory requests, waiting for memory response. The increase of memory access latency stops in turn. This dependency actually reflects the process of practical applications. For example, a processor has to receive data from memory before any related operation. The characteristics of the above workload pattern can be represented as follows.

$\boldsymbol{N^{\mu}_{RQ}(\mu_i)}$ is outstanding request number from a client with index i. A client can issue memory requests successively until this limit. Then the client stalls, waiting for memory response. Only when there is any memory response returned to this client, another new memory request can be issued. In addition, outstanding request number to the shared memory $N^{\mu}_{RQ}(D)$ is the sum of $N^{\mu}_{RQ}(\mu_i)$ in the entire interconnect.

$\boldsymbol{T^{\mu}_{RQ}(\mu_i)}$ is request interval between two successive memory requests. A client issues successive requests with intervals, normally in clock cycles. It reflects necessary processor execution time or the time across the data path in practical applications. In addition, variation of request interval is also introduced that memory requests are more distributed in time. By contrast, when the

request interval is fixed as 1 $T^{\mu}_{RQ}(\mu_i) = 1$, memory requests will be issued into the system more intensively.

Workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$ describes memory workloads with limited outstanding requests and dependent on memory response time. With either an increased $N^{\mu}_{RQ}(\mu_i)$ or a decreased $T^{\mu}_{RQ}(\mu_i)$, memory workloads from client μ_i to relevant memory access path increase.

4.2.1 Locally Arbitrated Architecture and Globally Arbitrated Architecture

Based on the previous analysis, the *locally arbitrated* architecture allows multiple memory requests in transfer leading to varying blocking behaviour. The generic analytical flow to predict the memory access behaviour across the *locally arbitrated* architecture is proposed in Section 4.1, and the Bluetree design is shown as an example. In general, the *locally arbitrated* architecture can allow average-case timing behaviour to be much lower than the worst case. By contrast, the *globally arbitrated* architecture provides the contention-free data paths based on the global scheduling interval, provided that the strict synchronisation can be guaranteed. This limits the average-case memory access latency to be similar to the worst case, facilitating the timing behaviour analysis.

The reminder of this section continues to explore the timing behaviour of *locally arbitrated* architectures and *globally arbitrated* architectures with experiments. The root memory latency is assumed as a constant $t(D) = 20$ in clock cycles. Taking the design of Bluetree and TDM Tree as examples, both 8-client architectures are running with the same clock frequency. According

to the proposed analytical method, different Bluetree blocking factor leads to different blocking behaviour of Bluetree multiplexer at a local distributed stage. This allows Bluetree blocking factor to be set with a specific value at a specific stage across a specific data path to benefit specific memory workloads. In this section, Bluetree blocking factor is set as $\alpha = 1$, and each local arbiter is set with the same value in the entire Bluetree interconnect. This can provide relatively fair accesses for all clients regardless of memory workloads. By comparison, the global scheduling interval of TDM Tree is set as 160 in clock cycles that 8 clients share the root memory with $t(D) = 20$ in this architecture. (It is to be noted that the global scheduling interval is roughly decided for observations only.) In addition, traffic generators are employed as clients with synthetic memory workloads which follow the above workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$.

Multiple experiments with varying memory workloads has been conducted, and different groups of memory workloads lead to different experimental results. In this section, 5 groups are selected to demonstrate and compare the difference of timing behaviour across the *locally arbitrated* Bluetree-based architecture and the *globally arbitrated* TDM Tree-based architecture. These include workload conditions such as balanced or unbalanced path workloads and varying or increasing request intervals.

Hardware Simulations: Increasing Memory Workloads

The initial experiment is conducted by hardware simulations with relatively simple workload patterns. Each traffic generator issues 36 memory requests totally. Request interval is fixed as 1 $T^{\mu}_{RQ}(\mu_i) = 1$, and outstanding request number $N^{\mu}_{RQ}(\mu_i)$ varies as shown in Table 4.2. The column is for client μ_i, and the row is for 3 groups of outstanding request number. For example,

Table 4.2. Increasing Outstanding Requests for 8-Client Architectures

	μ_0	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
group a	0	0	0	2	1	0	0	0
group b	1	0	1	2	2	0	0	1
group c	2	1	1	3	3	1	1	1

outstanding request number $N^{\mu}_{RQ}(\mu_1) = 0$ is that client μ_1 is with no memory workloads. In this way, the table content shows increasing memory workloads from *group a* to *group c* with increasing outstanding request number for clients. In addition, each client is with different outstanding request number $N^{\mu}_{RQ}(\mu_i)$ in each group, leading to unbalanced path workloads.

In this experiment, the root memory module is designed using Bluespec BRAM package [107] with extra delays as a constant $t(D) = 20$ in clock cycles. Both the 8-client Bluetree-based system and the 8-client TDM Tree-based system are implemented using Bluespec System Verilog [107][108], with simulations running on BlueSim simulator [107][108]. This experiment measures memory access latency across both 8-client architectures that the latency of each memory access is measured. In addition, memory request release time of each memory access is also measured. The measured results are shown in Figure 4.5 and Figure 4.6 with scatter plot. The horizontal axis is for memory request release time in clock cycles, and the vertical axis is for memory access latency in clock cycles.

Figure 4.5 (a) shows memory access latency across the 8-client Bluetree-based architecture with only memory requests in path P_3 and path P_4. At the start period of the simulation, outstanding request number to the shared memory is $N^{\mu}_{RQ}(D) = 3$, and thus memory access latency increases to approximately 60 very quickly referring to the figure. With different outstanding request num-

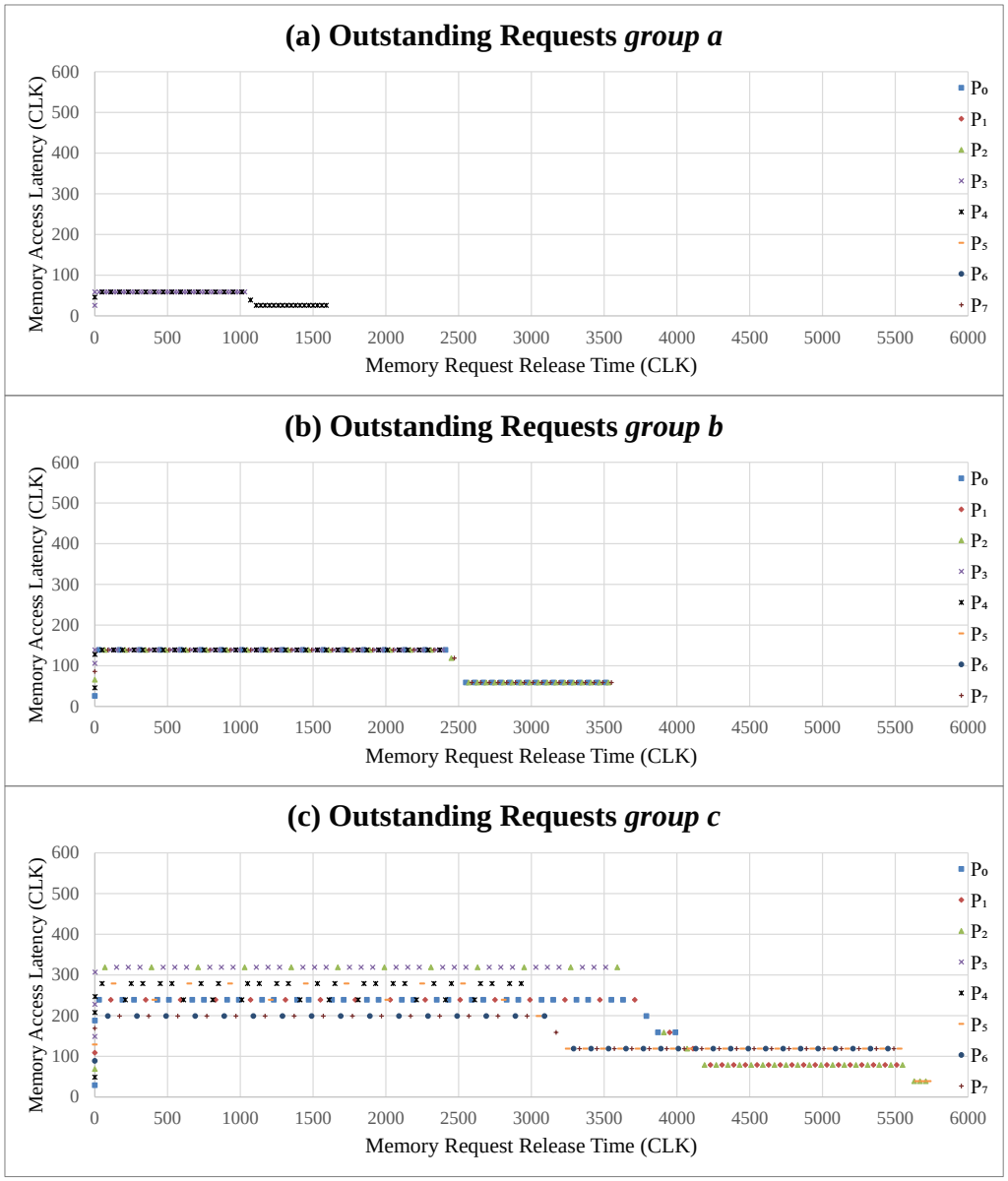

Figure 4.5. Memory Access Latency across 8-Client Bluetree-based Architecture

ber $N^{\mu}_{RQ}(\mu_3) = 2$ and $N^{\mu}_{RQ}(\mu_4) = 1$ but with the same total request number, simulations in different paths complete at different time instants. For example, the simulation in path P_3 with $N^{\mu}_{RQ}(\mu_3) = 2$ completes at approximately 1000. Afterwards, with the decreasing of outstanding request number to the shared memory $N^{\mu}_{RQ}(D) = 1$, memory access latency across path P_4 reduces

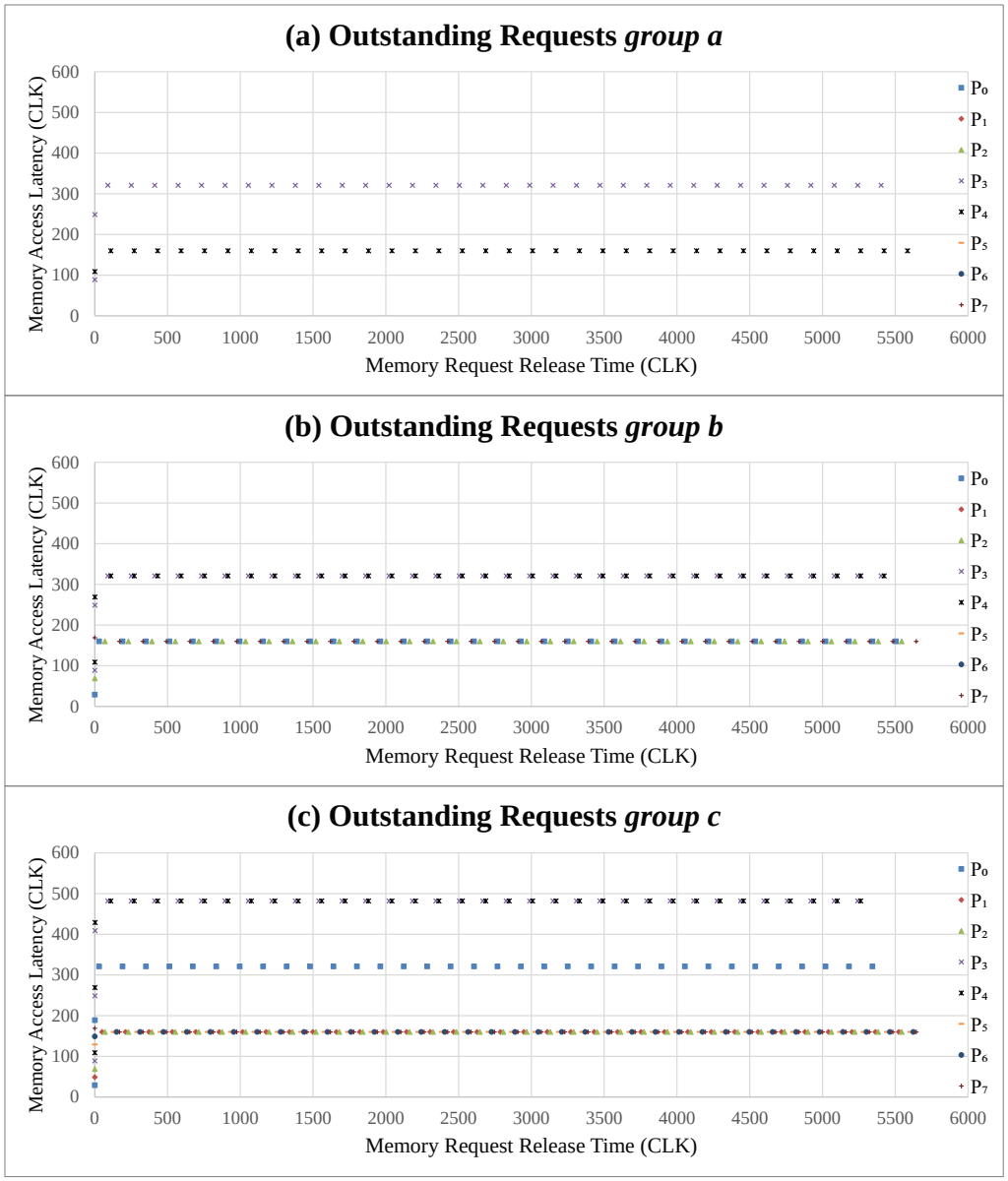

Figure 4.6. Memory Access Latency across 8-Client TDM Tree-based Architecture

to approximately 20 until the end of the simulation. Figure 4.5 (b) shows memory access latency with increased memory workloads. The distribution of scatters shows similar trend as that in Figure 4.5 (a). By contrast, with increased outstanding request number, the highest measured memory access latency increases to approximately 150.

Figure 4.5 (c) shows memory access latency across the Bluetree-based architecture with further increased memory workloads. Memory access latency increases sharply in a very short period of time from the start period of the simulation, with intensively issued memory requests into the system. Essentially, as the memory workload pattern is dependent on the response time, the release of memory requests drops that the increase of memory access latency stops in turn. Memory access latency in each path tends to reach the corresponding maximum limit. With fixed request interval as $T^{\mu}_{RQ}(\mu_i) = 1$, regular values of memory access latency can be observed. For example, memory access latency in path P_4 and path P_5 is approximately 280 or 240. As the *locally arbitrated* Bluetree allows varying blocking behaviour, the inter-path interference also affects paths nearby. Referring to the above example, memory access path P_5 with only 1 outstanding request $N^{\mu}_{RQ}(\mu_5) = 1$ is severely affected by path P_4 with $N^{\mu}_{RQ}(\mu_4) = 3$, and thus memory access latency in path P_5 varies, either 280 or 240.

Figure 4.6 shows memory access latency across the 8-client TDM Tree architecture. Compared with Figure 4.5 (a), Figure 4.6 (a) shows that the *globally arbitrated* TDM Tree does not support work conservation. With only memory requests in path P_3 and path P_4, the interconnect or the memory module can be idle. However, the strict TDM scheme only allows 1 memory request to be relayed to the empty data path at a time. In this case, the measured memory access latency in path P_4 is approximately 160 which reflects the global cycle. As a result, the simulation finally completes at approximately 5500. Figure 4.6 (b) shows similar scatter distribution with increased memory workloads, and the highest measured memory access latency increases in Figure 4.6 (c) with further increased memory workloads. Compared with Figure 4.5 (c), memory access latency in Figure 4.6 (c) is identical in either in path P_4 or path P_5 with different path outstanding request number.

Table 4.3. Balanced Outstanding Requests for 8-Client Architectures

μ_i	μ_0	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7
$N^{\mu}_{RQ}(\mu_i)$	2	2	2	2	2	2	2	2

FPGA Experiments: Balanced Path Workloads

This experiment is conducted on FPGA implementations with synthetic memory workloads. Each traffic generator issues 100 memory requests totally following workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$. Outstanding request number is set as $N^{\mu}_{RQ}(\mu_i) = 2$ as shown in Table 4.3, thus balanced path memory workloads, and request interval varies with randomly generated values between 1 to 64 as $T^{\mu}_{RQ}(\mu_i) \in [1, 64]$. The values used as request intervals are shown in Appendix A.1. This aims to provide varying memory workloads closer to practical applications. The root memory module is designed using FPGA BRAM [110] with extra delays as a constant $t(D) = 20$ in clock cycles. Both the 8-client Bluetree-based system and the 8-client TDM Tree-based system are synthesised using Xilinx Vivado [111][112] and implemented on Zedboard [113] with 100MHz of clock frequency. This experiment measures memory access latency across both 8-client architectures that the latency of each memory access is measured. In addition, memory request release time of each memory access is also measured.

Figure 4.7 shows scatter plot of memory access latency with memory request release time in this measurement. The horizontal axis is for memory request release time in clock cycles, and the vertical axis is for memory access latency in clock cycles. Compared with Figure 4.7 (a) and Figure 4.7 (b), the distribution of scatters shows similar trend. It is also observed in Figure 4.7 (a) that memory access latency drops in the end period of the experiment.

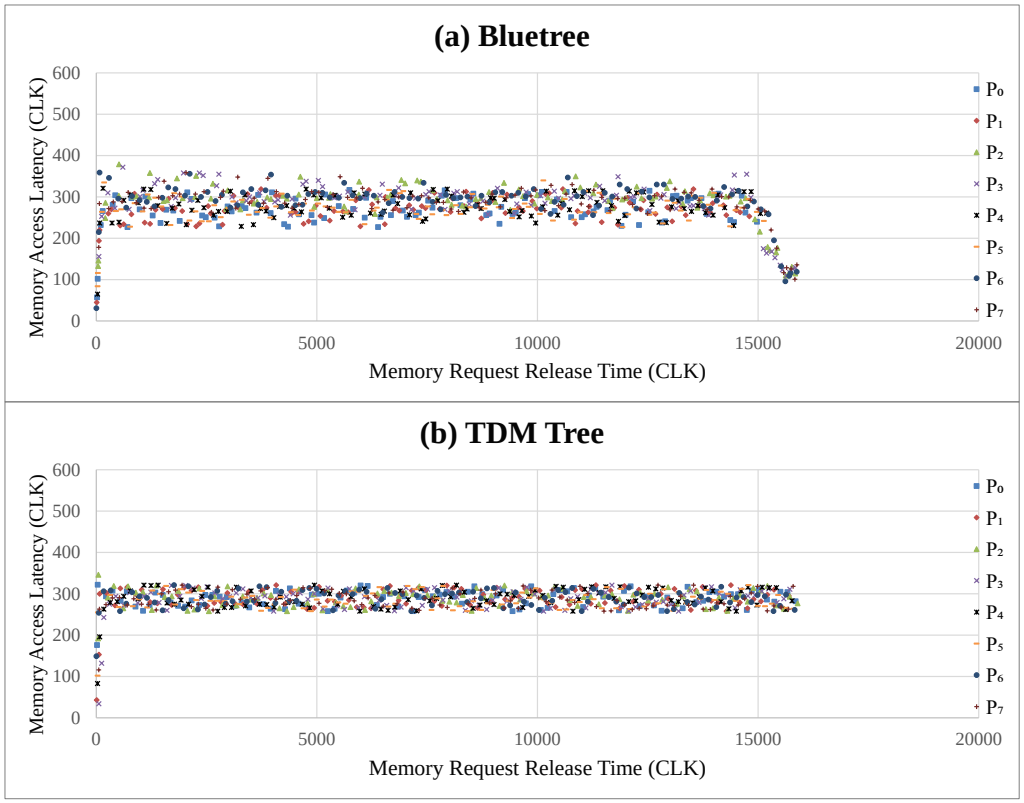

Figure 4.7. Memory Access Latency with Balanced Path Workloads across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

Figure 4.8 shows boxplot of memory access latency in this measurement. The horizontal axis is for memory access path, and the vertical axis is for memory access latency in clock cycles. Following the above analysis, the distributions show similar trend in Figure 4.8 (a) and Figure 4.8 (b), although in each memory access path, the interquartile range in Figure 4.8 (a) is slightly larger than that in Figure 4.8 (b). It is also observed that there are more outliners in Figure 4.8 (a) than that in Figure 4.8 (b). This reflects the reduced memory access latency in the end period of the experiment in Figure 4.7 (a). With such decreasing of outstanding request number, memory access latency reduces across the Bluetree-based architecture. By comparison, TDM Tree limits the average case to be similar to the worst case.

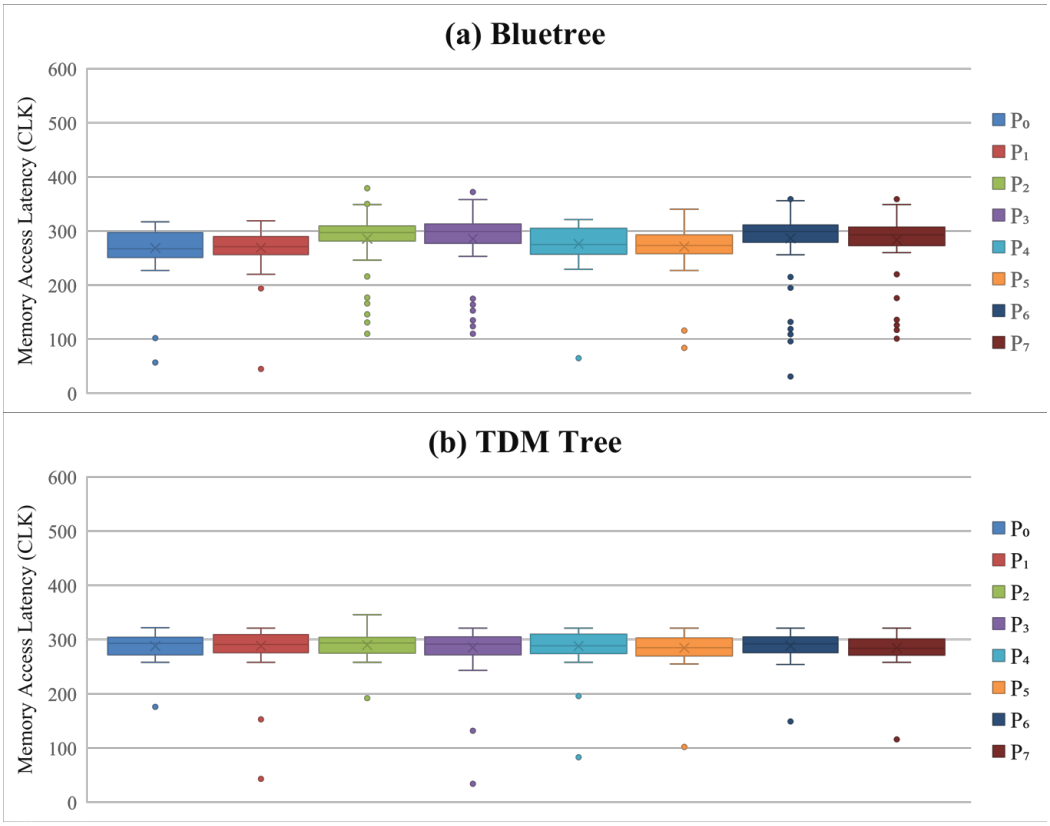

Figure 4.8. Boxplot of Memory Access Latency with Balanced Path Workloads across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

FPGA Experiments: Increasing Request Intervals

This experiment is conducted using the similar setup of the above experiment. Based on balanced outstanding requests $N^{\mu}_{RQ}(\mu_i) = 2$ as shown in Table 4.3, the variation of request interval increases as $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$. The values used for request intervals are randomly generated as shown in Appendix A.2. Figure 4.9 shows scatter plot of memory access latency with memory request release time in this measurement. Compared with Figure 4.7 (a) of the above measurement, scatters are more distributed in Figure 4.9 (a). Compared with Figure 4.7 (b), much more distributed scatters can be observed in Figure 4.9

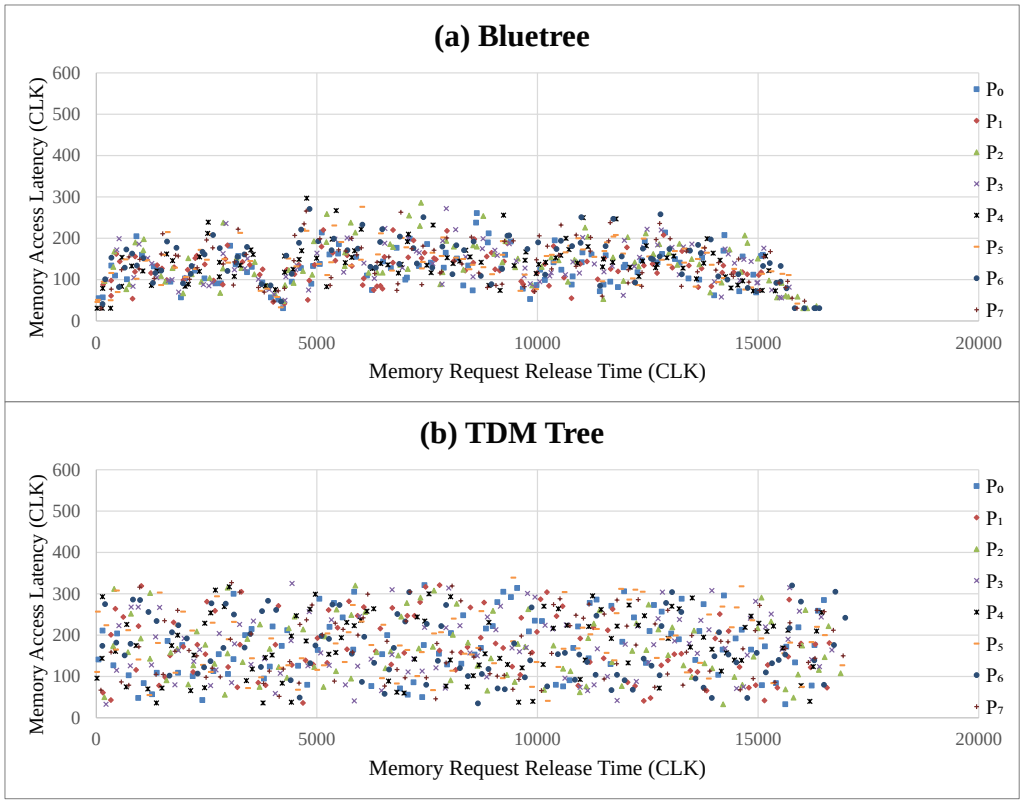

Figure 4.9. Memory Access Latency with Increasing Request Intervals across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

(b), although the highest measured memory access latency almost remains the same.

