Scalable and Bandwidth-Efficient Memory Subsystem Design for Real-Time Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 7 september 2015 om 16:00 uur

door

Manil Dev Gomony

geboren te Trivandrum, India

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. A.C.P.M. Backx
1^{e} promotor:	prof.dr. K. Goossens
copromotor:	dr. B. Akesson (Czech Technical University in Prague)
leden:	prof. N. Audsley (University of York)
	Prof. Dr.-Ing. habil. M. Hübner (Ruhr-Universität Bochum)
	dr.ir. L. Jóźwiak
	prof.dr.ir. C.H. van Berkel

Scalable and Bandwidth-Efficient Memory Subsystem Design for Real-Time Systems

Manil Dev Gomony

Doctoral Committee:

prof.dr. K. Goossens	Eindhoven University of Technology *promotor*
dr. B. Akesson	Czech Technical University in Prague *co-supervisor*
prof.dr.ir. A.C.P.M. Backx	Eindhoven University of Technology *chairman*
prof. N. Audsley	University of York
Prof. Dr.-Ing. habil. M. Hübner	Ruhr-Universität Bochum
dr.ir. L. Jóźwiak	Eindhoven University of Technology
prof.dr.ir. C.H. van Berkel	Eindhoven University of Technology

Cover design by Deepti Thorat

Printed in The Netherlands

A catalogue record is available from the Eindhoven University of Technology Library. ISBN: 978-90-386-3897-3

Acknowledgements

I would like to express my sincere gratitude to all the people who have contributed to this thesis in different ways. First, I thank Prof. Kees Goossens, Prof. Kees van Berkel and Prof. Henk Corporaal for selecting me as one of the PhD candidates in the COBRA project. I am extremely grateful to Prof. Kees Goossens for being my first promotor and for his understanding, wisdom, enthusiasm, and encouragement that has pushed me to achieve more than I thought I could do. I have always enjoyed the regular meetings with Kees which were full of positive energy and resulted in effective decision making. I am extremely thankful to my co-supervisor Dr. Benny Akesson for the generous guidance and support that he has provided me all the way along this journey. I greatly appreciate his involvement in detailed technical discussions and critical review of research results with an eye for detail that helped me to make my work more concrete.

I would like to thank Christian Weis and Prof. Norbert Wehn of University of Kaiserslautern and Jamie Garside and Prof. Neil Audsley of University of York for the valuable collaborations that resulted in a couple of publications in a top conference. Especially, I thank Christian for his valuable contributions on 3D-stacked DRAMs and Jamie for the time and effort in implementing and evaluating our proposed GAMT architecture on an FPGA. My special thanks to Prof. Neil Audsley, Prof. Michael Hübner, Prof. Kees Van Berkel, Dr. Lech Jóźwiak, and Prof. Ton Backx for being the members of the Doctoral Committee and for their time and effort in reviewing this manuscript.

I was fortunate to be a part of the Memory team, where I learned more about memory subsystems than anywhere else. I enjoyed being a part of the discussions at the regular memory meetings where I got critical feedback on my research as well. Many thanks to the members of memory team Karthik Chandrasekar, Sven Goossens, Yonghui Li, Tim Kouters, and Jasper Kuijsten for their continuous enthusiastic support. I thank all the members of CompSOC team, especially, Andrew Nelson, Ashkan Beyranvand Nejad, Martijn Koedam, Shubhendu Sinha, Gabriela Breaban, Rachana Kumar, Reinier van Kampenhout, Juan Valencia, and Rasool Tavakoli for all the discussions that helped me gain knowledge about various aspects of embedded system design and for all the fun-filled activities.

I am very greatful for the friendship of all members of the Electronic Systems group. Many thanks to Cedric Nugteren, Francesco Comaschi, Shreya Adyanthaya, Umar Waqas, Hamid Reza Pourshaghaghi, Rosilde Corvino, Erkan Diken, Maurice Peemen, Marcel Steine, Gert-Jan van den Braak, Roel Jordans, Luc Vosters, Marc Geilen, Dip Goswami and Hailong Jiao. My sincere thanks to our group secretaries Rian van Gaalen, Marja de Mol, and Margot Gordon for their special care and making my stay in the office very comfortable and pleasant.

I am deeply thankful to Firew Siyoum, Tiblets Demewez, Davit Mirzoyan, Karthik Chandrasekar, Massimiliano de Leoni and Sahel Abdinia for those great moments that we spent together outside work. My heartily thanks to Damiano Scanferla for introducing me to the world of PDEngs Manolis Chrysallos, Kyveli Kompatsiari, Rimma Dzhusupova, Aleksey Dubok, Bedilu Befekadu, Maxim Bogomolov and many others who made my stay in Eindhoven filled with lots of fun and joy. Many thanks to Bipin Balakrishnan and Tomy Varghese for always giving me reasons to cheer.

I am forever indebted to my parents for their continuous encouragement and support throughout my life. Finally and most importantly, I thank Deepti Thorat for all her faith in me and for being such a loving wife.

Abstract

Scalable and Bandwidth-Efficient Memory Subsystem Design for Real-Time Systems

In heterogeneous multi-processor platforms for real-time systems, Dynamic Random Access Memory (DRAM) is typically used as a shared resource to reduce cost and enable communication between memory clients, i.e. the processing elements. Since multiple applications with firm real-time requirements run concurrently in such platforms, the memory clients impose strict worst-case requirements on main memory performance in terms of bandwidth and/or latency. These requirements must be guaranteed at design time to reduce the verification effort. This is made possible using a real-time memory subsystem consisting of a real-time memory controller and a memory interconnect in front of it that multiplexes requests arriving from different clients. Existing real-time memory controllers bound the execution time of a memory request by fixing the memory access parameters, such as burst size and page policy, at design time. To bound the response time, predictable arbitration policies, such as Time Division Multiplexing (TDM) and Round-Robin (RR), are employed in the memory interconnect. The performance of real-time memory subsystems can be analyzed using formal performance analysis based on e.g., such as network calculus and data-flow analysis.

To meet the ever increasing demand for memory bandwidth with more applications being integrated into multi-core platforms, the maximum clock speeds of memory devices were increased by over a factor of two for every memory generation with the help of technology node scaling. Moreover, memory devices with multiple memory channels (multi-channel memories) and wider interfaces, such as Wide IO, were introduced, targeting battery-operated mobile devices. To support the upcoming memory generations in multi-processor platforms with increasing number of clients, scalable memory subsystems are essential. However, existing bus-based memory interconnects with centralized implementation of predictable arbitration policies are not scalable in terms of clock frequency, and current distributed interconnects either suffer from poor performance in terms of area, power

consumption and latency, or do not provide differential treatment to the memory clients according to their diverse real-time requirements. Also, there is currently no real-time memory controller for the efficient utilization of multi-channel memories.

Structured design methodologies are essential for cost-efficient design of memory subsystems in real-time systems since the system designer needs to make design choices of several system-level parameters, such as selecting the memory type, memory controller configuration and mapping of memory clients to memory channels in a multi-channel memory. Selection of these parameters need to be done carefully as it impacts the efficient use of the memory bandwidth. However, currently there is no structured methodology for bandwidth-efficient design of memory subsystems in real-time systems.

This thesis addresses the key issues with the current memory subsystems, i.e. non-scalable architectures in terms of clock frequency and number of memory channels and the lack of design methodologies for cost-efficient design with the following four main contributions: 1) A generic, globally arbitrated memory tree (GAMT) architecture for distributed implementation of five different predictable arbitration policies. GAMT runs four times faster compared to existing centralized memory interconnects and provides better performance in terms of area/bandwidth and power/bandwidth trade-offs. 2) A coupled memory interconnect (CMI) architecture that allows coupling of any existing globally arbitrated memory interconnect, such as TDM Network-on-Chips (NoC) or GAMT, with the memory controller. CMI provides lower area usage, power consumption and worst-case latency compared to decoupled architectures. 3) A configurable real-time multi-channel memory controller (MCMC) with a novel method for logical-to-physical address translation that allows memory requests of clients to be interleaved across memory channels with different interleaving granularities. 4) An automated design-flow for the design of bandwidth-efficient memory subsystems in real-time systems, which performs memory type selection, memory controller configurations and mapping of memory clients to memory channels, while considering the real-time requirements of the clients. We demonstrate the effectiveness of our proposed design-flow using a case-study of designing the memory subsystem in a HD video processing system.

Table of contents

Chapter 1

Introduction

Integrated circuits are used to perform a wide variety of tasks in almost all present-day electronic systems. Today, with over one billion CMOS transistors in an integrated circuit [41], multi-processor platforms with multiple cores interconnected using an on-chip communication protocol are available in the market [38, 89, 42]. Such platforms run multiple applications at the same time, offering high performance at very low power consumption compared to traditional multi-chip platforms. In contemporary multi-processor platforms, main memory (off-chip DRAM) is typically a shared resource for cost reasons and to enable communication between the between the processing elements [80, 128, 89]. Multi-processor platforms for real-time systems run a mix of applications with different real-time requirements on main memory performance in terms of bandwidth and/or latency [129, 123]. However, memory resource sharing causes interference between the applications that may lead to violation of their real-time requirements. These real-time requirements must be guaranteed at design time and efforts must be made to minimize the *time to market*. This is made possible using *real-time memory subsystems* [104, 13, 108] and employing predictable arbitration policies for resource sharing in the memory interconnect.

To meet the memory performance demands in future systems with a large number of processing elements, which we refer to as *memory clients*, faster memories and memories with *multiple memory channels* are introduced [9]. Existing memory interconnects are not scalable with the increasing number of clients and cannot be run at higher clock frequencies. Moreover, selecting the right arbitration policy in the interconnect according to the diverse and dynamic client requirements on memory bandwidth and latency in re-usable platforms requires a generic re-configurable architecture supporting different arbitration policies.

There is currently no re-configurable architecture supporting different arbitration policies. Existing memory controllers either interleave all memory requests of all clients across all memory channels or do not interleave at all. However, a multi-channel memory controller that interleaves memory requests across the memory channels according to the client requirements is essential for the efficient utilization of multi-channel memories. Additionally, with the increasing complexity of future systems, design methodologies are essential for a faster and efficient design of systems [3]. However, existing design methodologies either does not support configuration of a multi-channel memory subsystem or do not provide performance guarantees to real-time systems.

This chapter is organized as follows: We start with a general discussion on the current trends in various aspects of real-time systems, such as the applications, hardware platforms and the memory subsystems in Section 1.1. Then in Section 1.2, we introduce the existing research problems related to real-time memory subsystem design that need to be addressed. The main contributions of thesis are then introduced in Section 1.3, and finally, we conclude this chapter in Section 1.4.

1.1 Trends in Real-Time Systems

This section presents some of the general trends in real-time applications, hardware platforms, memories and memory subsystems. First, we introduce the properties and requirements of different real-time applications. Then, the trends in hardware platforms, memories and real-time memory subsystems are presented.

1.1.1 Application Requirements

Applications in real-time systems are typically classified according to their time and safety criticality as *hard*, *firm*, *soft* and *non* real-time [30, 87]. Both hard and firm real-time applications have strict timing requirements in order to meet their *deadlines*, and missing deadlines are not acceptable. Missing the deadlines of hard real-time applications have negative implications on human safety. For example, the Full Authority Digital Engine Controller (FADEC) in the aircraft jet engine should report abnormal effects in the engine within a predetermined time to avoid catastrophe [22]. On the other hand, firm real-time requirements are usually set by standards, such as the software defined radios [98] for LTE [21], or derived, such as the LCD controller in a video processing system [123], to maintain a sufficient *Quality of Service* to the users. Hence, missing deadlines of firm real-time applications is highly undesirable as it may lead to incorrect functionality of the system.

Soft real-time applications also have real-time requirements, but they are not as strict as for hard and firm real-time applications. Soft real-time applications have statistical real-time requirements, and hence, the deadlines can be missed once in a while still guaranteeing an acceptable performance on average. For

example, a Video-on-Demand server needs to provide each segment of video at the exact time to maintain continuity without jitter, but the frame rate can be reduced under resource-constrained conditions [77]. Lastly, non-real-time applications, such as web browsing, do not have any timing requirements, but they must run as fast as possible, i.e. have a good average-case performance.

In this thesis, we consider only applications with firm real-time requirements. However, the ideas presented can be applied to hard real-time applications as well, but with additional mechanisms to ensure safety, such as redundancy. In addition to firm real-time applications, we assume that the soft and non-real-time applications are present in the system as well. Note that this thesis does not address the issue of efficient resource utilization by soft real-time applications and they require systems with statistical service provisioning [77].

1.1.2 Hardware Platform

With the help of CMOS process technology scaling, the number of transistors in an integrated circuit are doubled approximately every year following Moore's law. This drastic reduction in feature size helped to move a large amount of off-chip circuitry from the printed circuit board to inside the integrated circuit, minimizing the production cost, power consumption and the complexities involved in high-speed board design. Moreover, with such a smaller feature size, the integrated circuit can be clocked at higher speeds allowing more applications to be run in a single core at the same time. However, there are some limitations in continuing the process-technology scaling down the line. The leakage power starts dominating the overall power consumption, the fabrication cost increases, and it is hard to keep the process variation within acceptable levels [58]. Hence, in order to meet the processing power requirements of future applications, processing platforms with multiple processors, such as *System on Chip (SoC)*, were introduced [75, 89, 54].

Multi-processor platforms can be found in almost all present-day electronic systems used in consumer electronics [80, 128, 38], telecommunication systems [26], automobiles [99, 136] and avionics [102, 84]. Multi-processor platforms typically consist of multiple homogeneous or heterogeneous processing elements interconnected using a bus, such as AXI [23] or DTL [106], or a *Network-on-Chip (NoC)* [27, 51]. Such platforms allow multiple tasks of the same application to be mapped efficiently to the multiple processing elements to achieve a higher overall performance in terms of power consumption and execution time or throughput. Main memory, i.e. off-chip Dynamic Random Access Memory (DRAM) is typically a shared resource in multi-processor platforms to enable communication between the applications running on different processing cores and minimize the cost.

1.1.3 Memories

There are several DRAMs available in the market that have been standardized by the Joint Electron Device Engineering Council (JEDEC) [9]. They are of different generations and have different interface widths, operating frequencies, and number of memory channels [9, 96]. As mentioned before, the number of memory clients is ever increasing with more applications being integrated in multi-processor platforms [59]. To meet this continuous demand for memory bandwidth with more applications being integrated into multi-processor platforms, the maximum clock frequency of memories were increased by over a factor of two every memory generation with the help of technology node scaling. This trend can be clearly seen by observing the clock speeds of memories in every generation of a DRAM type, such as LPDDR, LPDDR2 and LPDDR3 [9]. Due to the ever increasing memory bandwidth requirements with strict power budget in battery-operated mobile devices, memories with multiple memory channels in the same die, i.e. *multi-channel memories*, and wider interfaces, such as Wide IO [4] and Wide IO2 [8], were introduced. This is because a higher memory bandwidth-to-power ratio is achieved by increasing the number of memory channels and/or the memory interface width than by increasing the operating frequency [48]. The current trend in the scaling of memory clock frequency and/or the number of memory channels to satisfy the ever increasing bandwidth demands is expected to continue at least for the next few years [5, 61].

1.1.4 Memory Subsystems

In real-time systems, real-time guarantees on memory performance in terms of bandwidth and/or latency need to be provided to the memory clients to meet the firm real-time requirements of the applications, which are often quite quite diverse [129]. For example, in a H264 video processing system, the video decoding engine have high bandwidth requirements, while the LCD controller and CPU have low latency requirements [123]. These real-time requirements must be guaranteed at design time to reduce the cost of verification. Existing real-time memory controllers [105, 13, 108, 115, 25, 131, 82, 62] with a memory interconnect (IC) in front of it employing one or more predictable arbitration policies [14, 50, 110, 55, 113], as shown in Figure 1.1, provide guarantees on memory performance in terms of bandwidth and/or latency.

Real-time memory controllers typically bound the execution time of a memory request by fixing the memory access parameters of the request, such as burst length, number of banks over which a request is interleaved and the number of read/write commands, at design time. These parameters determine the *access granularity* and *memory-map* of the memory controller. The access granularity defines the amount of data read/written from/to the memory per request and the memory-map defines the physical placement of the request internally in the memory. A dedicated hardware block, *atomizer* (AT), is typically used to split every

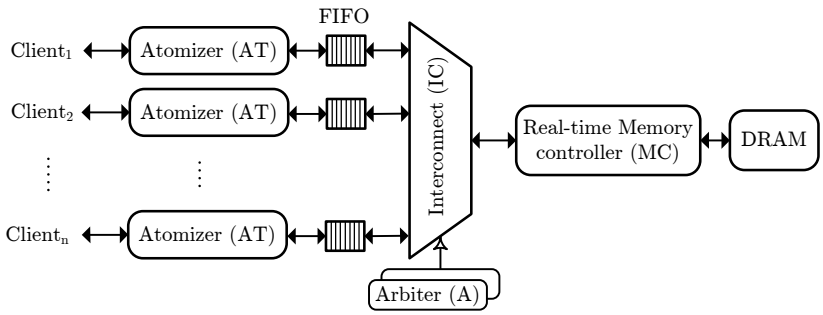


Figure 1.1: Real-time memory subsystem consisting of a real-time memory controller and memory interconnect. The *atomizer* (AT) splits a larger request into smaller sized requests according to the fixed access size of the real-time memory controller.

request of a memory client into smaller *service units* of size equal to the fixed request size of the real-time memory controller. Note that the atomizer can either be on the client side, i.e. an atomizer per client, or in front of the memory controller. In this thesis, we consider an atomizer per client, as shown in Figure 1.1, that splits large requests to smaller service units such that other clients can be served in bounded time [16, 49]. For a fixed access granularity, statically and semi-statically scheduled real-time memory controllers [25, 13, 108] use a fixed memory command schedule according to the command timing requirements provided by the memory data-sheet, which bounds the worst-case execution time of a read-/write request. In dynamically scheduled memory controllers [131, 115, 62, 82], the worst-case command schedule is determined to bound the execution time. Also, the *gross bandwidth* offered by a memory for a fixed access granularity can be computed [16]. The gross bandwidth of a memory is the worst-case bandwidth for a given access granularity configuration and it is computed after considering the overhead in memory access. Note that the gross bandwidth of a memory will always be less than or equal to its *peak bandwidth*, which is the maximum achievable bandwidth of a memory defined as the product of its interface width, operating frequency and data rate. In this thesis, we refer to a memory request of fixed size as a service unit and the time taken to execute a service unit as *service cycle*.

For resource sharing between multiple memory clients, memory interconnects employing predictable arbitration policies, such as Time Division Multiplexing (TDM) and Round-Robin, are used to provide real-time guarantees to the memory clients [13]. Existing interconnect architectures can be classified as *centralized* and *distributed* according to the implementation of the arbitration policy. In a centralized implementation, such as in [14, 50], the arbitration policy is implemented in a single physical location. Centralized architectures are easy to implement as the arbitration decision is made at a central location using a single arbiter for all clients.

In distributed architectures, arbitration of memory clients is performed in a distributed manner using multiple arbitration nodes [43, 110, 55, 113, 107]. The arbitration nodes in a distributed architecture are connected in a tree-like structure with the clients at the leaves of the tree and the memory controller at the root. Distributed memory interconnects can be either *locally arbitrated* or *globally arbitrated* depending on whether the arbitration nodes work independently or in a coordinated manner.

In a locally arbitrated (distributed) interconnect [43, 107, 110], the multiple arbitration nodes operate independently of each other and they have local first input first output (FIFO) buffer per input port, which buffers the incoming requests until they are served[1]. For example, the routers in a Round-Robin (RR) [110] and priority-based [117] NoCs forwards the packets according to a local arbitration policy. The high-level architecture of a locally arbitrated memory interconnect (IC) with distributed implementation is shown in Figure 1.2. It can be seen that decoupling buffers are required in between every arbitration stage as the arbiters operate independently. This means that the memory requests arriving a node might have to wait for their turn in the local buffers until they get service. On the other hand, the arbitration nodes in a distributed memory interconnect with global arbitration, such as statically scheduled TDM NoCs [55], serve requests according to a single global schedule, as shown in Figure 1.3. Hence, every arbitration node (implicitly) is aware of the scheduling decisions of other nodes such that buffering of requests are not required at every node. Note that we assume separate request and response paths, i.e. the read responses from the memory do not interfere with the read/write requests. Table 1.1 shows a summary of features of the different state-of-the-art memory interconnects.

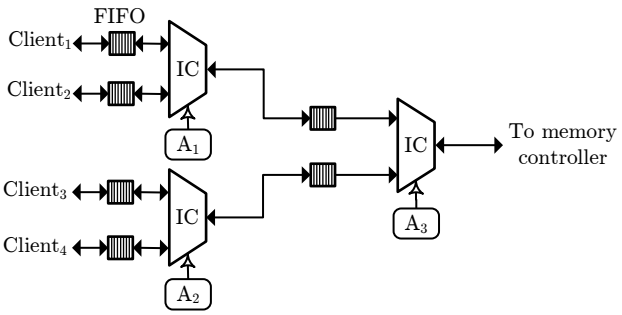


Figure 1.2: Locally arbitrated distributed memory interconnect (IC) with four memory clients. The FIFOs at the input of every arbitration stage store the requests temporarily until they get served.

[1]Other buffering schemes also are possible, but in essence there is always a decoupling buffer of size at least equal to a request size between the routers.

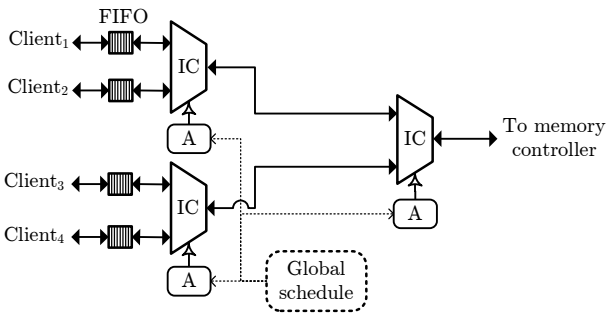


Figure 1.3: Globally arbitrated distributed memory interconnect (IC) with four memory clients. All of the arbitration stages works according to a single global schedule, and hence, FIFOs are not required in between the arbitration stages.

Table 1.1: Summary of different state-of-the-art memory interconnects.

Interconnect	*Scope*	*Arbitration*
Bus-based [14], PMAN [50]	Centralized	Global
TDM NoCs [55, 45, 113]	Distributed	Global
Other NoCs [110, 117]	Distributed	Local

1.2 Research Problems

This section introduces the two main research problems addressed in this thesis. First, we discuss the limitations of the existing real-time memory subsystem architectures in terms of scalability. Then, the necessity for design methodologies for faster design of memory subsystems in future real-time systems is shown.

1.2.1 Scalability

Traditional *bus-based* memory interconnects employing predictable arbitration policies having a centralized implementation, such as PMAN [50], suffer from poor scalability with respect to the number of clients. This is because the priorities of all memory clients are compared, i.e. priority resolution, using combinatorial logic consisting of a tree of multiplexers [118, 39, 88, 20], which increases the critical path of the logic for a large number of clients. The main drawback of this approach is that the critical path of the multiplexer tree increases with the number of clients, which reduces the maximum clock frequency at which the logic can be synthesized [33]. Moreover, the implementation of slack management in any of the predictable arbitration policies requires implicit priority resolution as one client needs to be selected out of many based on the *slack management policy*, which again is not scalable using centralized architectures [118].

Existing locally arbitrated memory interconnect suffer from large overhead in

terms of area, power and latency with increasing number of clients due to their decoupling buffers [45]. Moreover, the real-time performance analysis of such memory interconnects are difficult. On the contrary, the arbitration nodes of a globally arbitrated interconnect, such as a TDM NoC [55, 45, 113], work in a coherent manner, i.e. according to a single global schedule, such that no FIFOs are required at every arbitration stage. The arbitration decisions made at multiple arbitration nodes in a globally arbitrated interconnect are combined to determine the final arbitration decision. Existing NoC-based memory interconnects using a single global schedule, i.e. globally arbitrated interconnects only support TDM, which is not suitable in systems where the client requirements are diverse. This is because the TDM arbitration policy inherently couples the latency and bandwidth, which typically increases the *over-allocation* of bandwidth to the clients with low latency requirements [14].

In this thesis, we consider only the globally arbitrated distributed memory interconnects as they are scalable and have lower area consumption, power usage and latency compared to locally arbitrated interconnects. In a globally arbitrated interconnect, there is a dedicated *virtual circuit* between each source (client) and destination (memory controller). Since the clients use the interconnect concurrently and the requests may arrive interleaved at the memory controller, each client requires a dedicated buffer in the memory controller to avoid deadlock. Then, a local bus-based interconnect with an arbitration policy can be used to serve the requests to the memory controller as shown in Figure 1.4. However, this increases the area usage, power consumption and latency.

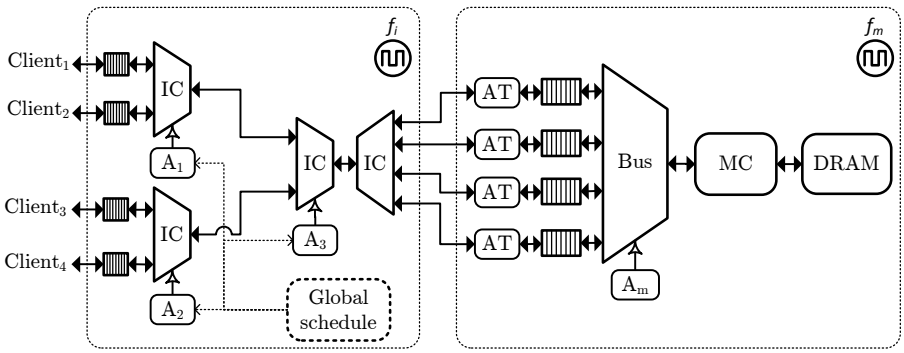


Figure 1.4: Globally arbitrated memory interconnect (IC) with distributed implementation with four memory clients, decoupled from the memory controller (MC) using FIFOs.