Figure 4.10 shows boxplot of memory access latency in this measurement. Compared with Figure 4.8, median and mean are both reduced in Figure 4.10 that the increased request interval reduces memory workloads. However, both the interquartile range and the difference between the maximum line and the minimum line significantly increases in Figure 4.10. Variation of memory access latency becomes more severe in this experiment. Compared with Figure 4.10 (a), the interquartile range is much larger in Figure 4.10 (b). In this case, the TDM Tree-based system suffers more severe variation

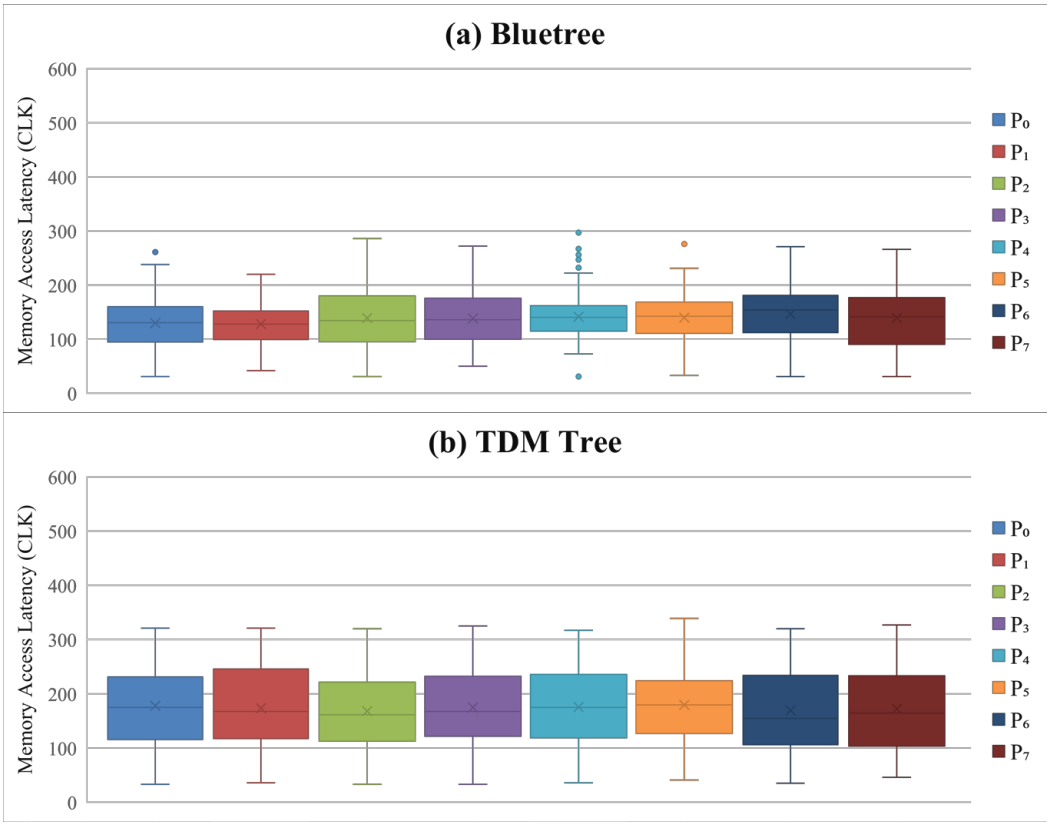

Figure 4.10. Boxplot of Memory Access Latency with Increasing Request Intervals across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

of memory access latency than the Bluetree-based system. Memory requests are more distributed in time that request interval varies as $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$. It does not satisfy the global cycle of TDM Tree which is 160.

FPGA Experiments: Unbalanced Path Workloads

Based on the above setup, this experiment is conducted using unbalanced path workloads and varying request intervals. Outstanding request number $N^{\mu}_{RQ}(\mu_i)$ is set as *group c* in Table 4.2, and request interval varies as $T^{\mu}_{RQ}(\mu_i) \in [1, 64]$ with randomly generated values in Appendix A.1. Figure 4.11 shows

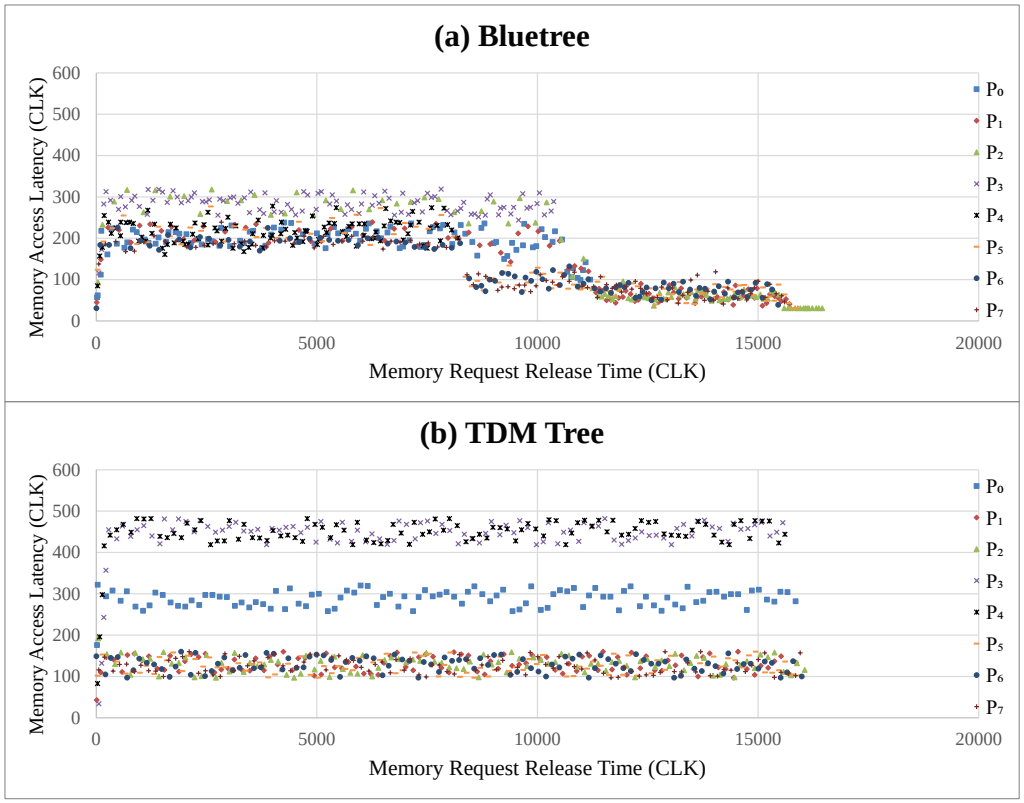

Figure 4.11. Memory Access Latency with Unbalanced Path Workloads across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

scatter plot of memory access latency with memory request release time in this measurement. Referring to the figure, scatters in Figure 4.11 (a) distributes with similar trend in Figure 4.5 (c), and scatters in Figure 4.11 (b) distributes with similar trend in Figure 4.6 (c).

Figure 4.12 shows boxplot of memory access latency. Compared with Figure 4.12 (b), in each memory access path, although median and mean are lower, either the interquartile range or the difference between the maximum line and the minimum line is much larger in Figure 4.12 (a), especially in path P_1 and path P_2. Based on the previous analysis, Bluetree allows varying blocking behaviour, and the inter-path interference also affects paths nearby.

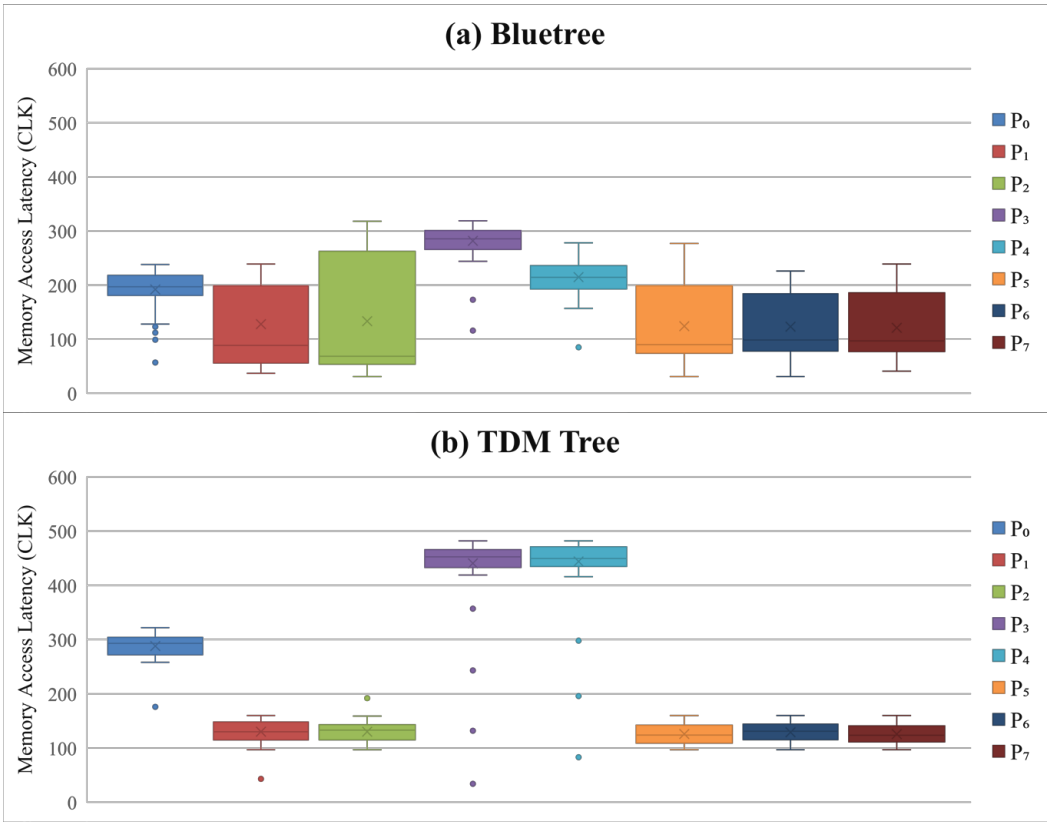

Figure 4.12. Boxplot of Memory Access Latency with Unbalanced Path Workloads across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

In this experiment, the Bluetree-based system suffers severe variation of memory access latency.

FPGA Experiments: Varying Request Intervals

Based on the above unbalanced outstanding request number $N^{\mu}_{RQ}(\mu_i)$, variation of request interval increases in this experiment as $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$ from Appendix A.2. Figure 4.13 shows scatter plot of memory access latency. Compared with Figure 4.11 (a) of the above measurement, the highest measured memory access latency is much lower in Figure 4.13 (a) that memory

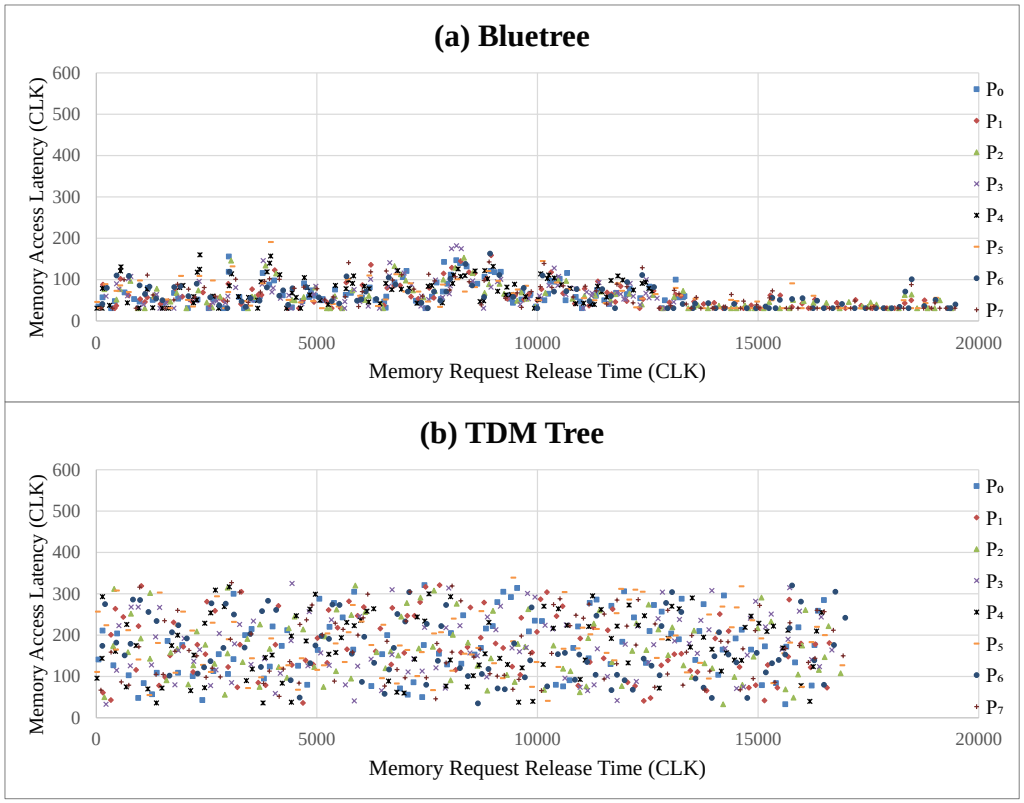

Figure 4.13. Memory Access Latency with Varying Request Intervals across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

workloads decrease in this experiment due to the increased request interval. Compared with Figure 4.11 (b), scatters are much more distributed in Figure 4.13 (b).

Figure 4.14 shows boxplot of memory access latency in this measurement. Compared with Figure 4.14 (a), in each memory access path, both the interquartile range and the difference between the maximum line and the minimum line are much larger in Figure 4.14 (b), as well as much higher median and mean. In this case, varying request interval $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$ does not satisfy the global cycle of TDM Tree. As a result, the TDM Tree-based system suffers more severe variation of memory access latency than the Bluetree-

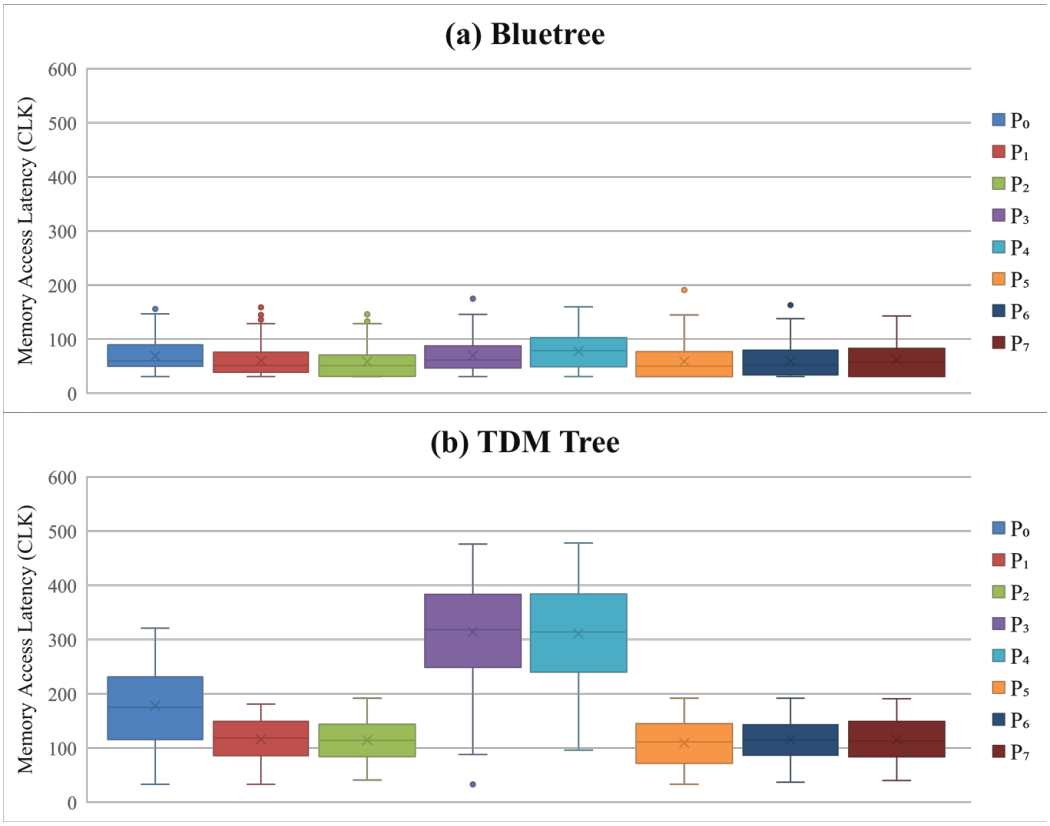

Figure 4.14. Boxplot of Memory Access Latency with Varying Request Intervals across 8-Client Bluetree-based Architecture and 8-client TDM Tree-based Architecture

based system, especially in path P_3 and path P_4 with increased outstanding request number.

Discussion

This section explores and analyses timing behaviour of *locally arbitrated* and *globally arbitrated* architectures with experiments, and Bluetree and TDM Tree are taken as examples. The *locally arbitrated* architecture allows varying blocking behaviour, and it potentially suffers variation of memory access latency that the average-case timing behaviour is much lower than the worst

case. Taking figures of Bluetree experimental results as examples, the upper scale limit of the vertical axis (600 in clock cycles) is approximately set according to the analytical worst-case memory access latency (623 in clock cycles which is statically bounded with the proposed analytical method in Section 4.1), and memory access latency varies below this extreme limit.

By contrast, the *globally arbitrated* architecture potentially limits the average case to be similar to the worst case. However, varying memory workloads may not satisfy the global scheduling interval, thus leading to substantial varying memory access latency. Based on the previous analysis, the deployment of the *globally arbitrated* architecture can rely on effective analysis of accurate application behaviour thus to benefit specific applications. For example, additional rate control schemes can be employed based on the reserved time slots to benefit a sequence of successive memory requests (i.e., GAMT).

4.3 Summary and Discussion

This chapter analyses the timing behaviour of the *locally arbitrated* architecture and the *globally arbitrated* architecture. Section 4.1 analyses the resource contention and the blocking effect across the data paths within *locally arbitrated* architectures and proposes the generic analytical flow to predict the memory access behaviour and statically bound the worst-case memory access latency when there is uncertainty on memory access profile. This contributes to solve the research question *Q1*. In addition, Section 4.2 explores the timing behaviour of the *locally arbitrated* architecture and the *globally arbitrated* architecture using experiments with synthetic memory workloads.

The main contribution presented in this chapter is summarised as follows.

The generic analytical flow is proposed for time-predictable behaviour of memory accesses across multi-core architectures with locally arbitrated interconnects. Without exact memory access profiles, this static analysis can guarantee the safe worst-case bound for real-time applications applying calculations.

It is to be noted that the worst-case memory access latency is bounded when there is uncertainty on memory access profile, thus with pessimistic results. The bound provided can also be tightened in the future work, e.g., by restricting the demand from processors with limit, and the discussion on the tightness also requires sufficiently representative memory workload patterns to be fair.

Chapter 5

Reducing Variation of Memory Access Latency across Multi-Core Architectures with Shared Distributed Memory Interconnects

This chapter proposes the root queue modification with the root queue management to reduce variation of memory access latency across the *locally arbitrated* architecture. It employs and utilises an additional hardware queue with queue management between the distributed interconnect root and the shared memory module. This aims to solve the research question *Q2: Can multi-core architectures with shared distributed memory interconnects be modified at the hardware level to reduce variation of memory access latency?*

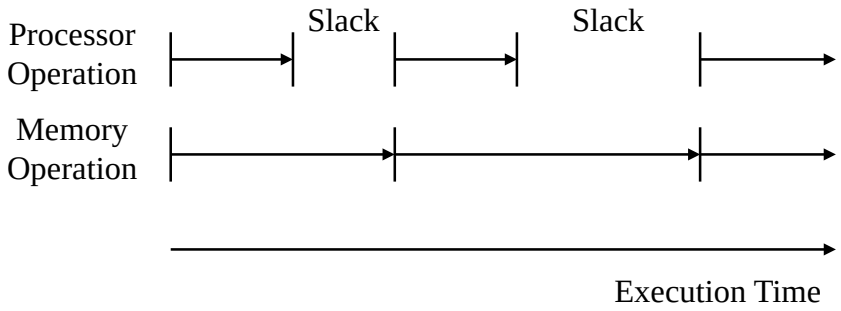

Figure 5.1. Processor Operation vs. Memory Operation

The reminder of this chapter is structured as follows. Section 5.1 analyses variation of memory access latency within the *locally arbitrated* architecture and the *globally arbitrated* architecture. Section 5.2 proposes the root queue modification and explains its operation with analysis on the time-predictable behaviour of memory accesses. Section 5.3 evaluates the effectiveness of the root queue modification on reducing variation of memory access latency with hardware simulations, and Section 5.4 continues to evaluate the effectiveness of this architectural modification with FPGA experiments. Afterwards, Section 5.5 summarises this chapter and presents discussion.

5.1 Problem Analysis

The multi-core architecture is typically designed for good average-case performance, and the resource contention within such architecture is inevitable. This potentially causes contention over memory accesses due to resource sharing issue, and such contention can lead to substantial varying memory access latency. Wide variation of memory access latency leads to wide fluctuation of the overall system performance as memory access latency is the main part forming the overall execution time. Figure 5.1 shows an example that the

processor stalls with varying slack time, depending on the varying memory response time. In this case, variation of memory access latency directly impacts the processor utilisation and the dependent processes. In addition, the variation of memory access latency can also lead to very pessimistic worst-case assumptions in the timing analysis — where the maximum contention has to be assumed for most, if not all, memory accesses — thus with large safety margins.

The multi-core architecture with the shared distributed memory interconnect appears to be more sensitive to the resource contention due to the tree-based structure that the overlapped data paths are also shared by all clients as well as the root memory module. The *locally arbitrated* architecture allows multiple memory requests in transfer simultaneously and thus leads to varying memory access latency due to varying blocking behaviour within such architecture. By contrast, the *globally arbitrated* architecture budgets processors based on the global scheduling interval. This aims towards contention-free data paths, potentially limiting the average case to be similar to the worst case. However, memory requests can be more distributed in time, and varying memory workloads may not perfectly satisfy the global scheduling interval, thus leading to substantial varying memory access latency.

Based on the above analysis, both the *locally arbitrated* architecture and the *globally arbitrated* architecture potentially suffer variation of memory access latency, which can also be illustrated with experimental results in Section 4.2. By comparison, the *globally arbitrated* architecture is more suitable for specific applications. Therefore, the reminder of this research focuses on to reduce variation of memory access latency across the *locally arbitrated* architecture. The *locally arbitrated* Bluetree-based architecture is taken as an example in the following research.

Although the deployment of the Blurteee-based architecture does not require to model memory requests in applications, it potentially suffers varying blocking behaviour caused by the contention to critical resource, especially the contention to the overlapped data paths. The interaction between the *inter-path blocking* and *intra-path blocking* complicates the analysis of the varying blocking behaviour. First, due to the architectural feature of tree-based structure, any blocking closer to the tree root blocks the entire interconnect. Many requests are blocked waiting in the shared data paths, which also blocks subsequent requests. Second, with blocking along the interconnect paths, new issued requests can overtake and get ahead of pending requests due to the local arbitration at distributed stages. In this case, the sequence of the pending requests is broken which leads to additional blocking, and the system resource is not fairly shared. Third, due to the design of the local arbitration, Bluetree shows varying blocking behaviour at distributed stages. Even with $\alpha = 1$ which provides relatively fair resource sharing, the blocking behaviour still varies. In addition, varying memory workloads aggravates the varying blocking behaviour within the architecture.

The varying blocking behaviour across the Bluetree-based architecture causes resource sharing issue that pending requests are not fairly served. This leads to varying memory access latency. First, if the sequence of pending requests is broken, the system resource is not fairly shared. This causes additional blocking to these pending requests, which aggravates the variation of memory access latency in turn. The inter-path interference also affects paths nearby. As a result, a portion of memory requests inevitably suffer much higher latency than the average case at runtime. Second, it complicates the timing behaviour analysis which requires to derive the detailed status of the memory flow and the local arbiter at every pipelined stage with exact memory access profiles. However, such analysis becomes much more complicated with an expanding system configuration (i.e., an increasing number of clients) as

discussed in Section 4.1. This potentially leads to conservative system design with enough safety margin to guarantee the memory response. If a client suffers variation of memory access latency with uncertainty on memory access profile, it has to refer to the worst-case assumption to determine memory access latency, thus with pessimistic results.

The methods to alleviate critical resource contention has been widely studied on multi-core architectures. A method is to regulate accesses to critical resource based on resource reserving. However, it relies on effective program analysis to benefit specific applications. This analysis is similar to that of the *globally arbitrated* architecture, and the applicability and the effectiveness very much depend on memory access patterns. A different method is message combining which can potentially combine memory requests and reduce the contention to the overlapped data paths within the tree-based interconnect. However, this significantly increases data width to either the shared root memory controller or the pipelined arbiter stages especially the stages closer to the tree root. It actually tends to move the workloads and leave the burden to the centralised location, i.e., interconnect root, where the increasing logic size with an expanding system severely harms the maximum synthesisable clock frequency.

Instead, an alternative method is to invest additional hardware resources to enhance the multi-core architecture, such as employing virtual channel with flow control to alleviate the router contention from multiple communication flows in NoC applications. Following this idea, this research proposes the root queue modification with the root queue management to enhance the *locally arbitrated* Bluetree-based architecture. It is to employ and utilise an additional hardware queue with queue management between the Bluetree interconnect root and the shared memory module to smooth resource sharing and reduce variation of memory access latency across the Bluetree-based architecture.

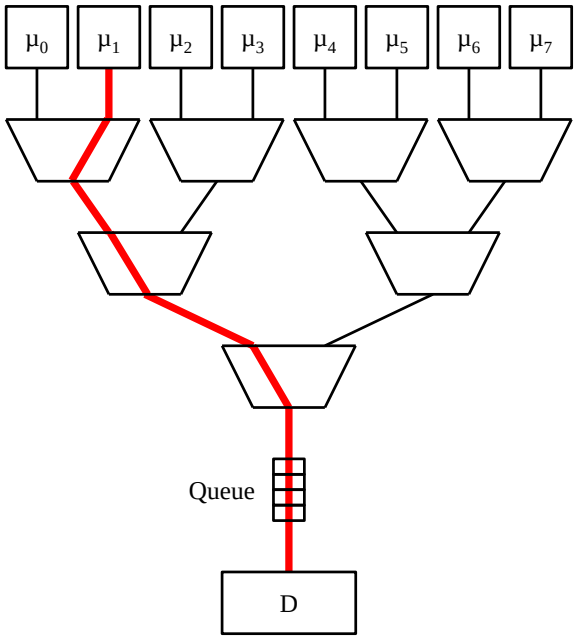

Figure 5.2. Bluetree-based Architecture with Root Queue Modification

5.2 Root Queue Modification

This section enhances the *locally arbitrated* architecture and modifies the Bluetree design with an additional hardware queue. As shown in Figure 5.2, the queue is employed to connect the root of the Bluetree interconnect and the shared memory module. As request paths overlap to the root of the tree-based interconnect, every single request will be relayed into the shared hardware queue. The root queue buffers all the requests that arrive at the Bluetree root.

The design of the root queue is based on bypass FIFO buffer. If the queue is empty, a request can be relayed to the root memory module directly without any additional delay. If the queue is not empty, it temporarily stores the requests that arrive but can not be immediately processed by the root memory module. The FIFO buffer also treats the queued requests equally, and the

first-arrived request will be relayed to the memory module first. In this way, it remains the arrival sequence of memory requests from the Bluetree interconnect, alleviating the contention over the overlapped data paths.

With the introduction of the root queue at the Bluetree root, the pending memory requests are relayed across the FIFO buffers in sequence, instead of blocking each other. It alleviates the inter-path interference. This architectural modification actually introduces additional resources to influence the timing behaviour. With sufficient root queue size, all the outstanding memory requests can be stored in the buffers, rather than blocking the overlapped interconnect. In this way, there is no contention to the shared request paths. The root memory responses to these requests in FIFO sequence, and new arrival requests have to wait in queue behind. This can be defined as the *queued service* which smooths the resource sharing and thus reduces the variation of memory access latency across this *locally arbitrated* architecture.

The premise of *queued service* is that the size of the root queue is sufficiently large enough to store all outstanding requests in the system. Due to the architectural features, the *locally arbitrated* Bluetree interconnect also provides buffers as well as the root queue. The amount of the total queued buffers in this architecture is analysed as follows.

- The root memory provides 1 buffer - a request occupying the memory module can be considered as stored locally.
- The employed root queue provides Q buffers (size).
- The Bluetree root multiplexer provides 1 pipelined buffer.
- Either the Bluetree multiplexer adjacent to the root stage provides 1 buffer. If buffers from both Bluetree multiplexers are considered, there

may be path contention. With the aim to guarantee the *queued service*, only a single buffer can be considered as applicable.

Based on the above analysis, the total size of the buffers at the Bluetree root is $Q+3$. On the other hand, as for practical applications, the number of memory requests issued to a system is limited, and the memory access pattern is dependent on memory response. This potentially follows the workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$. In this case, outstanding request number to the shared memory $N^{\mu}_{RQ}(D)$ is assumed as the sum of $N^{\mu}_{RQ}(\mu_i)$ in the entire Bluetree interconnect. With these above assumptions, the minimum size of the root queue Q_S for the *queued service* is $Q_S = N^{\mu}_{RQ}(D) - 3$. The *queued service* requirement can be summarised as follows.

$$Q \geq Q_S \; where \; Q_S = N^{\mu}_{RQ}(D) - 3 \tag{5.1}$$

When the *queued service* requirement is satisfied, the system stores the outstanding memory requests into the root buffers in sequence. The queue modification effectively smooths the sharing of the critical interconnect paths within the multi-core architecture. Besides that, this method requires no modification to software operations.

5.2.1 Timing Behaviour Analysis

The timing behaviour analysis of memory accesses across the modified *locally arbitrated* architecture follows the generic analytical flow in Section 4.1. The employment of the root queue introduces additional blocking within this architecture, and memory requests stalled in the root queue only leads to *intra-path blocking*. With blocking at the tree root, the entire interconnect stalls, blocking the flow of memory requests in each path. However, this does

not complicate the blocking behaviour within the shared distributed interconnect, and the maximum increase of memory access latency is proportional to the root memory latency $t(D)$.

Memory request ω in *priority path* gives $\omega \in P_i$. The maximum blocking number up to and including any given Bluetree stage β_k can be computed with (4.6) and (4.7). The maximum blocking number that the request ω experiences across the request path $N_{RQ}^{WC}(\omega)$ can be calculated iteratively, starting from the Bluetree leaf stage to the Bluetree root stage β_0 within the interconnect. With the root queue size Q, the maximum blocking number in the request path $N_{RQ}^{WC}(\omega)$ can be determined with the sum calculation that the iterative process result accumulated to the root stage $N_{RQ}^{WC}(\beta_0)$ plus Q as follows. For example referring to Table 4.1, as for the 8-client Bluetree architecture with the root queue size $Q = 8$, the maximum blocking number in the worst case is $N_{RQ}^{WC} = 27 + 8 = 35$ with the local blocking factor $\alpha = 1$.

$$N_{RQ}^{WC}(\omega) = N_{RQ}^{WC}(\beta_0) + Q \tag{5.2}$$

Afterwards, the worst-case memory access latency $t^{WC}(\omega)$ can be calculated with (4.5). With this worst-case assumption, memory requests suffer pessimistic blocking, thus no variation.

As for practical applications, the number of memory requests issued to a system is limited. Outstanding request number $N_{RQ}^{\mu}(\mu_i)$ can be determined by workload pattern from clients such as exact memory access profiles. This relies on effective analysis on accurate behaviour of application software. Instead, outstanding request number $N_{RQ}^{\mu}(\mu_i)$ can be determined by the architecture of a processor, i.e., maximum number of outstanding memory requests before the processor stalls. For example, a processor can be designed employing AXI protocol, which allows only a single outstanding request between

the master-slave pair. An alternative method is to utilise traffic shaping to limit the number of outstanding memory requests and thus determine outstanding request number $N^{\mu}_{RQ}(\mu_i)$. Afterwards, outstanding request number to shared memory $N^{\mu}_{RQ}(D)$ can be determined with the sum calculation in this architecture. The increasing of $N^{\mu}_{RQ}(D)$ complicates the timing analysis in the original *locally arbitrated* Bluetree-based architecture, and it requires to derive the detailed status of the memory flow and the local arbiter at every pipelined stage with exact memory access profiles. By contrast, with the root queue modification, the value of $N^{\mu}_{RQ}(D)$ can be used to determine the minimum size Q_S with (5.1) thus to satisfy the *queued service* requirement.

When the *queued service* requirement is satisfied, this architecture stores the outstanding memory requests into the root buffers, waiting for the service of the shared memory module in FIFO sequence. In this case, the root queue modification smooths the resource sharing and thus reduces variation of memory access latency. Besides that, the root queue modification also facilitates timing analysis for real-time applications. The *queued service* allows memory requests to experience the same maximum queued delay, and the pending period due to the root memory latency $t(D)$ can mask the data path latency across the pipelined buffers. Therefore, the worst-case memory access latency of ω across this architecture can be bounded as follows.

$$t^{WC}(\omega) \leq N^{\mu}_{RQ}(D) \times t(D) \tag{5.3}$$

It is to be noted that if traffic shaping is employed to determine outstanding request number $N^{\mu}_{RQ}(\mu_i)$, the bound provided by the above analysis only guarantees the worst-case memory access latency between the traffic shaping components across the interconnect and the shared root memory module, instead of the end-to-end latency from clients. This requires additional anal-

ysis to determine the time consumed between clients and the traffic shaping components in such design.

In addition, request interval $T^{\mu}_{RQ}(\mu_i)$ also affects memory workloads. With a very small request interval such as $T^{\mu}_{RQ}(\mu_i) = 1$, memory requests will be issued arriving to the interconnect root more intensively. This actually quickly fills the shared root queue. If request interval $T^{\mu}_{RQ}(\mu_i)$ remains identical, memory access latency will be identical. By contrast, the varying request interval $T^{\mu}_{RQ}(\mu_i)$ leads to varying memory access latency. Considering the memory workload pattern which is dependent on the response time, the new issued requests arrive at the root queue distributed in time. In this case, such memory requests suffer varying queued delays.

To sum up, with sufficient root queue size to satisfy the *queued service* requirement in (5.1), memory access latency across this architecture only varies with varying memory workloads, but no longer due to the resource sharing issue. In this case, memory access latency can be bounded applying (5.3). However, memory access latency can still vary with either the decreasing of outstanding request number $N^{\mu}_{RQ}(D)$ or the increasing of request interval $N^{\mu}_{RQ}(\mu_i)$. If memory workloads change dramatically, these pending memory requests can suffer widely varying queued delays at the root of the interconnect (potentially in the root queue). Accordingly, memory access latency varies widely.

5.2.2 Root Queue Management

A potential improvement method to further reduce variation of memory access latency or even keep memory access latency identical is to utilise dummy packets at root of the *locally arbitrated* interconnect. Dummy packet is gen-

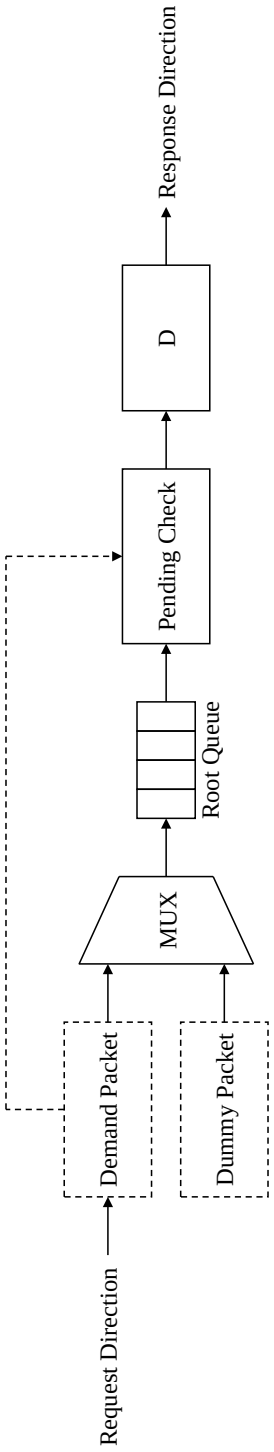

Figure 5.3. Root Queue Management with Hardware Design

erated locally without any information from clients. It is used to fill in the root queue until the queue is full. When a dummy packet is relayed out of the root queue, it stalls consuming a time period which equals to the root memory latency $t(D)$.

The root queue management is then deployed to both dummy packets and demand packets which are memory requests. First, when demand packets arrive at the root queue, they have to be allowed into the queue first rather than any dummy packet. Second, when there is any demand packet arriving but pending out of the root queue, the first dummy packet in the FIFO queue has to be discarded. This aims to guarantee that all demand packets experience same queued delays and do not suffer any additional delay due to dummy packets.

Figure 5.3 shows an example of the hardware design of this root queue management. Dummy packets are generated locally at the root of the interconnect and used to fill in the root queue. In this case, before demand packets arrive, such as no memory requests issued by clients during the cold starting of the system, the hardware queue is already full with dummy packets. Dummy packets essentially consume time period $t(D)$ however with no memory response. When demand packets arrive at the root queue, the 2-to-1 multiplexer simply employs the static priority-based arbitration scheme, always allowing the demand packet to have the higher priority and get relayed into the queue first. Besides that, the pending check process is used to guarantee there is no demand packet waiting out of a full root queue. If any, the queue discards the first dummy packet in the FIFO sequence. This hardware root queue management can allow demand requests with similar queued delays, and the relevant delays vary within a period which equals to the root memory latency $t(D)$.

The root queue modification can eliminate the resource sharing issue within the *locally arbitrated* architecture, and thus memory access latency varies with varying memory workloads. Based on this, the root queue management further reduces variation of memory access latency that memory access latency only varies within a single root memory time — no longer varies with memory workloads. With further reduced variation of memory access latency, the average-case memory access latency is closer to the worst-case memory access latency across this modified architecture. In this case, the root queue management leads to increased average memory access latency, which potentially increases the overall program execution time, especially harming those applications with not intensive root memory accesses.

The above hardware design illustrates an example of the root queue management. It is more applicable to the hardware platform with fixed architectural feature, such as a system employing AXI protocol with fixed outstanding request number. Besides that, this design requires additional hardware resource including buffers and deployment of the root management. As the queue management keeps checking and filling processes, this design also increases power consumption at runtime. An alternative design can rely on the aid of compiler with explicit instruction on the root queue management or the effective application behaviour analysis for flexibility in utilisation of the root queue, which remains the future work.

5.3 Evaluation: Hardware Simulations

This section evaluates the effectiveness of the root queue modification on reducing variation of memory access latency across the *locally arbitrated* Bluetree-based architecture by hardware simulations. Multiple experiments

with varying experimental parameters has been conducted, and 1 group is selected in this section to evaluate memory access latency across 8-client Bluetree-based architectures with increasing root queue size.

Traffic generators are employed as clients with synthetic memory workloads which follow the workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$. Each traffic generator issues 36 memory requests totally. Request interval is fixed as 1 $T^{\mu}_{RQ}(\mu_i) = 1$ for a relatively simple workload pattern, and outstanding request number $N^{\mu}_{RQ}(\mu_i)$ varies as *group c* in Table 4.2. In this case, the analysis of experimental results partially follows that of Figure 4.5 (c). Bluetree blocking factor is set as $\alpha = 1$, and each local arbiter is set with the same value in the entire Bluetree interconnect to provide relatively fair accesses for all clients. The root queue is designed using bypass FIFO in Bluespec SpecialFIFOs package [107], and the root queue is reconfigurable with the root queue size Q. In this architecture, the *queued service* requirement is $Q_S = N^{\mu}_{RQ}(D) - 3 = 13 - 3 = 10$, referring to the sum of the outstanding request number $N^{\mu}_{RQ}(\mu_i)$. In addition, $Q = 0$ indicates no root queue modification. The root memory module is designed using Bluespec BRAM package [107] with extra delays as a constant $t(D) = 20$ in clock cycles. Bluetree-based systems are implemented using Bluespec System Verilog [107][108], with simulations running on BlueSim simulator [107][108].

The experimental parameter is the root queue size Q which increases from 0, 5, 10, 15 to 20. This experiment measures memory access latency across the modified 8-client Bluetree-based architectures that the latency of each memory access is measured. In addition, memory request release time of each memory access is also measured. The measured results are shown in Figure 5.4 and Figure 5.5 with scatter plot. The horizontal axis is for memory request release time in clock cycles, and the vertical axis is for memory access latency in clock cycles.

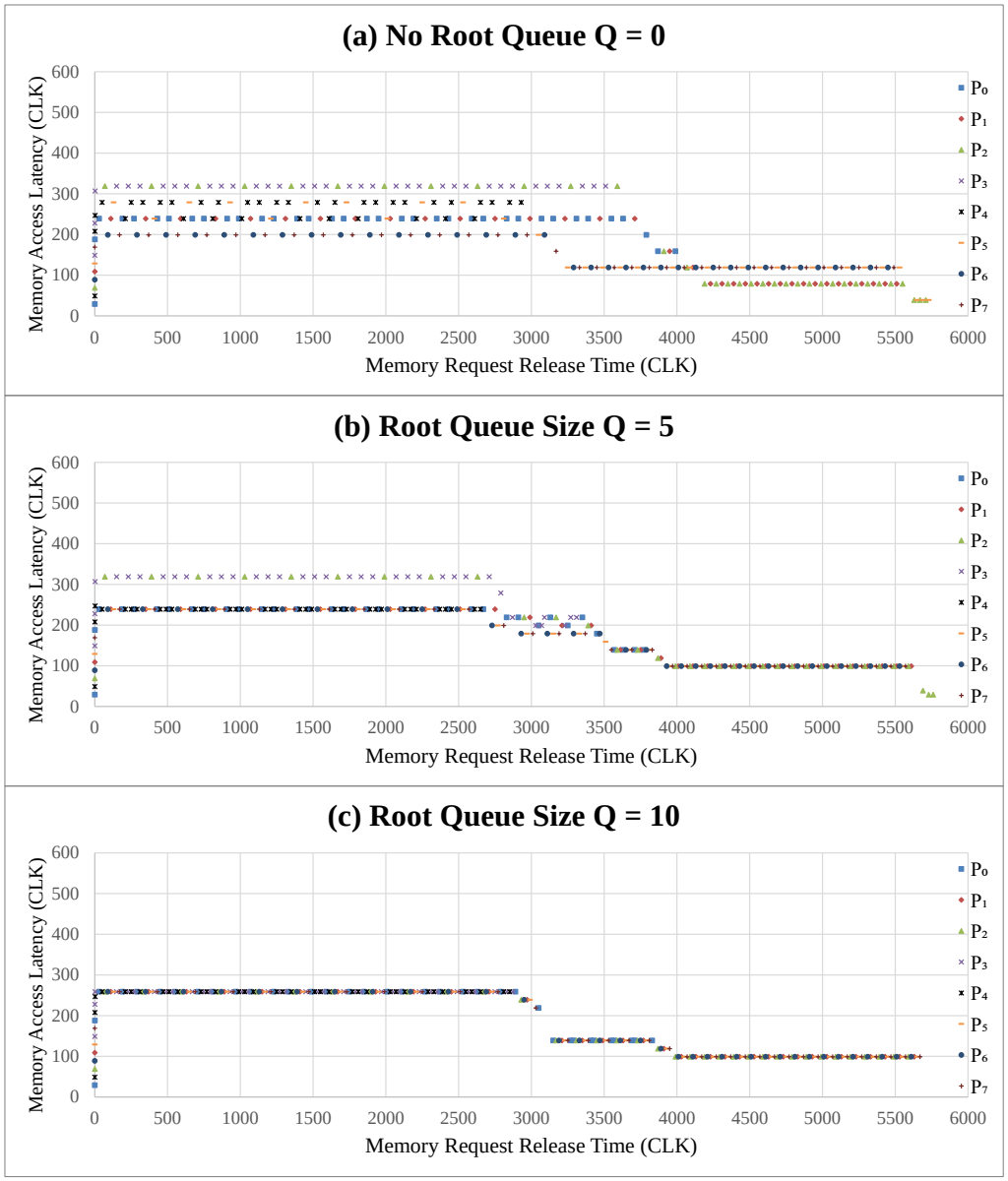

Figure 5.4. Memory Access Latency with Increasing Root Queue Size

Figure 5.4 (a) demonstrates same measured results as in Figure 4.5 (c). Memory access latency increases sharply in a very short period of time from the start period of the simulation, with intensively issued memory requests into the system. Essentially, as workload pattern is dependent on response time, the release of memory requests drops that the increase of memory access latency stops in turn. Memory access latency in each path tends to reach the

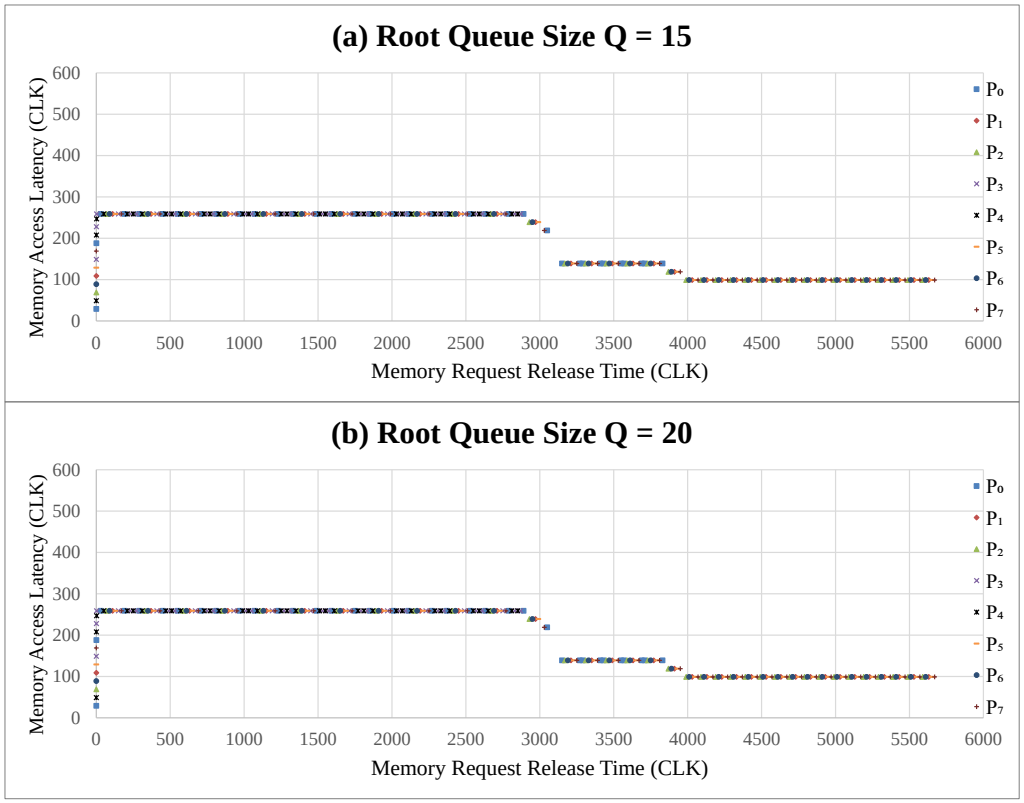

Figure 5.5. Memory Access Latency with Increased Root Queue Size

corresponding maximum limit. With fixed request interval, regular values of memory access latency can be observed. As Bluetree allows varying blocking behaviour, varying memory access latency can be observed. For example, memory access latency in path P_4 and path P_5 varies, approximately either 280 or 240. Besides that, the inter-path interference also affects paths nearby. For example, path P_2 with $N^{\mu}_{RQ}(\mu_2) = 1$ is severely affected by path P_3 with $N^{\mu}_{RQ}(\mu_3) = 3$, and thus it suffers high memory access latency at approximately 325 due to additional blocking. With the decreasing of outstanding requests $N^{\mu}_{RQ}(D)$ in the end period of the simulation, the contention to the shared resource reduces and memory access latency reduces.

Figure 5.4 (b) shows memory access latency across modified Bluetree-based architecture with root queue size $Q = 5$. Compared with Figure 5.4 (a),

memory access latency tends to coincide to be identical (as with fixed request interval) in Figure 5.4 (b). With the root queue, some pending requests can be stored in the shared FIFO buffer instead of blocking in the overlapped data paths. In this way, the root memory is able to response to these pending requests in sequence. This smooths resource sharing, and thus variation of memory access latency is reduced. However, in this measurement, the root queue modification benefits some memory access paths that memory access latency in paths with high memory workloads reduces. For example, memory access latency in path P_4 with $N^{\mu}_{RQ}(\mu_4) = 3$ no longer varies between 280 and 240, instead remaining at approximately 240. By contrast, memory access latency in paths with relatively lower workloads increases due to the smoothing effects. For example, memory access latency in path P_7 with $N^{\mu}_{RQ}(\mu_7) = 1$ increases from 200 to 240. Besides that, root queue size $Q = 5$ is not enough to buffer all the outstanding requests in this architecture. As shown in the figure, memory access latency in either path P_2 or path P_3 still suffers high memory access latency due to the resource sharing issue.

When the root queue is reconfigured with $Q = 10$, the *queued service* requirement is satisfied that the resource sharing issue is eliminated where $Q \geq Q_S$ and $Q_S = N^{\mu}_{RQ}(D) - 3 = 13 - 3 = 10$ in this architecture. Figure 5.4 (c) shows the measured results. The root queue is fully filled quickly from the start period of the simulation, and then memory access latency remains identical (due to fixed request interval). Afterwards, the worst-case memory access latency can be bounded as $t^{WC}(\omega) \leq N^{\mu}_{RQ}(D) \times t(D) = 13 \times 20 = 260$, and the highest measured value is 259. It drops significantly compared with Figure 5.4 (a) (where the highest observed memory access latency is approximately 325). In the end period of the simulation, memory access latency reduces with the decreasing of outstanding request $N^{\mu}_{RQ}(D)$. In this case, memory access latency across the Bluetree-based architecture with root queue modification varies with varying workloads, but not due to the resource sharing issue.

Figure 5.5 shows memory access latency with further increased root queue size $Q = 15$ and $Q = 20$. Both Figure 5.5 (a) and Figure 5.5 (b) show the same results as Figure 5.4 (c). In this case, the root queue modification with a larger root queue size (larger than the *queued service* requirement) has no effect to the timing behaviour.