Multi-channel memories allow memory requests to be *interleaved* across different memory channels with different interleaving granularities after splitting it into smaller sized requests, as the memory channels are independent of each other. Previous studies on multi-channel memories show that mapping soft real-time memory clients to multiple memory channels according to their memory request sizes benefit average-case performance [111, 31, 101]. Interleaving a mem-

ory request across multiple memory channels allows parallel access to the different channels, which minimizes the latency. In addition to different request sizes, firm real-time memory clients in real-time multi-processor platforms come with different requirements on memory bandwidth, latency, communication and memory capacity as well. The memory requests of the different memory clients need to be interleaved across the different channels according to the client requirements and request sizes for efficient utilization of the multi-channel memory. Existing real-time memory subsystems only allow either interleaving of memory requests across all the memory channels or statically allocating memory clients to single memory channels, i.e. no interleaving. However, interleaving memory clients across all memory channels or not interleaving at all may result in poor memory utilization.

To summarize, we define our scalability problem as the lack of a scalable memory interconnect and a multi-channel memory controller. The memory interconnect must be scalable in terms of clock frequency to support faster memories and a large number of clients, and with lower area usage, power consumption and worst-case latency. The memory interconnect architecture must furthermore be configurable with different arbitration policies according to the diverse client requirements in re-usable platforms. For the efficient worst-case utilization of the multi-channel memories, the real-time multi-channel memory controller must allow interleaving of memory requests across memory channels with different interleaving granularities and with different bandwidth allocated to them in each channel. Note that the (single channel) memory controller architecture remains unchanged with increasing number of clients, and hence, we do not consider it as a bottleneck for scalability.

1.2.2 Design Methodologies

As we have discussed before, the ever increasing number of transistors integrated into a chip enables us to design a multi-processor platform for a real-time system with multiple simultaneously running applications. However, the design complexity of such multi-processor platforms is increasing with the number of applications being integrated into such platforms. Existing computer-aided tools for designing and configuring the hardware architecture for a large number of clients do not catch up with the speed at which the semiconductor feature size is decreasing [90]. To minimize the design time (time-to-market), the *design gap* [3] between hardware process technology capability and design methodologies need to be reduced.

Off-chip DRAM is expensive in terms of area and power consumption, and the memory price typically increases with bandwidth and memory capacity [1]. Hence, we need to design the memory subsystem such that the memory bandwidth utilization is maximized and the bandwidth allocated to the clients is minimized while meeting their requirements. There are plenty of DRAMs available in the market, of different generations, capacities, interface widths, operating frequencies and number of memory channels [9, 96]. These system-level parameters need

to be selected such that all of the memory client requirements are satisfied with minimal bandwidth allocated to them. Apart from these system-level parameters, the memory controller configuration, such as the memory-map configuration, decide the memory bandwidth utilization [18]. There are several memory-map configurations possible for a memory, which increases the design-space. Moreover, the clients typically have different memory request sizes and their bandwidth and/or latency requirements are quite diverse. Hence, determining the memory-map configuration is not a trivial problem. Additionally, the presence of multiple memory channels (multi-channel memory) introduces a new mapping problem, i.e. optimal mapping of memory clients to the memory channels. The total memory bandwidth allocated to the clients in a multi-channel memory depends on the interleaving granularities of memory requests of each memory client and the bandwidth allocated to them in each memory channel. Currently, there exist no methodology for optimal mapping of memory clients to a multi-channel memory. We define our memory subsystem design optimization problem as follows:

Given a set of real-time memory clients with different request sizes and diverse requirements on memory bandwidth and/or latency, select the memory, configure the memory controller and arbiter, and determine the mapping of memory clients to memory channels, such that the memory bandwidth utilization is maximized and the bandwidth allocated to the clients is minimized.

1.3 Thesis Contributions

In this section, we introduce the two main contributions of this thesis. First, we address the scalability issue by presenting our proposed scalable memory subsystem architecture for real-time systems. Then, an automated methodology for bandwidth-efficient design of memory subsystems for real-time systems is presented.

1.3.1 Scalable Architecture

Our proposed solutions for a scalable memory subsystem architecture consists of three main innovations that build on each other: (1) A *generic, and globally arbitrated memory tree (GAMT)* [47] that can be configured with five different arbitration policies. (2) A *coupled memory interconnect (CMI)* architecture that can be used to couple existing globally arbitrated interconnects with the memory controller [45]. (3) A *multi-channel memory controller (MCMC)* that allows interleaving memory requests across memory channels with different interleaving granularities and with different bandwidth allocated to them in different channels [44, 46].

To address the scalability issue in terms of clock frequency in existing memory interconnects, this thesis proposes a distributed memory interconnect, generic and globally arbitrated memory tree (GAMT). The high-level architecture of GAMT,

shown in Figure 1.5, consists of dedicated *accounting and priority assignment (APA)* logic per client, which keeps track of its eligibility status to get scheduled and assigns a unique priority level according to an arbitration policy. All clients are scheduled according to the notion of a *global scheduling interval*, which means that the scheduling decisions are taken by the different APAs at the same time. The *priority resolution* among the clients is done using a tree of multiplexers with pipeline registers in between them. When the service unit of a client with the highest priority in a scheduling interval reaches the memory controller, it is removed from the request FIFO. The remaining service units that are dropped at the multiplexer stages are re-scheduled in the next scheduling interval. The distributed APA logic and the priority resolution enables GAMT to be synthesized up to *four times faster* than traditional bus-based architectures. Moreover, GAMT outperforms the centralized implementations by over 51% and 37% in terms of area and power consumption for a given bandwidth, respectively. *(Chapter 3)*

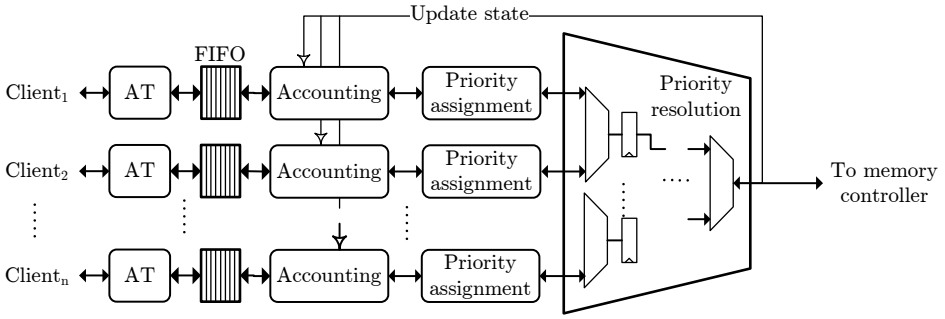


Figure 1.5: A generic scalable memory interconnect architecture. *Accounting* keeps track of eligibility status of a client to get service. *Priority assignment* assigns a unique priority to each client. The fully-pipelined *priority resolution* grants access to the client with the highest priority.

To address the issue of large area usage, power consumption and worst-case latency due to the decoupled memory interconnect and memory controller, this thesis proposes a novel coupled memory interconnect (CMI) architecture. The basic idea of CMI is to generate the interconnect and memory controller clock frequencies from the same clock source and align the clock cycles at the boundaries of their service cycles. The service unit size is made same in the interconnect and the memory controller. This helps to remove the decoupling buffers and the bus-based arbiter between the interconnect and the memory controller, which reduces the area usage, power consumption and the worst-case latency. The high-level architecture of the coupled memory interconnect is shown in Figure 1.6. It can be seen that the arbitration is only done in a single point compared to the decoupled architecture shown in Figure 1.2, where the arbitration is done twice. Our proposed CMI architecture can be used to couple a globally arbitrated memory interconnect, such as TDM NoC and GAMT, with the memory controller.

Coupling a TDM NoC and memory controller using our approach saves 45% in guaranteed latency, 20% in area, 19% in power consumption, with different DRAM generations, for a system consisting of 16 memory clients. (Chapter 3)

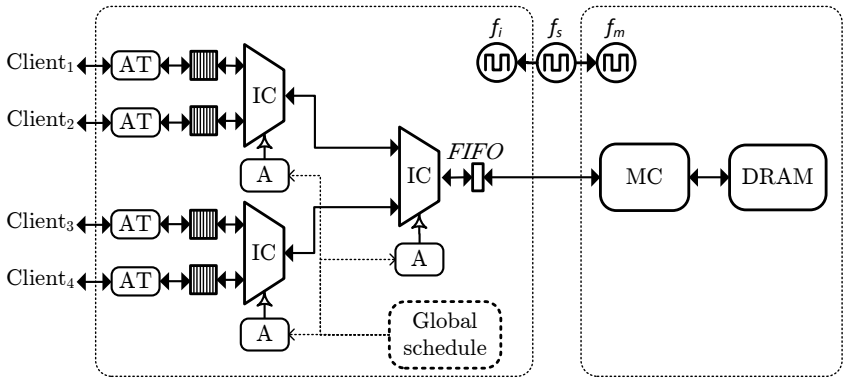


Figure 1.6: Proposed coupled memory interconnect (CMI) architecture. The interconnect and memory clock frequencies f_i and f_m, respectively, are derived from the source clock frequency f_s.

For efficient use of a multi-channel memory, a configurable multi-channel memory controller (MCMC) architecture, as shown in Figure 1.7, is proposed. MCMC consists of a dedicated channel selector (CS) per client, which routes the service units to the different channels according to the configuration programmed in the sequence generator (SG). Each memory channel is controlled by a *channel controller (CC)* with a memory interconnect employing a predictable arbitration policy, which multiplexes the requests arriving from different channel selectors. Note that the channel controller is the same as the (single channel) memory controller (MC) and we use a different name here to avoid confusion with the multi-channel memory controller. Also, we propose a novel method for logical-to-physical address translation, that allows each client to be mapped with different interleaving granularities and allocated bandwidth in each memory channel. Note that the logical-to-physical address translation performs service unit to channel mapping, whereas the memory-map performs service unit to physical memory address mapping. (Chapter 3)

Combining the three innovations, i.e. GAMT, CMI and MCMC, a scalable real-time memory subsystem can be realized, as shown in Figure 1.8. MCMC enables efficient utilization of the multi-channel memory. GAMT allows the memory subsystem to be synthesized at higher clock frequencies and configured with different arbitration policies according to the diverse client requirements. Moreover, by coupling GAMT with the memory controller (channel controller) using the CMI architecture, its area, power and the worst-case latency is minimized. Note that a globally arbitrated interconnect, such as TDM NoC and GAMT, can be coupled with the memory controller by making the service unit size and schedul-

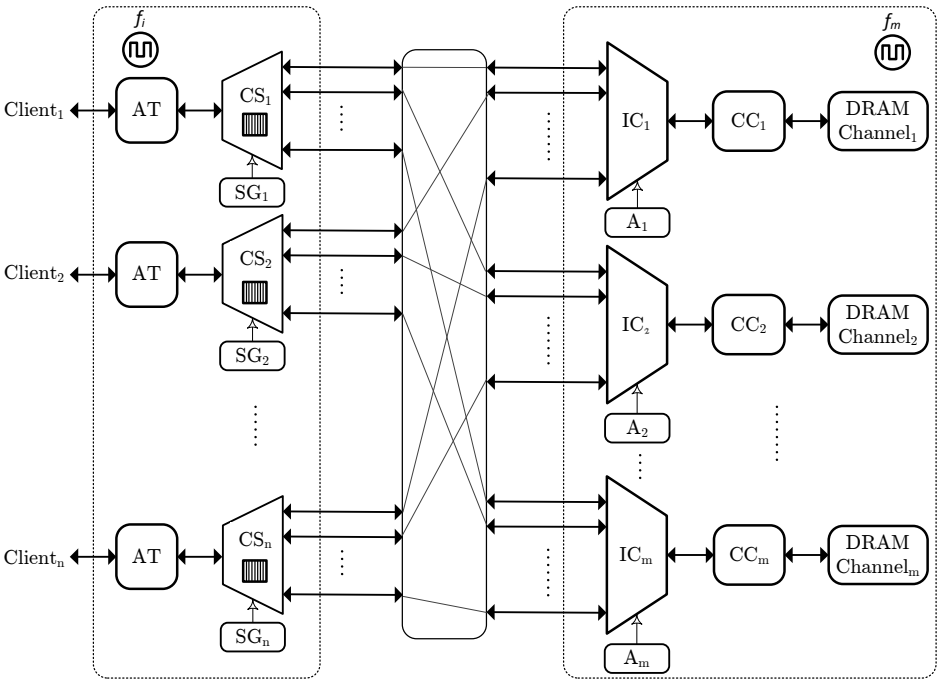


Figure 1.7: High-level architecture of the proposed multi-channel memory controller (MCMC). The Channel Selector (CS) routes the service units to the different memory channels according to the configuration in the sequence generators (SG). Note that the point-to-point connections between the CS and the Interconnect (IC) are short wires.

ing interval same as the memory controller and by ensuring non-blocking delivery of the service unit. Hence, a completely scalable memory subsystem, both in terms of clock frequency and number of memory channels can be realized using the contributions presented in this thesis.

1.3.2 Bandwidth-Efficient Design Methodology

This thesis proposes an automated design-flow for a *bandwidth-efficient* memory subsystem design, i.e. the worst-case memory bandwidth is maximized and the bandwidth allocated to the clients is minimized, in real-time systems, as shown in Figure 1.9. At first, a pre-selection of memories is made from all available memory types. In this step, only the memories with peak bandwidth greater than or equal to the gross bandwidth requirement of all clients together are selected. Then for all those memories, we compute the worst-case gross bandwidth using our proposed *design guidelines* for memory-map selection. We propose the design guidelines for memory-map selection, which maximizes the worst-case gross bandwidth, based on a worst-case analysis of memory types across and within

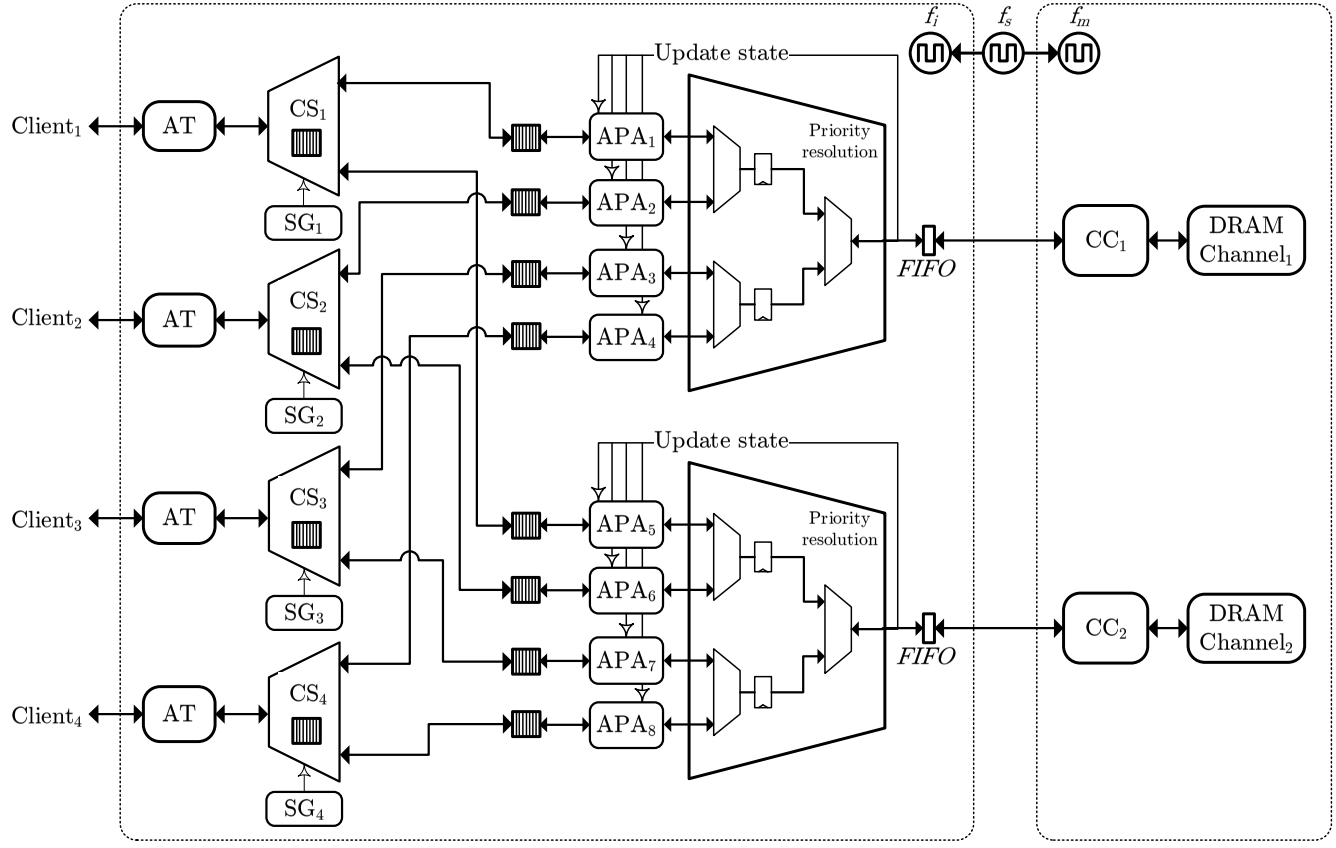


Figure 1.8: An example instance of the scalable memory subsystem with four memory clients and two memory channels realized by combining the proposed MCMC of Figure 1.7, GAMT of Figure 1.5 and CMI of Figure 1.6.

generations [48]. The design guidelines reduce the design space of memory-map selection drastically. Note that we compute the worst-case gross bandwidth using the methods presented in [14]. Then, using our proposed method, we compute the *aggregate bandwidth* requirements for the different service unit sizes. The aggregate bandwidth requirement is computed to consider the impact of *data efficiency*, which defines the fraction of fetched data that is useful to the clients [14]. The aggregate bandwidth computation takes in to account the different request sizes and bandwidth requirements of all clients. Finally, for all those service unit sizes, we perform mapping of clients to memory channels, with the objective to minimize the bandwidth allocated to them while satisfying their requirements, using our proposed algorithms. To determine the mapping of memory clients to memory channels with minimum allocated bandwidth, this thesis proposes two algorithms, one an optimal algorithm based on an integer programming formulation of the mapping problem, and the other a fast heuristic algorithm to determine the number of service units and the bandwidth that needs to be allocated to each client in each memory channel. With up to 4 memory channels and 100 memory clients, our heuristic algorithm finds a valid mapping in less than one second while the optimal algorithm in a solver takes 2 hours. However, this performance gain comes at a cost of 7% reduction in successfully mapped use-cases, which is significantly lower than the failure ratios 19% and 33% of two traditional heuristic mapping algorithms on the same input set [44, 46]. (Chapter 4)

1.4 Summary

With the drastic reduction in feature size of an integrated circuit over the years, the number of processing cores integrated into a chip has increased significantly. Such multi-processor platforms allow a large number of applications running at the same time with different application tasks communicating with each other using a shared memory. Real-time memory controllers with a memory interconnect employing a predictable arbitration policy are used to provide guarantees on memory bandwidth and/or latency to the firm real-time applications running in the system. However, current memory subsystems are not scalable for future systems with a large number of clients. This is because the existing memory interconnect cannot be synthesized at higher clock frequencies and are decoupled from the memory controller, i.e. they consume more power and area and have larger worst-case latencies. Moreover, we need a reconfigurable memory interconnect that can be configured with different predictable arbitration policies according to the diverse client requirements in re-usable platforms. On the other hand, efficient utilization of a multi-channel memory needs interleaving memory requests of clients with different granularities according to their bandwidth and/or latency requirements, and currently there is no real-time memory controller for multi-channel memories.

With a large number of applications integrated into multi-processor platforms,

the system design complexity increases as well. For the design of a real-time memory subsystem, there are several design parameters that need to be selected. This includes parameters related to the selection of the memory type, configuration of the memory controller and arbiter, and mapping of the memory clients to the memory channels. The selection and configuration of these parameters impact the efficient utilization of the memory. As the memory resource is scarce and systems are getting more complex, we need automated design methodologies for faster and bandwidth-efficient design of memory subsystems for future real-time systems.

To address the scalability issue in the existing real-time memory subsystems, we propose three innovations in this thesis: (1) A generic, globally arbitrated memory tree (GAMT), which runs four times faster than traditional bus-based interconnects and can be configured with five different predictable arbitration policies. (2) A coupled memory interconnect (CMI) architecture to couple any existing globally arbitrated memory interconnect with the memory controller for lower area usage, power consumption and latency compared to a decoupled architecture. (3) A real-time multi-channel memory controller (MCMC) with a novel method for logical-to-physical address translation, together allowing memory requests of different clients to be interleaved across the memory channels with different interleaving granularities.

For faster and bandwidth-efficient design of memory subsystem in real-time systems, we propose a novel automated design-flow. The inputs to the design-flow are the set of memory type specifications, client bandwidth, latency, capacity and communication requirements, and client request sizes. The design-flow includes methodologies for memory type selection, memory-map configuration in the memory controller, and algorithms for bandwidth-efficient mapping of memory clients to memory channels. The final output of the design-flow is the memory type, memory-map configuration and mapping of clients to the channels.

In the remainder of this thesis, Chapter 2 gives an introduction to DRAMs, state-of-the-art real-time memory controllers, predictable arbitration policies and existing memory interconnects. In Chapter 3, the proposed GAMT, CMI and MCMC architectures, and their experimental evaluations are presented. Chapter 4 then presents the proposed automated design-flow for bandwidth-efficient design of DRAM subsystems in real-time systems and applies it to a case-study where the memory subsystem of a High-Definition (HD) video processing system is designed. Previous works related to the contributions presented in this thesis are discussed in Chapter 5. Finally, the thesis is concluded in Chapter 6 with future work.

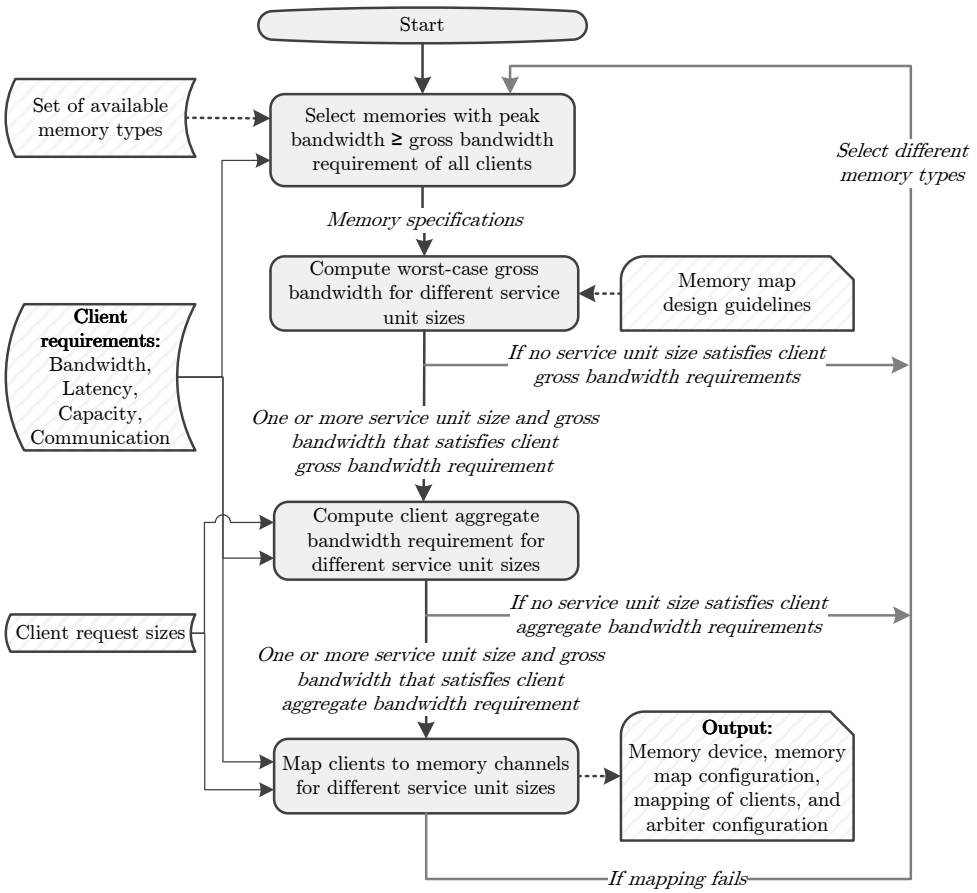


Figure 1.9: Proposed automated design flow for bandwidth-efficient DRAM subsystem design in real-time systems. Note that the final output of the design-flow is a single optimal configuration although there are one or more service unit sizes that gives a valid mapping.

Chapter 2

Background

DRAM is typically a shared resource for cost reasons and to enable communication between the processing elements in multi-processor platforms. As introduced in Section 1.1.4, real-time memory subsystems consists of a real-time memory controller and a memory interconnect employing predictable arbitration policies multiplexing the requests arriving from different clients. The memory interconnect architecture can be centralized (bus-based) or distributed (TDM NoCs). Real-time memory subsystems provide performance guarantee on memory bandwidth and/or latency to the memory clients in the system. Real-time memory subsystems can be analyzed using shared resource abstractions, such as the Latency-Rate ($\mathcal{LR}$) [126] server model, which can be used in formal performance analysis based on e.g., network calculus [35] or data-flow analysis [121].

In this chapter, we give an overview of the high-level DRAM architecture, its operation and available DRAM configurations in Section 2.1. We introduce the concept of real-time memory controllers in Section 2.2, and the $\mathcal{LR}$ server model and different predictable arbitration policies in Section 2.3. In Section 2.4, we introduce statically-scheduled TDM NoCs.

2.1 Dynamic Random Access Memories (DRAM)

This thesis proposes memory subsystem architectures and design methodologies primarily for Dynamic Random Access Memories (DRAM). In this section, we first present the high-level architecture of DRAM and its operation, and then the different DRAM devices and their configurations.

2.1.1 DRAM Architecture and Operation

In a DRAM device, each bit is stored using a single transistor-capacitor pair known as *storage cell* [64]. The storage cells are arranged to form a memory array with a matrix-like structure, as shown in Figure 2.1. The intersection of rows and columns, specified by a *row address* and a *column address*, identifies the storage cells inside the memory array. The memory array and a *row buffer* constitute a *bank*. Current DRAM devices contain either 4 or 8 banks that can be accessed concurrently, although they share command, address, and data buses to reduce the number of off-chip pins.