5.4 Evaluation: FPGA Experiments

This section continues to evaluate the effectiveness of the root queue modification on reducing variation of memory access latency across the *locally arbitrated* Bluetree-based architecture by FPGA experiments. Multiple experiments with varying experimental parameters has been conducted, and 3 groups are selected in this section following experimental setup in previous chapters. These experiments evaluate memory access latency across 8-client Bluetree-based architectures with no root queue modified, with sufficient root queue modification that *queued service* requirement is satisfied, and with root queue management, by varying memory workloads.

Traffic generators are employed as clients with synthetic memory workloads which follow the workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$. Each traffic generator issues 100 memory requests totally. Bluetree blocking factor is set as $\alpha = 1$, and each local arbiter is set with the same value in the entire Bluetree interconnect to provide relatively fair accesses for all clients. The root queue is designed using bypass FIFO in Bluespec SpecialFIFOs package [107] with reconfigurable root queue size Q. The root memory module is designed using FPGA BRAM [110] with extra delays as a constant $t(D) = 20$ in clock cycles. Bluetree-based systems are synthesised using Xilinx Vivado [111][112] and implemented on Zedboard [113] with 100MHz of clock frequency.

In the following experiments, synthetic memory workloads vary with either varying outstanding request $N^{\mu}_{RQ}(\mu_i)$ or varying request interval $T^{\mu}_{RQ}(\mu_i)$. These experiments measure memory access latency across Bluetree-based architectures that the latency of each memory access is measured. In addition, memory request release time of each memory access is also measured.

5.4.1 Memory Access Latency with Unbalanced Path Workloads

Following experimental setup of hardware simulations, the initial FPGA experiment is conducted with unbalanced path workloads. Outstanding request number $N^{\mu}_{RQ}(\mu_i)$ varies as *group c* in Table 4.2, and request interval $T^{\mu}_{RQ}(\mu_i)$ varies as $T^{\mu}_{RQ}(\mu_i) \in [1, 64]$ with randomly generated values in Appendix A.1.

Figure 5.6 shows scatter plot of memory access latency with memory request release time in this measurement. The horizontal axis is for memory request release time in clock cycles, and the vertical axis is for memory access latency in clock cycles. Figure 5.6 (a) with no root queue shows the same results as Figure 4.11 (a). Compared with Figure 5.6 (a), scatters in Figure 5.6 (b) tends to coincide. This follows similar trend from Figure 5.4 (a) to Figure 5.4 (c) in the measurement of hardware simulations. However, with varying request interval, memory access latency varies in this measurement. In addition, the highest measured memory access latency reduces to 258 in Figure 5.6 (b) where the worst-case memory access latency is bounded as 260 with sufficient root queue size (referring to the analysis of hardware simulations). However, with the variation of outstanding request to shared memory $N^{\mu}_{RQ}(D)$, memory access latency still varies in Figure 5.6 (b), such as the reduction of memory access latency in the end period. By comparison, scatters in Figure 5.6 (c)

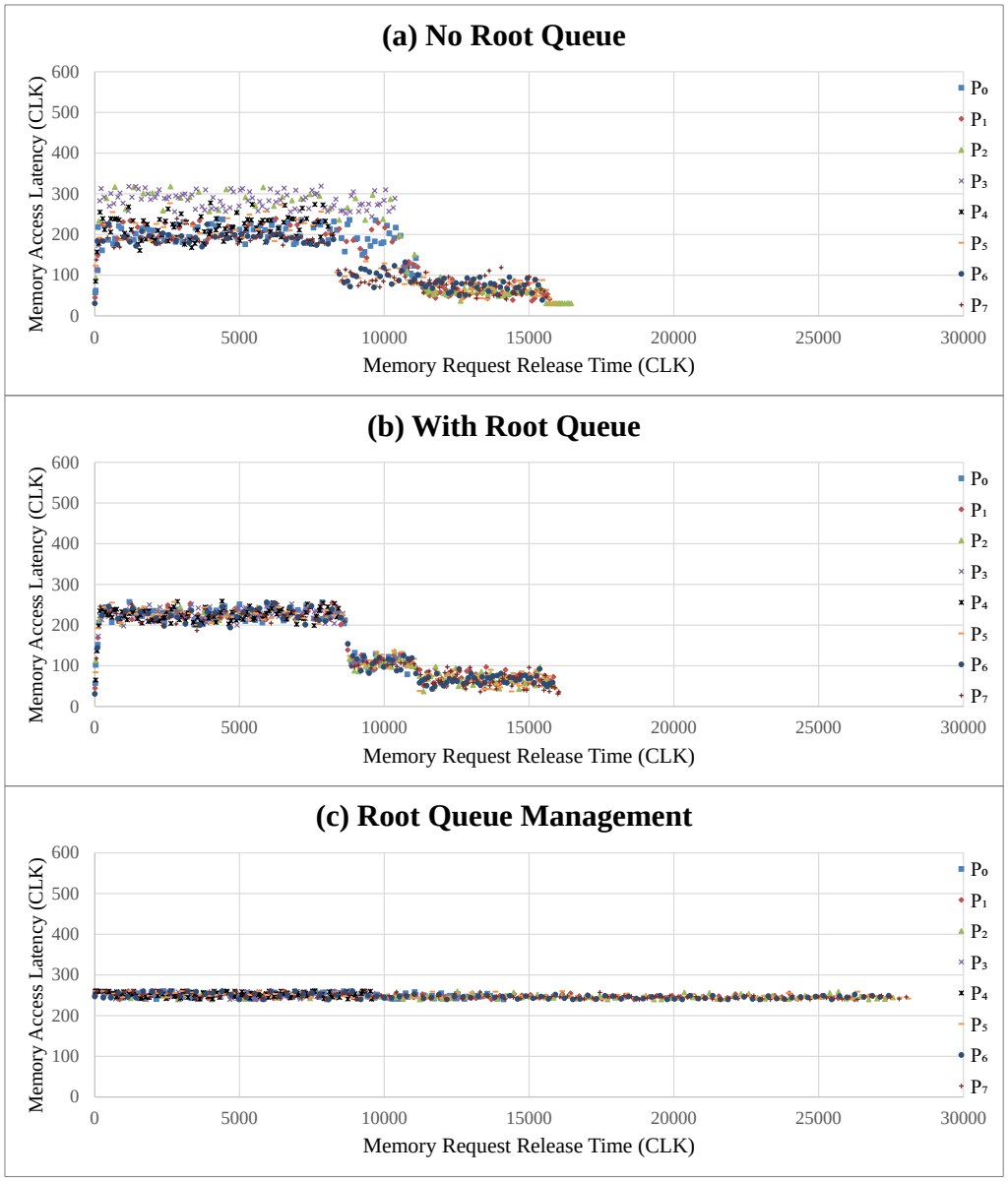

Figure 5.6. Memory Access Latency with Varying Workloads

almost coincide from the start to the end in this measurement. However, a much longer horizontal axis can be observed that the execution time of this experiment is much longer than others.

Figure 5.7 shows boxplot of memory access latency in this measurement. Similar to the analysis of hardware simulations, the root queue modification tends

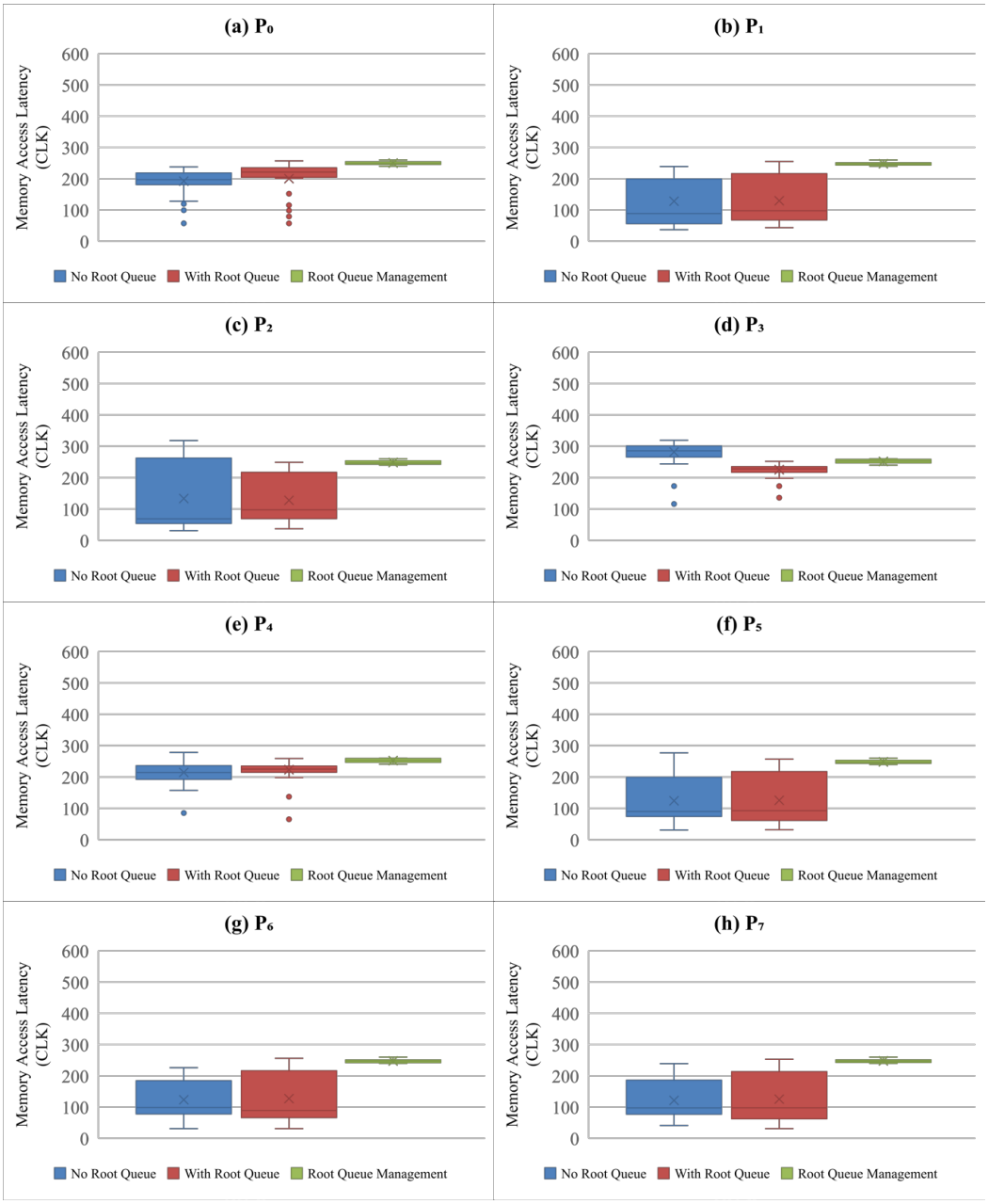

Figure 5.7. Boxplot of Memory Access Latency with Varying Workloads

to benefit memory access paths with relatively higher memory workloads. For example, compared with the measured results of no root queue and with root queue, median and mean in Figure 5.7 (d) for path P_3 with $N_{RQ}^{\mu}(\mu_3) = 3$ drops, and the difference between the maximum line and the minimum line is

noticeably reduced in in Figure 5.7 (e) for path P_4 with $N^{\mu}_{RQ}(\mu_4) = 3$. However, either the interquartile range or the difference between the maximum line and the minimum line remains or even increases in some memory access paths, such as in Figure 5.7 (f) for path P_5 and in in Figure 5.7 (h) for path P_7, even with only 1 outstanding request. With the variation of outstanding request to shared memory $N^{\mu}_{RQ}(D)$, the root queue is not fully filled and thus memory requests suffer varying queued delays. In this case, only with the root queue modification, memory access latency still varies with varying memory workloads. By comparison, the root queue management guarantees similar queued delays. Referring to the figure, at the expense of higher median and mean, both the interquartile range and the difference between the maximum line and the minimum line are significantly reduced, in each memory access path.

5.4.2 Memory Access Latency with Balanced Path Workloads

This experiment is conducted with balanced path workloads. Outstanding request number is fixed as $N^{\mu}_{RQ}(\mu_i) = 2$ as shown in Table 4.3, and request interval varies as $T^{\mu}_{RQ}(\mu_i) \in [1, 64]$ with randomly generated values in Appendix A.1. Figure 5.8 shows scatter plot of memory access latency with memory request release time in this measurement. The horizontal axis is for memory request release time in clock cycles, and the vertical axis is for memory access latency in clock cycles. Compared with scatter plot of Figure 5.8 (a), Figure 5.8 (b) and Figure 5.8 (c), scatters of memory access latency with memory request release time least distributed in Figure 5.8 (c). It is also observed similar horizontal axises, thus similar execution time of these experiments.

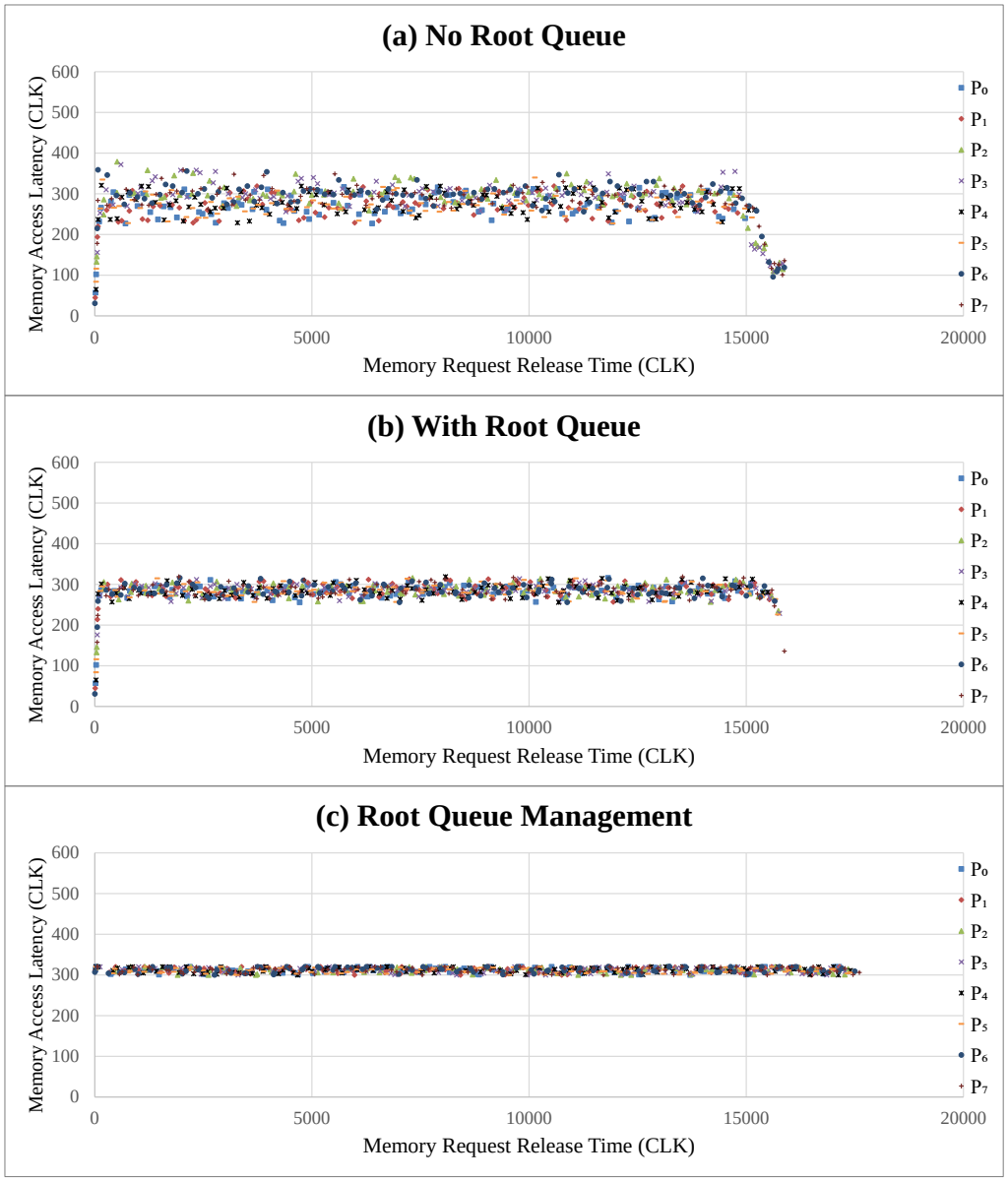

Figure 5.8. Memory Access Latency with Balanced Path Workloads

Figure 5.9 shows boxplot of memory access latency in this measurement. Compared with the measured results of no root queue, both the interquartile range and the difference between the maximum line and the minimum line are noticeably reduced with root queue in each memory access path, and median and mead almost remains the same values. By comparison, the interquartile range and the difference between the maximum line and the minimum line

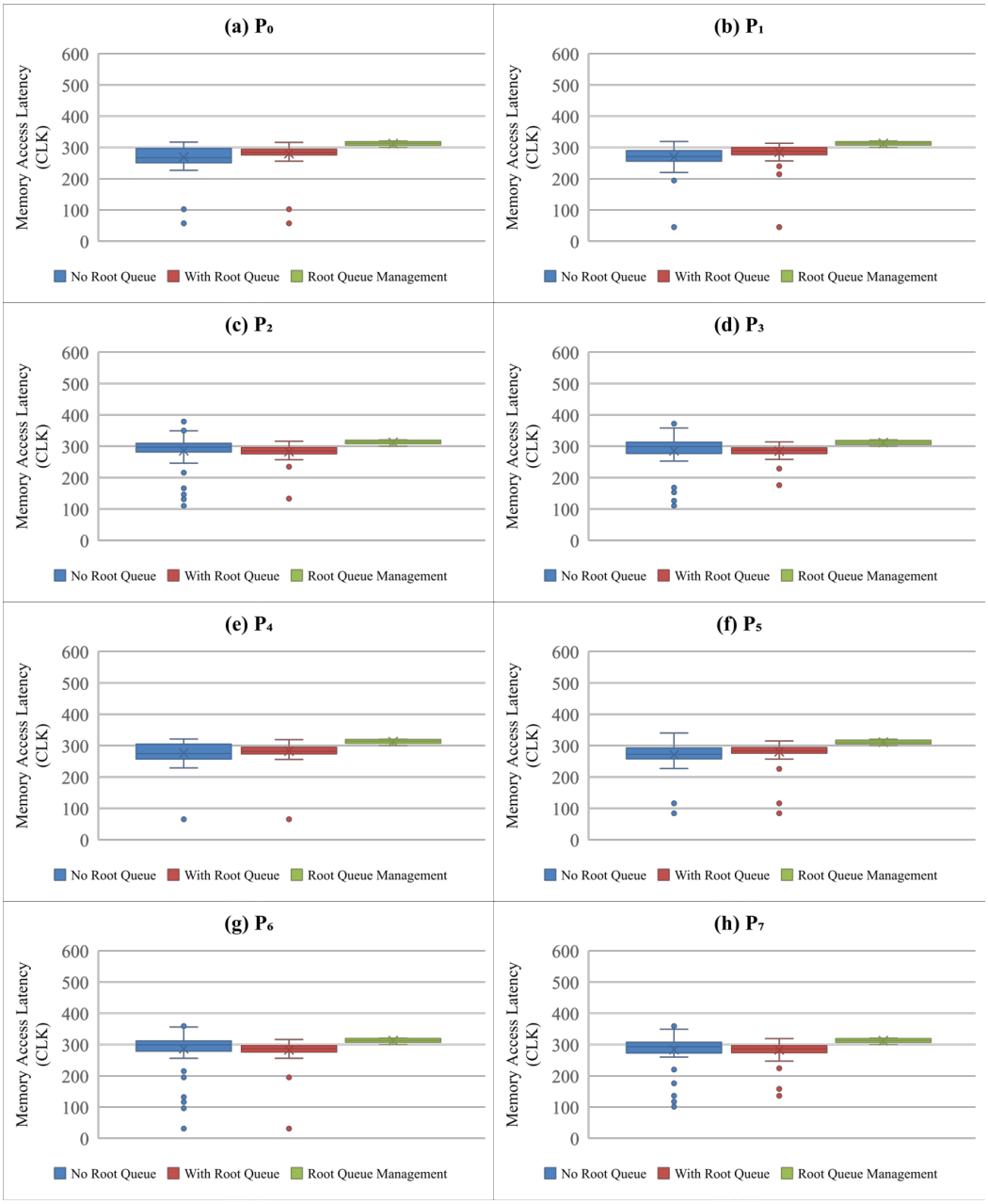

Figure 5.9. Boxplot of Memory Access Latency with Balanced Path Workloads

is further reduced with root queue management in each memory access path, with slightly increased median and mean referring to the figure. In addition, no outliners are observed with root queue management.

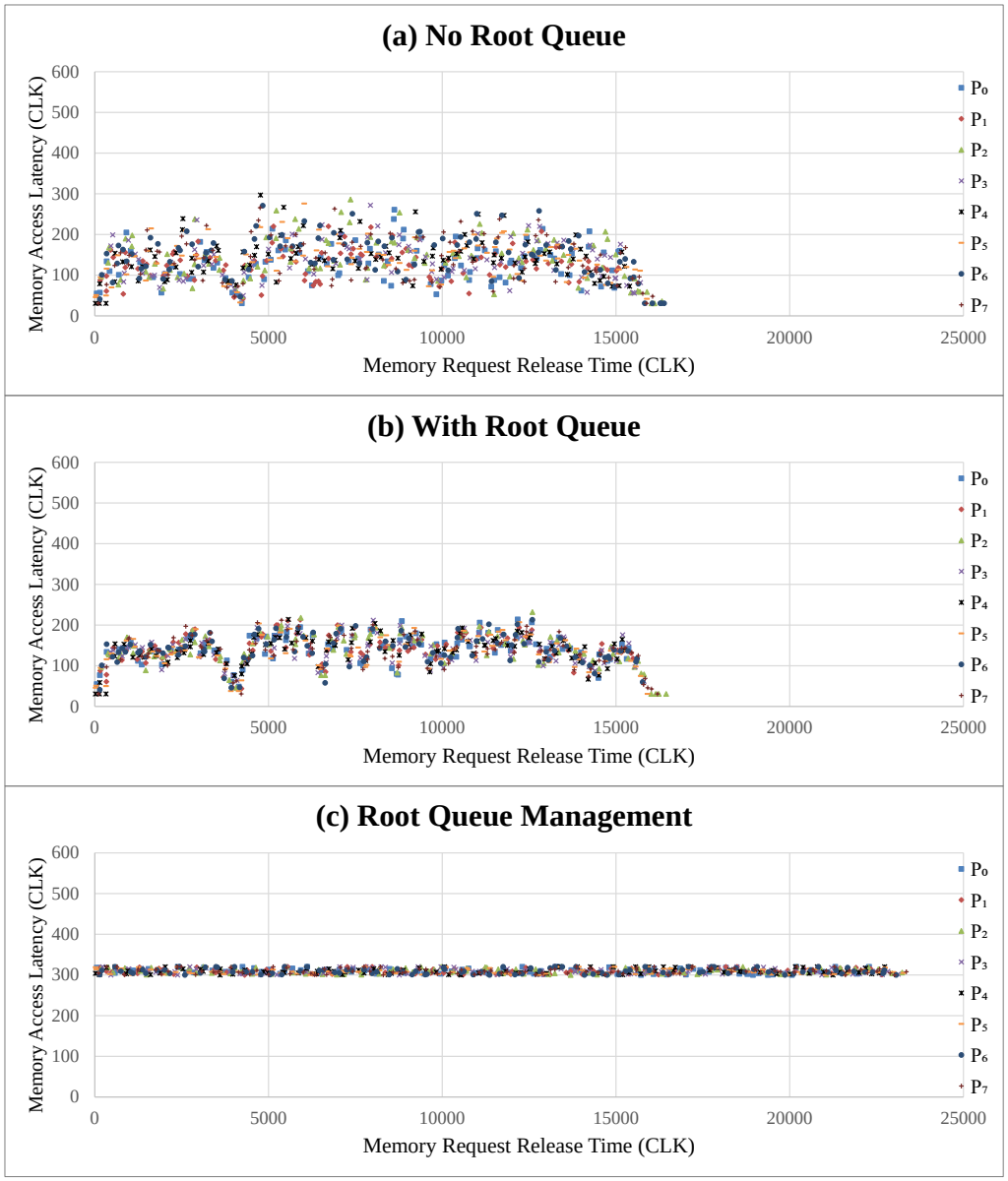

Figure 5.10. Memory Access Latency with Increasing Request Intervals

5.4.3 Memory Access Latency with Increasing Request Intervals

Based on similar setup of the above experiment, the variation of request interval increases as $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$ in this experiment, with randomly

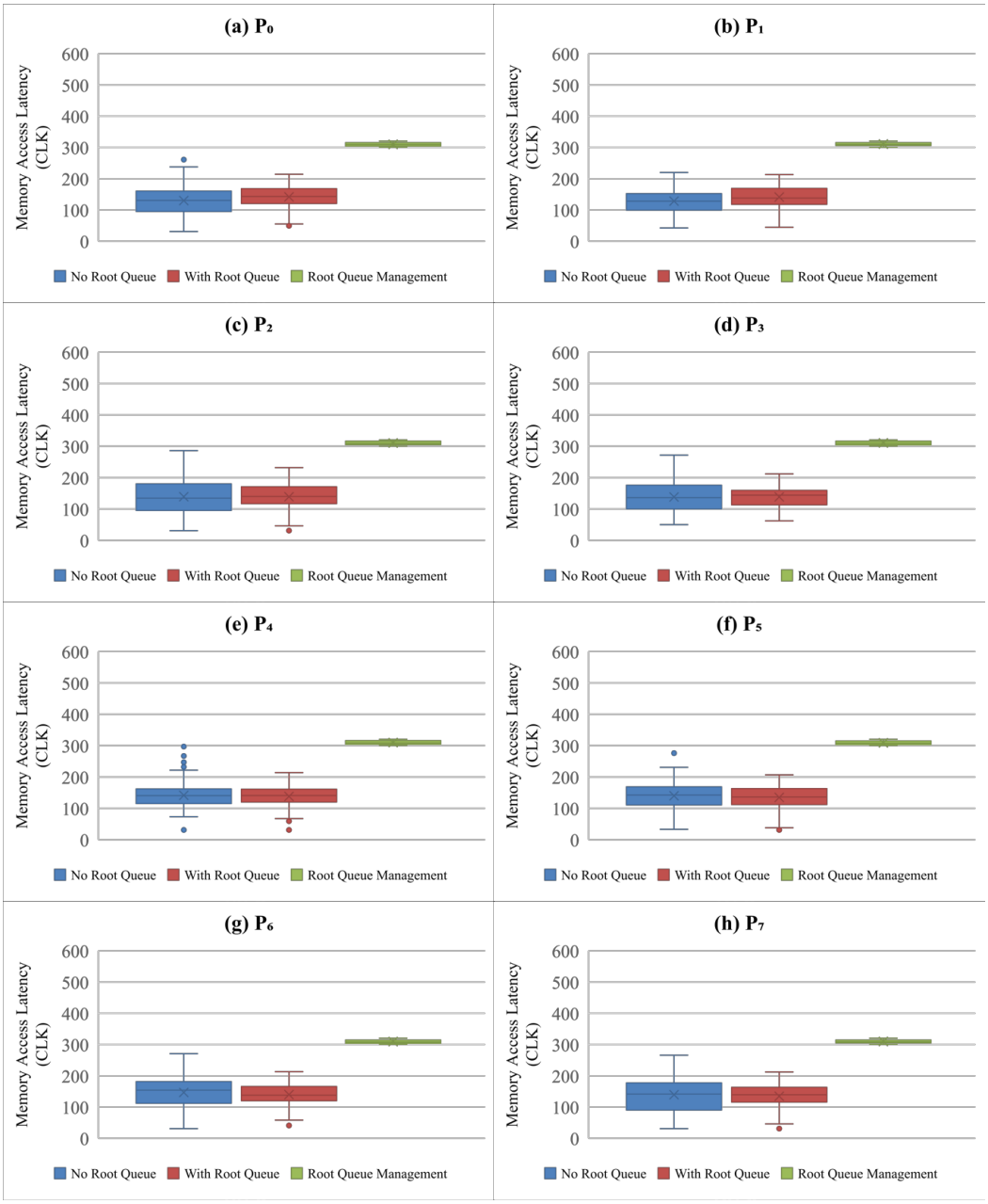

Figure 5.11. Boxplot of Memory Access Latency with Increasing Request Intervals

generated values in Appendix A.2. Figure 5.10 shows scatter plot of memory access latency with memory request release time in this measurement. The horizontal axis is for memory request release time in clock cycles, and the

vertical axis is for memory access latency in clock cycles. Compared with Figure 5.10 (a) and Figure 5.10 (b), scatters less distributed, and the highest measured memory access latency noticeably drops. By comparison, scatters tend to coincide in Figure 5.8 (c). However, a longer horizontal axis can be observed referring to the figure, and thus the execution time of this experiment is much longer than others.