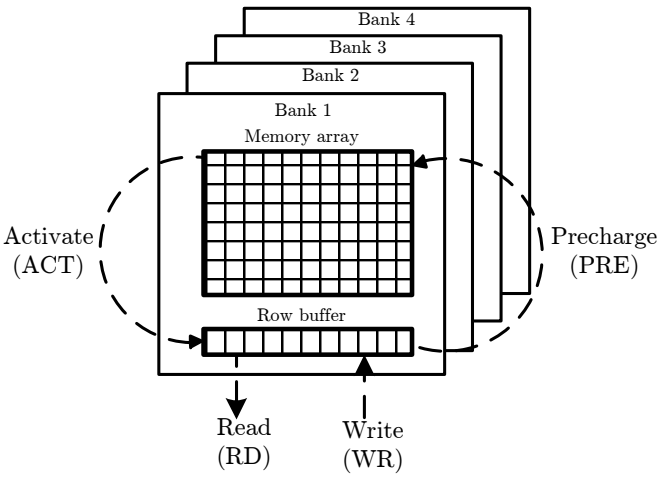


Figure 2.1: High-level DRAM architecture showing the organization of memory array, row buffer and banks.

During a memory access, the data from the storage cells of target row are copied to the row buffer before performing a read/write operation. Data is then transferred over the data bus with a *data rate* of one or two words per clock cycle, depending on if the memory device uses a Single Data Rate (SDR) or a Double Data Rate (DDR). The data rate affects the *peak bandwidth* of the memory, which is defined as the product of its operating frequency, data rate, Interface Width (IW) and number of memory channels (NC).

The memory controller interacts with the DRAM by sending DRAM commands. There are several timing constraints that must be considered while issuing these commands. To understand these timing constraints, an example scenario for a read operation is shown in Figure 2.2. The contents of a row inside the memory array is copied to the row buffer by issuing an *activate (ACT)* command. It takes *tRCD* cycles to fetch the data from the storage cells and copy it to the row buffer, which is the minimum time before the *read (RD)* command can be issued. Once the read command is issued, it takes additional *tRL* cycles before

the first words of data is available on the data bus, as indicated by D0-D1 for the DDR device in the figure. A read/write command accesses the memory as a burst with a predefined *Burst Length (BL)* (in words). Before another row in the memory array can be read, the existing row must be closed by writing back the contents to the storage cells using a *precharge (PRE)* command. The precharge command can only be issued *tRAS* cycles after the activate command. Also, the next activate command is allowed to be issued only after *tRP* cycles from the precharge command as shown in the figure.

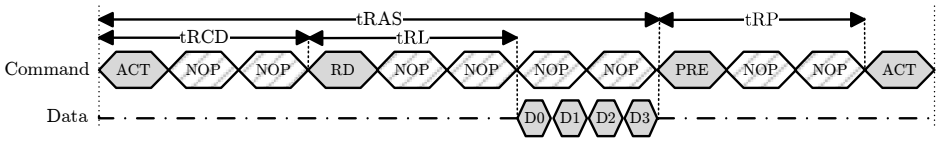


Figure 2.2: DRAM command timing diagram for an example read operation.

In addition, there are other constraints that needs to be satisfied for the correct functioning of the memory device. The four-active window constraint specifies the maximum number of activate commands in a window of duration *tFAW* cycles. As there will be leakage of charge from the storage cells over the time, they must be recharged using a *refresh* command every refresh interval, *tREFI*, to prevent loss of data. The DRAM data-bus is bi-directional and setting the bus direction for a read operation after a write will take *tWTR* clock cycles. Please refer to the data-sheet of the memory for an exhaustive list of timing constraints [9, 96]. Note that due to the various command timing constraints of DRAM, the maximum achievable memory bandwidth will always be less than the peak bandwidth.

2.1.2 DRAM Generations and Configurations

DRAM devices standardized by the Joint Electron Device Engineering Council (JEDEC) [9] can be broadly classified into *standard* DRAMs and *mobile* DRAMs. Standard DRAM generations, such as DDR2, DDR3, and DDR4, are targeted towards high-performance computing systems, such as workstations and servers, and can be clocked at higher speeds compared to mobile DRAMs. Mobile DRAM generations, such as LPDDR, LPDDR2, LPDDR3, LPDDR4, WideIO and WideIO2, are designed specifically for battery-operated mobile devices, such as smart phones and notebook computers, due to their lower power consumption compared to the standard DRAMs. Mobile DRAMs differ from the standard DRAMs in the initialization sequence, input/output circuitry and clocking [92]. Table 2.1 shows an overview of the standard and mobile memories across and within generations based on the JEDEC specifications [66, 67, 68, 72, 69, 70, 71, 73, 4, 8]. It can be seen that the operating frequency increases every generation to increase the memory bandwidth, and supply voltage is reduced to minimize the power consumption. Moreover, the memory capacities are increased in order to meet the

application demands.

Due to the ever increasing memory bandwidth requirements in mobile devices with strict power budget, memories with multiple memory channels in the same die, i.e. *multi-channel memories*, are proposed for the LPDDR3, LPDDR4, WideIO and WideIO2 memory generations. In addition to having multiple memory channels, WideIO and WideIO2 have wider interfaces which further reduces their power consumption [48]. WideIO is a single data rate (SDR) device consisting of four independent memory channels, each having an interface width of 128 bits, while its second generation, WideIO2, consists of eight channels, each having a 64-bit interface.

2.2 Real-Time Memory Controllers

Existing real-time memory controllers can be classified as *static*, *dynamic* and *semi-static*, according to their scheduling policy of memory commands. Memory controllers with static [25] command schedule require the complete sequence of memory requests in advance for the analysis of the worst-case execution time of a request. Dynamic memory controllers [131, 115, 62, 82] make the memory command scheduling decisions at run-time. The worst-case command schedule is analytically determined to bound the execution time in dynamically scheduled memory controllers. Semi-static memory controllers [13, 108] use a pre-computed (fixed) command sequence to perform the basic memory operations, such as read, write and refresh, and dynamically schedule the command sequences according to the incoming memory requests. Figures 2.3 (a) & (b) show example pre-computed command schedules for read and write operations, respectively, for a memory request interleaved across two memory banks. The read and write may have different command schedules depending on the command timing constraints, as explained in Section 2.1. The NOPs in the command schedule are inserted such that the different command timing constraints are satisfied. The worst-case execution time of a memory request can be computed from the pre-computed command sequences as explained in [17].

(a) Read operation

(b) Write operation

Figure 2.3: Example pre-computed command schedules for memory read and write operations in semi-static real-time memory controllers. The NOPs in the command schedule are inserted such that the different memory command timing constraints are satisfied.

Real-time memory controllers bound the execution time of a memory request

Table 2.1: Overview of DRAM configurations across and within generations.

Memory	*Operating Speeds (MHz)*	*Capacities*	*Operating Voltage (V)*	*IO widths (bits)*	*Banks*	*Channels*
DDR [66]	100 - 200	128 Mb - 1 Gb	1.8	4, 8, 16	4	1
DDR2 [67]	125 - 333	128 Mb - 4 Gb	1.8	4, 8, 16	4, 8	1
DDR3 [68]	400 - 1066	512 Mb - 8 Gb	1.8	4, 8, 16	8	1
DDR4 [72]	800 - 1600	2 Gb - 16 Gb	1.8	4, 8, 16	8	1
LPDDR [69]	100 - 266	64 Mb - 2 Gb	1.8	16, 32	4	1
LPDDR2 [70]	100 - 533	64 Mb - 8 Gb	1.2	8, 16, 32	8	1
LPDDR3 [71]	667 - 800	4 Gb - 32 Gb	1.2	16, 32	8	1, 2
LPDDR4 [73]	800 - 2133	4 Gb - 32 Gb	1.1	16, 32	8	2
WideIO [4]	200 - 266	1 Gb - 32 Gb	1.2	128	4	4
WideIO2 [8]	200 - 266	8 Gb - 32 Gb	1.1	64	4, 8	4, 8

by fixing the memory access parameters of a request, such as burst length and number of read/write commands, at design time [14]. Memory accesses by real-time memory controllers can be characterized by three parameters: Burst Length (BL) (as explained in Section 2.1), *Banks Interleaved (BI)*, and *Burst Count (BC)*. These are collectively referred to as the *memory map* [52] as they determine the physical location of data in the memory array. BI specifies the number of banks over which the data is interleaved and BC specifies the number of bursts per bank [17]. These parameters define the *access granularity* (AG) of the memory controller, which defines the amount of data read/written from/to the memory per request. The access granularity of a memory in bytes is given by $AG = BI \cdot BC \cdot BL \cdot IW_m$, where IW_m is the interface width of the memory. The choice of memory map is done at design time and determines the memory efficiency that is guaranteed for a given mix of request sizes [52].

In this thesis, we consider the amount of data accessed in the memory while serving a single request to be fixed and we refer to these memory requests of a fixed size as *service units (SU)* with size (in Bytes) SU^{bytes}, and the time taken to serve such a service unit is a *service cycle*. The service unit size of DRAMs is typically in the range of 16-256 Bytes. Note that although the dynamic memory controllers can support multiple request sizes, we consider only a single service unit size as all the atomizers are configured to split the incoming requests to the same size. The time (in ns) taken by the memory controller to finish the execution of a service unit is called a *memory service cycle* and is denoted by SC^{ns}. For a given memory with operating frequency f_m, the memory service cycle length of a service unit size of SU^{bytes}can be computed according to [18]. The service cycle for a read and a write request can be different and depends on the memory type ($tWTR$ constraint as explained in Section 2.1) and the memory controller. For simplicity, we assume the same service cycle length for read and write requests, as it is shown in [53] that the memory service cycle for read and write requests can be made equally long with negligible loss of gross bandwidth. Note that the request size of a memory client may be smaller than the service unit size of the memory controller. In that case, the *data efficiency*, defined as the ratio of request size to the service unit size, will be lower than 100%. For a service unit size with a given memory map configuration, *gross bandwidth* (b_m^{gross}), which is defined as the maximum achievable memory bandwidth in the worst-case without taking data-efficiency into account, can be computed according to the analysis presented in [14]. The gross bandwidth accounts for various overheads, such as activating and precharging of rows, write-to-read switching and refresh operation. Note that although we use the analysis techniques presented in [14] for the computation of gross bandwidth, the techniques can in general be applied to static and dynamic memory controllers as well.

2.3 Predictable Arbitration

Current real-time memory controllers provide real-time guarantees to the clients and use a predictable arbitration policy in the memory interconnect in front of them to multiplex requests from different clients. In this thesis, we use the *Latency-Rate* $\mathcal{LR}$ server [126] model as the shared resource abstraction to derive bounds on service provided by predictable arbiters. First, we introduce the $\mathcal{LR}$ server model and then the predictable arbitration policies considered in this thesis.

2.3.1 Latency-Rate ($\mathcal{LR}$) Servers

Latency-Rate ($\mathcal{LR}$) servers [126] are a general model to capture the worst-case behavior of various scheduling algorithms or arbiters in a simple unified manner, which helps to formally verify the service provided by a shared resource. There are many arbiters belonging to the class of $\mathcal{LR}$ servers, such as TDM, Round-Robin and its variants Weighted Round-Robin (WRR) [78], Deficit Round-Robin (DRR) [119], and priority-based arbiters with a rate-regulator, such as Credit-Controlled Static Priority (CCSP) [20] and Priority Based Scheduler (PBS) [124]. The $\mathcal{LR}$ abstraction capture behavior of many different arbiters, and is compatible with a variety of formal analysis frameworks, such as data-flow analysis [121] or network calculus [35].

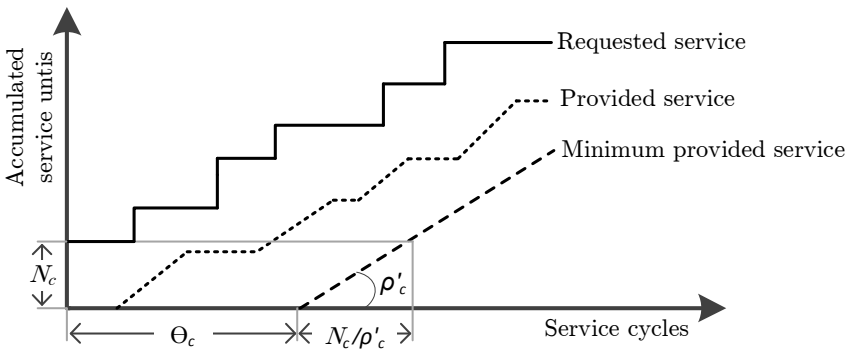


Figure 2.4: Example service curves of a $\mathcal{LR}$ server showing service latency (Θ_c) and completion latency (N_c/ρ'_c).

Using the $\mathcal{LR}$ abstraction, a lower linear bound on the service provided by an arbiter to a client can be derived. In this thesis, we assume a simplified $\mathcal{LR}$ abstraction and we do not consider clients with multiple outstanding requests, although it can be added if the characterizations of the arriving traffic is taken into consideration to bound the waiting time in the queue [126]. Otherwise, it is also possible if the $\mathcal{LR}$ abstraction is embedded in a formal analysis framework,

such as data-flow, that considers back-pressure [100].

Figure 2.4 shows example service curves of a $\mathcal{LR}$ server. The requested service (request size) by a client at a time consists of one or more service units, indicated on the y-axis of the figure. The minimum service provided to the client c is the service guaranteed by the $\mathcal{LR}$ abstraction, which depends on two parameters, namely the *service latency* Θ_c and the *allocated rate* ρ'_c (bandwidth). The *service latency* is defined as the maximum time before the allocated rate is provided, as seen in the figure, and depends on the choice of arbiter and its configuration, e.g. allocated rate and/or priority [14]. After a request consisting of N_c service units is scheduled to be served, it receives service at the allocated rate ρ'_c and it hence takes N_c/ρ'_c *service cycles* to finish serving the request, called the *completion latency* of the client. The *worst-case latency* $\hat{L}'_c$ (in service cycles) of a client c is the total time taken by its request of size N_c service units at the head of its request queue to get served in the worst case, which is the sum of the service latency and the completion latency, given by Equation (2.1).

$$\hat{L}'_c = \Theta_c + \lceil N_c/\rho'_c \rceil \tag{2.1}$$

The worst-case latency (in ns) of a memory client c in a real-time memory subsystem consisting of a memory m operating at frequency f_m is given by Equation (2.2), where δ_m the internal pipeline delay of the memory subsystem and depends on the number of pipeline stages in the RTL implementation of the memory controller.

$$\hat{L}'_c = \frac{(\Theta_c + \lceil N_c/\rho'_c \rceil) \cdot SC^{cc}_m + \delta_m}{f_m} \tag{2.2}$$

2.3.2 Predictable Arbitration Policies

In this thesis, we consider five different arbitration policies, Time Division Multiplexing (TDM), Round Robin [78], Frame-Based Static Priority (FBSP) [14], Priority-Based Scheduler (PBS) [124, 115] and Credit-Controlled Static-Priority (CCSP) [20], which have been proposed for shared memory access in real-time systems.

TDM is a frame-based arbitration policy with a fixed frame size, f, consisting of one or more TDM slots and each slot is of size equal to the SI. Each client is statically assigned to one or more slots and the fraction of number of assigned slots to the total number of slots in the frame is the allocated *rate*, ρ'_c (corresponds to the fraction of total gross memory bandwidth). With the progress of time, the clients are served every SI according to the static order in the TDM schedule and the frame repeats itself at the end of every frame. RR is a special case of TDM where the frame size is equal to the number of clients and each client is assigned to exactly one slot such that the clients are served one after the other. Note that TDM and RR are the same except that RR is more restrictive.

Since a client can issue a read or write request in a TDM slot, we consider a slot size equal to the maximum of read or write service cycles. Figures 3.5(a) & 3.5(b) show a TDM frame of size f with client c allocated two slots using contiguous and distributed slot allocation strategies, respectively. Here, c gets a rate $\rho'_c = 2/6$, since two out of six slots are allocated to c. The service latency (Θ_c) of c is 4 and 2 for contiguous and distributed (equidistant) TDM allocations, respectively, because of the interference from other clients that occupy the remaining set of TDM slots. In terms of rate ρ'_c and/or frame size f, the service latencies of contiguous and distributed TDM are given by $\Theta_c = f \times (1 - \rho'_c)$ and $\Theta_c = f/(f \times \rho'_c) - 1$, respectively, as shown in [14]. A more complex method for determining the service latency for arbitrary slot allocations is presented in [19].

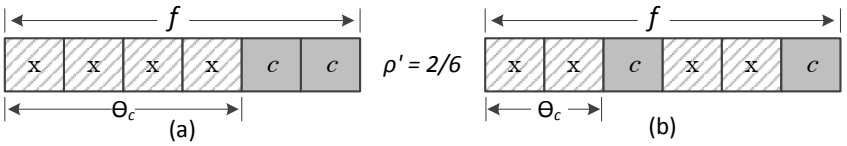


Figure 2.5: Example TDM frame of size f showing service latency (Θ_c) of client c with its slots allocated using (a) contiguous and (b) distributed allocation strategies.

Hence, for a TDM arbiter with a frame size f, the worst-case latency (in slots) of a client c with an allocated rate of ρ'_c for contiguous and distributed TDM is given by Equations (2.3) & (2.4), respectively, in which both service latency and completion latency are rounded up to make the bound conservative.

$$\hat{L}'_c = \lceil f \times (1 - \rho'_c) \rceil + \lceil N_c/\rho'_c \rceil \tag{2.3}$$

$$\hat{L}'_c = f/\lceil f \times \rho'_c \rceil - 1 + \lceil N_c/\rho'_c \rceil \tag{2.4}$$

Similar to TDM, FBSP [14] is also frame-based with a fixed frame size and each client is assigned a *budget* of slots corresponding to its rate, ρ'_c. However, unlike in TDM there is no static assignment of clients to the slots. Instead, each client is assigned a unique static priority and at every SI, the (backlogged) client with the highest priority and one or more remaining budget slots is granted service. When a client is granted service during an SI, its budget is reduced by one. Note that in this thesis, we refer to the clients with sufficient budget to get scheduled as *eligible clients*. At the beginning of every new frame, the budgets of all clients are reset to their initial values, i.e. all clients will be eligible. PBS is a special case of FBSP where only one of the clients is assigned with the highest priority [124] and every other client has equal (lowest) priority.

Unlike frame-based arbitration policies, CCSP [20] does not use the notion of frames for the replenishment of the client budgets. Instead, the budget of each client is *replenished continuously*, i.e. for every SI. This means that the replenishment interval in CCSP is a lot less than the frame-based arbitration

policies. The service provided to a client c depends on allocated *burstiness* (σ_c), rate (ρ'_c), and its static priority. To start, a client is credited with initial budget, which depends on σ_c. During every SI, the budget level is incremented at a constant fractional rate ρ'_c and decremented by one when it is granted service. When the client is not backlogged, it is only allowed to build up its budget until its initial budget value.

One important aspect of the various arbitration policies is that each of them comes with different properties. TDM is suitable for providing temporal isolation among the clients, and RR when all clients need fair treatment. FBSP, PBS and CCSP are priority-based with different benefits [14] and suitable when differentiated treatment needs to be provided to the clients. CCSP provides lower latency to high priority clients without using the notion of frame. Using PBS is beneficial when there is one low latency client in the system and high switching overhead between the remaining clients. This means that an arbitration policy needs to be selected according to the requirements of clients running in the system.

In all the arbitration policies that we discussed above, *work conservation*, i.e. assigning the unused time slot of a client to another client with pending request(s), can be implemented according to a certain *slack management policy*. For example, higher priorities can be given to the bandwidth-demanding soft or non-real-time clients to improve their average-case performance or using the same static-priority levels in priority-based arbitration policies in work-conserving mode as well. Note that the budget of the scheduled client in work conservation mode is not deducted.

2.4 Statically Scheduled TDM NoCs

In this section, we explain statically scheduled TDM NoCs in detail as we consider it as the globally arbitrated memory interconnect for the evaluation of our Coupled Memory Interconnect (CMI) architecture explained later in Chapter 3.

State-of-the-art TDM NoCs are either packet switched [51, 57, 97, 112, 11] or circuit switched [122, 130], and are globally arbitrated as the arbitration in the routers is done using a statically computed global TDM schedule. The routers in the NoCs are non-blocking, and hence, the Network Interface (NI) performs end-to-end flow control to avoid overflow of the buffers in the routers and NIs. The NI converts a memory request into one or more smaller units, called *flits*, and transports them to a destination NI according to the global TDM schedule. Typically, the service unit (flit) size of the NoC is in the order of few words (4-12 Bytes) [55]. The time taken to transport a complete service unit over a NoC link i is called a *NoC (or interconnect) service cycle* and is given by SC_i^{ns} (in ns) and SC_i^{cc} (in clock cycles). For a service unit of size SU^{bytes}, SC_i^{cc} can be computed using Equation (2.5), where IW_i is the interface width of the NoC (in bits) and δ_{ov} the cycles to carry a header, such as address and command that is specific to a NoC architecture.

$$SC_i^{cc} = \lceil (SU^{bytes} \times 8)/IW_i \rceil + \delta_{ov} \tag{2.5}$$

In this thesis, we use the TDM NoC connected in the form of a tree with the memory controller at the root of the tree and a fully pipelined response path without arbitration as in [55, 113]. The worst-case latency of a memory client c in the NoC (in ns), $\hat{L}_{i,c}$, to transport a read request from a source to a destination NI and read back the response consisting of N_c service units is given by Equation (2.6), where f_i is the NoC operating frequency (in MHz), $\Theta_{i,c}$ is the service latency in the global TDM arbitration, $\rho'_{i,c}$ the rate allocated to the client in the NoC, n_{hops} the number of hops between the source and destination NI, and δ_p the fixed pipeline delay of a NoC router [16]. For a write request, only the request path need to be considered, and hence, there will be a single $n_{hops} \cdot \delta_p$ as given by Equation (2.7). Note that we assume no self-interference/back-pressure and no flow control considering sufficiently large buffers after the atomizers (in front of the memory controller), and equal-length request and response paths.

$$\hat{L}_{i,c}(RD) = \frac{(\Theta_{i,c} + \lceil N_c/\rho'_{i,c} \rceil) \cdot SC_i^{cc} + 2n_{hops} \cdot \delta_p}{f_i} \tag{2.6}$$

$$\hat{L}_{i,c}(WR) = \frac{(\Theta_{i,c} + \lceil N_c/\rho'_{i,c} \rceil) \cdot SC_i^{cc} + n_{hops} \cdot \delta_p}{f_i} \tag{2.7}$$

Decoupled Memory Interconnect

TDM NoCs and the memory controller with a bus-based memory interconnect in front of it are used in conjunction, with the NoC connected in a tree-like structure and the memory controller at the root of the tree as shown in Figure 2.6. Since the NoC and the memory controller run in different clock domains, the virtual circuit for each client in the NoC needs to be decoupled using dedicated buffers in the memory controller clock domain. Then, a bus-based interconnect with a predictable arbitration policy is used to schedule the requests in different buffers to the memory controller. Moreover, the NI, which interfaces with the memory controller, requires a separate port for each virtual circuit increasing its area and power consumption. Note that this is the current setup in *CompSOC* [15, 57, 122, 49], which is a hardware platform for real-time systems.

In such a *decoupled architecture*, the worst-case latency of a memory request can be computed as the sum of worst-case latencies of the memory interconnect (Equation (2.6)) and the bus-based arbitration stage and memory controller (Equation (2.2)). It can be seen that this approach could introduce a large overhead in terms of worst-case latency, since there are multiple arbitration stages for the same request and each of them introduces a service latency in the worst-case latency estimation.

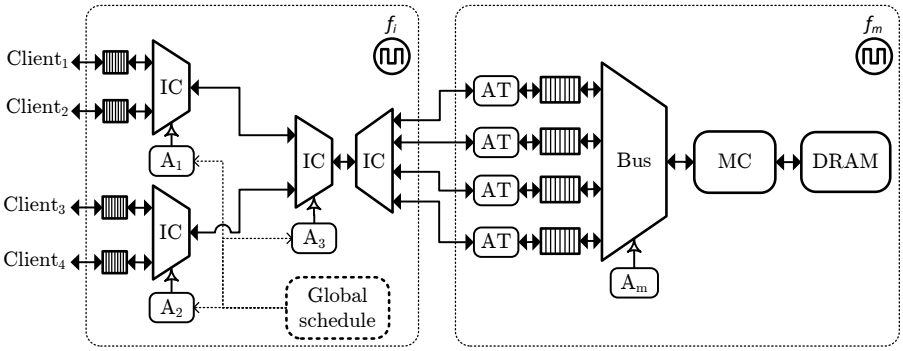


Figure 2.6: High-level architecture of a decoupled distributed memory interconnect and a memory controller. The interconnect and the memory controller with the bus-based arbiter run in different clock domains using clock frequencies f_i and f_m, respectively. Every arbitration stage (A_n) in the interconnect makes scheduling decisions according to a single global schedule.

2.5 Summary

A DRAM typically consists of multiple memory banks with each bank consisting of several storage elements organized in the form of a matrix-like structure. The contents in the memory are accessed by issuing different memory commands with strict timing constraints. The contents stored in the memory elements are first copied into a row buffer before a read/write operation is performed. There are plenty of DRAM device options available in the market, of different interface widths, operating frequencies, memory capacities, banks, and memory channels. With every memory generation, the memory capacities and operating frequencies are increased for the standard DRAMs, while the supply voltage is reduced for mobile DRAMs. The ever increasing demands for main memory bandwidth while keeping the power budget to the minimum led to the introduction of memories with multiple memory channels and wider interfaces, such as WideIO DRAM.

Existing real-time memory controllers bound the execution time of a memory request by fixing the memory access parameters of a request, such as burst length and number of read/write commands, at design time, which define the access granularity of the memory controller. For a fixed access granularity, real-time memory controllers use a fixed memory command schedule which bounds the worst-case execution time of a read/write request and also allows the worst-case bandwidth offered by a memory to be computed.

Predictable arbitration policies, such as Time Division Multiplexing (TDM), Round Robin (RR), Frame-Based Static Priority (FBSP), Priority-Based Scheduler (PBS) and Credit-Controlled Static-Priority (CCSP), are used to provide firm real-time guarantees to clients sharing a single memory resource (DRAM) between the multiple memory clients in multi-core real-time systems. The Latency-Rate ($\mathcal{LR}$) *server* model can be used to analyze real-time memory subsystems, which

can be used in formal performance analysis based on e.g., network calculus or data-flow analysis. $\mathcal{LR}$ is an abstraction that captures the worst-case behavior of a resource shared by any predictable arbiter with two parameters.

TDM NoCs connected in a tree-like structure are used as the memory interconnect to provide real-time guarantees to the memory clients. However, they need to be decoupled from the memory controller using dedicated buffers as they run in different clock domains.

Chapter 3

Scalable Memory Subsystem Architecture

Scalable memory subsystem architectures are essential to support the ever growing memory bandwidth demands of future real-time systems. Scalability in terms of clock frequency is achieved by either a local or global arbitration policy implemented in a distributed way, i.e. arbitration is pipelined in a distributed implementation in the memory interconnect. Distributed memory interconnects with local arbitration policies, i.e. *locally arbitrated*, have decoupling buffers between every arbitration stage which incur a large area, power and latency overhead. Although there exists distributed memory interconnects with global TDM arbitration, i.e. *globally arbitrated*, they are decoupled from the memory controller due to the different clock domains of interconnect and memory controller. This increases area, power and latency with the number of clients. Moreover, TDM arbitration is not suitable in systems where the clients have diverse requirements in terms of memory bandwidth and latency [14]. Scalability of the memory subsystem in terms of number of memory channels requires a memory controller that can be configured to interleave memory client requests across the memory channels with different interleaving granularities and rates according to the client specifications for the efficient utilization of the multi-channel memory.

This chapter starts with our proposed generic distributed globally arbitrated memory tree (GAMT) in Section 3.1 that can be configured with five different arbitration policies supporting work conservation. Section 3.2 presents a coupled memory interconnect (CMI) architecture that couples a globally arbitrated distributed memory interconnect with the memory controller reducing area, power

and latency. For scalability of the memory subsystem in terms of number of memory channels, we present multi-channel memory controller (MCMC) in Section 3.3 including a novel method for logical-to-physical address translation that allows clients to be mapped across multiple memory channels with different interleaving granularities. Finally, in Section 3.4, the performance evaluations of the proposed CMI and GAMT against the state-of-the-art approaches and functional verification of the MCMC are presented.

3.1 Generic Distributed and Globally Arbitrated Memory Tree (GAMT)

This section presents our proposed scalable memory tree (GAMT) [47] that can be configured with five different arbitration policies and supports work conservation for improving average-case performance by distributing slack. Before we present the detailed architecture and operation of GAMT, we first discuss the novel concept by which we achieve scalability, global arbitration and genericness in GAMT.

Scalability

We propose a distributed architecture, as shown in Figure 3.1, with dedicated *Accounting* and *Priority assignment* logic for each client and *Priority resolution* among the N clients using a tree consisting of N-1 pipelined multiplexer stages. The Accounting logic keeps track of the eligibility status of a client to get service, the Priority assignment logic assigns a unique priority to the client based on the arbitration scheme and whether or not the client is eligible, and the Priority resolution grants service to the client with the highest priority. Once a client is granted service, a feedback signal from the output of the Priority resolution logic updates the client's eligibility status in its Accounting logic. Also, the eligibility status of all clients is updated every scheduling interval. The use of pipelined multiplexer stages for priority resolution breaks the critical path and enables the logic to be synthesized at higher clock frequencies. However, this increases the time between selecting of a client and the update of the eligibility status, which sets a minimum limit for the scheduling interval length.

Global Arbitration

The Accounting logic of all clients use a *global scheduling interval* of fixed duration determined by the fixed access duration of the memory controller and the pipeline depth of the multiplexer stages (which depends on the number of clients in the system) in Priority resolution. Since the scheduling decision is made at the Accounting logic at the leaves of the tree, the pipeline registers in the multiplexer

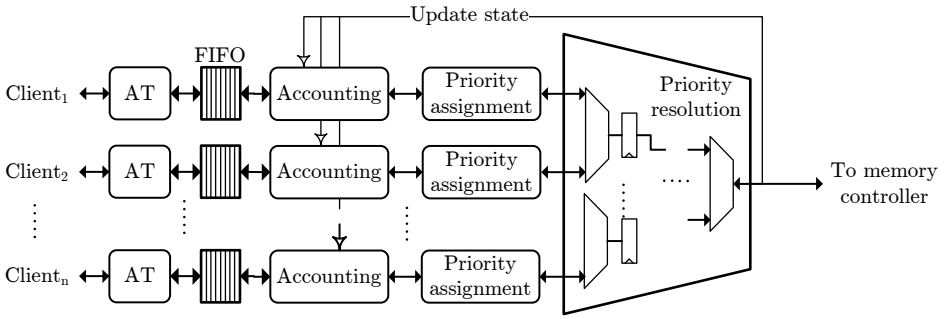


Figure 3.1: A generic, distributed and globally arbitrated memory interconnect architecture.

tree are simple registers of width equal to the data-path width, unlike the flit-sized buffers at every arbitration stage in the existing distributed implementations.

Generalization

Several predictable arbiters can be realized by configuring the Accounting logic. In TDM and RR, the responsibility of the Accounting logic is to keep track of the current slot, which essentially is the deciding factor for a client to get service. In FBSP, PBS, and CCSP the Accounting logic keeps track of the budget of the client. The priority level assigned to an eligible client by the Priority assignment logic is based on the arbiter configuration, which guarantees a minimum bandwidth and/or a maximum latency according to the $\mathcal{LR}$ model. In TDM and RR, there can only be one eligible client at a time, and hence, the highest priority is assigned to the client that is statically assigned to the slot. For FBSP, PBS and CCSP, the priority levels that are computed at design time to meet a certain bandwidth/latency requirement [14] are assigned to the eligible clients. At run time, multiple eligible clients may be eligible and enter the tree. Note that for slack management in work-conserving mode, i.e. when none of the eligible clients are backlogged, the backlogged non-eligible clients are assigned with unique priorities that are lower than the lowest priority level assigned to an eligible client. The priority levels in the work-conserving mode depends on the slack management policy, which could be the same or different from the regular arbitration policy.

3.1.1 Detailed GAMT Architecture and Operation

Given that we have presented the high-level concept of GAMT, now we proceed with the detailed GAMT architecture and its operation.

GAMT Architecture

Figure 3.2 shows the detailed architecture of GAMT in which the clients are at the leaves of the tree and the memory controller and DRAM at the root. The Accounting and Priority Assignment (APA) logic for each client is located in the network interface (NI) to which the client is attached. Note that we show a NI to emphasize that GAMT can be implemented as a NoC. The 2-to-1 multiplexers (Mux) implementing the priority resolution are interconnected in a tree-like structure with a NI (NI_d) at the root of the tree which interfaces with the memory controller. We assume a real-time memory controller with a fixed service unit size, as discussed in Section 2.2.

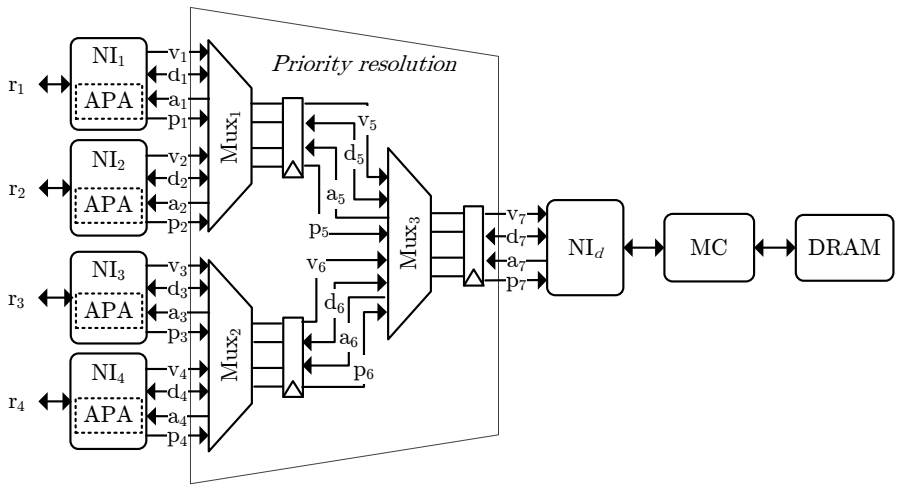


Figure 3.2: Detailed architecture of GAMT along with the memory shared by four clients $c_1 - c_4$. The APA logic for each client is located in their network interfaces (NI). The valid signal used to indicate a new request is denoted by v, the data/command lines d, priority lines p and the acknowledgement signal a. The multiplexer (Mux) stages perform the priority resolution by selecting the client with the highest priority. The APA status is updated using the acknowledgement signal.

The incoming memory requests from a client is split into equal-sized service units with a service cycle duration of SC^{cc} by the atomizer at the NI (not shown in the figure) according to the fixed access size of the memory controller. Each service unit is then scheduled by the arbitration policy at fixed scheduling intervals (SI). When an eligible request is scheduled, the request valid (v) signal is asserted and the data/command (d) and the priority (p) of the client are transmitted over the bus. When two inputs of the multiplexer stage arrive at the same clock cycle, the one which carries the highest priority is granted access and the other is dropped. Note that the notion of a global scheduling interval makes this possible. When a service unit arrives at the root, NI_d generates an acknowledgement (a) signal that is sent back to the client, which removes the request from the head of its

request queue and the current state of the Accounting logic is updated (details are presented later in Section 3.1.2). The dropped service units are not removed from their request buffers (no acknowledgement) and they are re-scheduled during the next SI. Note that if the request of an eligible client is dropped during a scheduling interval, the client remains eligible in the next scheduling interval. One drawback of this approach is that dropping and rescheduling requests could increase the switching activity, and hence, the power consumption.

It can be seen that for functional correctness, the minimum SI duration (SI_{min}), must at least be equal to or greater than the total time when a request is scheduled until its acknowledgement arrives back at the source NI. This depends on the number of multiplexer stages in the tree, which in turn depends on the number of clients in the system. For a balanced tree, this constraint is given by $SI_{min} \geq 2 \times \log_2(C)$, where C is the number of memory clients since each multiplexer stage introduces one cycle delay in both the request and response paths. The WCET for a memory read and write request is then given by $2 \times \log_2(C) + SC^{cc}$ and $\log_2(C) + SC^{cc}$, respectively. For a 16-bit IO DDR3-800 memory device, the SC^{cc} for the smallest request size of 16 Bytes is 25 clock cycles [91] assuming a *close-page policy* [14]. If we assume that GAMT runs at the same clock frequency as the memory, the minimum SI of 12 cycles for up to 64 clients is less than the SC^{cc} for the smallest request size. Hence, the memory bandwidth is not negatively impacted due to the pipeline delays in GAMT. Moreover, with larger request sizes and faster memories the WCET of requests in clock cycles increases making this constraint insignificant. Note that GAMT may not be suitable for SRAMs where data can be accessed in a single clock cycle.

For a read request, the response arrives back at the source on a pipelined response path. In this section, we assume the same clock domain and data-path width for both GAMT and the memory controller to ensure that their SIs are of the same duration. Hence, the buffer in the memory controller does not overflow as the service unit (if any) scheduled by the tree will be consumed by the memory during the same SI. However, it is possible to have different data-path widths for the memory tree and the controller and run them at different speeds by coupling the GAMT and memory controller, as proposed later in Section 3.2. During the periodic DRAM refresh operation in the memory controller, the service unit arriving at the root is dropped and rescheduled again. The refresh duration need not be an integer multiple of service cycle duration and the pending request will be served immediately in the first service cycle after the refresh operation is finished. Note that the impact of refresh needs to be taken into account for the worst-case memory bandwidth and latency computation [14].

Operation

Figure 3.3 shows an example timing behavior of the GAMT when there are pending requests to be scheduled in the FIFOs of NI_1 and NI_3 from clients c_1 and c_3, respectively (irrelevant signals are omitted for clarity). We consider a SI duration

of 7 clock cycles in this example. At the beginning of the first SI (grey vertical lines), the APA logic in NI_1 and NI_3 assert the valid signals, v_1 and v_3, and the data/command of the requests are issued on d_1 and d_3, respectively. We assume that client c_1 has higher priority than c_3 and their priorities are sent over p_1 and p_3, respectively (not shown in Figure 3.3). Since there are no pending requests in NI_2 and NI_4, the multiplexers Mux_1 and Mux_2 grant access to both requests arriving from NI_1 and NI_3, respectively. The requests arrive at Mux_3 after a delay of one clock cycle introduced by the first multiplexer stage. However, Mux_3 grants access to the request arriving from NI_1 since it has the highest priority, and the request from NI_3 is dropped. Once the root NI receives the valid signal on v_7, it sends back an acknowledgement on a_7 after one clock cycle delay as shown. The acknowledgement is sent back to the source NI_1 over a fully-pipelined response path and arrives back at NI_1 after three clock cycles. The request is then removed from the head of the FIFO in NI_1 and the APA status is updated. In the next scheduling interval, NI_3 reschedules the dropped request as it did not receive an acknowledgement.

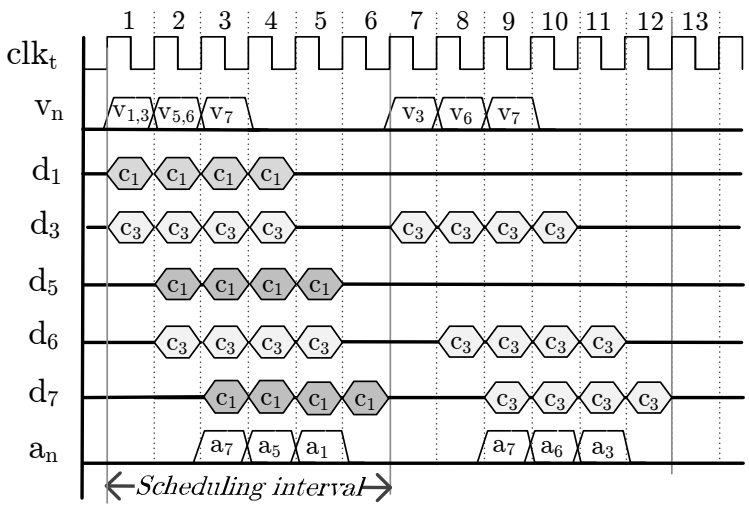


Figure 3.3: Example timing diagram showing scheduling of write requests from clients c_1 and c_3. All valid and accept signals are combined and shown together as (v_n) and (a_n), respectively.

3.1.2 APA Architecture and Configuration

In this section, first the proposed generic RTL architecture of the Accounting and Priority assignment (APA) logic is presented and then we show how it can be configured to operate as either TDM, RR, FBSP, PBS or CCSP, which typically are suitable to share the DRAM resource in real-time systems.

The RTL architecture of the proposed generic APA logic is shown in Figure 3.4. In the NI, the atomizer splits an incoming request into smaller service units (corresponding to the fixed request size in a real-time memory controller) and the FIFO buffer stores all pending service units from a memory client. Work-conserving mode of the arbitration policy is enabled by setting the register WC to one, which enables the data valid signal (v) to be asserted whenever there is a request pending in the FIFO. Note that in the work conserving mode, the priority level of the memory client will be lower than its any priority level in the non work conserving mode. Work conservation is disabled by setting WC to zero, which means the valid signal is asserted only when a client is eligible (has enough budget) to get service and is backlogged.

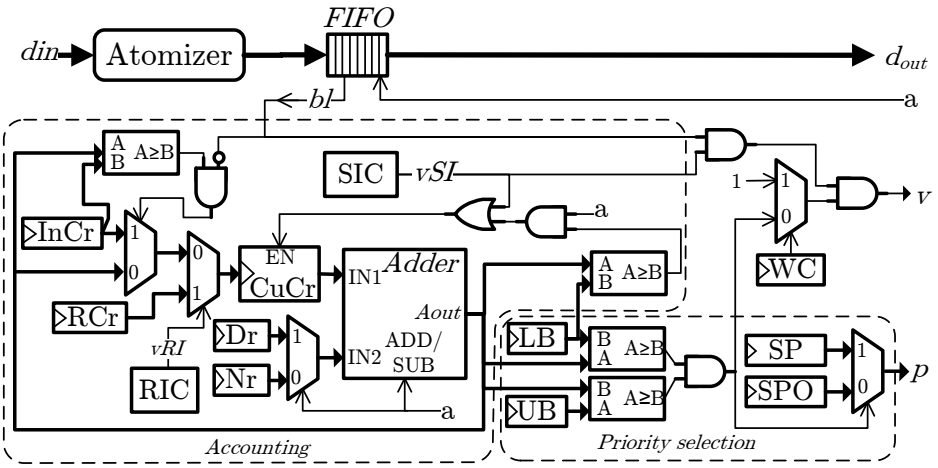


Figure 3.4: Generic APA architecture that can be configured to operate as either TDM, RR, FBSP, PBS or CCSP arbitration.

Algorithm 3 shows the logical operation of the Accounting and Priority assignment blocks[1]. In Accounting, the value in register Current credits (CuCr) is incremented by the value in the register Numerator (Nr) at every SI (line 8). The SI counter (SIC) asserts a valid signal, *vSI*, indicating the start of every new SI. Addition is performed using a full-adder, Adder, with one of its inputs connected to CuCr and the second input to Nr when it is in addition (ADD) mode. The Adder is in the ADD mode by default and the subtract (SUB) mode is enabled when the acknowledgement signal is valid. Note that CuCr is updated once every SI, since it is enabled (EN) when *vSI* is asserted. As explained in Section 2.3, the building up of budget in CCSP mode is not allowed when the client is not backlogged. Hence, when *bl* is not asserted, the value in CuCr is set to the initial

[1]For clarity in presentation, the pseudo code is split into two procedures, Accounting and Priority assignment. Accounting is triggered by signals acknowledgement (a) and backlogged (bl) signals, whereas Priority assignment is purely combinatorial logic.

budget stored in register Initial credits (InCr) using the multiplexer logic which selects InCr when the output of the Adder is greater than or equal to the initial budget (lines 3-4). On a valid acknowledgement signal, CuCr is decremented by the value in register Denominator (Dr) (lines 10-11). Note that the value of *Aout* returned (line 13) at the end of the procedure is passed to the Priority assignment block and also used as the next state value of CuCr during the next iteration of the Accounting procedure. The RI counter (RIC) is used by frame-based arbitration policies to replenish the budget every replenishment interval by asserting *vRI* which causes CuCr to be reset to the value in RCr (line 6).

Algorithm 1 Accounting and priority assignment logic

Input signals: Acknowledgement (a), Backlogged (bl)
Output signal: Priority (p)

```
1:  procedure ACCOUNTING(a, bl)
2:      if vSI then
3:          if (!bl) & (Aout ≥ InCr) then
4:              CuCr ← InCr
5:          else if vRI then
6:              CuCr ← RCr
7:          else
8:              CuCr ← CuCr + Nr
9:          end if
10:     else if (a) & (Aout ≥ LB) then
11:         CuCr ← CuCr - Dr
12:     end if
13:     return Aout
14: end procedure

15: procedure PRIORITY ASSIGNMENT(Aout)
16:     if LB ≤ Aout ≤ UB then
17:         p ← SP
18:     else
19:         p ← SPO
20:     end if
21:     return p
22: end procedure
```

The Priority assignment logic selects a priority level stored in the register Static priority (SP) when the value of the Adder output, *Aout*, falls in between the values stored in registers Lower bound (LB) and Upper bound (UB) (lines 16-17). A different priority level with a constant offset as configured in the register SP plus offset (SPO) is selected (lines 18-19) in work conservation mode, i.e, when

the client is not eligible to get service in the current SI such that SP has a higher priority than SPO. The value of the offset needs to be selected according to the slack management policy, discussed in Section 2.3. When a client is scheduled in work-conserving mode, no credits are deducted from its budget, and to ensure this its current budget level is checked against the sufficient budget limit in LB before enabling CuCr (line 10). Note that TDM does not have the notion of budget for the clients. In FBSP and CCSP, all non-eligible clients, i.e. with budget less than LB, are in work-conserving mode by definition.

3.1.3 APA Configurations

A summary of the different programmable registers in APA and the initial values that need to be configured to implement the different arbitration policies are shown in Table 3.1, which we will discuss in detail in this section.

Table 3.1: APA programmable registers and their configuration for TDM/RR, FBSP/PBS and CCSP arbitration policies.

	Arbitration policy		
Register	*TDM/RR*	*FBSP/PBS*	*CCSP*
InCr	f	$f \cdot \rho$	$\sigma \cdot dr$
CuCr	0	$f \cdot \rho$	$\sigma \cdot dr$
RCr	0	$f \cdot \rho$	Not used
Nr	1	0	nr
Dr	0	1	dr
SP	Unique for each client	Unique for each client	Unique for each client
SPO	SP + Offset	SP + Offset	SP + Offset
UB	End slot in TDM frame	$> f \cdot \rho$	High value
LB	Start slotp in TDM frame	1	$nr - dr$
SIC	SI	SI	SI
RIC	$f \cdot SI$	$f \cdot SI$	Not used

TDM and RR

When configured in TDM mode, the Accounting logic keeps track of the progress of the current frame in terms of number of slots and the Priority assignment logic sets the priority of a client to the highest value available during its allocated slot(s) in the frame. Note that the GAMT supports only TDM with contiguous slot allocation strategy. In Accounting, CuCr is initialized to zero and is incremented by one every SI by configuring Nr with a value of one, which basically keeps track of the current slot in the frame. To identify the start of a new frame, the RIC is configured to assert *vRI* every frame. This resets CuCr to zero by loading the value from RCr which needs to be initialized to zero to restart counting the slots for the new frame. In TDM, there is no budgeting required, and hence, the value in Dr is initialized to zero so that *ack* does not affect the value in CuCr as it switches the Adder to SUB mode. Note that RR is a special case of TDM where

only one slot is allocated to each client, i.e. the frame length is equal to the total number of clients.

In Priority selection, LB needs to be configured with the starting slot number of the client in the frame and UB with the ending slot number according to the continuous number of slots allocated to the client in the frame. In non-work-conserving mode, the priority level of clients configured in SP does not matter as only a single client schedules its request in a scheduling interval. For operation in work-conserving mode, we need to assign unique priority to each client in SP such that there is no conflict of priorities when SPO is selected, i.e. the priority levels of all clients in SPO must be less than in SP. For example, when the slack management policy is such that the average-case performance of bandwidth-demanding clients needs to be increased, as explained in Section 2.3.2, SPO can be assigned priority levels in descending order starting from the client with largest bandwidth requirement. Note that InCr is not used in TDM mode, but is configured to the maximum value of f to ensure that CuCr is not updated from InCr.

Consider three memory clients, c_1, c_2 and c_3, allocated in a TDM frame of size five, as shown in Figure 3.5. Table 3.2 shows the initial values that need to be set to the different programmable registers for the three clients. We have used a constant offset value of 10 to configure SPO. Table 3.3 shows the values of CuCr and the priority selected at different scheduling intervals for the three clients. In general, it can be seen that the highest priority is assigned to a client during its slot in a TDM frame according to the schedule shown in Figure 3.5. Note that the acknowledgment signal does not have any impact on the values of CuCr and priority as Dr is set to zero.

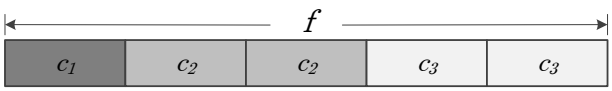


Figure 3.5: Example TDM slot allocation for clients c_1, c_2 and c_3.

Table 3.2: Initialization of the programmable registers for clients c_1 - c_3 using TDM.

Register	c_1	c_2	c_3
CuCr	0	0	0
RCr	0	0	0
Nr	1	1	1
Dr	0	0	0
SP	1	2	3
SPO	11	12	13
UB	1	3	5
LB	1	2	4

Table 3.3: Example TDM operation: values of CuCr and priority lines every scheduling interval of clients c_1 - c_3.

SI #	CuCr(c_1)	CuCr(c_2)	CuCr(c_3)	p(c_1)	p(c_2)	p(c_3)
1	0	0	0	1	12	13
2	1	1	1	11	2	13
3	2	2	2	11	2	13
4	3	3	3	11	12	3
5	4	4	4	11	12	3
6	0	0	0	1	12	13

FBSP and PBS

In FBSP and PBS modes, the Accounting logic keeps track of the current budget of a client in terms of number of slots in a frame of size f, and the Priority assignment logic sets the priority level of the client on the priority lines as long as sufficient budget is available. At the start of every frame, CuCr is initialized with $f \cdot \rho'_c$, which corresponds to the number of slots allocated to the client c in a frame, i.e, the maximum budget. The current budget needs to be decremented by one whenever a service unit gets scheduled, i.e. when an acknowledgement arrives back, and hence, Dr is configured with one. To replenish the budget at the start of every new frame, the RIC enables the multiplexer logic to update the initial budget from RCr to CuCr at the end of every frame. Note that Nr is set to zero as it is not used for budget replenishment. SP needs to be configured with the priority (determined at design time to meet the latency requirements) of the client and SPO with a constant offset. For example, consider the case when SP is assigned with values starting from zero, being the highest priority, and upwards with decreasing priority. As explained in Section 2.3.2, if the slack management policy requires the same relative priorities as SP for all the clients, a constant offset needs to be added to SP to get SPO such that the priority levels in SPO of all clients are lower than in SP. Note that SP is used while within budget and otherwise SPO is used, which ensures that the allocated bandwidth is guaranteed to the clients before the slack bandwidth is distributed among them. LB needs to be configured with a value of one and UB with a value greater than $f \cdot \rho'_c$ such that the priority in SP is selected for a number of service units equal to $f \cdot \rho'_c$ in a frame. Note that InCr is not used in FBSP mode, but is set to a maximum value of $f \cdot \rho'_c$ to avoid initialization of CuCr from InCr.