Figure 4.14 shows boxplot of memory access latency in this measurement. By comparison, the root queue management reduces variation of memory access latency to the minimum, with both the smallest interquartile range and the smallest difference between the maximum line and the minimum line in each memory access path. With widely varying request intervals $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$, memory requests are not intensively issued to this system. Referring to the measured results of the original Bluetree-based architecture with no root queue, both median and mean are reduced to half of those in the above experiment with request interval $T^{\mu}_{RQ}(\mu_i) \in [1, 64]$. The similar reduction trend can also be observed in measurements with root queue modification. The modified Bluetree system is actually not sufficiently loaded with request interval $T^{\mu}_{RQ}(\mu_i) \in [1, 256]$, and thus the root queue is not fully filled. In this case, memory requests suffer varying queued delays. By contrast, the root queue management guarantees a full root queue, thus almost the same queued delays. As shown in the figure, either the interquartile range or the difference between the maximum line and the minimum line is approximately 20, and this time period equals to the root memory latency $t(D)$. In this experiment, the worst-case memory access latency in the *queued service* can be bounded as $t^{WC}(\omega) \leq N^{\mu}_{RQ}(D) \times t(D) = 16 \times 20 = 320$. Compared with the root queue modification, the root queue management keeps memory access latency much closer to the worst-case bound in the *queued service*.

5.5 Summary and Discussion

This chapter proposes the root queue modification with the root queue management to the *locally arbitrated* architecture by employing and utilising an additional hardware queue with queue management between the shared distributed interconnect root and the shared memory module. With sufficient root queue size, the *queued service* eliminates the resource sharing issue within the *locally arbitrated* architecture that memory access latency only varies with varying memory workloads. This root queue modification also facilitates the timing behaviour analysis that the worst-case memory access latency can be bounded applying calculations instead of deriving with exact memory access profiles. Besides that, the root queue management is further proposed by utilising dummy packets to reduce variation of memory access latency. This guarantees that memory requests experience the same queued delays. Based on the root queue modification, the root queue management keeps the average-case memory access latency closer to the worst case in the *queued service*.

Experiments with hardware simulations and FPGA implementations demonstrate the effectiveness of the proposed work on reducing the variation of memory access latency. Experimental results from hardware simulations demonstrate more noticeable effectiveness with the increasing of the root queue size. The root queue modification reduces variation of memory access latency and also reduces the highest measured memory access latency. Experimental results from FPGA implementations demonstrate that memory access latency only varies due to varying memory workloads with the root queue modification, but no longer due to resource sharing issue. By comparison, the root queue management further reduces variation of memory access latency

to minimum that memory access latency can only vary with varying root memory latency, regardless of memory workloads.

In summary, the root queue modification with the the root queue management effectively reduces variation of memory access latency across the *locally arbitrated* architecture and facilitates the timing behaviour analysis. This contributes to solve the research question *Q2*.

The main contribution presented in this chapter is summarised as follows.

The root queue modification with the root queue management is proposed for multi-core architectures with locally arbitrated interconnects that variation of memory access latency is effectively reduced and the timing behaviour analysis is also facilitated, contributing towards real-time multi-core systems.

It is to be noted that this research implements the root queue management in hardware design as an example. An alternative design can rely on the aid of compiler with explicit instruction on the root queue management or the effective application behaviour analysis for flexibility in utilisation of the root queue.

Chapter 6

Meshed Bluetree: Distributed Time-Predictable Multi-Memory Interconnect for Multi-Core Architectures

This chapter proposes a novel distributed multi-memory interconnect, Meshed Bluetree, for multi-core architectures. It is the extension of the tree-based distributed memory interconnect employing mesh-of-trees topology. The Meshed Bluetree architecture is constructed by the coupling of a router network and multiple *locally arbitrated* Bluetree-based architectures in parallel, allowing multiple processors to simultaneously access multiple memory modules with time-predictable behaviour. This aims to solve the research question *Q3: Can multi-core architectures with shared distributed memory interconnects be improved by architectural enhancement for increasing memory workloads?*

The reminder of this chapter is structured as follows. Section 6.1 analyses resource contention over the *locally arbitrated* architecture and the *globally arbitrated* architecture with increasing memory workloads. Section 6.2 proposes the design of Meshed Bluetree and explain the operation with analysis on the time-predictable behaviour of memory accesses. Section 6.3 evaluates the hardware consumption of Meshed Bluetree. Section 6.4 evaluates memory access latency across the Meshed Bluetree architecture with FPGA experiments using synthetic memory workloads. Section 6.5 evaluates the overall system performance of the Meshed Bluetree architecture with FPGA experiments using real-world benchmarks. Afterwards, Section 6.6 summarises this chapter and presents discussion.

6.1 Problem Analysis

In the emerging real-time application scenarios, such as autonomous vehicles and robotics, there is a stringent requirement on the execution time of application being both bounded in the worst case (thus time predictability) and low. To deal with complex functionality and achieve high performance, multi-core architectures are widely deployed where multiple processors share a single memory module. With the trend of either integrating more applications or employing more processors into the shared memory multi-core architecture, the contention over memory accesses potentially aggravates.

The multi-core architectures with shared distributed memory interconnects are able to provide the time-predictable behaviour over memory accesses. Besides that, the distributed pipelined stages can allow high synthesisable clock frequency, scaling to a large number of processors. The *locally arbitrated* architecture allows multiple memory requests in transfer simultaneously that

the contention over either the shared root memory module or the overlapped data paths increases with increasing memory workloads. This leads to increasing memory access latency. By contrast, the *globally arbitrated* architecture avoids contention over memory accesses based on global scheduling interval. However, it does not alleviate memory workloads. For example, with the increasing number of processors, memory access latency increases as well as the global scheduling cycle. In this case, both the *locally arbitrated* architecture and the *globally arbitrated* architecture potentially suffer increasing memory access latency with increasing workloads.

The methods to alleviate critical resource contention has also been widely studied on multi-core architectures. Intuitively, more effective root memory subsystem or local memory modules at processors can be deployed to alleviate the contention over memory accesses. The effectiveness of such methods essentially relies on the analysis of accurate application behaviour thus to exploit data efficiency as well as to predict the memory access behaviour to bound the worst case. A method is to regulate accesses to critical resource based on resource reserving. The design of real-time system also tends to achieve temporal isolation. Similar to the analysis of the *globally arbitrated* architecture, it aims to provide contention-free behaviour without alleviating memory workloads. A different method is message combining which can potentially combine memory requests to reduce the contention over the overlapped data paths within the tree-based interconnect. However, it fails to alleviate workloads to the shared root memory module.

Instead, an alternative method is to invest additional hardware resources, such as employing virtual channel to alleviate the router contention from multiple communication flows in NoC applications. As for the tree-based structure, Audsley et al. [105] proposes that multiple memory modules or memory banks can be independently employed at the root of the *locally ar-*

bitrated Bluetree-based architecture (potentially through a shared memory controller), aiming to provide diverse memory features to support mixed-criticality systems. This potentially increases memory bandwidth. However, it moves the design burden to the shared memory controller, and the shared tree root remains the architectural bottleneck of the *locally arbitrated* interconnect.

Following the idea of multiple root memory modules being engaged, this research proposes an architectural enhancement that the tree-based distributed memory interconnect can be extended to a multi-memory interconnect based on the mesh-of-trees topology [78][79]. In this way, the new distributed multi-memory interconnect allows multiple processors to simultaneously access multiple memory modules. This potentially alleviates the contention to a single shared memory module as well as the shared distributed memory interconnect. It aims towards time-predictable behaviour (i.e., with the analytical memory access latency and bounded worst case), memory access latency reduction in the average case, as well as scalability for multi-core architectures. The *locally arbitrated* Bluetree-based architecture is employed as an example which does not require to model memory requests in applications. This design can be extended to other configurations than the Bluetree design.

6.2 Meshed Bluetree

This section proposes the design of Meshed Bluetree, the distributed multi-memory interconnect for multi-core architectures. The topology of this design is based on mesh-of-trees (MoT) [78][79]. In the research [77], MoT is developed with single-clock-cycle data paths, using a set of switches coordinated by a global control signal to establish a complete memory access path dedicated

for a specific processor at a time. This MoT operates in the circuit-switched round-robin manner with centralised control, allowing data transfer between processors and memory modules within a single clock cycle and enabling relatively simple timing analysis. However, with an expanding system configuration (i.e., the number of processors and memory modules), the logic size of this centralised design increases logarithmically, which severely limits the maximum synthesisable clock frequency.

By contrast, the design of Meshed Bluetree employs distributed data paths with local arbitration. Although additional clock cycles are introduced, it allows much higher synthesisable clock frequency, scaling to a large system. The Meshed Bluetree architecture is proposed to alleviate the resource contention within the conventional *locally arbitrated* architecture, enabling multiple processors to share multiple memory modules. This aims to achieve good and scalable average-case performance, whilst providing time-predictable behaviour.

Figure 6.1 illustrates the architecture of Meshed Bluetree, which is constructed by the coupling of a distributed router network (the upper half) and multiple Bluetree-based architectures in parallel (the lower half). In this particular example, the Meshed Bluetree architecture employs 8 clients sharing 4 independent memory modules. Each client μ_i has a memory access path $P_{(i,j)}$ to connect to the memory module D_j where i is the client index and j is the memory module index. For example, the path $P_{(1,1)}$ for the client μ_1 to connect to the memory module D_1 is highlighted in the figure. The memory modules can be paralleled memory banks within a single DRAM module as analysed in [105]. The design can also be extended with paralleled scratchpad memory (SPM), cache, or mixed types of memory components. The Meshed Bluetree architecture allows sufficient design flexibility to support multi-core applications.

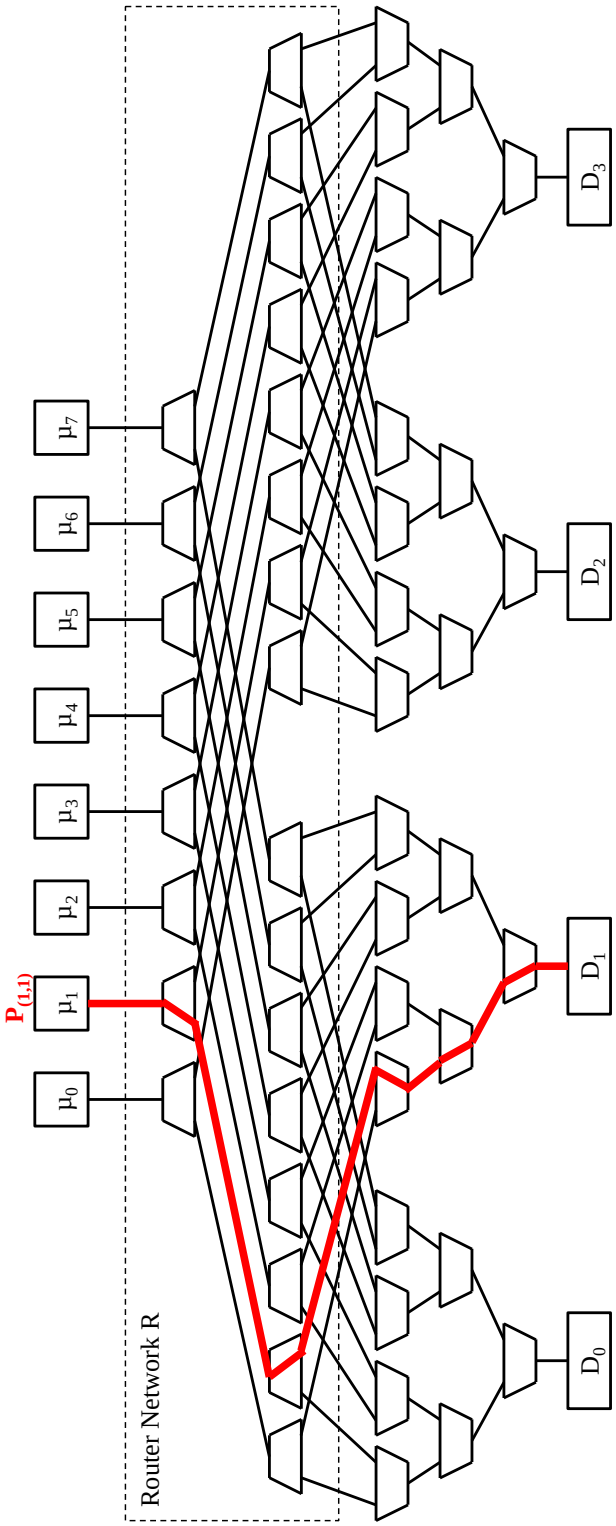

Figure 6.1. 8×4 Meshed Bluetree

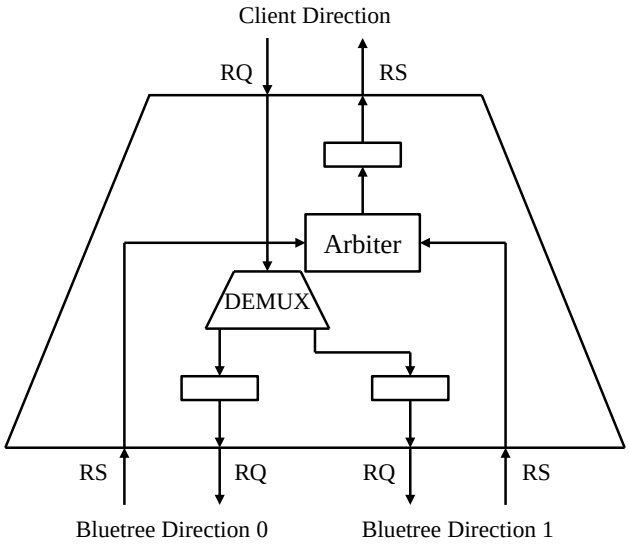

Figure 6.2. Bluetree Router

When a client μ_i issues a memory request, the router network R first decides the routing path and relays the request to a specific Bluetree-based architecture. Then the corresponding Bluetree interconnect B_j further multiplexes and relays this request to the destination memory module D_j. It is to be noted that the same subscript j indicates a one-to-one relationship between a Bluetree interconnect and a memory module. The memory response returns across the bi-directional meshed interconnect in a reverse process. As the memory address range can be partitioned across these paralleled memory modules, the simultaneous accesses to different memory modules can be processed concurrently. This potentially reduces the contention over a single memory module as well as a single shared memory interconnect and thus reduces memory access latency.

The router network R is constructed with multiple stages of Bluetree routers. With the number of memory modules N_D growing, the router network R scales with more pipelined router stages, which increases the router depth N_R in the tree-based architecture. In Figure 6.1, N_R is equal to 2. The de-

sign of the Bluetree router is shown in Figure 6.2. The local request path (named RQ as before) of Bluetree router is non-blocking, and the internal demultiplexer decides the route direction of memory requests. Pipelined buffers and client-server interfaces are also implemented, similar to the design of Bluetree multiplexer.

Arbitration occurs in the local response path (named RS as before) to decide which Bluetree direction of memory response to be relayed to the client direction, and potentially to next Bluetree routers. An applicable local arbitration scheme can be round-robin, which provides locally fair access for both Bluetree directions. It is also feasible to employ static-priority arbitration at the local router stage, always allowing the memory response from a single direction to have higher priority and get relayed first. The consecutive responses along a specific path have time intervals in between, related to the responding speed of the memory modules. Therefore, a memory response will not be blocked at a single router stage for long, even with a lower priority. The exact amount of blocking along the response path depends on the number of responses ahead in transfer. This Meshed Bluetree architecture allows memory modules with different response time, potentially supporting mixed-criticality applications.

The term system cardinality is introduced to describe the configuration of the Meshed Bluetree architecture. It is expressed as the product of the number of clients N_μ and the number of memory modules N_D. For example, the system cardinality of the Meshed Bluetree in Figure 6.1 is 8×4. With an increasing system cardinality, the Meshed Bluetree scales with either higher router depth N_R or higher Bluetree depth N_β, indicating larger hardware consumption.

The number of components required to construct the Meshed Bluetree interconnect is analysed as follows, including Bluetree multiplexers, Bluetree

routers and Bluetree wires. For a single Bluetree memory architecture within Figure 6.1, the number of Bluetree multiplexers N_{mux} increases with the number of clients N_{μ}, considering the tree topology. For the Meshed Bluetree architecture, the total number of Bluetree multiplexers N_{mux} also increases with the number of memory modules N_D as follows. Taking Figure 6.1 as an example, the number of Bluetree multiplexers N_{mux} is equal to $(8-1)\times 4 = 28$.

$$N_{mux} = (N_{\mu} - 1) \times N_D \tag{6.1}$$

Similarly, the number of Bluetree routers N_{router} required to construct the tree-based router network increases with both the number of memory modules N_D and the number of clients N_{μ} as follows. Taking Figure 6.1 as an example, the number of Bluetree routers N_{router} is equal to $(4-1)\times 8 = 24$.

$$N_{router} = (N_D - 1) \times N_{\mu} \tag{6.2}$$

Bluetree wire refers to the data bus for the communication between any two Bluetree components within the interconnect, i.e., clients, memory modules, Bluetree multiplexers and Bluetree routers. The number of Bluetree wires N_{wire} is calculated as follows. For each Bluetree multiplexer, there is a Bluetree wire (pointing towards the memory direction), thus $(N_{\mu} - 1) \times N_D$ in the architecture. For each Bluetree router, there is a Bluetree wire (pointing towards the client direction), thus $(N_D - 1) \times N_{\mu}$ in the architecture. Then the rest $N_D \times N_{\mu}$ Bluetree wires connect Bluetree multiplexers and Bluetree routers. Taking Figure 6.1 as an example, the number of Bluetree wires N_{wire} is equal to $(8-1)\times 4 + (4\times 2 - 1)\times 8 = 84$.

$$N_{wire} = (N_{\mu} - 1) \times N_D + (N_D \times 2 - 1) \times N_{\mu} \tag{6.3}$$

The width of the data bus within the Meshed Bluetree interconnect depends on the communication packet format as shown in Figure 6.3. Developed from

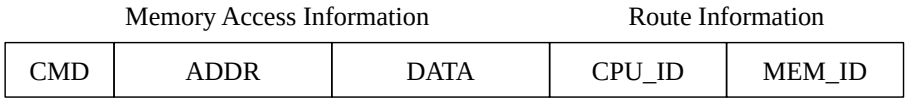

Figure 6.3. Meshed Bluetree Communication Packet Format

the Bluetree communication packet format, Figure 6.3 includes the memory access information and the route information in general. The memory access information is generated or received by the client or the root memory, for example, including the 1-bit command field CMD (i.e., the memory command type such as memory read or memory write), the 32-bit address field ADDR and the 32-bit data field DATA. In the memory request packet, CMD '0' indicates a read request, and CMD '1' indicates a write request. In the memory response packet, CMD '0' indicates a read response, and CMD '1' indicates a write acknowledgement.

The route information is required for the packet transfer across the interconnect, and it is used for for each distributed multiplexing stage to track or decide the route. The route information can include the 8-bit client identifier field CPU_ID and the 8-bit memory identifier field MEM_ID as an example. In this case, it can support a maximum Bluetree depth $N_\beta = 8$ and a maximum router depth $N_R = 8$. When a client issues a request, the corresponding CPU_ID is encoded by the local arbiter at each Bluetree multiplexer to track the route: left shift by 1 bit with '0' for the local high-priority path, or left shift 1 bit with '1' for the local low-priority path. CPU_ID is also used by the demultiplexer along the response path to decide the route back to the corresponding client, decoded by the right shift operation at each local stage. Similarly, MEM_ID is required by Bluetree routers.

In the above example, the total bit-width of a packet is 81, which is also the width of the data bus as well as the Bluetree multiplexers and the Bluetree

routers. It is to be noted that this design is reconfigurable and allows flexible extension, such as additional bits for the priority field of a priority-based arbitration in the route information. An extra interface is needed for the conversion of the packet format, for example, converting the packet format between the Meshed Bluetree interconnect and the AXI bus. In addition, this design is independent of memory addressing schemes.

In general, a single memory access across the Meshed Bluetree architecture experiences higher latency, due to the longer pipelined data path with the router network. However, simultaneous memory accesses can be processed by the paralleled memory modules concurrently, which effectively alleviates the contention over a shared memory module. In this way, latency of intensive memory accesses can be reduced, and thus the overall system performance is improved. Besides that, the Meshed Bluetree architecture supports memory isolation, potentially simplifying software or OS development for multi-core systems. This architecture also provides sufficient flexibility for mixed-criticality systems with diverse memory bandwidth or memory latency requirements.

6.2.1 Timing Behaviour Analysis

The timing behaviour analysis of memory accesses across the Meshed Bluetree architecture follows the generic analytical flow in Section 4.1 which is proposed for the *locally arbitrated* architecture. The reminder of this section focuses on the bound of the worst-case latency which is particularly important for the real-time applications. In general, the calculation on the worst-case latency t^{WC} of the memory access ω consists of the worst-case request path latency $t^{WC}_{RQ}(\omega)$, the root memory latency $t(D_j)$, and the worst-case response path latency $t^{WC}_{RS}(\omega)$ as follows. It is to be noted that *inter-path blocking*

and *intra-path blocking* potentially occur along both the request path and the response path in the Meshed Bluetree architecture.

$$t^{WC}(\omega) = t^{WC}_{RQ}(\omega) + t(D_j) + t^{WC}_{RS}(\omega). \tag{6.4}$$

For the request path, the employment of the router network R introduces *intra-path blocking* to the memory request ω before Bluetree stages. With the router depth N_R, the maximum blocking number in the router request path is equal to N_R, under the assumption that all the buffers are occupied at every pipelined stage. This blocking within the router network aggravate the *inter-path blocking* in the overlapped Bluetree request paths, which gets more severe closer to the root memory modules.

In this case, the maximum blocking number along the full request path $N^{WC}_{RQ}(\omega)$ can be statically determined applying calculations, and the worst-case assumption remains that the system is flooded by memory requests. Path $P_{(i,j)}$ gives *priority path* in a Bluetree-based architecture. The maximum blocking number that the request ω experiences across the request path $N^{WC}_{RQ}(\omega)$ can be calculated iteratively, starting with the value N_R from the router network R to the Bluetree root stage β_0 within the interconnect B_j. Finally, the maximum blocking number in the request path $N^{WC}_{RQ}(\omega)$ equals to the maximum blocking number accumulated to the root stage $N^{WC}_{RQ}(\beta_0)$.

With the increasing Bluetree local blocking factor α, the maximum blocking number in the request path $N^{WC}_{RQ}(\omega)$ decreases with more local high-priority tracks. Afterwards, the worst-case request path latency $t^{WC}(\omega)$ can be calculated with (4.4). According to the Bluetree arbitration design, when the blocking factor is set as $\alpha = 1$, the Bluetree interconnect provides relatively fair accesses for all requests regardless of the client index.