Consider three memory clients c_1, c_2 and c_3, with c_1 having the highest priority and c_3 the lowest, and rates $\rho'_1 = 1/5$, $\rho'_2 = 2/5$ and $\rho'_3 = 2/5$, respectively, allocated to them with an FBSP arbiter frame of size five. Table 3.4 shows the initialization values of the different programmable registers for the three clients. We have selected a constant offset value of 10 to configure SPO. The numbered lines in first column of Table 3.5 shows the values of CuCr at the start of every scheduling interval and the priority selected for the three clients. The successive

line shows the updated values of CuCr and the priority line when an acknowledgement of a client arrives back. Note that we assume all clients are always backlogged in this example. It can be seen that the budget of a client is decremented by one whenever it gets scheduled successfully, i.e. when it receives an acknowledgement. Moreover, the priority level in SP is selected when a client have sufficient credits (i.e. more than one) and SPO otherwise.

Table 3.4: Initialization of the programmable registers for clients c_1 - c_3 using FBSP.

Register	c_1	c_2	c_3
CuCr	1	2	2
RCr	1	2	2
Nr	0	0	0
Dr	1	1	1
SP	1	2	3
SPO	11	12	13
UB	3	3	3
LB	1	1	1

Table 3.5: Example FBSP operation: values of CuCr and priority lines every scheduling interval of clients c_1 - c_3.

SI #	a	CuCr(c_1)	CuCr(c_2)	CuCr(c_3)	p(c_1)	p(c_2)	p(c_3)
1	-	1	2	2	1	2	3
-	c_1	0	2	2	11	2	3
2	-	0	2	2	11	2	3
-	c_2	0	1	2	11	2	3
3	-	0	1	2	11	2	3
-	c_2	0	0	2	11	12	3
4	-	0	0	2	11	12	3
-	c_3	0	0	1	11	12	3
5	-	0	0	1	11	12	3
-	c_3	0	0	0	11	12	13
6	-	1	2	2	1	2	3

CCSP

In CCSP mode, the Accounting logic keeps track of the current budget level of a client based on a continuous replenishment policy and the Priority assignment logic sets a higher priority for the client on the priority lines based on its current budget. Each client is initialized with an initial budget of $\sigma \cdot dr$ in CuCr and InCr. The budget stored in CuCr is replenished by incrementing at a rate of nr, configured in Nr, every SI and depleted by subtracting dr, configured in Dr, when an acknowledgement arrives back, where nr and dr are integers used to represent the allocated rate, $\rho = nr/dr$. In the Priority assignment logic, SP and SPO are

configured with the client's priority level and with a constant offset, respectively. LB is set to dr as $dr - nr$ is the minimum budget required to select SP and that $Aout$ is CuCr+nr at the beginning of every SI which determines the priority level. UB needs to be set to a sufficiently large value such that it is larger than the maximum budget that can ever built up, which is bounded in [20].

Consider three memory clients, c_1, c_2 and c_3, with c_1 having the highest priority and c_3 the lowest, rates $\rho_1 = 1/4$, $\rho_2 = 1/5$ and $\rho_3 = 2/7$, and burstiness $\sigma_1 = 1$, $\sigma_2 = 1$ and $\sigma_3 = 2$, respectively, allocated using CCSP arbitration. Table 3.6 shows the initialization of different programmable registers for the three clients. We have selected an offset value of 10 to configure SPO. Table 3.7 shows the values of CuCr and the priority levels selected for the three clients, at different scheduling intervals and when the acknowledgement arrives back. For simplicity, we assume all clients are always backlogged in this example. It can be seen that the priority level in the SP is selected when a client have sufficient credits and SPO otherwise. We can also see that the credits of c_2 and c_3 increase beyond their initial credits until they get scheduled, which is valid in CCSP arbitration scheme as long as the clients are backlogged. The continuous replenishment scheme of CCSP allows a high-priority client to interfere more with low-priority clients since they may build up new credits before the low-priority client depletes its budget. This behavior can be seen when both c_1 and c_2 replenish and interfere again with c_3 (SI 4 & 5) before it depletes its budget. Note that this cannot happen in FPSP, where replenishment is based on frames. As a result, CCSP gives lower latency to high-priority clients and higher latency to low-priority clients [14].

Table 3.6: Initialization of the programmable registers for clients c_1 - c_3 using CCSP.

Register	c_1	c_2	c_3
CuCr	4	10	14
InCr	4	10	14
Nr	1	1	2
Dr	4	5	7
SP	1	2	3
SPO	11	12	13
UB	100	100	100
LB	4	5	7

3.2 Coupled Memory Interconnect (CMI)

In this section, we first introduce the architecture of the proposed coupled memory interconnect (CMI) that can be used to couple a globally arbitrated memory interconnect with the memory controller. Then, a methodology to compute the interconnect parameters for a CMI and the worst-case guarantees provided by the CMI is presented.

Table 3.7: Example CCSP operation: values of CuCr and priority lines every scheduling interval of clients c_1 - c_3.

SI #	a	CuCr(c_1)	CuCr(c_2)	CuCr(c_3)	p(c_1)	p(c_2)	p(c_3)
1	-	4	10	14	1	2	3
-	c_1	0	10	14	11	2	3
2	-	1	11	16	11	2	3
-	c_2	1	6	16	11	2	3
3	-	2	7	18	11	2	3
-	c_2	2	2	18	11	12	3
4	-	3	3	20	1	12	3
-	c_1	0	3	20	11	12	3
5	-	1	4	22	11	2	3
-	c_3	1	0	22	11	12	3
6	-	2	1	15	11	12	3

3.2.1 Architecture

The basic idea is to use a single arbitration policy for the memory subsystem, in the memory interconnect, without the decoupling buffers and the bus-based interconnect in front of the memory controller. The interconnect and memory controller are treated as a single (deeply) pipelined resource. For this, a single clock source is used and the different clock frequencies for both the interconnect and the memory controller are generated from the same clock source, which helps to align their clock edges at the service cycle boundaries. This enables us to schedule the service units to the memory controller using the single global schedule without the decoupling buffers and the bus-based arbiter in front of the memory controller. The arbiter works on a single service unit size in the interconnect and the memory controller. This implies that the memory service cycle must be equal to the interconnect service cycle (in ns), i.e., $SC^{ns} = SC_i^{ns}$. This allows the service units to be scheduled at fixed scheduling intervals (SI).

Figure 3.6 shows the high-level architecture of a TDM NoC coupled with a memory controller, shared by five memory clients. The clocks for the NoC and the memory controller represented by clk_i (f_i) and clk_m (f_m), respectively, are derived from a single clock source, clk_s, by dividing in different ratios, m and n. The memory controller is a real-time memory controller, as discussed in Section 2.2, attached to a memory operating at a frequency (in MHz) given by f_m. Compared to the decoupled architecture, previously shown in Figure 2.6, the decoupling buffers and the atomizer in the memory controller domain are moved to the leaves of the memory interconnect tree. Since the service units are scheduled using the global arbitration in the interconnect, the bus and the arbiter in front of the memory controller are not required anymore. The NI attached to the memory controller needs only a single port instead of a number of ports equal to the number of clients in the decoupled architecture. Since there are no decoupling buffers in the memory controller clock domain, the NI requires a

buffer of size equal to the service unit. This is because we assume that the memory controller does not serve the request until the complete service unit (including the payload of a write request) is available in its input.

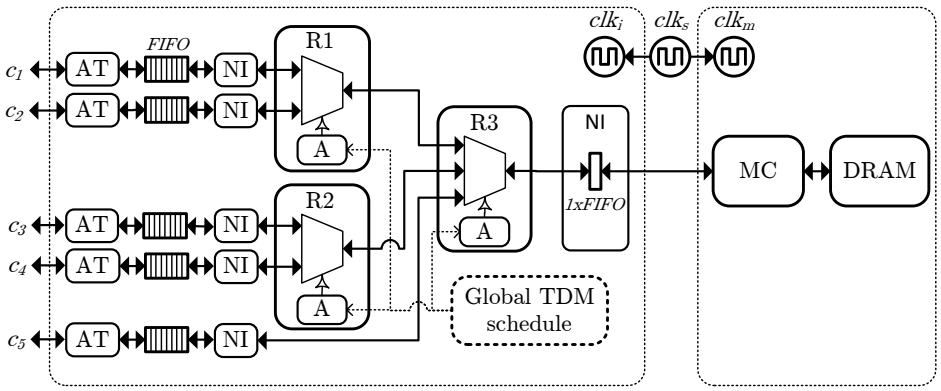


Figure 3.6: High-level architecture of a coupled interconnect (TDM NoC) with a memory controller shared by five memory clients c_1 to c_5. R1, R2 and R3 represent the routers of the NoC.

3.2.2 Operation

Consider an example memory subsystem with five memory clients c_1 to c_5, and a TDM NoC used as the memory interconnect coupled to the memory controller, shown in Figure 3.6, with three routers R1, R2, and R3, respectively. Let us assume a global TDM schedule with allocation for the five clients according to Figure 3.7, where each client is allocated one slot in a frame of size five. Figure 3.8 shows the clock-cycle-level behavior of the interconnect. For simplicity, we assume $SC^{cc} = 4$ cycles and $SC_i^{cc} = 6$ cycles and $f_i = \frac{3}{2} f_m$, i.e., three clock cycles of clk_i correspond to two clock cycles of clk_m. In the first scheduling interval (SI 1), router R1 schedules the service unit of client c_1 and the first word arrives at router R3 after a pipeline delay of R1 equal to δ_p (one clock cycle in this example). Router R3 immediately forwards the service unit to the memory controller after introducing another pipeline delay, and hence, the service unit of c_1 arrives after $2\delta_p$ cycles. The memory controller has to wait for one service cycle for the complete service unit to be delivered by the NoC. Once the memory controller issues a read/write command, the internal pipeline stages in the memory controller introduce a delay of δ_m cycles after which the command reaches the memory. Note that δ_m can be several clock cycles depending on the number of pipeline stages in the memory controller. In our memory controller, δ_m was around 20 clock cycles [53].

After the first read command of a request is issued to the memory, the response data comes out after a delay of several clock cycles, δ_t, which depends on the tRL

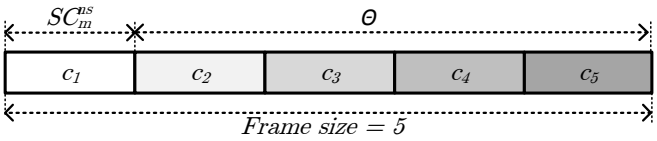


Figure 3.7: Global TDM allocation for clients c_1 to c_5.

and *tRCD* timing constraints of the memory type, as explained in Section 2.1. The response path is fully pipelined and there is no flow control as in the channel trees proposed in [55]. Hence, the memory controller sends back a read response as soon as it buffers the complete data (in the memory controller FIFO) from the memory device. As shown in Figure 3.8, the response for the service unit of c_1 is completely buffered in the memory controller after δ_x, which takes $BL/2$ clock cycles for a single burst in a double data-rate memory. However, at this point the end of the service cycle may not be aligned with the NoC clock and hence, it has to wait for the next rising edge of the clock before the response can be forwarded back to the client through the routers. Similarly, clients $c_2 - c_5$ are scheduled during the successive scheduling intervals $SI\ 2 - SI\ 5$, respectively, as shown in the figure.

3.2.3 Bandwidth Matching

To ensure that the memory interconnect delivers a complete service unit in a service cycle for a given memory operating at frequency f_m and a service unit size of SU^{bytes} with a service cycle duration SC^{cc}, the interconnect link bandwidth must be same as the memory controller in a service cycle, as given by Equation (3.1). The left-hand side of the equation corresponds to the bandwidth of the interconnect link during an interconnect service cycle considering the communication overhead, δ_{ov}, of the request (e.g. flit header in NoC as explained in Section 2.4) and right-hand side the bandwidth of the memory controller interface in a memory service cycle. In addition, we need to make sure that the NoC and the memory controller clocks are aligned on the boundary of the service cycle. In order to satisfy this constraint, the clock frequencies of the interconnect and the memory controller need to be selected according to Equation (3.2). Intuitively, this means a memory service cycle is equal to the interconnect service cycle in *ns*.

$$f_i \times (IW_i/8) \times \frac{SC_i^{cc} - \delta_{ov}}{SC_i^{cc}} = f_m \times \frac{SU^{bytes}}{SC_m^{cc}} \tag{3.1}$$

$$SC_i^{cc} \times f_i = SC_m^{cc} \times f_m \tag{3.2}$$

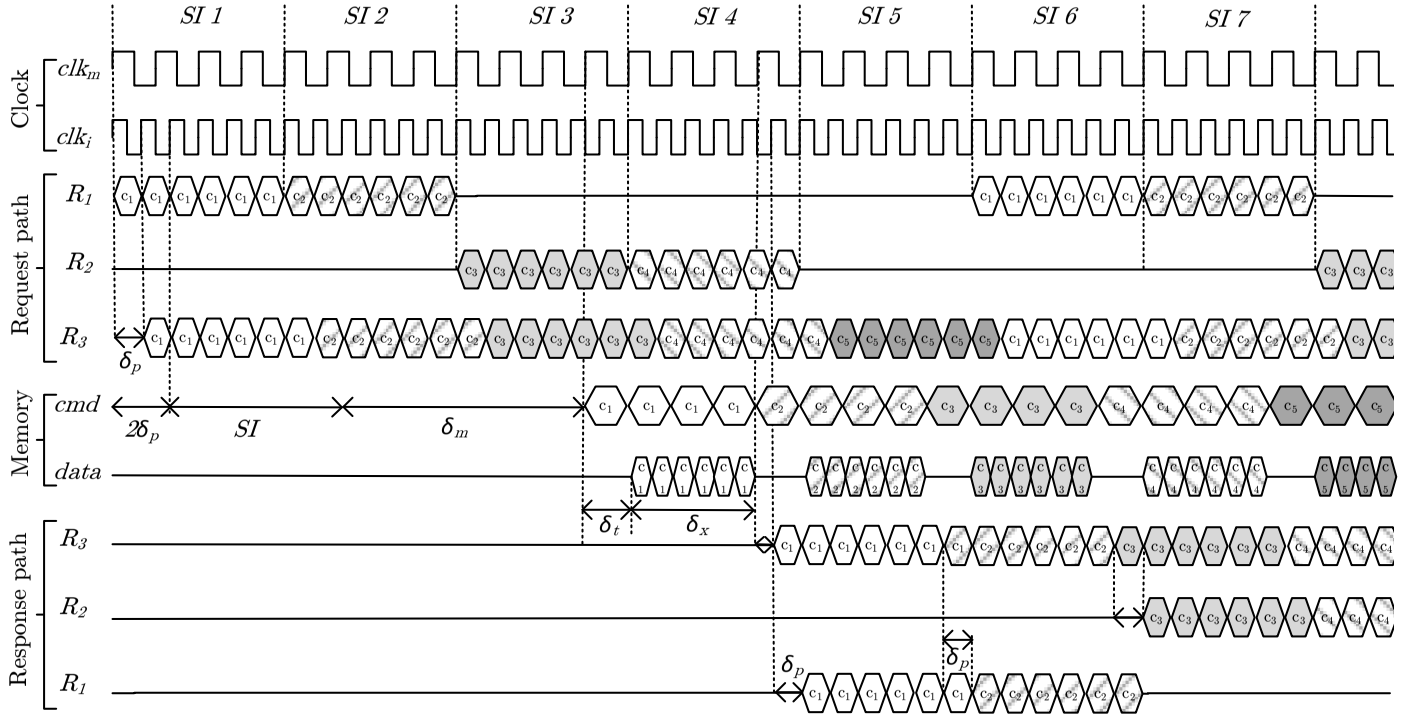


Figure 3.8: Timing diagram showing the coupled architecture operation. The service units are scheduled every scheduling interval (SI). The activity of the clients c_1 to c_5 in the request and response paths are shown separately.

3.2.4 Computation of Interconnect Parameters

Given the memory frequency (f_m), service unit size (SU^{bytes}) and memory service cycle duration (SC_m^{cc}) and interconnect overhead (δ_{ov}), we need to determine all valid combinations of interconnect frequency and interface width (f_i, IW_i). Note that the interconnect overhead is a constant for a given interconnect type. Since we use a single clock source for both the interconnect and the memory controller, the possible f_i and f_m are integer divisions of the system clock frequency. Hence, at first a set, Z, containing all possible values of f_i that are integer multiples and common fractions of f_m need to be computed. Our proposed algorithm for the computation of interconnect parameters is given in Algorithm 2. We need to make sure that $SC_i^{ns} = SC_m^{ns}$ and the interconnect and the memory controller clocks are aligned at the boundary of the service cycle. This is because the bandwidth matching is done over a window of a service cycle, which is measured in clock cycles. Hence, those values of f_i that satisfy Equation (3.2) need to be selected (line 6), which ensures that the number of cycles in SC_i^{cc} (previously defined in Equation (2.5)) corresponding to an f_i will also be an integer multiple or common fraction (same used for computing the f_i from f_m) of SC_m^{cc}, i.e., the clocks will be aligned at the edge of the service cycle. Finally, the values of IW_i corresponding to the different f_i can be computed by substituting the values of the known and the computed parameters in Equation (3.1)(line 9). The computed value of IW_i is rounded up if the result is a non-integer. Note that our approach might result in IW_i configurations which may not be integer multiples of IW_m, requiring appropriate width converters.

3.2.5 Real-Time Guarantees

In the coupled architecture, the worst-case latency (in ns) of an interconnect i, $\hat{L}_{i,c}$, for a read request consisting of N_c service units at the head of the request queue of a client c is given by Equation (3.3), where, $\rho'_{i,c}$ is the rate allocated to the client in the arbiter of the coupled architecture and $\Theta_{i,c}$ is the service latency (in service cycles) caused by interfering clients. This worst-case latency consists of an additional memory clock cycle since the response from the memory may not be aligned with the interconnect clock and hence it has to wait for one interconnect clock cycle. The worst-case latency of write request, given by Equation (3.4), is similar to the read request, except that it experiences the number of hops, n_{hops}, across the interconnect only once as there is no response for a write request. It can be seen in Equation (3.3) that the coupled architecture introduces only a single service latency component in the worst-case latency as opposed to the two in the decoupled architecture, one by the bus-based interconnect in the memory controller clock domain (Equation (2.2)) and the other by the memory interconnect (Equation (2.6)).

Algorithm 2 Computation of memory interconnect parameters

Input: Memory frequency (f_m), Service unit size (SU^{bytes}), Memory service cycle (SC_m^{cc}), Interconnect overhead (δ_{ov}), Set of integer multiples and common fractions (Z)
Output: Interconnect interface width and frequency combinations (F_i, W_i)

procedure ComputeICParam(f_m, SU^{bytes}, SC_m^{cc}, δ_{ov}, $F(t)$)
  $F_i \leftarrow \emptyset$
  $W_i \leftarrow \emptyset$
  for all $\zeta \in Z$ **do**
    $f_i' \leftarrow f_m \times \zeta$
    if $(SC_m^{cc} \times f_i') \bmod f_m = 0$ **then**
      $F_i \cup f_i'$
      $SC_i^{cc} \leftarrow \zeta \times SC_m^{cc}$
      $IW_i' \leftarrow \lceil \frac{8 \cdot f_m}{f_i} \times \frac{SU^{bytes}}{SC_m^{cc}} \times \frac{SC_i^{cc}}{SC_i^{cc} - \delta_{ov}} \rceil$
      $W_i \cup IW_i'$
    end if
  end for
  return F_i, W_i
end procedure

$$\hat{L}_{i,c}(RD) = \frac{(\Theta_{i,c} + \lceil N_c/\rho'_{i,c} \rceil) \cdot SC_i^{cc} + 2n_{hops} \cdot \delta_p + 1}{f_i} + \frac{\delta_m + SC_m^{cc}}{f_m} \tag{3.3}$$

$$\hat{L}_{i,c}(WR) = \frac{(\Theta_{i,c} + \lceil N_c/\rho'_{i,c} \rceil) \cdot SC_i^{cc} + n_{hops} \cdot \delta_p + 1}{f_i} + \frac{\delta_m + SC_m^{cc}}{f_m} \tag{3.4}$$

To summarize, a globally arbitrated memory interconnect can be coupled with a real-time memory controller by configuring its request size equal to the memory service unit size (using an atomizer), selecting a buffer of size equal to the memory service unit size for the NI attached to the memory controller, and configuring the response path to be fully pipelined. Table 3.8 shows a summary of comparison between the coupled and decoupled architectures in terms of number of clocks, buffers and arbiters.

3.3 Multi-Channel Memory Controller (MCMC)

In this section, we present our proposed real-time multi-channel memory controller (MCMC) architecture. We start this section with an analysis of the impact of

Table 3.8: Comparison between coupled and decoupled globally arbitrated memory interconnect architectures.

	Coupled architecture	*Decoupled architecture*
Clocks	Two (synchronous at SC boundary)	Two (asynchronous)
Buffers	*Destination NI*: One SU size for bandwidth matching	*Destination NI*: FIFO $\geq$ one SU size per client
	Router/Mux: Pipeline register (for global arbitration)	*Router/Mux*: One SU size per input port
	MC: One SU size	*MC*: One SU size
Arbiters	One (interconnect)	Two (MC and interconnect)

interleaving memory requests across multiple memory channels on the guaranteed service provided by arbiters belonging to the class of $\mathcal{LR}$ servers, which we refer to as $\mathcal{LR}$ *arbiters*. Then, we present our proposed architecture of MCMC which builds on the architectures of GAMT and CMI, followed by a novel method for logical-to-physical address translation.

3.3.1 Multi-Channel Memories and $\mathcal{LR}$ Servers

As we have seen in Section 2.1.2, multi-channel memories allow a memory request to be interleaved across multiple memory channels. When the memory request of a firm real-time client is interleaved across multiple memory channels with each channel consisting of an $\mathcal{LR}$ arbiter, the worst-case latency is *the maximum of the worst-case latencies among all the memory channels to which the request is interleaved*. Using the $\mathcal{LR}$ model, the worst-case latency of a client, c with a required rate (bandwidth) $\check{\rho}'_c$ *increases* when the number of channels to which its request is interleaved increases. This counter-intuitive result is shown in Equation (3.5), which shows the worst-case latency (in service cycles) for a TDM arbiter in each memory channel, assuming the required rate $\check{\rho}'_c$ and the total number of service units N_c in a memory request are distributed evenly across the number of channels to which the request is interleaved, n_{Chc}. It can be seen that the service latency increases with n_{Chc}, however, the completion latency remains constant. Note that the results are same even when the service units are not evenly distributed across the memory channels. This conclusion is valid for all $\mathcal{LR}$ arbiters that couple latency and rate since they all have the rate term, $\check{\rho}'_c$, which will always get split across channels and the completion latency remains constant. This is evident from their worst-case latency equations [14]. However, note that the worst-case latency can be reduced by interleaving a memory request across multiple memory channels and by allocating a higher rate than requested, i.e., *over-allocating* rate.

$$\hat{L}'_c = \left\lceil f \times \left(1 - \frac{\rho'}{n_{Chc}}\right) \right\rceil + \left\lceil \frac{N_c/n_{Chc}}{\rho'/n_{Chc}} \right\rceil = \left\lceil f \times \left(1 - \frac{\rho'}{n_{Chc}}\right) \right\rceil + \left\lceil \frac{N_c}{\rho'} \right\rceil \qquad (3.5)$$

Interleaving a memory request to more than one memory channel is unavoidable under the following four conditions: (1) When the latency requirement of a client cannot be met in a single channel even after allocating a 100% bandwidth ($\check{\rho}'_c = 1$) to the client. This could happen with larger request sizes as can be seen in Equation (2.3) if the request size is so large that even after allocating a 100% bandwidth ($\check{\rho}'_c = 1$) of a channel, it does not meet its latency requirement it must then be interleaved across multiple channels with an over-allocated rate. (2) When the bandwidth requirement of a client could not be satisfied with the available bandwidth in a single shared memory channel. (3) When the memory capacity requirements cannot be met with the capacity available in a memory channel. (4) When a client needs to communicate with another client whose requests are interleaved across multiple memory channels for any of the previous three reasons, since communicating requestors must be mapped to the same channels.

In a real-time system consisting of several memory clients with different request sizes and diverse requirements on bandwidth, latency, communication and memory capacity, the optimal mapping of clients to the memory channels for minimal bandwidth utilization may result in different degrees of interleaving across the memory channels for each client. This implies that the existing methods [132], in which all clients are interleaved in the same fashion to the memory channels are not always optimal. Hence, we need a configurable memory controller architecture that can be configured to interleave memory requests of a client to any number of available memory channels at different granularities. The next section presents our proposed memory controller architecture for multi-channel memories, while methodologies to determine the optimal mapping of memory clients to channels are presented in Chapter 4.

3.3.2 MCMC Architecture

As we have seen in Section 3.3.1, for optimal bandwidth utilization, memory clients need to be interleaved across multiple memory channels according to their requirements. To enable mapping of memory requests from memory clients to memory channels at different granularities, MCMC performs *channel interleaving*, by which an incoming memory request is split into several service units of equal size and routes them to different memory channels. The architecture of MCMC, shown in Figure 3.9, consists of a dedicated (single channel) *channel controller* (CC) (we call it channel controller to avoid confusion with multi-channel memory controller) for each memory channel, and a memory interconnect (IC) employing a predictable arbitration policy. The channel controller is a real-time memory controller with a fixed service unit size configuration, as explained in Section 2.2. The interconnect can be either a centralized memory interconnect or a globally arbitrated interconnect, such as a TDM NoC or GAMT, coupled with the memory controller using our proposed CMI architecture. The atomizer (AT) splits an incoming memory request into a number of smaller units equal to the service unit size and the *channel selector (CS)* routes the service units to the different

channels according to the configuration in the sequence generator (SG). The CS has separate *request* and *response* paths for each client. As shown in the figure, point-to-point connections are used to interconnect the channel selectors and the memory interconnects.

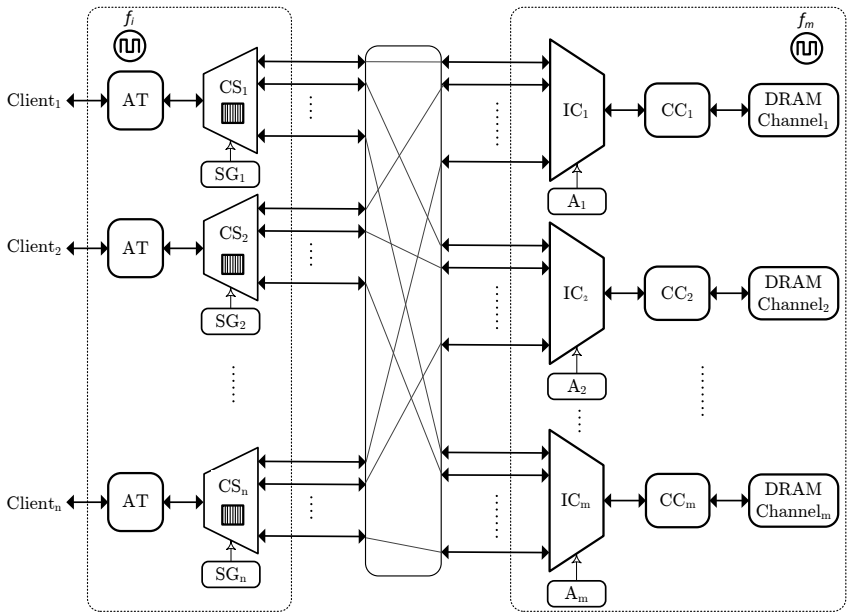


Figure 3.9: High-level architecture of the proposed multi-channel memory controller (MCMC). The atomizer (AT) splits a memory request in to smaller service units and the *channel selector (CS)* routes these service units to the different memory channels according to the configuration in the sequence generators (SG).

A detailed architecture of the channel selector showing both request and response paths is shown in Figure 3.10. In the request path, the atomizer first splits an incoming memory request into a number of service units, and then the sequence generator routes them to the respective memory channels. The sequence generator performs logical-to-physical address translation (explained in the next section) for each of the service units before routing them to the memory channels. The buffers at each output of the channel selector ensure non-blocking delivery of service units (write data) to the different memory channels. The service units routed to the different memory channels may get served at different time instants, and hence the (read) responses from the memory channels may arrive at different times and even out-of-order. Hence, the incoming responses are buffered in the response path at the channel selectors until all of the responses from the different channels have arrived, and then the atomizer forwards the complete response to the client. The pattern stored in the sequence generator is used to re-order data in the response path. Hence, the size of the output and input buffers (for write

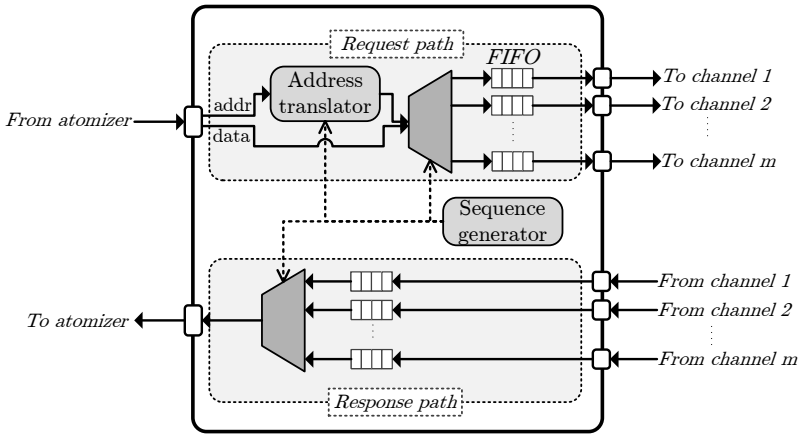


Figure 3.10: Detailed architecture of the channel selector, showing request and response paths. The sequence generator routes the service units to the memory channels to which they are mapped after performing the logical-to-physical address translation. The responses from different channels are buffered in the response path before forwarding the complete response to the atomizer.

data and read response, respectively) should be equal to the maximum number of outstanding requests of a client times its request size (assuming fixed request size for all requests from a client) to enable maximum throughput. Since the sequence generators, arbiters and atomizers can be configured at design time, this architecture enables all possible connections of a client to any of the memory channels with any level of interleaving, and different rate allocated to each client in each channel. Later in Chapter 4, we present our proposed methodology for optimal mapping of clients to channels and configuring the arbiters.

3.3.3 Logical-To-Physical Address Translation

As discussed before, an optimal mapping of clients to memory channels could result in each channel allocated a different number of clients and different memory capacities allocated to the clients in different memory channels. Hence, the service units of a memory request can end up in different physical addresses in each channel when interleaved across multiple memory channels. However, *the application programmer must be able to view the entire memory space (including all memory channels) as a single continuous logical address space to avoid explicit data partitioning and data movement while writing the application program.* In other words, the application programmer need not worry about the number of memory channels in the system and how memory requests are interleaved across them.

Consider an example scenario consisting of a client c_1 with a capacity requirement of 512 B (we consider a small capacity requirement for ease of presentation)

and request size of 256 B interleaved across two memory channels, Channel 1 and Channel 2. Figures 3.11a and 3.11b illustrate the logical and physical views of the memory, respectively. Assuming a service unit size of 64 B, every request from the client consists of four service units. Figure 3.11b shows the physical memory map of the two memory channels, each having an address space of 1 GB. Two service units (SU1, SU2) of request Q1 are allocated to Channel 1, and the remaining two (SU3, SU4) are allocated to Channel 2. Request Q2 is also shown in the figure and is allocated in the same fashion. To access an incoming memory request, say Q2 starting at logical address 0x10010200, the address needs to be translated to the corresponding physical addresses 0x10000180 and 0x10000080 in Channel 1 and Channel 2, respectively. To reduce complexity in the logical-to-physical address conversion and to avoid the use of complex look-up table (LUT) based addressing schemes, we propose a method to compute the logical address in each channel, expressed by Equation (3.6). Note that *Request size* is in service units.

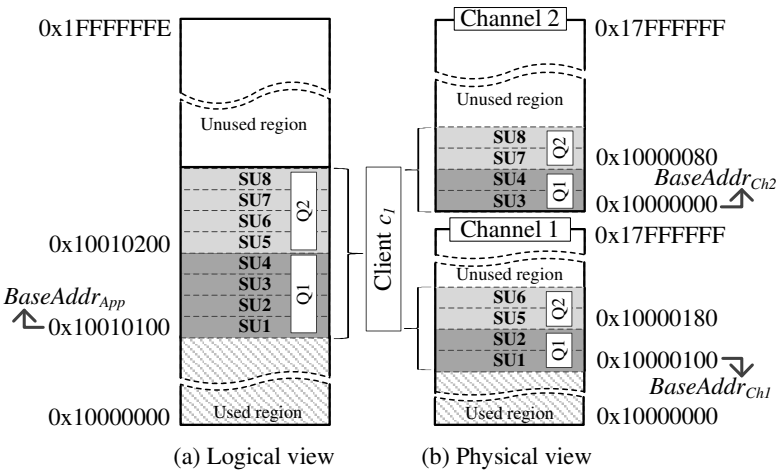


Figure 3.11: Example memory map showing client c_1 allocated to two memory channels, with every request Q1 and Q2 interleaved across the two memory channels.

$$ReqAddr_{Ch} = (ReqAddr_{App} - BaseAddr_{App}) \gg \log_2(Request\ size/N_{Ch_n}) + BaseAddr_{Ch_n} \tag{3.6}$$

The logical address offset between the requested logical address, $ReqAddr_{App}$, and the logical base address of the application, $BaseAddr_{App}$, is computed first, and then added to the physical base address of the application in the corresponding channel, $BaseAddr_{Ch_n}$. When a request is interleaved across multiple channels, the logical address offset is divided by the ratio of service units allocated to each memory channel. This is because the memory capacity allocated

to a client in each channel is proportional to the number of service units of its request allocated to the channel as all the requests of a client are interleaved with the same proportions in each memory channel. For a fast and simple hardware implementation, division is performed using a logical shift operation. We hence require the number of service units allocated to each channel and request sizes (in service units) to be a power of two. Note that the request sizes of most of the real-world memory clients, such as CPUs, DSPs, LCD & DMA controllers are in the order of power of two [123, 127].

The logical base address of an application, $BaseAddr_{App}$, is generated by the application compiler/linker, while the number of service units allocated to each channel, N_{Ch_n}, is decided by the one of our two mapping methods, later presented in Section 4.3. We generate the base addresses for all the clients mapped to each of the channels, $BaseAddr_{Ch}$, based on the memory capacity allocated to them by adding the capacity required by each client in a channel.

3.4 Experiments

In this section, we first compare the performance of the coupled memory interconnect (CMI) with a decoupled architecture, for two different TDM NoCs, in terms of area, power and guaranteed latency. Also, we show the trade-off between area usage and power consumption for the different interconnect parameters using GAMT and the two different TDM NoCs. Then, the performance comparison of the scalable memory tree (GAMT) with respect to centralized implementations of two different arbitration policies is presented. Finally, we experimentally evaluate the real-time guarantees provided by the proposed multi-channel memory controller (MCMC) architecture.

3.4.1 CMI Performance

Experimental Setup

Our experimental setup consists of RTL implementations of the following modules: (1) Router and NI of two different TDM NoCs, one is the packet-switched *Aelite* [57] and the other circuit-switched *Daelite* [122]. (2) A TDM arbiter. (3) A Bus-based interconnect using the Device Transaction Level (DTL) protocol, which is comparable in functionality and complexity to the AXI and OCP protocols [106]. For logic synthesis and for power/area estimation of the designs, we used the Cadence Encounter RTL compiler and the 40 nm nominal V_t CMOS standard cell technology library from TSMC with worst-case process corner.

Comparison Between CMI and Decoupled NoC-Based Interconnects

This section compares the performance of CMI and decoupled NoC-based memory interconnects in terms of area, power consumption and guaranteed latency.

Since the architectural differences between the CMI and decoupled architectures are in the destination NI, the bus-based memory interconnect and TDM arbiter, we synthesized these modules independently to find the differences in area and power consumption. For simplicity, we do not compare the power consumption of the clock sources in CMI and decoupled architectures. However, note that the single clock source of CMI consume less power than the multiple sources in the decoupled architecture. To determine the impact of the savings in area and power in a real system, we consider a system consisting of 16 memory clients with a three-stage NoC tree consisting of eight two-input NIs and seven routers. Figures 3.12 and 3.13 show the decoupled and coupled architectures, respectively, of the example system under consideration. The NoC and the DTL bus was configured with a data-path width of 32-bits assuming a 16-bit DDR memory device that reads/writes 32-bits every clock cycle, and synthesized with a TDM arbiter for a target frequency of 500 MHz. For the decoupled architecture, we configured the NI, DTL bus and the TDM arbiter with 16 ports, as shown in Figure 3.12. Assuming that the clients have diverse requirements on bandwidth and latency, the TDM arbiter was configured with a frame size of 64 considering an average allocation of four TDM slots per client, which enables fine-grained allocation in chunks of $100/64 \simeq 1.5\%$ giving sufficient flexibility in allocating slots to the clients according to their requirements.

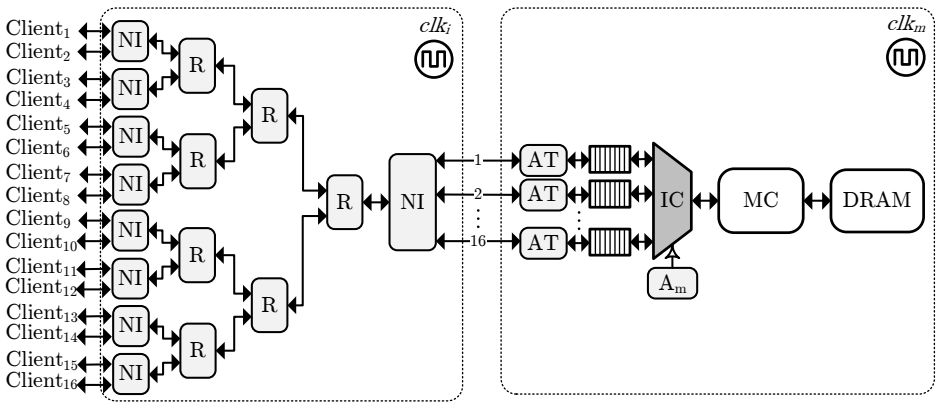


Figure 3.12: Decoupled architecture with 16 memory clients.

To compare the worst-case guaranteed latency of both CMI and decoupled architectures, we assume the same TDM allocation for the NoC in both architectures. For a system consisting of 16 clients with a worst-case bandwidth requirement of each client equal to $1/16^{th}$ of the gross memory bandwidth, we consider a TDM wheel of frame size 64 with four slots allocated to each client using the distributed (equidistant) allocation strategy. Hence, the service latency of a client c in the interconnect i, $\Theta_{i,c}$, is 15 service cycles. For a three-stage NoC tree, i.e., $n_{hops} = 3$. The value of δ_p of Aelite and Daelite are 3 and 2 clock cycles, re-

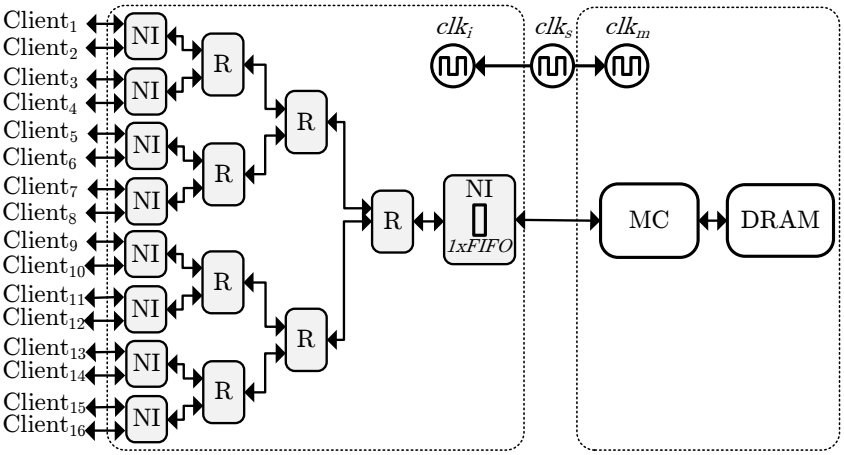


Figure 3.13: Coupled architecture with 16 memory clients.

spectively, according to the number of pipeline stages in their routers [57, 122]. Since we consider the worst-case latency for a read, we use a value of 20 cycles for δ_m based on the pipeline stages for a read operation in the RTL implementation of our memory controller [53]. For the decoupled architecture, we assume the same TDM slot allocation in the bus-based arbiter in front of the the memory controller as in the NoC for simplicity, and hence, their service latencies are 15 service cycles each.

The savings in area and power consumption by using CMI over a decoupled architecture with Aelite and Daelite NoCs are shown in Figure 3.14. It can be seen that the CMI consumes much less power and area and has a lower latency compared to the decoupled architecture. The savings in area are 24% and 20% and in power are 27% and 19% compared to Aelite and Daelite NoCs, respectively. The larger savings in Aelite compared to Daelite is because of the larger area usage of its NI when a new port is added compared to the NI of Daelite. It can be seen that the guaranteed latency with the CMI architecture is 45% lower than the decoupled architecture with both Aelite and Daelite NoCs. This is due to the double service latency in the decoupled architecture because of its two decoupled arbitration stages. However, the decoupled architecture has the flexibility of selecting any arbiter type in the bus-based arbiter in front of the memory controller irrespectively of the arbitration policy in the memory interconnect. For example, by using a priority-based arbiter and assigning the highest priority to one of its clients, its service latency can be reduced to zero as in the case of CMI. However, this will affect the guaranteed latencies of the low-priority clients in the system adversely. Another benefit of using a decoupled architecture is that the memory controller with the arbiter in front of it and the interconnect (NoC) can be run at different clock frequencies that are not aligned. Hence, there is no restriction on

the selection of clock frequencies unlike in the coupled architecture, as we have seen in Section 3.2.4.

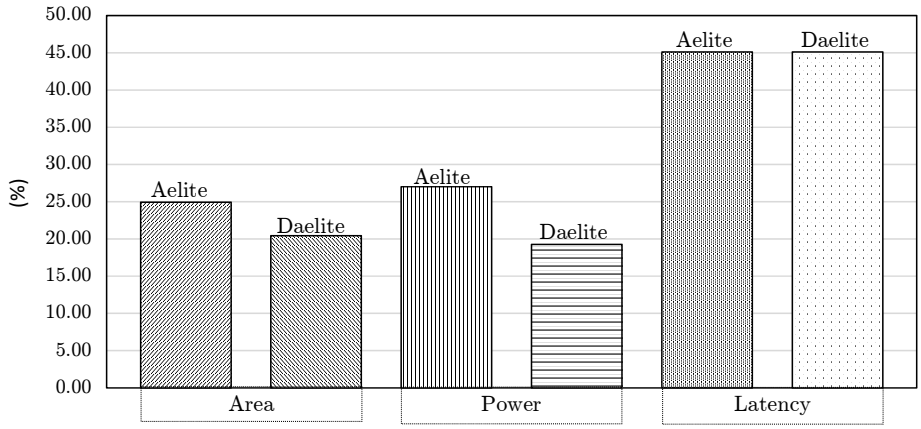


Figure 3.14: Area and power savings and latency reduction of CMI with respect to the decoupled architecture for Aelite and Daelite NoCs.

Impact of Interface Width and Operating Frequency

We analyze the impact of different operating frequency and interface width (f_i, IW_i) combinations for the coupled NoC on its area and power consumption in this section. First, we compute the memory service cycle SC^{cc} and worst-case guaranteed bandwidth b_m^{gross} for different DRAM devices (of 16-bit IO) with a service unit size of 64 B according to [18]. For all those memories with different service unit sizes, we then computed the different (f_i, IW_i) combinations for both Aelite and Daelite NoCs, and GAMT according to the methodology presented in Section 3.2.4. The values of δ_{ov} for Aelite, Daelite and GAMT is 3, 2 and 0 cycles, respectively. Table 3.9 shows the different interconnect operating frequencies and interface widths for different memories for the service unit size of 64 B. In general, it can be seen that the interface width and operating frequency requirement of the interconnects increase with the gross bandwidth of the memory device, as expected. The three interconnects may have different interface width requirements because of their different overhead, δ_{ov}. In general, it can be seen that increasing service unit size increases the gross bandwidth of the memory because of more efficient memory accesses and hence a faster and/or wider interconnect is required.

To analyze the trade-off between area and power consumption with the different (f_i, IW_i) combinations in Table 3.9, we synthesized the RTL design of a single router-NI combination of both Aelite and Daelite NoCs and single APA-Mux of GAMT, with the different (f_i, IW_i) combinations. We selected a single router-NI combination for this analysis as these are the main building blocks

Table 3.9: Memory service cycle and gross bandwidth for different DRAMs with a service unit size of 64 B

Memory	SC^{cc}(cc)	b_m^{gross}(MB/s)	(f_i, Aelite IW_i, Daelite IW_i, GAMT IW_i) combinations
LPDDR-266	19	448.0	(133,32,31,27),(266,15,15,14),(399,10,10,9), (532,8,7,7),(665,6,6,6),(798,5,5,5),(931,4,4,4)
LPDDR-416	19	700.6	(208,32,31,27),(416,15,15,14),(624,10,10,9), (832,8,7,7),(1040,6,6,6),(1248,5,5,5)
LPDDR2-667	25	853.8	(199.8,43,40,35),(266.4,31,29,26),(333,24,23,21), (399.6,19,19,18),(466.2,16,16,15),(532.8,14,14,13), (599.4,13,12,12),(666,11,11,11),(999,8,8,7)
LPDDR2-1066	39	874.7	(177.6,52,47,40),(355.3,23,22,20),(533,15,14,14), (710.6,11,11,10),(888.3,9,9,8),(1066,7,7,7), (1243.6,6,6,6)
DDR3-800	25	1024.0	(240,43,40,35),(320,31,29,26),(400,24,23,21), (480,19,19,18),(560,16,16,15),(640,14,14,13), (720,13,12,12),(800,11,11,11),(1200,8,8,7)
DDR3-1600	44	1163.6	(200,65,57,47),(400,27,26,24),(600,18,17,16), (800,13,13,12),(1000,10,10,10),(1200,9,8,8)

of a TDM NoC and we assume that our power and area results scale accordingly in a larger network. Although we understand that there could be impact of physical routing of wires on the area usage and power consumption in a real implementation of a NoC, an exploration in this direction is beyond the scope of this work. Figures 3.15a, 3.15b and 3.15c show the trade-off between area and power consumption for Aelite, Daelite NoCs and GAMT, respectively, coupled with the different memories with a service unit size of 64 B. It can be seen that Daelite, which is a circuit-switched NoC has about 20% higher area and 35% higher power consumption compared to the packet-switched Aelite because of the additional logic required by the slot table (LUT) in the Daelite router. On the other hand, GAMT uses less than a third of the area compared to the NoC-based interconnects as it does not have slot tables, as we have seen in Section 3.1.2. However, the power consumption of GAMT is about 50% and 20% higher than Aelite and Daelite NoCs at higher operating frequencies. This is because of the larger switching activities due to the number of wires in GAMT as it uses DTL protocol on its wires between the routers.

For all the interconnects, the power consumption increases with the gross bandwidth of the memory device because of higher operating frequency and/or interface width requirements. For lower area usage (smaller interface widths and higher operating frequency), the power consumption is higher because of the dynamic power consumption at higher operating frequencies. The power consumption reduces with operating frequency, however, the area increases because of increase in interface width. The only exception is in the case of GAMT at higher frequencies when the additional area is added to the APA logic to make it syn-

thesizable at those frequencies due to the critical path in the APA logic. The area requirement increases linearly with the interface width and hence the leakage power consumption. In addition to leakage power, increase in area increases the dynamic power as well due to the added switching activity in the circuit. Figures 3.16a, 3.16b and 3.16c show the trade-off between energy efficiency and area usage of the three interconnects. It can be seen that although the area is increased with wider interfaces, the energy efficiency is higher due to the slower operating frequencies. However, increasing interface width beyond a certain limit does not increase the energy efficiency as the leakage power starts dominating. Note that we observed the similar results with different access granularities, 16 B, 32 B, 128 B and 256 B, and is shown in Appendix E.

To summarize, we have seen that the power consumption and area usage of a coupled memory interconnect depends on the memory device, its access granularity and the interconnect type. Hence, given a memory interconnect, its optimal interface width and operating frequency need to be selected based on power vs. area trade-off of all the different combinations computed using our proposed methodology. Comparing Aelite, Daelite and GAMT, we can see that Aelite and GAMT overall have low power consumption and GAMT has low area usage.

3.4.2 GAMT Performance

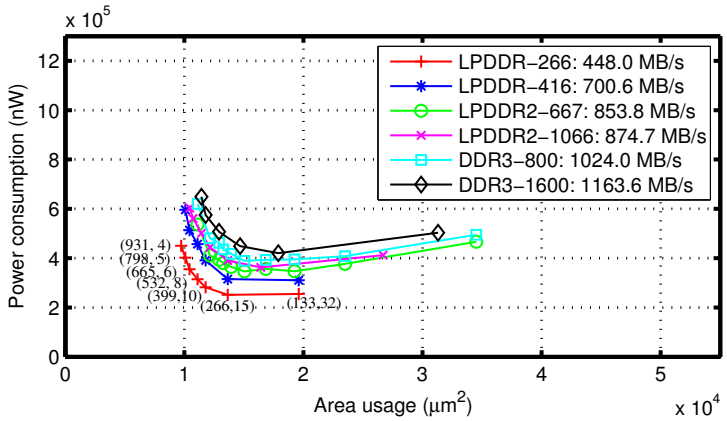
In this section, we present the functional verification and performance comparison of GAMT with centralized implementations of two different arbitration policies. First, we will introduce the experimental setup and then proceed with the experiments.

Experimental Setup

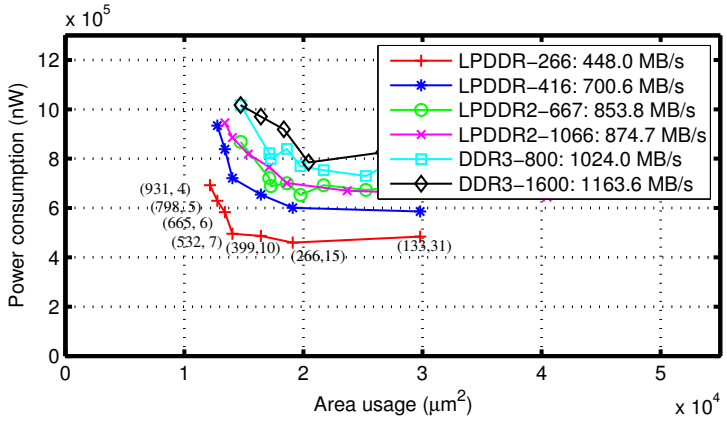
The experimental setup consists of the RTL implementation of GAMT and centralized implementations of two different arbitration policies, TDM [53] and CCSP [14], with a 32-bit data-path. We used Cadence Encounter RTL compiler and the 40 nm nominal Vt CMOS standard cell technology library from TSMC with the worst-case process corner for logic synthesis to determine the power and area usage and the maximum synthesizable frequency of the designs.