The analysis for blocking in the response path is different from that for the request path in the *locally arbitrated* architecture. According to the design of the Meshed Bluetree architecture, the consecutive memory responses are separated by certain time intervals, depending on the responding speed of the memory modules. Therefore, a response path will not be flooded by interfering responses. The maximum blocking that the memory access ω experiences in the response path is much less than that in the request path. In general, the response path is non-blocking within a Bluetree interconnect B_j, and the memory response can experience blocking in the router network R. The blocking analysis within the router network varies, depending on whether the root memory modules have homogeneous latency.

If all the paralleled memory modules have the identical root memory latency $t(D_j)$, there will be no blocking within the router network R. The memory requests from the same client are always issued successively. Therefore, there is only a single response arriving at each arbitration stage at a time, thus no *inter-path blocking*. If the root memory latency $t(D_j)$ varies on different memory modules over the paralleled Bluetree-based architectures, the *inter-path blocking* occurs in the router network R. A response only stalls in each pipelined stage for at most 1 clock cycle due to a single contending response from the other local path. Referring to the previous analysis, Bluetree router can locally employ either the round-robin arbitration scheme or the static-priority arbitration scheme. The worst case occurs when each local arbiter along the response path always harms the response flow, and the maximum blocking number under both schemes can be statically bounded applying calculations.

With the round-robin scheme at each router stage, a response can be blocked by at most a single response from the other local path. Considering the response intervals from the memory modules and the basic pipelined data path

latency (crossing routers and multiplexers without blocking), such *inter-path blocking* will not lead to any *intra-path blocking* of the responses behind. Therefore, the maximum blocking number in the response path is determined by the router depth N_R as $N_{RS}^{WC}(\omega) = N_R$. The worst-case response path latency $t_{RS}^{WC}(\omega)$ can be calculated as the sum of the basic pipelined path latency (through the router network and the Bluetree interconnect) plus blocking as follows.

$$\begin{aligned} t_{RS}^{WC}(\omega) &= N_\beta + N_R + N_{RS}^{WC}(\omega) \\ &= N_\beta + N_R + N_R \\ &= N_\beta + 2 \times N_R \end{aligned} \tag{6.5}$$

The local static-priority arbitration can lead to more *inter-path blocking*. With static priority at each router stage, the internal arbiter will always allow memory responses from a local path with higher priority to block the other local path. Following the architectural characteristics, the memory responses in a specific path are separated with intervals, and a single memory response experiences the basic pipelined data path latency. Therefore, a single response will not be stalled at a local router stage for long. The *inter-path blocking* does not cause any *intra-path blocking* to memory responses behind in the same path, as clients accept responses immediately, unlike memory modules which take $t(D_j)$ to respond to requests.

When a response ω crosses the leaf stage of the router network, there will be only a single interfering response from the other local path considering the memory responding intervals. Then the response ω experiences more *inter-path blocking* at the subsequent router stages closer to the client. The maximum blocking number in the response path can be bounded as $N_{RS}^{WC}(\omega) = N_D$, with the assumption that the response flow is always interfered. In this case, the worst-case response path latency $t_{RS}^{WC}(\omega)$ can be calculated as fol-

lows.

$$t_{RS}^{WC}(\omega) = N_\beta + N_R + N_D \tag{6.6}$$

Taking the static-priority arbitration in Bluetree router as an example, the worst-case latency t^{WC} of the memory access ω can be computed from the worst-case latency across the request path $t_{RQ}^{WC}(\omega)$ and the response path $t_{RS}^{WC}(\omega)$. The overall calculation can be reformed based on the above analysis as follows.

$$\begin{aligned} t^{WC}(\omega) &= t_{RQ}^{WC}(\omega) + t(D_j) + t_{RS}^{WC}(\omega) \\ &= N_{RQ}^{WC}(\beta_0) \times t(D_j) + t(D_j) + t_{RS}^{WC}(\omega) \\ &= (N_{RQ}^{WC}(\beta_0) + 1) \times t(D_j) + N_B + N_R + N_D. \end{aligned} \tag{6.7}$$

Analytical Results and Measured Results

This section compares the analytical worst-case memory access latency and the measured worst-case memory access latency across the Meshed Bluetree architecture under system cardinality 8×1, 8×2 and 8×4. Bluetree blocking factor is set as $\alpha = 1$, and each local arbiter is set with the same value in the entire Bluetree interconnect. Bluetree router is set with round-robin scheme. The latency of the paralleled memory modules are assumed as a constant 20 in clock cycles. In this case, the analytical results are calculated following the above analysis that there is no blocking within the router network.

The measured results are from hardware simulations. Traffic generators are employed as clients, and each traffic generator keeps pushing memory requests into its memory access path. In this case, the system can be flooded with memory requests (potentially pending), aiming towards that each memory request experiences its maximum blocking. The root memory modules are implemented using Bluespec BRAM package [107] with extra delays as a

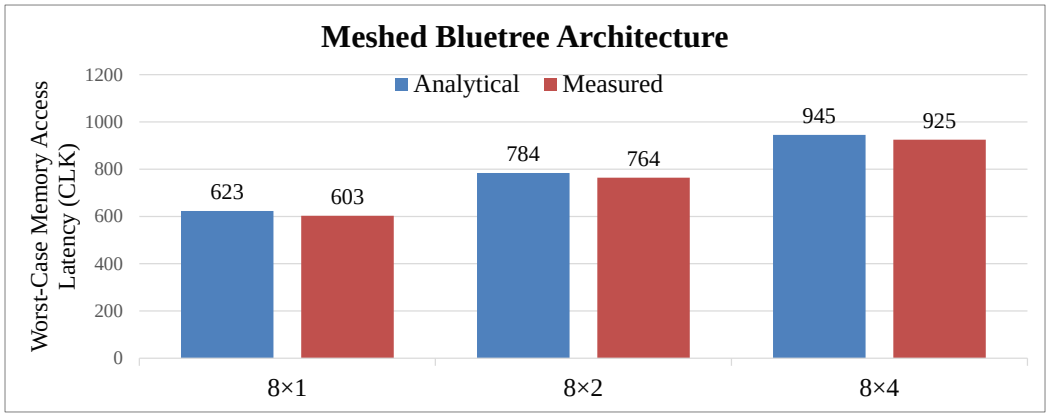

Figure 6.4. Worst-Case Memory Access Latency across Meshed Bluetree Architecture

constant 20 in clock cycles. The system is implemented using Bluespec System Verilog [107][108], with simulations running on BlueSim simulator [107][108]. This simulation measures memory access latency across the Meshed Bluetree architecture that the latency of each memory access is measured.

Similar to the analysis of Figure 4.4, the measured memory access latency gradually increases until at a constant value. Then the maximum measured constant is selected as the measured worst-case memory access latency across the relevant memory access path. In addition, the worst-case memory access latency is identical with blocking factor $\alpha = 1$ in each memory access path, both analytical and measured results. In this case, Figure 6.4 shows the comparison of analytical worst-case memory access latency and measured worst-case memory access latency over system cardinality with bar chart. The horizontal axis is for system cardinality 8×1, 8×2 and 8×4, and the vertical axis is for the worst-case results in clock cycles. It is observed that the measured results are smaller than the analytical results. It is to be noted that the 8×1 Meshed Bluetree architecture is the same as the conventional 8-client Bluetree-based architecture.

Discussion

Following the generic analytical flow to safely bound the worst case of the *locally arbitrated* architecture, the worst-case analysis inevitably produces pessimistic results across the Meshed Bluetree architecture. This can lead to conservative system design and resource dimensioning, as the memory access latency is the main part forming the overall program execution time. If the exact memory access profiles can be provided, the accurate memory access latency with no pessimism can be determined based on the detailed status of the memory flow and the local arbiter at every pipelined stage. With uncertainty on memory access profiles which is often the case in reality, the worst-case analysis reported in this section must be deployed for real-time applications even with pessimistic results.

The worst-case bound provided can also be tightened, e.g., by restricting the demand from processors with limit, and the discussion on the tightness also requires sufficiently representative memory workloads to be fair. As for practical applications, the number of memory requests issued to a system is limited and the memory access pattern is dependent on memory response. This potentially follows the workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$. In this case, the root queue modification with the root queue management can be appropriately employed to facilitate timing behaviour analysis of memory accesses across such architecture, which remains the future work.

In summary, compared with the conventional Bluetree-based architecture, the Meshed Bluetree architecture allows simultaneous memory accesses to be processed by the paralleled memory modules concurrently. It provides good average-case performance and guarantees the worst-case memory access latency which is particularly important for real-time applications.

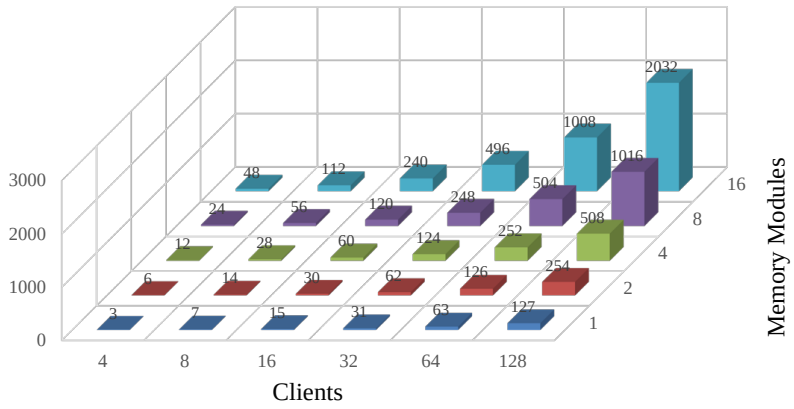

Figure 6.5. Hardware Consumption: Bluetree Multiplexer

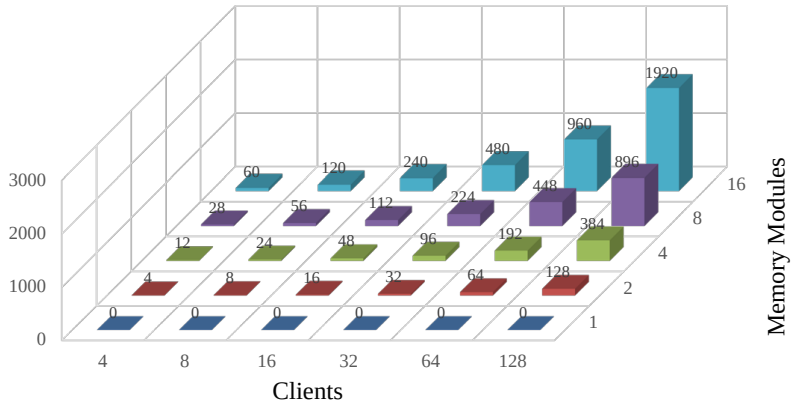

Figure 6.6. Hardware Consumption: Bluetree Router

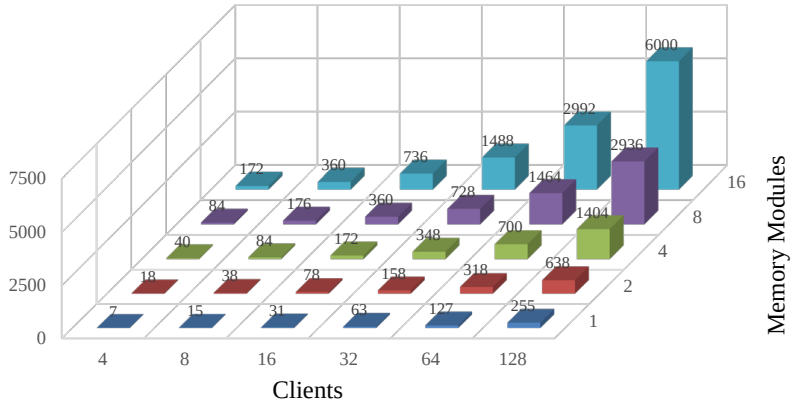

Figure 6.7. Hardware Consumption: Bluetree Wire

Table 6.1. Hardware Consumption at RTL Level

Component	LUT	Register	BRAM
Bluetree Multiplexer	105	269	0
Bluetree Router	88	251	0

6.3 Evaluation: Hardware Consumption

This section evaluates the hardware consumption of the Meshed Bluetree architecture. The numbers of components required to construct the Meshed Bluetree interconnect, including Bluetree multiplexers, Bluetree routers and Bluetree wires, are reported in Figure 6.5, Figure 6.6 and Figure 6.7, with the system cardinality increasing from 4×1 to 128×16. These results are calculated using (6.1), (6.2) and (6.3), which covers the entire interconnect. As shown in these figures, the numbers of components are proportional to the number of clients and memory modules, respectively.

The hardware consumption of Bluetree multiplexer and Bluetree router at the register-transfer level (RTL) is reported in Table 6.1, in terms of look-up tables (LUTs), registers, and BRAMs, which are the basic logic units on FPGA. Gate-level consumption, which depends on the fabrication technology, can be evaluated in the future work that more detailed information such as the width and length of wires, as well as the exact amount of area, is available. The design employs Bluetree multiplexers with the local blocking factor $\alpha = 1$ and the static priority-based arbitration within Bluetree routers. The entire Meshed Bluetree architecture is implemented using Bluespec System Verilog [107][108] and synthesised with Xilinx Vivado [111][112].

As shown in Table 6.1, a single Bluetree router consumes slightly fewer resources than a Bluetree multiplexer, and the relevant difference is mainly on

the design of internal arbiter. The BRAM consumption is 0 with the selected arbitration schemes. It is to be noted that this resource consumption is obtained from Vivado synthesis report, and the resource consumption can be much lower after optimisation. Based on Figure 6.5, Figure 6.6, Figure 6.7 and Table 6.1, the hardware consumption of the Meshed Bluetree interconnect increases linearly over the system cardinality.

6.4 Evaluation: Synthetic Memory Workloads

This section evaluates memory access latency across Meshed Bluetree architectures by FPGA experiments with synthetic memory workloads. Multiple experiments with varying experimental parameters has been conducted, and 2 groups are selected in this section to evaluate the timing behaviour of memory accesses across the Meshed Bluetree architecture under various system configurations.

In the following experiments, traffic generators are employed as clients with synthetic memory workloads which follow the workload pattern $N^{\mu}_{RQ}(\mu_i)$ and $T^{\mu}_{RQ}(\mu_i)$. Each traffic generator issues 100 memory requests totally. Outstanding request number is fixed as 2 $N^{\mu}_{RQ}(\mu_i) = 2$ as shown in Table 4.3, and request interval varies as $T^{\mu}_{RQ}(\mu_i) \in [1, 64]$ with randomly generated values in Appendix A.1. Bluetree blocking factor is set as $\alpha = 1$, and each local arbiter is set with the same value in the entire Bluetree interconnect to provide relatively fair accesses for all clients. Bluetree router is set with round-robin scheme. The Meshed Bluetree systems are synthesised using Xilinx Vivado [111][112] and implemented on Virtex-7 FPGA VC709 [114] with 100MHz of clock frequency.

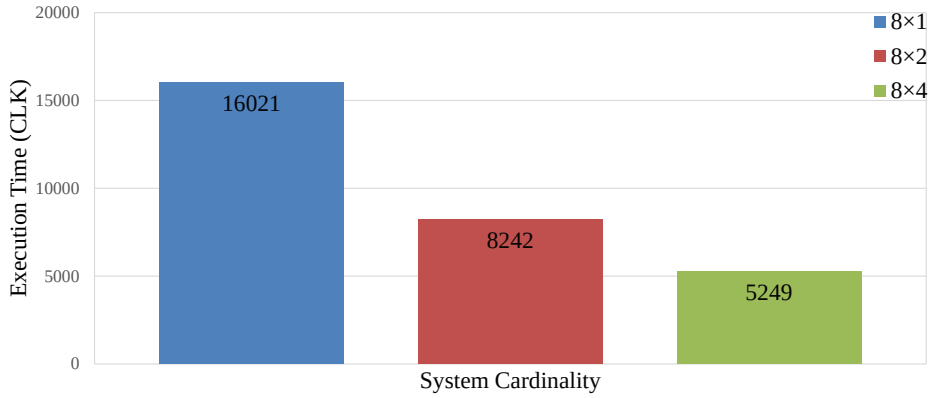

Figure 6.8. Execution Time with Multiple Homogeneous Memory Modules

The following experiments measure the execution time of each experiment which reflects the overall system performance. In addition, each experiment also measures memory access latency across the Meshed Bluetree architecture that the latency of each memory access is measured.

6.4.1 Memory Access Latency with Multiple Homogeneous Memory Modules

This experiment evaluates memory access latency across 8-client Meshed Bluetree architecture with multiple homogeneous memory modules, under system cardinality 8×1, 8×2, and 8×4. The memory modules are implemented based on FPGA BRAM [110] with additional 20 clock cycles. The memory accesses are randomly partitioned among these paralleled memory modules following the uniform distribution.

Figure 6.8 shows the execution time of experiments over system cardinality with bar chart. The horizontal axis is for system cardinality 8×1, 8×2 and 8×4, and the vertical axis is for the execution time in clock cycles. It is observed

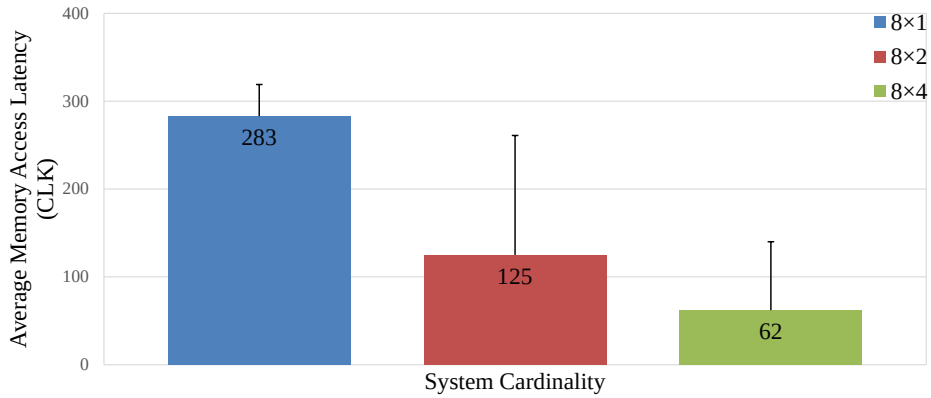

Figure 6.9. Average Memory Access Latency with Multiple Homogeneous Memory Modules

that the measured execution time is roughly reduced by half as the number of memory modules doubles. The reduction of the execution time is not exactly by half, instead slightly less than, because memory accesses experience longer data path latency across the Meshed Bluetree interconnect, according to the timing behaviour analysis. Although a portion of data path latency can be masked by the waiting for the root memory module, the latency of a single memory access increases. Besides that, following a randomised process, the memory accesses are not evenly partitioned to the paralleled architecture, neither the target memory modules nor the memory request issuing time instants. In this case, the contention over a heavier shared memory module increases the relevant memory access latency.

Latency of each memory access in this measurement is also analysed. Figure 6.9 shows the average memory access latency with the highest measured memory access latency over system cardinality. The horizontal axis is for system cardinality 8×1, 8×2 and 8×4, and the vertical axis is for average memory access latency with the whisker for the highest measured memory access latency, both in clock cycles. Although the blocking due to the shared resources still occurs, the simultaneous memory requests are partitioned into multiple

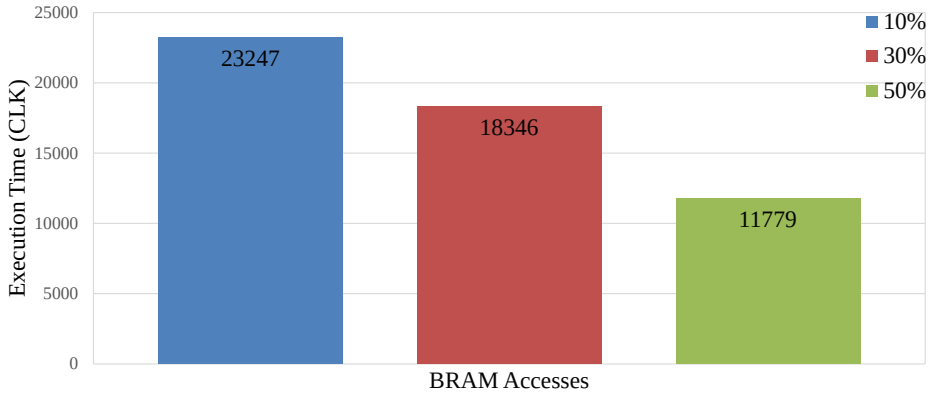

Figure 6.10. Execution Time with Mixed Memory Modules

memory modules in parallel through the Meshed Bluetree interconnect (although not evenly partitioned). This effectively alleviates the contention to a single memory module and thus reduces the average memory access latency as well as the highest measured memory access latency referring to the figure. It is also observed that the highest measured memory latency is 319, 246 and 117 in 8×1, 8×2 and 8×4 systems respectively. The results are much lower the analytical results which statically bounds the worst case without exact memory access profiles. It is to be noted that the Meshed Bluetree architecture is designed towards memory access latency reduction in the average case, whilst with analytical time-predictable behaviour and safe worst-case bound.

6.4.2 Memory Access Latency with Mixed Memory Modules

This experiment evaluates memory access latency across 8-client Meshed Bluetree architectures with mixed memory modules under system cardinality of 8×2, directly employing an FPGA BRAM module [110] and a VC709 DDR3 DRAM module [115]. In this experiment, the percentage of memory

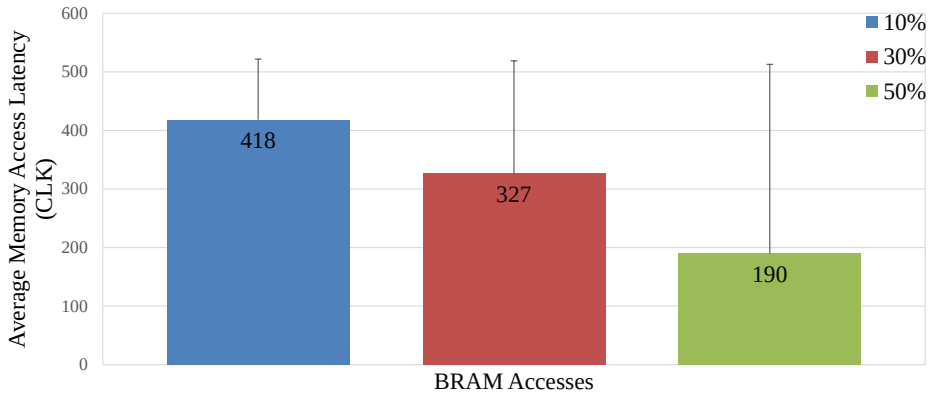

Figure 6.11. Average Memory Access Latency with Mixed Memory Modules

accesses to BRAM varies from 10%, 30%, to 50%, as the faster memory module tends to be in smaller size and with smaller memory address range. It is to be noted that memory write is not research focus, as write buffers are deployed in DRAM modules to expedite memory writes. By contrast, synthetic memory workloads focus on memory reads. In this case, traffic generator issues memory read requests with randomly generated memory addresses and thus leads to considerable DRAM latency on memory reads.

Figure 6.10 shows the execution time of experiments over the percentage of BRAM accesses with bar chart. The horizontal axis is for the increasing percentage of BRAM accesses 10%, 30% to 50%, and the vertical axis is for execution time in clock cycles. It is observed that the execution time is reduced with the increasing percentage of BRAM accesses. When this percentage increases from 10% to 30%, the execution time is reduced by approximately 20% due to the much faster response from the BRAM module. When the percentage further increases from 30% to 50%, the execution time drops even faster by approximately 40%. In this case, if the architecture scales with faster memory modules in parallel, the overall system performance can have more noticeable improvement.

Figure 6.11 shows the average memory access latency with the highest measured memory access latency over the percentage of BRAM accesses. It is observed that the average memory access latency is reasonably reduced with more accesses to the faster BRAM module. However, the highest measured memory access latency remains almost unchanged referring to the whiskers. As the memory accesses are randomly partitioned between the BRAM module and the DRAM module, the traffic generator can quickly issue next memory requests to the DRAM module after receiving the very fast response from the BRAM module. In this case, the contention to the shared slow DRAM module is not alleviated, and thus the highest measured memory access latency is not reduced.

6.5 Evaluation: Benchmarks

This section evaluates the overall system performance of Meshed Bluetree architectures under various system configurations with Mälardalen benchmark suite [116]. The experiments are based on 8-Microblaze [85] FPGA system running 8 calculation-intensive benchmarks of different functionality. Each Microblaze executes a benchmark as *cnt*, *compress*, *cover*, *expint*, *fdct*, *insertsort*, *jfdctint* and *qsort-exam*. It is to be noted that there is no local memory deployed, which makes the root memory modules with intensive memory workloads. Bluetree blocking factor is set as $\alpha = 1$, and each local arbiter is set with the same value in the entire interconnect to provide relatively fair accesses for each Microblaze. Bluetree router is set with round-robin scheme.