Functional Verification

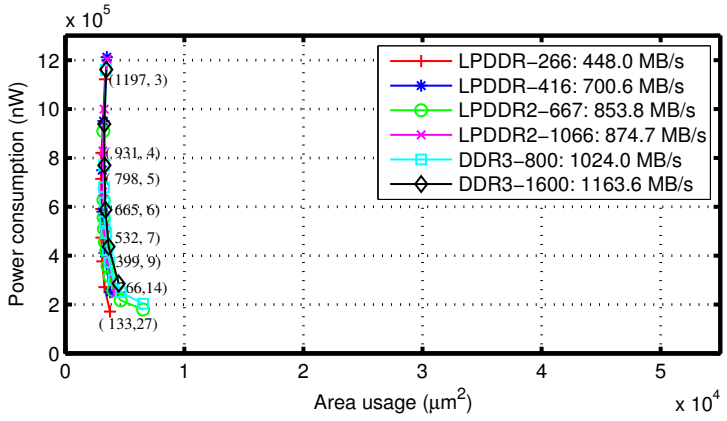
We ensured the functional correctness of GAMT by comparing the scheduling decisions made by the FPGA implementation of GAMT (with 16 clients) at every scheduling interval with C++ reference models of centralized implementations of TDM, FBSP, and CCSP. Note that the other supported arbitration policies are special cases of these three arbiters and do not require additional verification. We used synthetic traffic generators for the clients to generate random traffic to cover both backlogged and non-backlogged conditions and verified the functionality



(a) Aelite

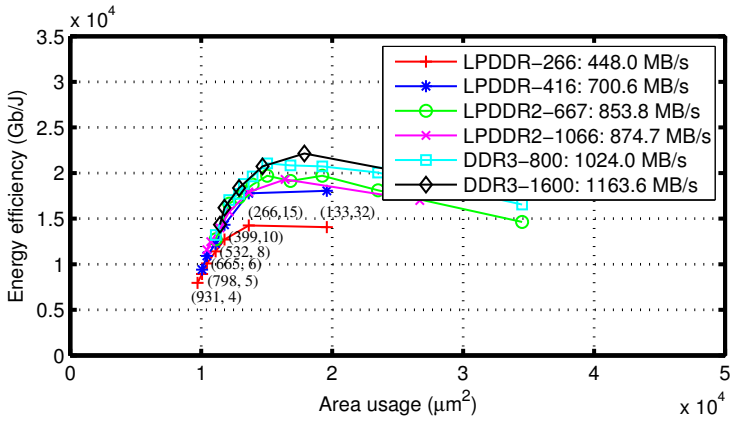


(b) Daelite

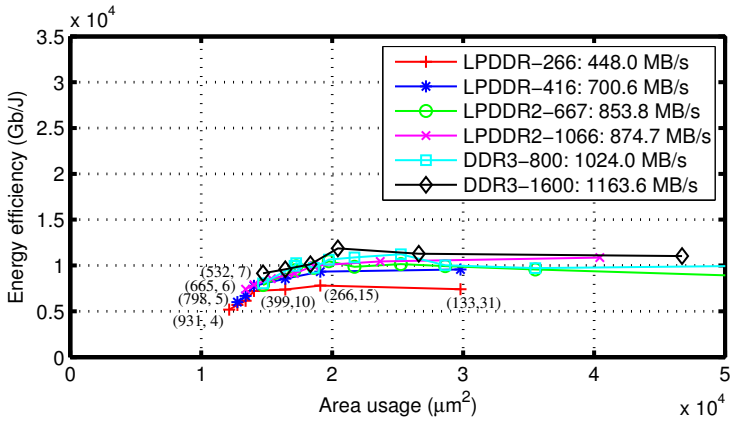


(c) GAMT

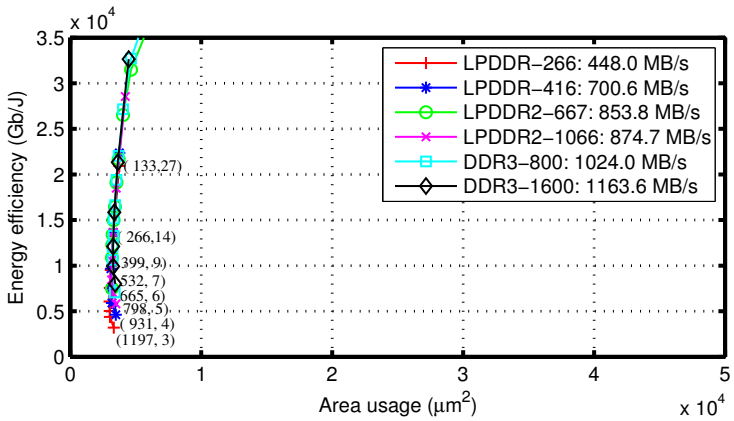
Figure 3.15: Area vs. power trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 64 B. For clarity, only the (f_i, IW_i) combinations for LPDDR-266 are shown.



(a) Aelite



(b) Daelite



(c) GAMT

Figure 3.16: Energy efficiency (Gb/J) vs. area trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 64 B. For clarity, only the (f_i, IW_i) combinations for LPDDR-266 are shown.

(for several thousands of scheduling decisions) in both work-conserving and non-work-conserving modes of all the three arbitration policies. We found that all scheduling decisions made by both GAMT and the centralized implementations were the same, and hence, *we conclude that GAMT correctly implements the different arbitration policies.* Since all decisions in GAMT are made identically to the centralized reference implementations, the timing analyses of the original arbiters can be used for GAMT as well with an only addition of the constant propagation delay in GAMT.

Performance Comparison

We synthesized the design of GAMT and the centralized implementations of TDM and CCSP for different number of clients, i.e. 4, 8, 16, 32 and 64 to determine the maximum synthesizable frequency. Note that we selected frame-based (TDM) and priority-based (CCSP) arbiters to compare GAMT with different types of hardware implementations. For TDM, we have used a frame size such that each client gets four slots and for CCSP, we used registers with 16-bit precision for the configuration registers (Numerator, Denominator, and Current Credits). Table 3.10 shows the area, power and maximum clock frequency of the GAMT and the centralized implementations of TDM and CCSP. In general, it can be seen that the maximum clock frequency, f_{max}, of centralized TDM and CCSP do not scale with the number of clients. With 64 clients, GAMT can be run up to a clock frequency of 1.2 GHz, whereas CCSP and TDM are limited to 0.3 GHz.

Table 3.10: Area, power and maximum clock frequency (f_{max}) of GAMT and centralized implementations of TDM and CCSP

	Area (mm^2)			Power (mW)			f_{max} (MHz)		
# Clients	*TDM*	*CCSP*	*GAMT*	*TDM*	*CCSP*	*GAMT*	*TDM*	*CCSP*	*GAMT*
4	0.016	0.020	0.017	5.19	5.35	4.55	588	526	1250
8	0.029	0.036	0.035	7.88	8.07	9.77	500	435	1250
16	0.061	0.077	0.070	16.13	14.94	20.20	435	357	1250
32	0.107	0.172	0.141	17.46	25.36	41.07	333	333	1250
64	0.203	0.417	0.282	35.60	63.18	82.81	333	303	1250

In general, the area and power consumption of all different designs increase linearly with the number of clients due to the additional logic added. The area usage of centralized CCSP increases significantly with increasing number of clients due to the extra logic added to break the critical path. The f_{max} for centralized TDM scales down as well with increasing number of clients due to the critical path in the priority resolution logic for the work-conserving mode. On the other hand, GAMT has better scalability in f_{max} with the number of clients, since its critical path in the APA logic remains constant irrespectively of the number of clients as it is dedicated for each client. However, it is worthwhile to note that GAMT consumes more power compared to the centralized implementations in most cases. This is primarily due to the addition of extra priority lines in the bus

and the dedicated APA logic for each client. One limitation of GAMT is that it can support only TDM with contiguous slot allocation strategy, whereas the centralized implementation of TDM using a Look-up-Table (LUT) can support distributed allocations [14].

To efficiently compare the centralized designs and GAMT in terms of frequency, area and power consumption, we define two cost-efficiency metrics, *bandwidth/area* and *bandwidth/power (bits/Joule)*. Bandwidth is computed by multiplying data-path width (in Bytes) with the clock frequency (f_{max}). Figures 3.17 & 3.18 shows the ratio of bandwidth (Bytes/s) to area usage (mm^2) and bandwidth (Bytes/s) to power consumption (mW), respectively, of centralized CCSP and TDM normalized to GAMT. It can be seen that for all configurations of clients, GAMT has over 51% and 37% performance gain in terms of area and power consumption, respectively, compared to traditional centralized implementations of CCSP and TDM. Hence, *we can conclude that GAMT is suitable when there are a large number of memory clients in the system that requires the arbiter to be clocked at higher speed or when the platform requires different arbiters for different sets of applications.*

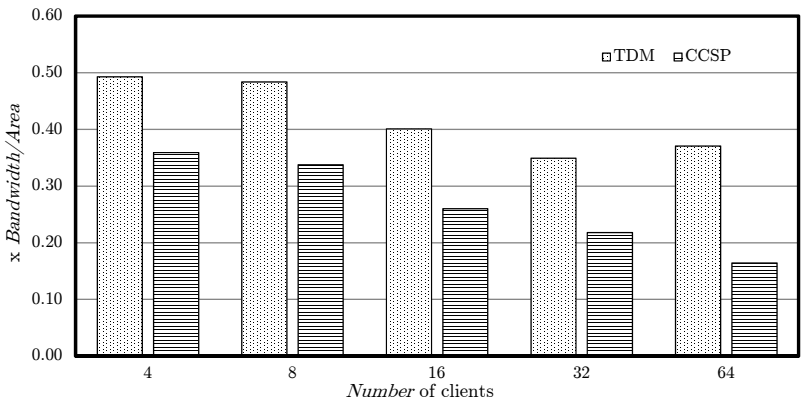


Figure 3.17: Bandwidth/Area performance of centralized CCSP and TDM arbiters normalized to GAMT.

3.4.3 MCMC Evaluation

Experimental Setup

The experimental setup consists of a cycle-accurate SystemC model of MCMC using *Predator* [13] as channel controllers attached to a Wide IO 200 MHz DRAM [4] memory model with each channel consisting of 4 banks and a data bus of 128-bits. TDM arbiters with contiguous slot allocations are used as the $\mathcal{LR}$ arbiter in the channel controllers.

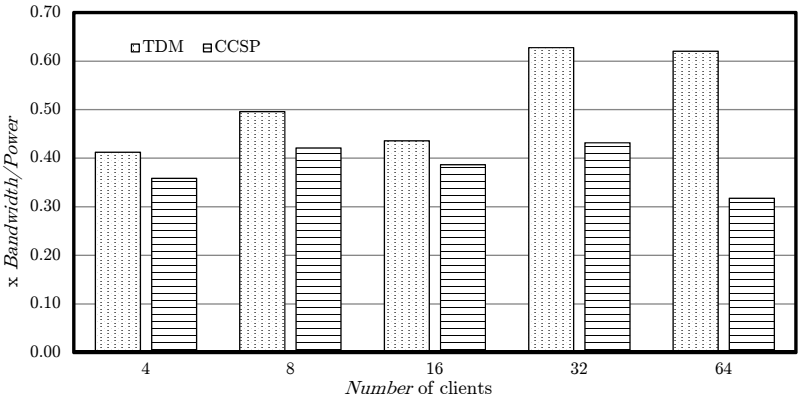


Figure 3.18: Bandwidth/Power performance of centralized CCSP and TDM arbiters normalized to GAMT.

To demonstrate the guarantees provided by MCMC on worst-case latency and bandwidth, we consider two clients: one with low latency requirements (c_1) and the other with worst-case bandwidth requirements (c_2), which corresponds to real-time low-latency and some streaming clients [129], respectively. We used the *mpeg2 decode* application from the MediaBench [81] benchmark suite applications as c_1. To emulate c_1, we used a SystemC traffic generator that can elastically replay transaction-level traces of memory requests of the application. The memory request traces are generated by running the application on a *SimpleScalar* out-of-order simulator [2] with a unified 64 KB and 128 KB L1 and L2 caches, respectively. The L2 cache is configured with 64 B cache lines, 512 sets and an associativity of 4. With this configuration, each request in the trace thus corresponds to a cache miss of 64 B. To measure the actual round-trip latency from the point at which a request is issued until the final response arrives back at the client without the impact of self-interference[2], we have configured the traffic generator with maximally one outstanding request, such that c_1 issues memory requests of size 64 B one at a time (for each cache-miss) and the successive requests are blocked until the response of last issued request has arrived. For c_2, we used a synthetic memory request generator, which generates requests of size 64 B according to a normal distribution with a sufficiently low mean to request more bandwidth than the client is allocated to ensure that the client is always backlogged. This implies self-interference between the requests. The synthetic client generates a mix of both read and write requests.

For the Wide IO SDR 200 MHz device, we selected an access granularity of 32 B in each channel that provides a worst-case bandwidth of 484.1 MB/s per channel (computed according to the analysis in [14]). We selected the service unit

[2]To be consistent with our system model that provides guarantees on end-to-end latency for a complete request without self-interference.

size equal to the access granularity of 32 B, which takes 13 clock cycles to read and write to the memory (service cycle). We selected this service unit size since it is smaller than the request size (of 64 B), giving us the flexibility of interleaving the memory requests across channels.

Bandwidth is computed counting the requests served by a channel controller during an interval. When a request is interleaved across multiple memory channels, we compute the bandwidth in each channel individually and sum them up to find the total provided bandwidth. To measure the latency of a request, we find the time difference between the time at which a read request arrives at the request buffer of MCMC until the complete response arrives back. Note that this is consistent with our system model presented in Section 2.3.

We measured the latency and bandwidth of c_1 and c_2, respectively, for different cases (discussed in the next section) and compared against the analytically computed worst-case bounds. The worst-case bandwidth of a client is computed as a fraction of the worst-case bandwidth provided by a channel using the fraction of TDM slots (rate) allocated to the client. We computed the worst-case latency bound using Equation (2.3), and also included the overhead due to the hardware pipeline stages (9 clock cycles) and one refresh duration (130 ns for Wide IO DRAM). The hardware pipeline delays are introduced by the atomizer, channel selector, interconnect and the memory controller. We add only one refresh duration to the latency of a request, since only one refresh operation can occur in a single TDM wheel considering the much larger refresh interval of 7.8 μs for the WIDE IO 200 MHz 2 Gb device compared to the TDM frame size of 6 (= 390 ns), which we used in our experiments.

Simulation Results

We need to evaluate the guarantees on latency and bandwidth provided by our MCMC to c_1 and c_2, respectively, under different interleaving schemes. Hence, we perform experiments by configuring the sequence generators in the channel selectors for the following four different cases of interleaving: 1) Neither client is interleaved across memory channels. 2) Only c_1 is interleaved across two memory channels. 3) Only c_2 is interleaved across two memory channels. 4) Both clients are interleaved across memory channels. Figures 3.19 & 3.20 show both TDM slot allocation and the simulation result for Cases 3 & 4, respectively. Due to similarity in results, we do not show Cases 1 & 2. The simulation results show both measured latency of c_1 and bandwidth of c_2 during the first 200 μs of the simulation with their respective worst-case bounds.

In all four cases and for the complete duration of simulation, we observed that the guaranteed latency bound is only about 15% higher than the maximum of the measured latencies (depicted by crosses) of all of the requests and the measured bandwidth (depicted by circles) is 0% off from the guaranteed bandwidth bound as expected. This is because the worst-case latency bound is computed according to the abstract $\mathcal{LR}$ model, which provides a conservative bound. However, the worst-

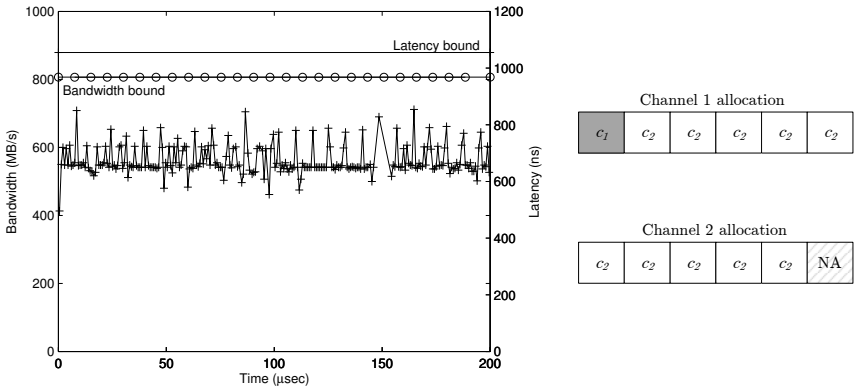


Figure 3.19: *Case 3*: c_2 is interleaved across two memory channels with a rate of 5/6 in each channel, and c_1 is interleaved to one memory channel with a rate of 1/6. The measured latency of every request of c_1 is depicted by a cross and the measured bandwidth of c_2 during every refresh interval is depicted by a circle.

case guaranteed bandwidth is a tight bound, since it is computed considering the actual DRAM command timing constraints including refresh. Refresh is periodic and its impact on bandwidth can be estimated accurately [14]. Note that c_2 is constantly backlogged to measure the guaranteed bandwidth. This shows that the analysis technique that we use in this work gives good bounds. Comparing Figures 3.19 & 3.20, it can be seen that the average latency of c_1 is lower by about 50% after interleaving across two memory channels, since it gets twice the rate. However, the guaranteed latency bound is lower by about 30% only, as the completion latency is reduced by half but the service latency remains the same, according to Equation (3.5).

To summarize, we have demonstrated that the bounds on bandwidth and latency given to the clients are conservative and we have verified the conservativeness for much longer simulation traces than shown in the figures. Also, we have seen that the worst-case latency and/or bandwidth bounds vary according to the number of service units allocated in each memory channel and the allocated rate. Hence, our configurable multi-channel memory controller enables configuring the memory subsystem for efficient utilization according to the latency and/or bandwidth requirements of the memory clients.

3.5 Summary

To address the scalability issue with present real-time memory subsystems, this chapter presented three main innovations: (1) A generic globally arbitrated memory tree (GAMT). (2) A coupled memory interconnect (CMI) architecture. (3) A

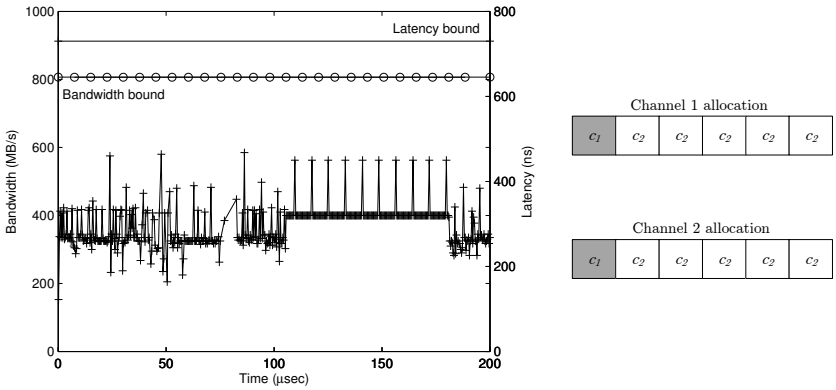


Figure 3.20: *Case 4*: Both clients c_1 and c_2 are interleaved across the two memory channels with a rate of 1/6 and 5/6, respectively, in each channel. The measured latency of every request of c_1 is depicted by a cross and the measured bandwidth of c_2 during every refresh interval is depicted by a circle.

multi-channel memory controller (MCMC). In addition, we have shown the performance evaluation of the different architectures by experimentation.

For a scalable memory interconnect in terms of clock frequency, we proposed a novel memory tree, GAMT, for distributed implementation of predictable arbitration policies. Moreover, our configurable RTL-level design of GAMT can be configured to operate as any of the five different predictable arbitration policies, TDM, RR, FBSP, PBS, and CCSP, which are proposed for shared memory access in real-time systems. Our experimental results show that GAMT outperforms the centralized implementations by more than four times in terms of clock speed and over 51% and 37% in terms of bandwidth/power and bandwidth/area trade-off, respectively. However, GAMT consumes more power than centralized implementations due to the larger switching activity.

To minimize the area usage, power consumption and worst-case latency in the existing distributed memory interconnects, we presented a novel coupled memory interconnect architecture (CMI) to couple the memory interconnect with a real-time memory controller. Coupling the NoC and memory controller using the proposed architecture saves 45% in guaranteed latency, 24% and 20% in area, 27% and 19% in power consumption, for two different TDM NoCs when used as the memory interconnect, for a system consisting of 16 memory clients. However, this comes at the cost of restricting the interconnect operating frequency to a limited set of values.

For a scalable memory subsystem in terms of number of memory channels, we presented a real-time multi-channel memory controller (MCMC) architecture including a novel method for logical-to-physical address translation. Our proposed MCMC architecture allows interleaving of memory requests across multiple mem-

ory channels with different granularities and rates allocated to each client in each channel. Finally, we demonstrated the real-time guarantees provided by MCMC by experimentation.

Chapter 4

Bandwidth-Efficient Memory Subsystem Design

Bandwidth-efficient design of the memory subsystem, i.e. maximum memory efficiency and minimum bandwidth allocated to clients, in a real-time system is challenging. This is because the design choice of several system-level parameters related to DRAM type selection and configuration, and configuration of multi-channel memory controller need to be made, while providing memory performance guarantees to the memory clients with diverse real-time requirements. Currently, there are plenty of memory type options available in the market with different interface widths and operating frequencies. These parameters need to be considered carefully while selecting a memory type in order to get maximum memory bandwidth for the price paid. In addition, the configuration of the memory map in the memory controller affects how efficiently the memory bandwidth is utilized. On the other hand, the bandwidth that needs to be allocated to the memory clients in the memory subsystem in order to meet their bandwidth and/or latency requirements must be minimized to maximize the slack bandwidth. The slack memory bandwidth can be allocated to the soft or non-real-time clients in the system to improve their average-case performance. The bandwidth that needs to be allocated to the clients in a multi-channel memory subsystem depends on the the client bandwidth and/or latency requirements, mapping of clients to memory channels, and the client request sizes. Existing design methodologies of memory subsystems either does not consider mapping of clients to multi-channel memories or do not provide real-time performance guarantees to the memory clients.

In this chapter, we first formally define the problem and give a high-level

overview of our proposed design flow for the bandwidth-efficient design of memory subsystem in Section 4.1. We present a worst-case analysis of memory types across and within generations in Section 4.2, and derive guidelines for the configuration of memory map parameters. In Section 4.3, we present our proposed algorithms for mapping memory clients to memory channels and arbiter configuration. Finally, a case study demonstrating the effectiveness of our proposed approach for a High-Definition (HD) video processing system is presented in Section 4.4.

4.1 Motivation and Proposed Solution

In this section, we first define the problem and then introduce the high-level overview of our proposed design-flow for bandwidth-efficient design of memory subsystems in real-time systems.

4.1.1 Problem Statement

There are plenty of DRAM types available in the market [9, 96, 10] of different generations, interface widths (IW_m), operating frequencies (f_m) and number of memory channels (NC). Once a memory type is selected, there are different configuration options for the memory map parameters (BI, BC), defined in Section 2.2, in the memory controller. The selection of a memory type and its configuration need to be done carefully as the worst-case bandwidth utilization of the memory depends on its interface width and operating frequency, the configuration of the memory map parameters in the memory controller, and the client requirements and request sizes [17, 48].

As we have seen in Section 1.1.4, the real-time memory subsystem consists of an arbiter in front of the memory controller, which multiplexes memory requests arriving from the different clients. The configuration of this arbiter, such as frame size (f) and number of TDM slots allocated to each client in the case of a TDM arbiter, needs to be done such that their bandwidth and/or latency requirements are satisfied. However, the bandwidth allocated to a client depends on other factors as well, such as the amount of over-allocation of bandwidth required due to the discretization of rate in the case of frame-based arbiters [46] and the impact on client latency and memory bandwidth of the memory map configuration.

Memories with multiple physical channels and wide interfaces, such as Wide IO DRAMs [4, 8], are essential to meet the main memory *power/bandwidth* demands of future real-time systems [48]. In multi-channel memories, a memory client can be mapped to multiple memory channels by *interleaving* its memory requests across different memory channels after splitting it into smaller sized requests. Previous studies on multi-channel memories show that mapping soft real-time memory clients to multiple memory channels according to their memory request sizes benefits average-case performance [111, 31, 101]. In addition to having different request sizes, and communication and memory capacity requirements, firm

real-time memory clients in real-time multi-processor platforms come with diverse requirements on memory bandwidth and latency, as well. *The bandwidth allocated to firm real-time memory clients must be minimized to maximize the slack bandwidth that can be allocated to the soft and non-real-time clients in the system, which improves their average-case performance* [83].

To summarize, we define our problem statement as follows: *Given a set of memory client requirements and memory type specifications, select the memory and configure the memory map parameters in the memory controller for maximum worst-case gross bandwidth, and determine the mapping of memory clients to the channels for minimum allocated bandwidth.*

4.1.2 Overview of Proposed Design-Flow

The proposed design-flow for the bandwidth-efficient design of memory subsystem in real-time systems shown in Figure 4.1 consist of *four* main steps as explained below.

Step 1: Given a set of memory types, first we need to compute the peak bandwidth of each memory, as explained in Section 2.1, and select those with peak bandwidth greater than or equal to the total bandwidth requirements of all the clients. Also, the total memory capacity requirements of all the clients need to be satisfied by the selected memories. This step helps to discard the memory types that trivially cannot satisfy the client bandwidth and capacity requirements. Both single and multi-channel memories are included in this step. Note that we do not consider DRAM prices in our design-flow as prices are volatile [1]. However, the system designer can consider the memory price when including the memories in this step.

Step 2: In this step, the worst-case bandwidth of the memories (from the previous step) is computed for different service unit sizes according to our *memory-map design guidelines*, which will be explained later in Section 4.2.1. Note that the worst-case gross bandwidth of different memory map configurations is computed as explained in [14]. Using our memory-map design guidelines in this step reduces the design-space significantly as there are several memory-map configurations possible for a memory.

Step 3: Here, the *aggregate bandwidth requirements* of all clients is computed, as explained in Section 4.2.2, considering the different request sizes of the clients. This step captures the impact of data efficiency, as explained in Section 2.2, on the bandwidth requirement. Then, those service unit sizes that provide a worst-case gross bandwidth larger or equal to the aggregate client bandwidth requirements are selected.

Step 4: For each memory type and for all the service unit sizes determined in the previous step (that provides sufficient bandwidth to meet the aggregate client bandwidth requirements) the clients are mapped to the memory channels, i.e. the interleaving granularities of each client in each memory channel and the arbiter configurations are determined. For this, we present two design-time methods,

that determine the number of service units and allocated rate allocated to each client in each memory channel, for minimal total allocated bandwidth. One is an optimal algorithm (Section 4.3.2) based on an integer programming formulation of the mapping problem, and the other a fast heuristic algorithm (Section 4.3.3).

If the client requirements are not met with any of the service unit size configurations in Steps 2, 3 or 4, a new set of memories with higher bandwidth are selected and the whole process is repeated from Step 1. Note that the new set of memories can be faster memories or memories with larger number of memory channels which were not considered previously.