This experiment configures systems as 8×1 with a single DRAM module and 8×2 with 2 DRAM modules, employing VC709 DDR3 DRAM module [115] (and there are totally 2 DRAM modules on VC709). The 8×1 system with a

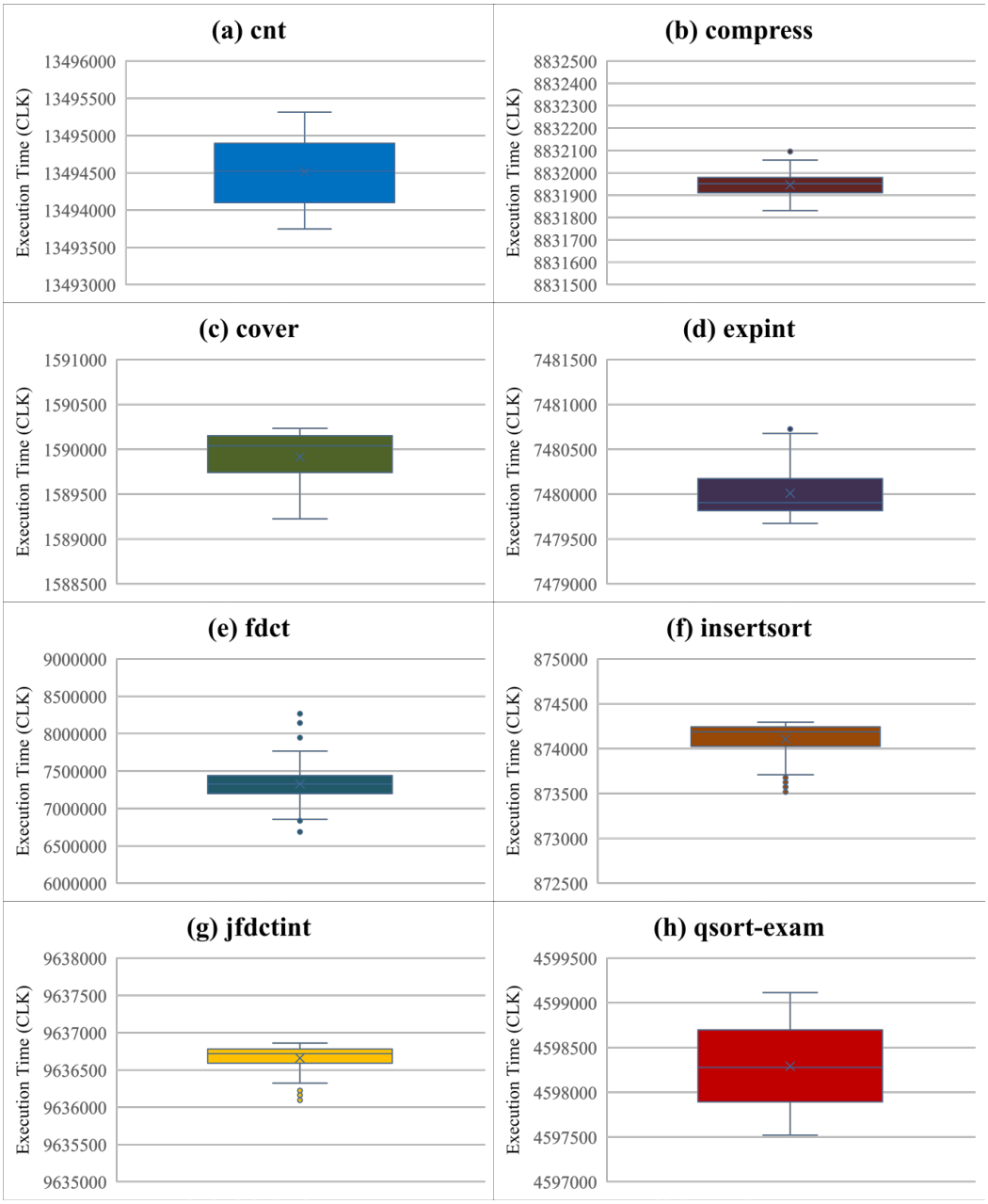

Figure 6.12. Boxplot of Execution Time in 8×1 Meshed Bluetree Architecture with Single DRAM Module

single DRAM module is denoted as Single DRAM for short. In this system, 8 Microblazes share a single DRAM module. The 8×2 system where memory accesses are partitioned as separate instruction DRAM accesses and data

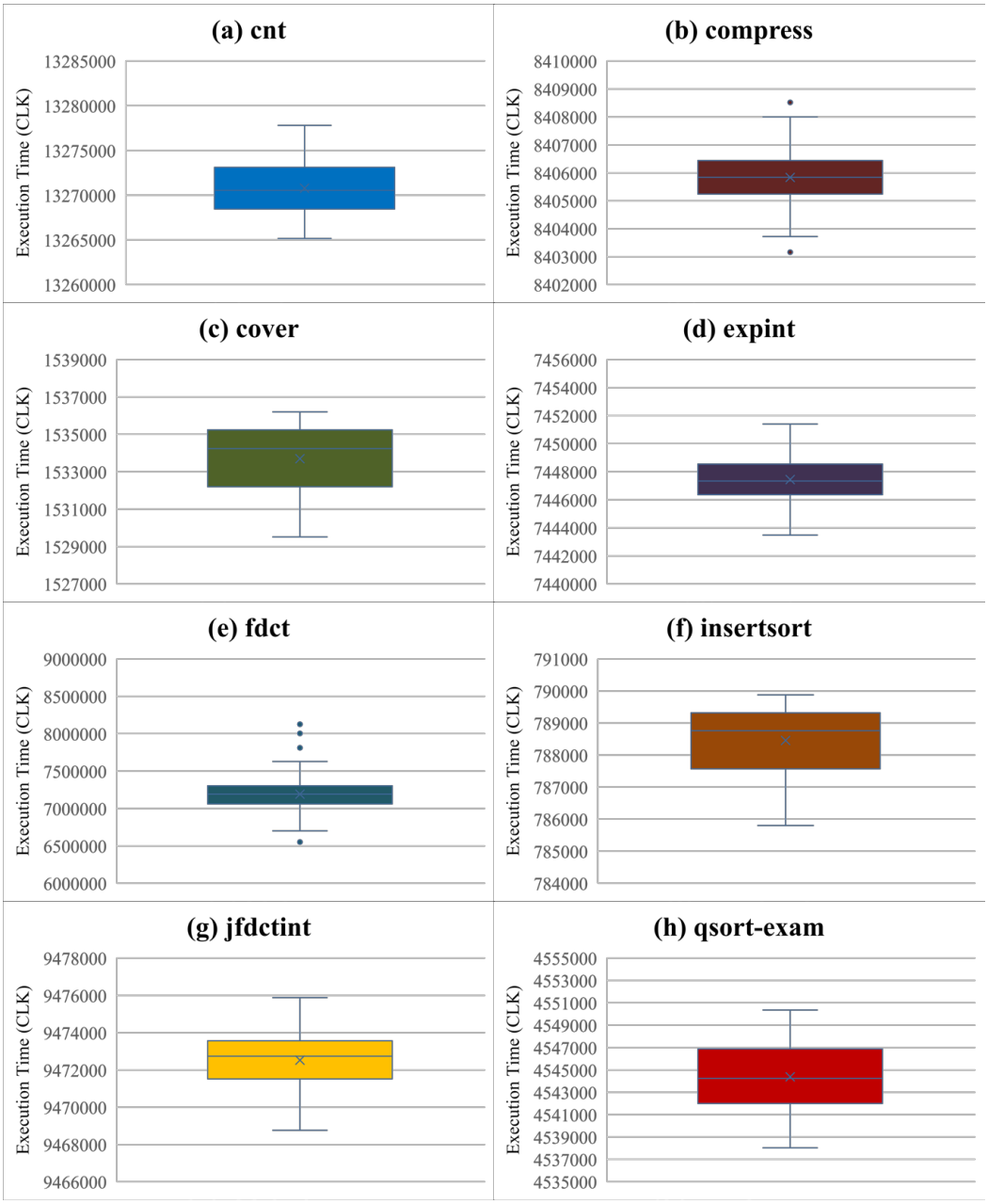

Figure 6.13. Boxplot of Execution Time in 8×2 Meshed Bluetree Architecture with Instruction DRAM Module and Data DRAM Module

DRAM accesses is denoted as Mixed DRAM. This system essentially employs a separate instruction DRAM module and a separate data DRAM module. The other 8×2 system where memory accesses are evenly partitioned to these

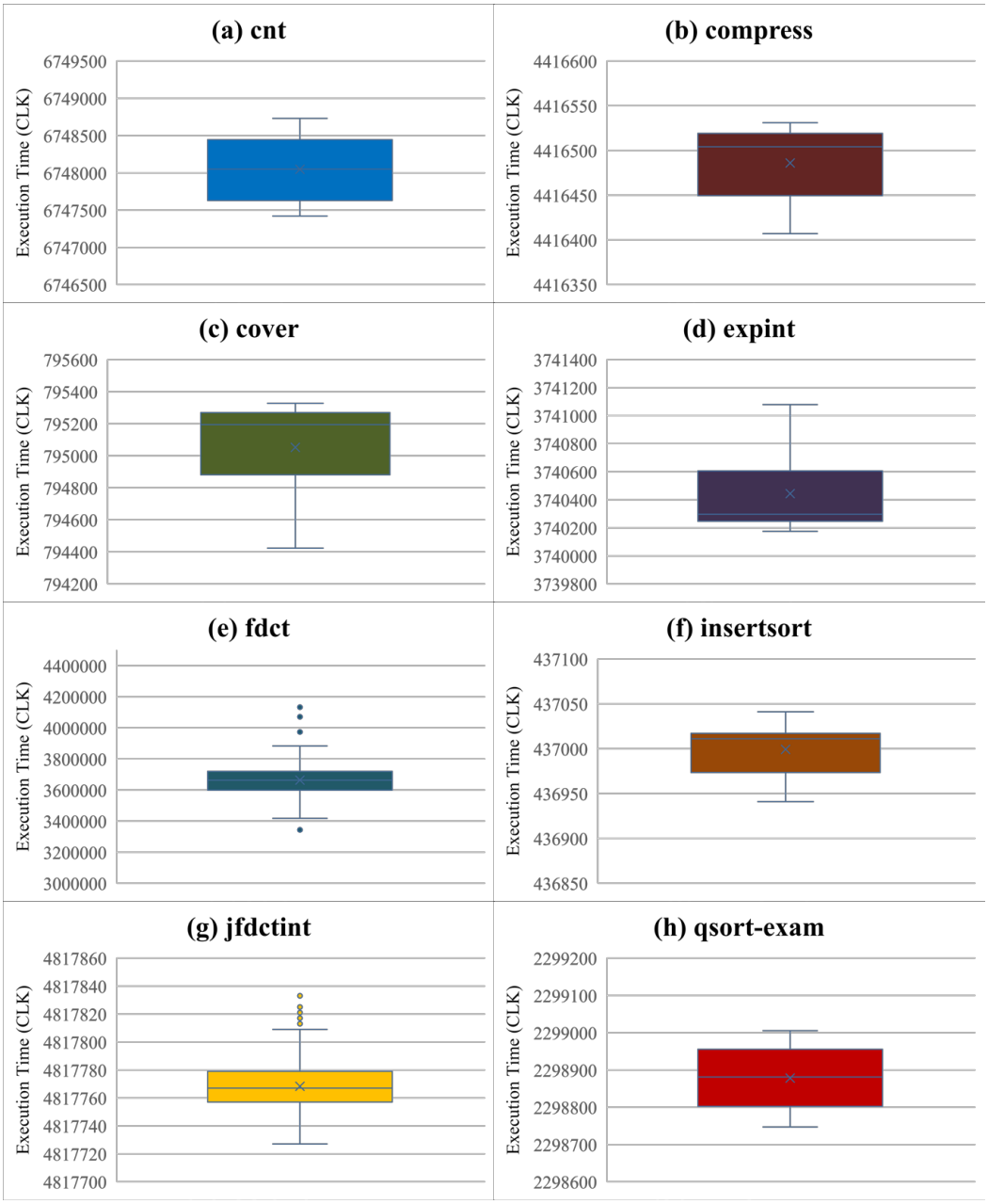

Figure 6.14. Boxplot of Execution Time in 8×2 Meshed Bluetree Architecture with Dual DRAM Modules

2 DRAM modules is denoted as Dual DRAM. In this system, every 4 Microblazes share a single DRAM module. It is to be noted that memory accesses issued by Microblazes include both memory reads and memory writes from

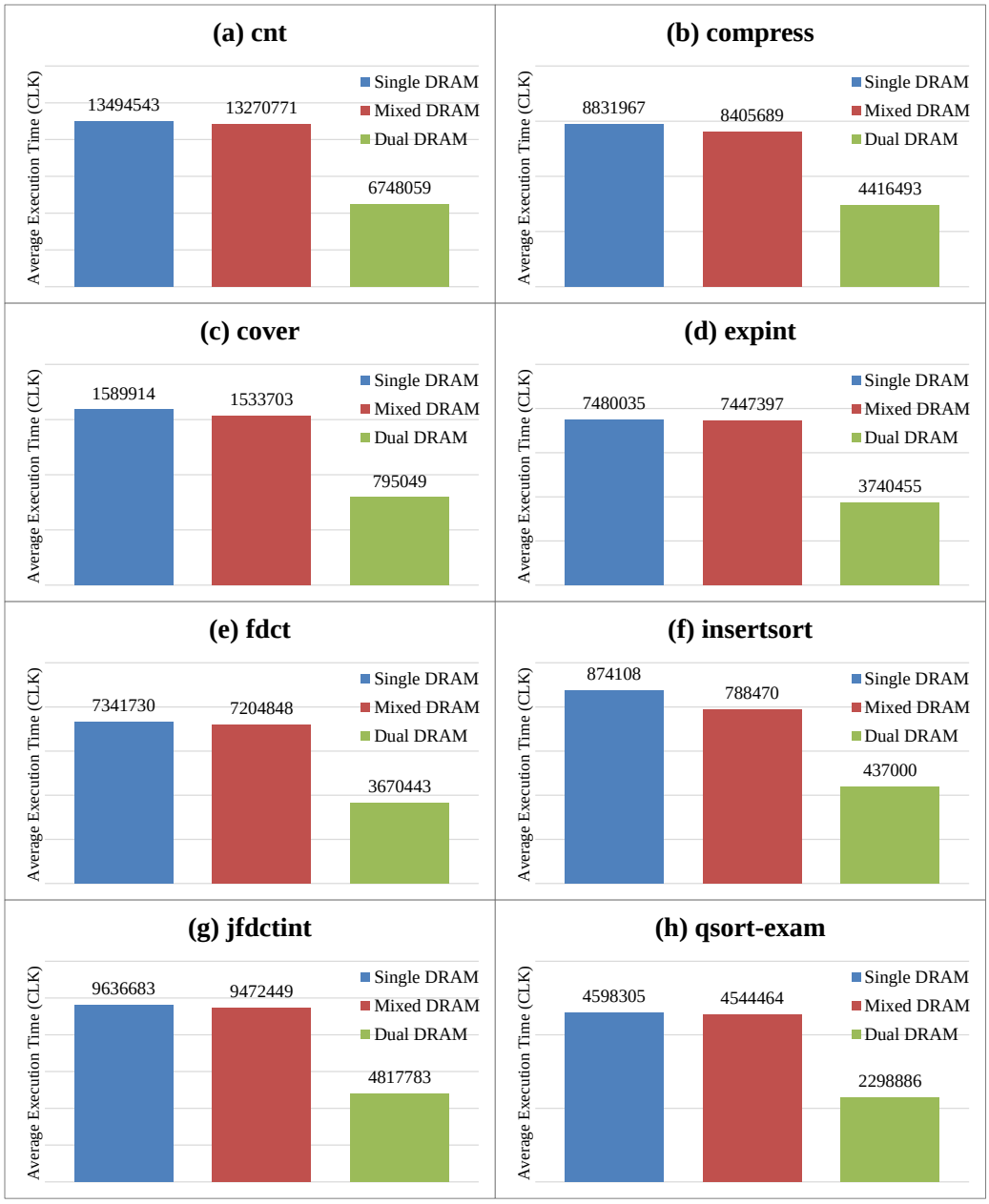

Figure 6.15. Average Execution Time in Meshed Bluetree Architectures

benchmarks in this experiment. These 3 Meshed Bluetree systems are synthesised using Xilinx Vivado [111][112] and implemented on Virtex-7 FPGA VC709 [114] with 100MHz of clock frequency. This experiment measures execution time of benchmarks over 100 repeated runs that the execution time of each run is measured.

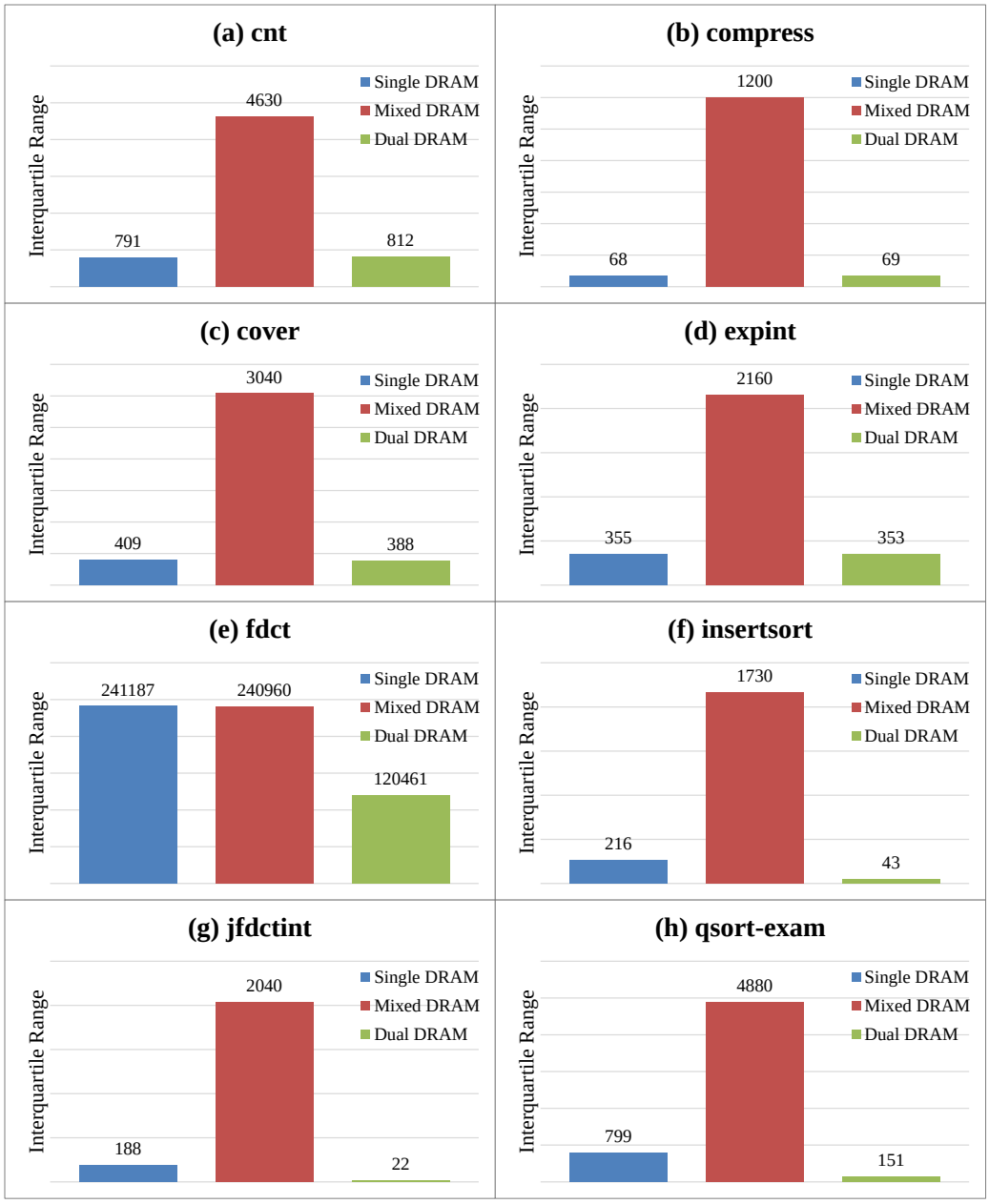

Figure 6.16. Interquartile Range of Execution Time in Meshed Bluetree Architectures

Figure 6.12, Figure 6.13 and Figure 6.14 show boxplot of execution time in this measurement. Each benchmark is plot separately, and the vertical axis is for execution time in clock cycles. It is to be noted that the scale of each vertical axis varies. In addition, the average execution time over system con-

figuration in this measurement is shown in Figure 6.15 with bar chart. The horizontal axis is for system configuration Single DRAM, Mixed DRAM and Dual DRAM, and the vertical axis is for the average execution time in clock cycles. Similarly, the interquartile range of execution time over system configuration in this measurement is shown in Figure 6.16 with bar chart. The horizontal axis is for system configuration Single DRAM, Mixed DRAM and Dual DRAM, and the vertical axis is for the interquartile range of execution time. Figure 6.15 and Figure 6.16 better compares the overall system performance of these Meshed Bluetree architectures over various system configurations within the same scale of vertical axis.

Compared with Single DRAM and Mixed DRAM, the average execution time slightly reduces for each benchmark. By contrast, the interquartile range of execution time dramatically increases, which partially reflects variation of execution time. Even though instruction accesses and data accesses are partitioned in Mixed DRAM, the number of outstanding memory requests to access a single memory module remains that all 8 Microblazes can simultaneously access a single DRAM module at a time. This leads to high memory access latency. On the other hand, when a single memory module is under high pressure, the other memory module is less accessed, due to the limited number of outstanding memory requests from these Microblazes. This leads to much lower memory access latency. In this case, wide variation of memory access latency leads to wide fluctuation of execution time, as memory access is the main part forming the overall benchmark execution. Following similar reduction trend of average execution time, the highest measured execution time is slightly reduced from Single DRAM to Mixed DRAM, for each benchmark, according to the accurate measured results. The system configuration Mixed DRAM fails to effectively alleviate the contention to a single shared memory.

By comparison, Dual DRAM has the lowest average execution time among these system configurations. Compared with Single DRAM and Dual DRAM, the average execution time is reduced roughly by half. As memory accesses are evenly partitioned to the paralleled memory modules for all benchmarks, the number of outstanding memory requests to access a single memory module is reduced to half, thus memory access latency dropping. In this case, execution time is reduced significantly. It is also observed that the interquartile range of execution time noticeably reduces for some benchmarks such as Figure 6.16 (g) *jfdctint* and Figure 6.16 (h) *qsort-exam*. However, the interquartile range of execution time remains almost unchanged for some benchmarks such as Figure 6.16 (a) *cnt* and Figure 6.16 (b) *compress*. This very much depends on behaviour of benchmarks. For example, due to the intrinsic variability of benchmark *fdct*, even the base memory address of arrays is unknown (which depends on function parameters). As a consequence, wide variation of execution time can be observed for benchmark *fdct* under each system configuration, referring to those large interquartile ranges in Figure 6.16 (e).

6.6 Summary and Discussion

This chapter proposes the Meshed Bluetree interconnect as the distributed time-predictable multi-memory interconnect for multi-core architectures. Constructed by the coupling of a router network and multiple *locally arbitrated* Bluetree-based architectures, the Meshed Bluetree architecture allows multiple processors to simultaneously access multiple memory modules with time-predictable behaviour. In general, a single memory access across the Meshed Bluetree architecture experiences higher latency due to the longer pipelined data path. However, simultaneous memory accesses can be processed by

the parallel memory modules concurrently, which effectively alleviates the contention over a single shared memory module as well as a single shared distributed memory interconnect. In this case, latency of intensive memory accesses can be reduced.

The hardware consumption to construct the Meshed Bluetree interconnect is reported, which increases linearly over the system cardinality. Experiments with FPGA implementations demonstrate the effectiveness of the proposed work. Experimental results from FPGA implementations with synthetic memory workloads demonstrate that with increasing number of paralleled memory modules employed, the average memory access latency is reduced with the same scale. Experimental results from FPGA implementations with Mälardalen benchmarks demonstrate that the Meshed Bluetree architecture can alleviate the contention to a single shared memory module. This reduces memory access latency and thus reduces overall execution time.

In summary, the Meshed Bluetree architecture allows multiple processors to simultaneously access multiple memory modules with time-predictable behaviour. It alleviates the contention to a single shared memory module as well as a single shared distributed memory interconnect. This effectively reduces memory access latency in the average case, contributing to solve the research question *Q3*.

The main contribution presented in this chapter is summarised as follows.

Meshed Bluetree is proposed as the distributed time-predictable multi-memory interconnect. Constructed by the coupling of a router network and multiple locally arbitrated Bluetree-based architectures in parallel, the Meshed Bluetree architecture allows multiple processors to simultaneously access multiple memory modules with time-predictable behaviour.

It to be noted that this architecture can be extended to other configurations than the *locally arbitrated* Bluetree design. Further improvement involves to investigate hardware-software co-design strategies for multi-core architectures with multi-memory interconnects.

Chapter 7

Concluding Remarks

This research explores timing behaviour of the multi-core architectures with shared distributed memory interconnects and improves the distributed time-predictable memory interconnect for multi-core architectures. This chapter draws the concluding remarks, and the reminder of this chapter is structured as follows. Section 7.1 summarises the proposed work in this research based on the given research questions. Section 7.2 revisits the main contributions which are summarised at the end of each chapter. Section 7.3 proposes the future work.

7.1 Research Summary

Chapter 4 analyses the timing behaviour of the multi-core architectures with shared distributed memory interconnects. First, Chapter 4 addresses the resource contention and the blocking effect across the data paths within shared memory multi-core architectures and proposes the generic analytical flow to

predict the memory access behaviour across the *locally arbitrated* architecture and statically bound the worst-case memory access latency when there is uncertainty on memory access profile. This contributes to solve the research question *Q1: Can analytical method predict timing behaviour of memory accesses and bound the worst-case memory access latency in multi-core architectures with shared distributed memory interconnects?* With the proposed analytical method, time predictability of the *locally arbitrated* architecture can be guaranteed. By contrast, the timing behaviour analysis of the *globally arbitrated* architecture reflects its global scheduling cycle. In addition, Chapter 4 also explores and analyses the timing behaviour of the *locally arbitrated* architecture and the *globally arbitrated* architecture by experiments with synthetic memory workloads.

Chapter 5 addresses the variation of memory access latency within the multi-core architectures with shared distributed memory interconnects. It proposes the root queue modification with the root queue management to the *locally arbitrated* architecture by employing and utilising an additional hardware queue between the distributed interconnect root and the shared memory module. With sufficient root queue size, the root queue modification smooths resource sharing across the *locally arbitrated* architecture and thus memory access latency only varies with varying memory workloads. Besides that, the root queue modification also facilitates the timing behaviour analysis that the worst-case memory access latency can be bounded applying calculations instead of deriving with exact memory access profiles. Based on the root queue modification, the root queue management is further proposed by utilising dummy packets to reduce variation of memory access latency. Experimental results from hardware simulations and FPGA implementations demonstrate the effectiveness of the proposed work. This contributes to solve the research question *Q2: Can multi-core architectures with shared distributed memory interconnects be modified at the hardware level to reduce variation of memory*

access latency? Applying the root queue modification with the root queue management, variation of memory access latency across the *locally arbitrated* architecture is effectively reduced. By contrast, the deployment of the *globally arbitrated* architecture can rely on effective analysis of accurate application behaviour to benefit specific applications.

Chapter 6 addresses the aggravated resource contention over the multi-core architectures with shared distributed memory interconnects due to increasing memory workloads. Based on the mesh-of-trees topology, it proposes Meshed Bluetree as the multi-memory interconnect for multi-core architectures. Constructed by the coupling of a router network and multiple Bluetree-based architectures in parallel, the Meshed Bluetree architecture allows multiple processors to simultaneously access multiple memory modules with time-predictable behaviour. This effectively reduces the contention to a single shared memory module as well as a single shared distributed memory interconnect. Experimental results from FPGA implementations with synthetic memory workloads and real-world benchmarks demonstrate the effectiveness of the proposed work. This contributes to solve the research question *Q3: Can multi-core architectures with shared distributed memory interconnects be improved by architectural enhancement for increasing memory workloads?* With the proposed distributed multi-memory interconnect, multiple processors can simultaneously access multiple memory modules with time-predictable behaviour. This potentially alleviates resource contention due to increasing memory workloads and thus reduces memory access latency in the average case. The *locally arbitrated* Bluetree and the Meshed Bluetree architecture are taken as examples, and this design can be extended to other configurations than the Bluetree-based architecture.

Based on the above analysis, the work presented in this thesis demonstrates the research hypothesis which is revisited as follows.

Distributed memory interconnect for multi-core architectures can be improved by architectural enhancement on hardware that the root queue modification with the root queue management reduces variation of memory access latency and the mesh-of-trees extension enables multiple processors to simultaneously access multiple memory modules, whilst guaranteeing the time-predictable behaviour.

7.2 Main Contributions

The summarised main contributions are revisited as follows.

- The generic analytical flow is proposed for time-predictable behaviour of memory accesses across multi-core architectures with *locally arbitrated* interconnects. Without exact memory access profiles, this static analysis can guarantee the safe worst-case bound for real-time applications applying calculations.
- The root queue modification with the root queue management is proposed for multi-core architectures with *locally arbitrated* interconnects that variation of memory access latency is effectively reduced and the timing behaviour analysis is also facilitated, contributing towards real-time multi-core systems.
- Meshed Bluetree is proposed as the distributed time-predictable multi-memory interconnect. Constructed by the coupling of a router network and multiple *locally arbitrated* Bluetree-based architectures in parallel, the Meshed Bluetree architecture allows multiple processors to simultaneously access multiple memory modules with time-predictable behaviour.

7.3 Future Work

The future work related to this research is proposed as follows.

A potential research is to investigate hardware-software integration or co-design strategies for multi-core architectures, such as investigating strategies that divide and map tasks to the paralleled memory modules in the Meshed Bluetree architecture or similar architectures. This potentially further improves the overall system performance, including execution time, power consumption, as well as reliability. In addition, more benchmarks, e.g., with more intensive demands or of mixed types, can also be adopted to evaluate the proposed design, and finer analysis can also be conducted on representative memory workload patterns.

Another potential research is to address the scalability of the shared distributed memory interconnect for multi-core architectures. With an expanding system configuration, i.e., an increasing number of processors, the contention over memory accesses keeps getting aggravated. Meshed Bluetree is proposed as an improvement method however with longer data path, thus higher memory access latency, and the responding speed of memory modules at the interconnect roots still limits the overall system performance. Further enhancement is to deploy memory modules at the distributed stages (potentially within the *locally arbitrated* design). This can raise research issues such as most beneficial memory types, appropriate memory size, as well as efficient memory management scheme especially for scratchpad memory (SPM).

Reference

[1] H. Wang, N. C. Audsley, and W. Chang. Addressing resource contention and timing predictability for multi-core architectures with shared memory interconnects. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 70–81, 2020.

[2] H. Wang, N. C. Audsley, X. S. Hu, and W. Chang. Meshed bluetree: Time-predictable multimemory interconnect for multicore architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3787–3798, 2020.

[3] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[4] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sep. 2006.

[5] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and Hon-Sum Philip Wong. Device scaling limits of si mosfets and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, March 2001.

[6] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 684–689, June 2001.

[7] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002.

[8] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[9] Hyung Gyu Lee, Naehyuck Chang, Umit Y. Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):23:1–23:20, May 2008.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[11] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[12] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[13] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.

[14] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-timeanalysis. *Real-Time Syst.*, 18(2/3):115–128, May 2000.

[15] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Fer-

dinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[16] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[17] Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 128–138, June 2000.

[19] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[20] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, pages 323–336, New York, NY, USA, 2016. ACM.

[21] M. Hassan. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 495–505, Dec 2018.

[22] Micron. *RLDRAM 3*.

[23] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable sdram memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, Sep. 2007.

[24] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time cmps. *IEEE Embedded Systems Letters*, 1(4):86–90, Dec 2009.

[25] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Sep. 2013.

[26] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[27] A. Milenkovic, M. Milenkovic, and N. Barnes. A performance evaluation of memory hierarchy in embedded systems. In *Proceedings of the 35th Southeastern Symposium on System Theory, 2003.*, pages 427–431, March 2003.

[28] Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, pages 213–232. Springer, 2003.

[29] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN 1998 Con-*

ference on Programming Language Design and Implementation, PLDI '98, pages 38–49, New York, NY, USA, 1998. ACM.

[30] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, April 1991.

[31] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, April 2001.

[32] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, June 1997.

[33] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez, and Víctor Viñals. Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems. *J. Syst. Archit.*, 57(7):695–706, August 2011.

[34] Alexis Arnaud and Isabelle Puaut. Dynamic instruction cache locking in hard real-time systems. In *In RTNS*, 2006.

[35] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1):4:1–4:38, December 2007.

[36] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 107–116, June 2000.

[37] Jason E. Miller and Anant Agarwal. Software-based instruction caching for embedded processors. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 293–302, New York, NY, USA, 2006. ACM.

[38] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[39] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[40] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[41] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 54–63, New York, NY, USA, 1991. ACM.

[42] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 176–186, New York, NY, USA, 1991. ACM.

[43] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual*

International Symposium on Microarchitecture, MICRO 25, pages 102–110, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[44] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 222–233, New York, NY, USA, 1996. ACM.

[45] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 111–121, Washington, DC, USA, 1999. IEEE Computer Society.

[46] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 279–290, New York, NY, USA, 2002. ACM.

[47] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 252–263, New York, NY, USA, 1997. ACM.

[48] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 96–, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th*

International Symposium on High Performance Computer Architecture, HPCA '07, pages 63–74, Washington, DC, USA, 2007. IEEE Computer Society.

[50] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[51] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[52] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st annual Design Automation Conference*, pages 238–243. ACM, 2004.

[53] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415. IEEE, 2002.

[54] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, 2000.

[55] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on Design and Test*, page 7. IEEE Computer Society, 1997.

[56] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10–pp. IEEE, 2005.

[57] Mahmut Kandemir, J Ramanujam, J Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conference*, pages 690–695. ACM, 2001.

[58] Guilin Chen, Ozcan Ozturk, M Kandemir, and M Karakoy. Dynamic scratch-pad memory management for irregular array access patterns. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 931–936. European Design and Automation Association, 2006.

[59] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109. ACM, 2004.

[60] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, Mahesh Balakrishnan, and Peter Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *System Synthesis, 2002. 15th International Symposium on*, pages 213–218. IEEE, 2002.

[61] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006.

[62] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330. ACM, 2006.

[63] Jack Whitham and Neil Audsley. Implementing time-predictable load and store operations. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 265–274. ACM, 2009.

[64] Jack Whitham and Neil Audsley. The scratchpad memory management unit for microblaze: Implementation, testing, and case study. *University of York, Tech. Rep. YCS-2009-439*, 2009.

[65] J. Whitham and N. Audsley. Mcgrep–a predictable architecture for embedded real-time systems. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 13–24, Dec 2006.

[66] J. Whitham and N. Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–316, April 2008.

[67] ARM. *AMBA 5 AHB Protocol Specification*, 2015.

[68] Xilinx. *AXI Interconnect*.

[69] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: An infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, 38(1):69–93, October 2004.

[70] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, September 2005.

[71] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, Oct 1991.

[72] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[73] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 3–14, Aug 2008.

[74] H. Shah, A. Raabe, and A. Knoll. Bounding wcet of applications using sdram with priority based budget scheduling in mpsocs. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 665–670, March 2012.

[75] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 248–259, June 2000.

[76] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[77] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[78] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[79] A. DeHon and R. Rubin. Design of fpga interconnect for multilevel metallization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1038–1050, 2004.

[80] A. O. Balkan, G. Qu, and U. Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 73–80, Sep. 2006.

[81] Igor Loi, Davide Rossi, Germain Haugou, Michael Gautschi, and Luca Benini. Exploring multi-banked shared-l1 program cache on ultra-low power, tightly coupled processor clusters. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 64:1–64:8, New York, NY, USA, 2015. ACM.

[82] A. O. Balkan, G. Qu, and U. Vishkin. Mesh-of-trees and alternative interconnection networks for single-chip parallelism. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(10):1419–1432, Oct 2009.

[83] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit. Evaluation of a connectionless noc for a real-time distributed shared memory many-core system. In *2012 15th Euromicro Conference on Digital System Design*, pages 727–730, Sep. 2012.

[84] ARM. *AMBA AXI and ACE Protocol Specification.*

[85] Xilinx. *MicroBlaze Processor Reference Guide.*

[86] Gary Plumbridge, Jack Whitham, and Neil Audsley. Blueshell: A platform for rapid prototyping of multiprocessor nocs and accelerators. *SIGARCH Comput. Archit. News*, 41(5):107–117, June 2014.

[87] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, and Alessandro Tocchi. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61, 04 2015.

[88] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A Time-Predictable Memory Network-on-Chip. In *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–62, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[89] M. Dev Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 193–198, March 2015.

[90] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, Feb 2017.

[91] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Syst.*, 52(3):272–322, May 2016.

[92] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12, July 2011.

[93] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core tdma resource arbitration delays. *Real-Time Syst.*, 50(2):185–229, March 2014.

[94] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[95] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

[96] Jamie Garside and Neil C. Audsley. Wcet preserving hardware prefetch for many-core real-time systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 193:193–193:202, New York, NY, USA, 2014. ACM.

[97] G. F. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, Oct 1985.

[98] Akbar Sharifi, Emre Kultursay, Mahmut Kandemir, and Chita R. Das. Addressing end-to-end memory access latency in noc-based multicores. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 294–304, Washington, DC, USA, 2012. IEEE Computer Society.

[99] I. Walter, I. Cidon, R. Ginosar, and A. Kolodny. Access regulation to hot-modules in wormhole nocs. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 137–148, May 2007.

[100] A. Hansson, M. Coenen, and K. Goossens. Channel trees: Reducing latency by sharing time slots in time-multiplexed networks on chip. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 149–154, Sep. 2007.

[101] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, pages 161–170, April 2008.

[102] N. Kavaldjiev, G. J. M. Smit, and P. G. Jansen. A virtual channel router for on-chip networks. In *IEEE International SOC Conference, 2004. Proceedings.*, pages 289–293, Sep. 2004.

[103] N. Kavaldjiev, G.J.M. Smit, P.G. Jansen, and P.T. Wolkotte. A virtual channel network-on-chip for gt and be traffic. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 6 pp.–, 2006.

[104] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual channels in networks on chip: Implementation and evaluation on hermes noc. In *2005 18th Symposium on Integrated Circuits and Systems Design*, pages 178–183, Sep. 2005.

[105] Neil Audsley. Memory architecturesfor noc-based real-time mixed criticality systems. *Proc. WMC, RTSS*, pages 37–42, 2013.

[106] G.G. Lee, C.P. Kruskal, and D.J. Kuck. On the effectiveness of combining in resolving hot spot contention. *Journal of Parallel and Distributed Computing*, 20(2):136–144, 1994.

[107] Bluespec. *Bluespec System Verilog Reference Guide.*

[108] Bluespec. https://bluespec.com/.

[109] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[110] Xilinx. *7 Series FPGAs Memory Resources*.

[111] Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[112] Xilinx. https://www.xilinx.com.

[113] ZedBoard. http://www.zedboard.org/product/zedboard.

[114] Xilinx Virtex-7 FPGA VC709. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html.

[115] Xilinx. *7 Series FPGAs Memory Interface Solutions*.

[116] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, July 2010. OCG.

Appendix A

Request Interval

FPGA experiments are conducted with synthetic memory workload. The path request interval $T_{RQ}(P_i)$ varies between two successive memory requests which employs the randomly generated values as follows.

A.1 Varying Request Interval [1, 64]

$T_{RQ}(P_0)$ = [17, 15, 28, 14, 39, 16, 53, 63, 50, 19, 25, 43, 51, 53, 38, 49, 25, 25, 29, 30, 51, 43, 55, 42, 47, 58, 15, 59, 9, 46, 52, 24, 22, 64, 58, 31, 14, 19, 2, 3, 49, 30, 22, 53, 28, 64, 30, 13, 28, 24, 18, 29, 46, 17, 4, 23, 40, 26, 12, 64, 60, 45, 4, 61, 56, 23, 13, 16, 8, 54, 26, 8, 29, 29, 62, 15, 4, 52, 29, 50, 63, 31, 48, 57, 5, 42, 39, 18, 17, 29, 23, 23, 61, 14, 12, 36, 41, 17, 18, 40];

$T_{RQ}(P_1)$ = [9, 51, 14, 20, 50, 9, 6, 22, 12, 42, 45, 11, 49, 24, 7, 14, 6, 51, 44, 42, 20, 47, 47, 17, 46, 13, 32, 1, 46, 17, 16, 59, 56, 5, 39, 41, 26, 37, 54, 55, 57, 28, 11, 3, 38, 46, 14, 7, 13, 40, 51, 2, 7, 10, 49, 56, 24, 42, 7, 35, 32, 29,

10, 19, 14, 45, 54, 30, 48, 13, 11, 46, 1, 44, 14, 57, 3, 41, 12, 46, 47, 22, 34, 10, 28, 61, 14, 5, 39, 43, 64, 1, 43, 57, 35, 40, 24, 12, 57, 27];

$T_{RQ}(P_2)$ = [41, 7, 31, 3, 57, 4, 19, 18, 61, 2, 30, 28, 15, 63, 60, 9, 64, 28, 49, 4, 50, 28, 36, 32, 21, 25, 57, 64, 54, 11, 44, 23, 62, 13, 43, 52, 54, 9, 61, 40, 21, 15, 19, 27, 20, 23, 4, 25, 28, 39, 41, 43, 25, 63, 8, 32, 26, 51, 2, 18, 36, 15, 21, 30, 26, 6, 10, 37, 51, 30, 30, 47, 26, 23, 35, 63, 24, 19, 38, 22, 36, 6, 46, 28, 11, 16, 59, 57, 31, 63, 34, 24, 41, 8, 14, 3, 50, 58, 29, 45];

$T_{RQ}(P_3)$ = [58, 63, 50, 47, 28, 50, 19, 44, 28, 18, 43, 62, 2, 28, 2, 8, 33, 5, 20, 34, 29, 20, 54, 31, 32, 27, 64, 25, 29, 22, 20, 35, 22, 58, 8, 51, 60, 10, 56, 29, 57, 38, 50, 63, 54, 7, 15, 50, 11, 7, 37, 15, 30, 62, 39, 50, 60, 37, 8, 40, 17, 35, 6, 23, 64, 57, 11, 62, 34, 29, 56, 11, 21, 19, 1, 53, 63, 54, 48, 33, 42, 41, 48, 64, 25, 15, 43, 7, 5, 21, 25, 56, 12, 27, 25, 16, 10, 43, 34, 11];

$T_{RQ}(P_4)$ = [29, 48, 59, 43, 41, 28, 18, 34, 1, 2, 1, 44, 47, 37, 47, 12, 28, 6, 64, 55, 55, 6, 51, 15, 47, 48, 54, 30, 43, 41, 47, 56, 1, 15, 22, 47, 16, 29, 64, 10, 53, 59, 62, 54, 15, 36, 63, 10, 39, 33, 2, 29, 1, 18, 57, 49, 39, 48, 22, 23, 26, 55, 28, 13, 26, 43, 4, 6, 64, 36, 11, 5, 19, 5, 33, 5, 41, 39, 25, 6, 10, 8, 38, 41, 42, 44, 29, 8, 15, 41, 58, 64, 15, 6, 49, 6, 7, 7, 60, 39];

$T_{RQ}(P_5)$ = [30, 8, 16, 25, 54, 50, 52, 35, 55, 14, 22, 3, 48, 11, 19, 37, 45, 39, 39, 40, 29, 47, 31, 27, 63, 56, 47, 30, 43, 12, 57, 64, 36, 53, 36, 43, 46, 16, 17, 27, 21, 6, 53, 46, 27, 3, 11, 6, 61, 33, 3, 62, 52, 64, 40, 61, 45, 13, 48, 39, 45, 19, 31, 9, 36, 44, 61, 60, 37, 42, 25, 35, 57, 6, 57, 10, 59, 24, 53, 22, 45, 11, 63, 18, 44, 29, 34, 13, 37, 17, 22, 39, 10, 1, 51, 52, 46, 25, 56, 58];

$T_{RQ}(P_6)$ = [3, 56, 16, 14, 64, 46, 18, 28, 32, 43, 62, 37, 1, 57, 4, 60, 48, 30, 15, 46, 40, 9, 41, 39, 4, 6, 44, 17, 40, 39, 13, 8, 15, 21, 62, 26, 16, 23, 12, 17, 58, 35, 8, 20, 23, 64, 26, 32, 50, 15, 23, 21, 19, 42, 17, 33, 9, 8, 57, 50, 19, 41,

49, 61, 32, 39, 13, 18, 8, 64, 41, 10, 19, 30, 41, 48, 7, 21, 30, 30, 25, 64, 59, 20, 42, 48, 29, 36, 54, 5, 43, 31, 30, 42, 25, 28, 64, 51, 24, 61];

$T_{RQ}(P_7)$ = [56, 14, 48, 32, 31, 49, 34, 47, 48, 61, 22, 17, 13, 1, 53, 47, 57, 61, 6, 47, 12, 35, 23, 35, 53, 7, 35, 47, 38, 26, 7, 9, 45, 8, 11, 21, 22, 39, 48, 32, 26, 52, 20, 46, 18, 57, 62, 33, 33, 49, 55, 8, 22, 46, 33, 13, 44, 22, 56, 39, 57, 3, 13, 32, 41, 13, 30, 37, 43, 4, 21, 53, 64, 63, 38, 49, 60, 57, 58, 51, 1, 49, 50, 28, 50, 1, 59, 60, 44, 46, 28, 49, 34, 20, 4, 63, 51, 54, 59, 4].

A.2 Varying Request Interval [1, 256]

$T_{RQ}(P_0)$ = [52, 91, 195, 84, 219, 216, 125, 188, 109, 245, 238, 94, 221, 219, 252, 120, 209, 125, 3, 122, 107, 227, 132, 64, 148, 110, 196, 207, 238, 37, 102, 44, 79, 157, 17, 245, 245, 115, 251, 176, 131, 197, 35, 1, 167, 187, 85, 191, 44, 169, 145, 106, 100, 42, 17, 30, 8, 113, 87, 88, 242, 165, 146, 83, 101, 178, 27, 36, 192, 20, 138, 16, 79, 189, 84, 49, 65, 118, 132, 34, 182, 91, 47, 218, 100, 26, 130, 109, 157, 243, 67, 238, 212, 22, 188, 76, 244, 63, 63, 37];

$T_{RQ}(P_1)$ = [152, 179, 3, 58, 78, 206, 55, 3, 244, 141, 108, 81, 150, 211, 95, 168, 34, 63, 199, 210, 58, 17, 237, 167, 224, 67, 194, 253, 58, 171, 97, 189, 33, 104, 156, 40, 75, 54, 161, 106, 193, 24, 165, 71, 75, 42, 5, 159, 1, 206, 119, 148, 55, 128, 43, 142, 153, 198, 181, 23, 164, 111, 35, 18, 76, 52, 244, 47, 187, 77, 178, 93, 59, 236, 170, 197, 154, 243, 149, 146, 160, 202, 164, 206, 196, 102, 112, 199, 211, 155, 198, 75, 216, 247, 23, 142, 200, 105, 249, 53];

$T_{RQ}(P_2)$ = [183, 198, 23, 163, 68, 130, 179, 2, 231, 94, 152, 241, 115, 79, 145, 191, 236, 61, 2, 187, 105, 247, 10, 155, 199, 187, 41, 77, 108, 199, 209, 39, 209, 12, 227, 7, 2, 66, 163, 200, 120, 171, 67, 44, 154, 31, 150, 14, 231, 51, 46, 140, 196, 221, 83, 46, 218, 178, 69, 150, 59, 254, 215, 217, 167, 212, 147, 44,

94, 62, 187, 219, 237, 94, 158, 48, 108, 170, 94, 195, 189, 49, 190, 173, 248, 256, 180, 254, 92, 31, 89, 253, 181, 105, 91, 61, 206, 130, 86, 214];

$T_{RQ}(P_3)$ = [220, 240, 53, 196, 19, 54, 194, 93, 55, 242, 78, 130, 185, 133, 244, 123, 73, 237, 67, 96, 193, 56, 179, 245, 112, 243, 23, 196, 30, 165, 191, 51, 242, 200, 77, 41, 180, 237, 235, 4, 211, 129, 191, 55, 201, 210, 15, 8, 99, 165, 168, 212, 16, 109, 52, 153, 162, 151, 21, 30, 98, 107, 226, 32, 248, 169, 147, 74, 222, 219, 186, 251, 27, 188, 118, 82, 18, 136, 84, 142, 166, 81, 109, 14, 154, 251, 157, 84, 72, 40, 102, 71, 156, 132, 7, 64, 181, 178, 41, 158];

$T_{RQ}(P_4)$ = [16, 113, 12, 161, 247, 10, 147, 252, 195, 125, 219, 136, 256, 75, 240, 5, 68, 13, 53, 5, 232, 133, 243, 51, 155, 229, 207, 1, 75, 136, 59, 23, 168, 157, 125, 91, 99, 86, 64, 58, 233, 188, 163, 240, 59, 88, 22, 245, 98, 8, 247, 134, 135, 134, 85, 178, 176, 252, 78, 242, 233, 20, 105, 58, 98, 229, 114, 80, 27, 60, 130, 100, 189, 21, 98, 75, 250, 202, 83, 58, 151, 32, 127, 156, 209, 117, 178, 162, 81, 76, 93, 113, 100, 153, 110, 244, 199, 77, 75, 67];

$T_{RQ}(P_5)$ = [21, 15, 98, 157, 14, 110, 181, 247, 196, 39, 213, 47, 65, 111, 194, 178, 42, 28, 132, 90, 250, 89, 207, 228, 232, 155, 220, 85, 171, 179, 150, 84, 142, 187, 89, 49, 89, 146, 226, 32, 239, 69, 158, 221, 103, 253, 24, 115, 208, 35, 167, 163, 145, 64, 190, 197, 233, 7, 247, 101, 255, 240, 117, 18, 154, 104, 120, 107, 165, 179, 47, 10, 36, 12, 17, 177, 116, 128, 123, 212, 172, 127, 199, 6, 117, 53, 103, 55, 4, 86, 130, 238, 250, 135, 247, 214, 124, 122, 143, 195];

$T_{RQ}(P_6)$ = [139, 60, 140, 171, 133, 36, 37, 66, 88, 220, 56, 98, 130, 215, 146, 146, 21, 153, 46, 72, 152, 119, 199, 26, 39, 61, 231, 203, 239, 118, 124, 131, 48, 49, 167, 229, 52, 189, 236, 102, 109, 71, 89, 18, 242, 14, 184, 77, 49, 202, 180, 227, 90, 149, 208, 163, 211, 243, 120, 33, 246, 244, 158, 226, 100, 179, 192, 18, 255, 124, 174, 184, 247, 178, 184, 42, 44, 18, 66, 227, 183, 186, 60, 103, 169, 180, 247, 213, 208, 139, 153, 148, 97, 2, 41, 183, 220, 226, 32, 80];

$T_{RQ}(P_7)$ = [105, 230, 233, 169, 229, 17, 217, 161, 31, 161, 62, 159, 172, 213, 139, 182, 231, 24, 18, 119, 239, 131, 222, 95, 143, 213, 19, 85, 49, 121, 233, 21, 108, 163, 120, 80, 23, 245, 53, 205, 181, 24, 17, 225, 212, 27, 189, 135, 1, 39, 175, 142, 215, 167, 159, 195, 137, 56, 166, 251, 232, 43, 34, 99, 74, 81, 47, 228, 4, 191, 7, 100, 146, 36, 248, 8, 216, 7, 243, 81, 121, 255, 115, 250, 162, 254, 62, 227, 45, 50, 236, 240, 99, 147, 188, 151, 45, 65, 110, 172].