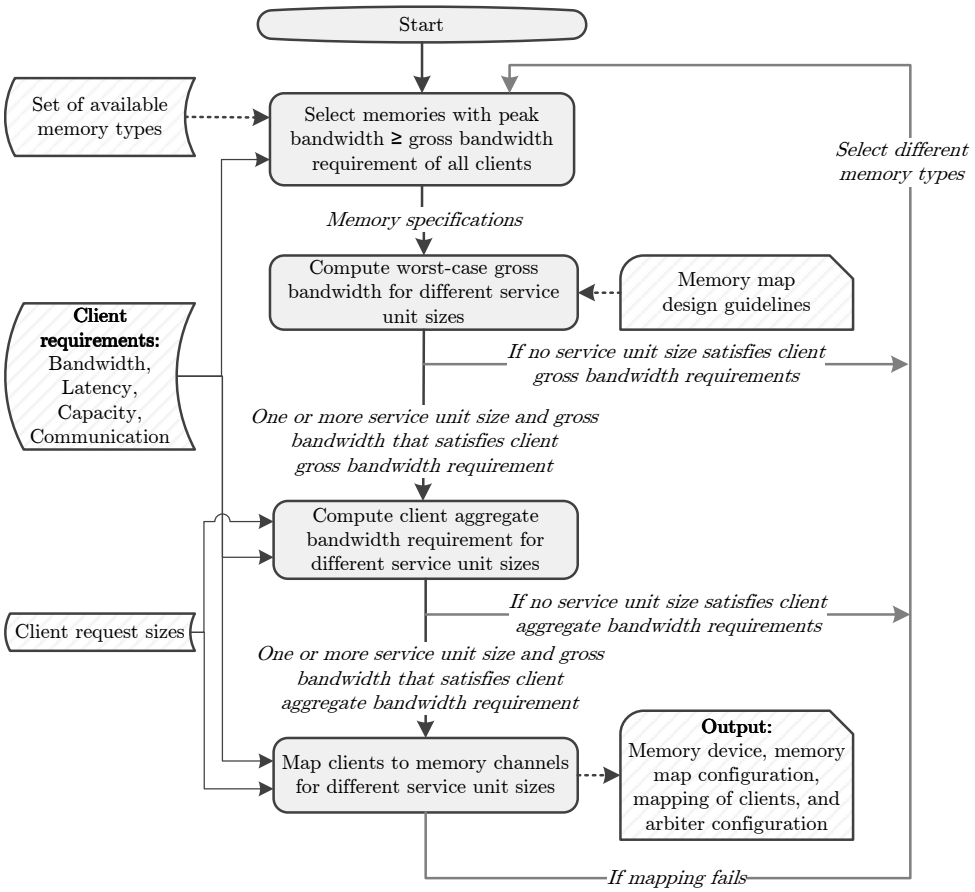


Figure 4.1: Proposed design-flow for bandwidth-efficient design of DRAM subsystem in real-time systems. Note that the final output of the design-flow is a single optimal configuration although there are one or more service unit sizes that gives a valid mapping.

4.2 Memory Map Selection and Aggregate Bandwidth Computation

In this section, we first start with a worst-case analysis where we show the guaranteed bandwidth results of DRAM types across generations and derive guidelines for the configuration of memory map parameters. Then, we present our proposed methodology for the selection and configuration of DRAM.

4.2.1 Memory Map Selection

Our worst-case analysis assumes a real-time memory controller, as explained in Section 2.2. The different memories selected for the worst-case analysis are given in Table 4.1. For simplicity, we consider only the mobile memory generations that target at battery-operated mobile devices, although our approach can be straightforwardly applied to other memory generations, such as DDR2 and DDR3, as well. We computed the worst-case bandwidth of the memories for different access granularities using the analysis methods in [14]. Note that a similar analysis can be applied to other existing real-time memory controllers as well. The memory command timings for LPDDR, LPDDR2 and LPDDR3 are based on Micron specifications [93, 94, 95], and WideIO (SDR) and WideIO2 (DDR) on JEDEC specifications [4, 8]. Note that the BL is fixed to the standard configuration for all the memory types.

Table 4.1: Memory types of different generations, capacities, operating frequencies, interface widths and burst length settings used for our worst-case analysis.

Memory type	*Memory capacity (Gb)*	*Operating frequency (MHz)*	*Interface width (bits)*	*Number of memory channels*	*Burst length (BL)*
LPDDR	2	133, 208	16	1	8
LPDDR2	2	333, 533	16	1	8
LPDDR	2	133, 208	32	1	8
LPDDR2	2	333, 533	32	1	8
LPDDR3	8	667, 800	32	2	8
WideIO	4	200, 266	128	4	4
WideIO2	8	400, 533	64	4	8

For the design-space exploration of memory map configuration, we computed the worst-case bandwidth for different combinations of BI and BC (BL is fixed as shown in Table 4.1) and for different access granularities. Figure 4.2 shows the worst-case gross bandwidth of all memory types for different access granularities, i.e. service unit sizes, and with different (BI,BC) combinations. In general, it can be seen that the gross bandwidth increases with the service unit size. This is because a larger burst access improves the memory efficiency by minimizing the negative impact of the overhead clock cycles, explained in Section 2.1, for a

memory access [14]. Now looking at the different (BI,BC) combinations, we can see that the gross bandwidth is larger when BI is increased first and then BC for all the memory types. This is because by increasing the number of banks over which a memory request is interleaved, bank-level parallelism can be exploited to hide the overhead in accessing data from a memory bank [14]. However, interleaving memory requests across 8 memory banks (8,1) results in poor memory bandwidth due to the negative effect of the four-activate window constraint, explained in Section 2.1, on the worst-case gross bandwidth [52]. Note that WideIO memories have a two-activate window constraint, and hence, the memory map configuration (4,1) has lower gross bandwidth than (2,2) for those memories. From our worst-case analysis results, we derive the following two guidelines for the selection of BI and BC to maximize the worst-case bandwidth:

1. *Maximize BI*, i.e. interleave data over the maximum number of banks until the activate command window constraint. This is because efficient pipelining of memory commands across different banks, i.e. bank-level parallelism, improves overall efficiency by hiding some of the DRAM timing constraints.

2. *Maximize BC* to amortize remaining overhead over a larger access granularity.

These guidelines should be respected in the order they are mentioned. Note that while this approach maximizes the worst-case bandwidth, it reduces the energy efficiency due to the use of a large number of memory banks [52].

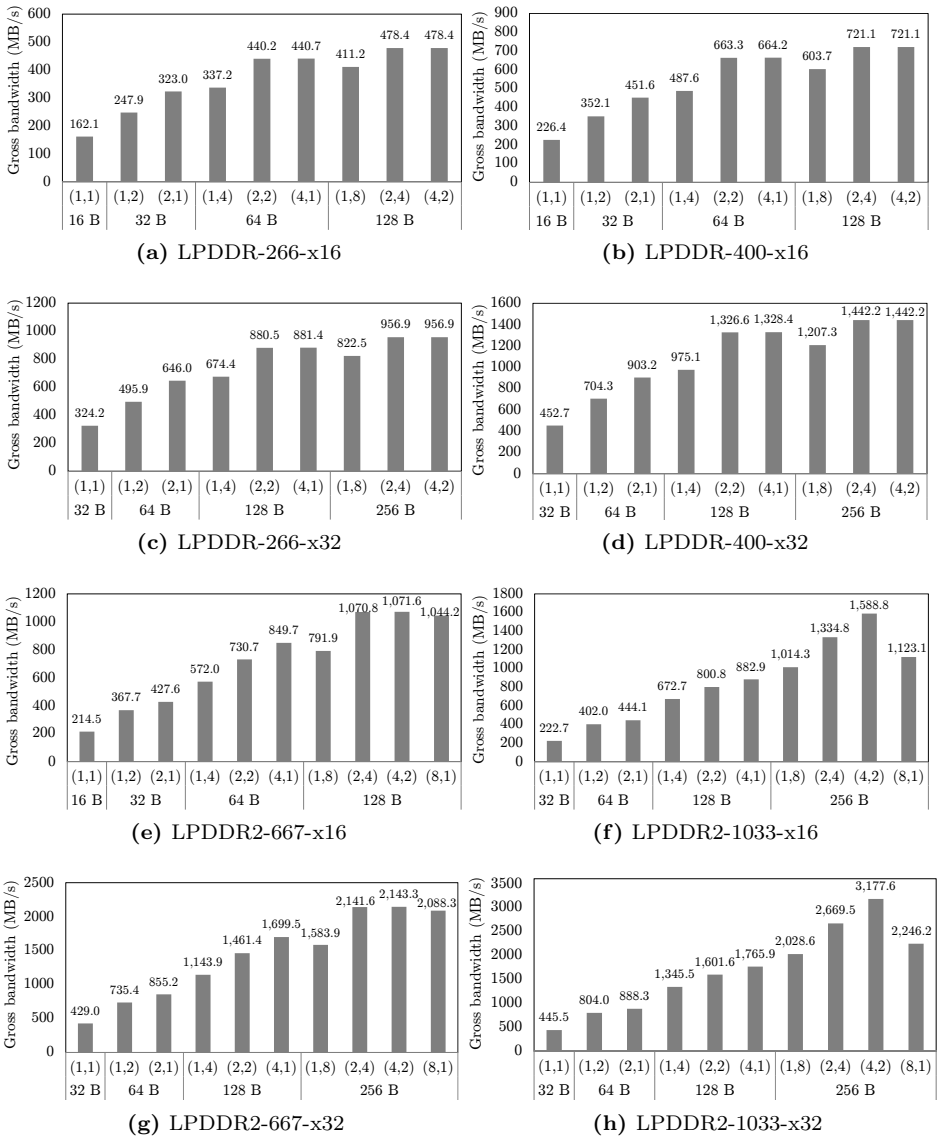
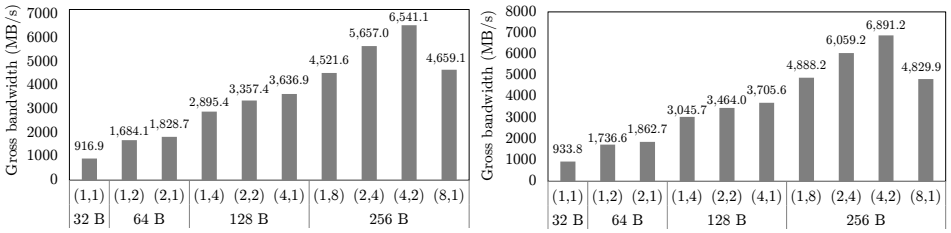
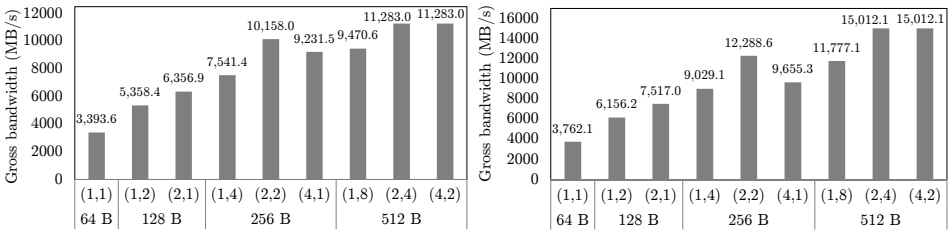


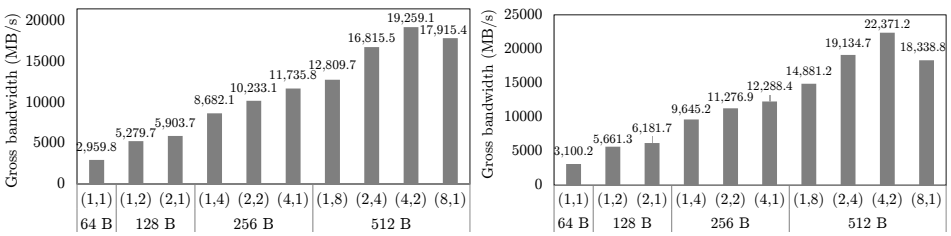
Figure 4.2: Worst-case gross bandwidth of slowest and fastest memory types with different interface widths across and within generations. The X axis shows the different access granularities (in Bytes) and different (BI,BC) combinations.



(i) LPDDR3-1333-x32 **(j)** LPDDR3-1600-x32



(k) WideIOSDR-200-x128 **(l)** WideIOSDR-266-x128



(m) WideIO2DDR-800-x64 **(n)** WideIO2DDR-1066-x64

4.2.2 Aggregate Bandwidth Computation

In a system consisting of multiple clients with different request sizes, it may not always be ideal to select the largest service unit size configuration for the memory controller to get the highest gross bandwidth. This is because there can be several clients with smaller request sizes than the service unit size, which results in poor memory utilization. Hence, we need to consider data efficiency before selecting a memory map configuration. In this section, we present an algorithm to compute the aggregate gross bandwidth requirements of all the clients considering the different request sizes, which helps determining whether or not the gross bandwidth provided by a given service unit size can meet the client requirements.

We use Equation (4.1) to compute the client aggregate bandwidth requirement ($\check{b}_u^{agg}$), where $u \in U$ is the set of service units. The inputs to the equation are the client request sizes s_c (in Bytes), worst-case bandwidth requirements $\check{b}_c$ of all the clients $c \in C$ in the system and set of service unit sizes SU_u^{bytes}. Typically,

the service unit sizes can be 16 B, 32 B, 64 B, 128 B, 256 B and 512 B, as 16 B is the minimum access granularity of a 16-bit DDR memory (with a burst length of 8) and the request sizes of most of the real-world memory clients are less than 512 B [123, 127]. In Equation (4.1), first the *aggregate bandwidth requirement* ($\check{b}_u^{agg}$) of all the clients, as proposed in [52], is computed. This is repeated for different service unit sizes u. For a given access granularity, the minimum bandwidth requirement of a client after considering its data efficiency is computed by dividing its bandwidth requirement with the ratio of its request size and the access granularity.

$$\forall u \in U : \check{b}_u^{agg} = \sum_{c \in C} \frac{\check{b}(c)}{min[1, (s_c / SU_u^{bytes})]} \tag{4.1}$$

Given that we have presented the guidelines for memory map configuration based on worst-case analysis and the algorithm for computation of aggregate client bandwidth requirements, we now proceed to our proposed algorithms to map memory clients to memory channels, in the next section.

4.3 Mapping Clients to Memory Channels

This section presents an optimal method, which is based on an integer programming formulation of the problem, and a heuristic algorithm for mapping memory clients to memory channels. First, we present a formal definition of our system and then a generic optimization problem formulation, which applies to any arbiter belonging to the class of $\mathcal{LR}$ servers, followed by our proposed heuristic algorithm. Finally, we present a performance comparison of the two approaches.

4.3.1 System Model

The set of memory channels is defined as $m \in M$, with each channel having a total memory capacity (in Bytes) given by κ_m. Note that we use the same notation m previously used to denote a memory for the memory channel as well as we consider the memory channels are a set of independent memories. As defined in Section 2.2, the service unit size (in Bytes) of each memory channel is given by SU_m^{bytes}, with a service cycle (in ns) given by SC_m^{ns}. For simplicity, we assume the same service unit size in all memory channels. Also, we assume the same $\mathcal{LR}$ arbiter in all memory channels, possibly with different configurations. For each memory channel $m \in M$, the worst-case gross bandwidth (in MB/s) can be computed for a fixed service unit size SU_m^{bytes} (e.g. see [14]), and is given by b_m^{gross}. Note that we assume the same service unit size configuration in all the memory channels.

Consider a set of clients denoted by $c \in C$, each with a worst-case latency requirement (in ns) given by $\hat{L}_c^{ns}$, minimum bandwidth requirement (in MB/s)

given by $\check{b}_c$, and a total (minimum) memory capacity requirement (in Bytes) given by $\check{\kappa}_c$. Note that the *minimum rate required* by a client can be computed as the ratio of its minimum bandwidth requirement $\check{b}_c$ and the worst-case bandwidth of a channel b_m^{gross}. The worst-case latency requirement (in service cycles) of a client $c \in C$, in each channel $m \in M$ is given by $\hat{L}_{m,c}$, and is defined as:

$$\hat{L}_{m,c} = \lfloor \hat{L}_c^{ns} / SC_m^{ns} \rfloor \tag{4.2}$$

The request size (in Bytes) of requests from a client $c \in C$ is given by s_c. We assume a constant (or maximum) request size for all requests from a single client since it typically holds for the real-time clients under consideration, such as CPUs, hardware accelerators, and LCD controllers. The number of service units in each request of a client $c \in C$ is given by q_c and is defined by Equation (4.3). Since the request sizes, s_c, and access granularity of a memory client, SU_m^{bytes}, are both powers of two, q_c will also be a power of two.

$$q_c = s_c / SU_m^{bytes} \tag{4.3}$$

Each client $c \in C$ has an associated group number given by g_c, which represents the communication dependencies with other clients. In other words, clients that need to communicate through shared memory are assigned the same group number. A summary of the memory system and client parameters and their corresponding notations are given in Appendix B.

4.3.2 Optimal Method for Mapping Clients to Channels

In this section, we present the formulation of the multi-channel mapping problem as an integer programming problem. As mentioned before, we need to minimize the bandwidth allocated to firm real-time clients to maximize the slack bandwidth, which improves the average-case performance of soft real-time clients in the system. Hence, we define our optimization problem as follows:

Find the mapping of clients to the memory channels, the number of service units allocated to those channels, $N_{m,c}$, and a rate, $\rho'_{m,c}$, for each client $c \in C$ in each memory channel $m \in M$, such that all client requirements are satisfied and the sum of rates allocated to all clients is minimized. The optimization problem is defined as:

$$\textit{Minimize:} \sum_{m \in M} \sum_{c \in C} \rho'_{m,c} \tag{4.4}$$

Such that the following nine constraints are satisfied:

Constraint 1: The worst-case latency of each client $c \in C$ after allocation $\hat{L}'_c$ must be less than or equal to its worst-case latency requirement $\hat{L}_c$, and is defined as:

$$\forall c \in C : \ \hat{L}'_c \leq \hat{L}_c \tag{4.5}$$

The service units of every request of a client are allocated across the memory channels such that each client has a (Θ, ρ') pair per channel describing its service. The worst-case latency of a client $c \in C$ in each channel $m \in M$ is then given by $\hat{L}'_{m,c}$, and is defined by Equation (4.6), where $\Theta_{m,c}$ is the service latency of a client c in each channel m. Note that Equation (4.6) is same as Equation (2.2), but adapted for multiple memory channels. For simplicity of presentation, we do not add the fixed delay that depends on the number of pipeline stages in the RTL implementation of the multi-channel memory controller architecture. Note that this fixed delay is about 20 clock cycles in the RTL implementation of our memory controller [53].

$$\forall m \in M, c \in C : \hat{L}'_{m,c} = \Theta_{m,c} + \lceil N_{m,c}/\rho'_{m,c} \rceil \tag{4.6}$$

The worst-case latency of a client $c \in C$ is then the maximum of the worst-case latencies among all the memory channels, which is defined as:

$$\forall m \in M, c \in C : \hat{L}'_{c} = \max_{m \in M} \hat{L}'_{m,c} \tag{4.7}$$

The *max* function is removed to enable formulation as an integer programming problem, and Constraint 1 is then defined as:

$$\forall m \in M, c \in C : \ \hat{L}_c - \hat{L}'_{m,c} \geq 0 \tag{4.8}$$

Constraint 2: The sum of rates allocated to all clients $c \in C$ in each memory channel $m \in M$ must not be greater than 1, i.e., 100%, defined as:

$$\forall m \in M : \sum_{c \in C} \rho'_{m,c} \leq 1 \tag{4.9}$$

Constraint 3: The sum of rates allocated to each client $c \in C$ across all memory channels $m \in M$ should be greater than or equal to its minimum required rate.

$$\forall c \in C : \frac{\check{b}_c}{\sum_{m \in M} b_m^{gross}} \leq \sum_{m \in M} \rho'_{m,c} \tag{4.10}$$

Constraint 4: The sum of service units $N_{m,c}$ of each client $c \in C$ allocated across all memory channels $m \in M$ must be equal to the total number of service units q_c in every request from the client, defined as:

$$\forall c \in C : q_c = \sum_{m \in M} N_{m,c} \tag{4.11}$$

Constraint 5: The number of service units $N_{m,c}$ of each client $c \in C$ allocated to each memory channel $m \in M$ must be a power of two.

To formulate this constraint as a linear constraint, we define two decision variables $t_{m,c}$ and $N'_{m,c}$ for each client in every channel. $t_{m,c}$ is a binary decision variable defined by Equation (4.12) and $N'_{m,c}$ can take a value in the range $0..\log_2[q_c]$. Constraint 5 is then defined by Equation (4.13)

$$t_{m,c} = \begin{cases} 1, & \text{if } N_{m,c} > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{4.12}$$

$$\forall m \in M, c \in C : N_{m,c} = 2^{N'_{m,c}} \times t_{m,c} \tag{4.13}$$

Constraint 6: Each pair of communicating clients, i.e. with the same group number g_c, must be allocated to the same set of memory channels, and the number of service units of the clients allocated in each channel must be proportional for data alignment since they may have different request sizes.

To understand this constraint, consider two communicating clients c_1 and c_2, each with a request size of eight and four service units, respectively, interleaved across two memory channels. Assume that c_1 issues a memory write request Q1 and c_2 reads the data with two read requests P1 and P2. In this case, four service units of request Q1 and two service units of requests P1 and P2 must be allocated to each memory channel, as shown in Figure 4.2, so that the ratio $\textit{Request size}/N_{Ch_n}$ remains same for both clients and results in coherent address translation according to Equation (3.6). Note that this constraint only ensures that the number of service units allocated in each channel are proportional. Furthermore, the service units of all communicating clients must be aligned in each memory channel. As shown in Figure 4.2, the first four service units SU1-SU4 of c_1 must be interleaved across two memory channels so that the response for the first read request from c_2 contains four service units (data) from the continuous logical address space. To ensure this, the sequence generator of c_1 in the multi-channel memory controller must be programmed to route the first four service units of a request to Channel 1 and the next four to Channel 2. For c_2, the sequence generator must be programmed to route the first two service units to Channel 1 and the next two to Channel 2.

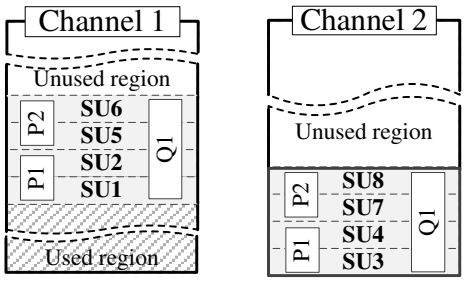


Figure 4.2: Physical memory maps of two memory channels showing request Q1 of size eight service units aligned with requests P1 and P2 of size 4 service units each.

For two communicating clients c_i and c_j, the constraint is defined by Equation (4.14). The decision variable $N'_{m,c}$ is the same as defined under Constraint 5.

The main purpose of this constraint to maintain a particular proportion of service units in channels to which two communicating clients are mapped. This constraint ensures that N'_{m,c_i} and N'_{m,c_j} are selected such that N_{m,c_i} and N_{m,c_j} are proportional. Hence, for every non-zero number of service units of c_i allocated to a memory channel, N_{m,c_i}, a corresponding number of service units in the order of power-of-two of client c_j, $2^{N'_{m,c_j}}$ is allocated to the same channel, and vice versa.

$$\forall m \in M, c_i, c_j \in C, g(c_i) = g(c_j) : N_{m,c_i} \times 2^{N'_{m,c_j}} = N_{m,c_j} \times 2^{N'_{m,c_i}} \tag{4.14}$$

Constraint 7: The total memory capacity of all clients in each channel $m \in M$ must be less than or equal to the channel capacity κ_m, defined by Equation (4.15).

This constraint along with Constraint 4 ensures that the sum of the memory capacities allocated to a client in all memory channels is equal or larger than its total memory capacity requirement.

$$\forall m \in M : \sum_{c \in C} \frac{N_{m,c}}{q_c} \times \check{\kappa}_c \leq \kappa_m \tag{4.15}$$

Constraint 8: For every service unit allocated to a memory channel $m \in M$, there must be a corresponding rate allocated, and vice versa, defined by Equations (4.16) and (4.17).

The decision variable, $t_{m,c}$, is the same as in Constraint 5 and D is a constant with a value larger than the maximum rate, i.e., $D > 1$. Equation (4.16) ensures that when service units are allocated to a channel (according to Constraint 5), a corresponding rate is allocated in the channel. Equation (4.17) ensures that the rate is set to zero when there are no service units allocated to the channel.

$$\forall c \in C, \forall m \in M : (1 - t_{m,c}) \times D + \rho'_{m,c} > 0 \tag{4.16}$$

$$\forall c \in C, \forall m \in M : t_{m,c} \times D - \rho'_{m,c} \geq 0 \tag{4.17}$$

In general, our optimization problem formulation can be used for $\mathcal{LR}$ servers whose service latency is linear or can be linearized, such as TDM with contiguous and distributed slot allocation strategies, by using their worst-case latency derivations in Constraint 1. However, the problem formulation might have to be extended with additional constraints which are specific to the arbiter. In this work, we show how to extend our problem formulation for a contiguous TDM arbiter. In the worst-case latency derivation of contiguous TDM (Equation (2.3)), we can see that for a given frame size, f, the rate that needs to be allocated depends on the latency requirement of a client and the discretization of rate when it is converted to TDM slots. This means that we need to make f a decision variable in the optimization problem formulation for an optimal allocated rate.

Moreover, the allocated rate needs to be optimized considering the over-allocation of bandwidth due to the discretization of rate. To ensure that the allocated rate, $\rho'_{m,c}$, is the discretized rate for a given frame size, we define a decision variable, $\alpha_{m,c}$, which can take a value between 0 and 1, and the allocated rate is then defined by Equation (4.18).

$$\rho'_{m,c} = (\lceil f \times \alpha_{m,c} \rceil)/f \tag{4.18}$$

In essence, this constrains $\rho'_{m,c}$ such that it gets a value which corresponds to an integer number of slots in the TDM table of a given frame size. Finally, we need to add Constraint 9 to the problem formulation to ensure that the frame size is sufficiently large to accommodate the number of slots required by all clients in each memory channel.

Constraint 9: For a TDM arbiter, the frame size, f, must at least be equal to or greater than the sum of the number of slots required by the clients allocated in each memory channel, defined by Equation (4.19)

$$\forall m \in M : f \geq \sum_{c \in C} f \times \rho'_{m,c} \tag{4.19}$$

After presenting the optimal algorithm for mapping memory clients to memory channels, now we proceed to our proposed heuristic algorithm.

4.3.3 A Fast Heuristic Algorithm to Map Memory Clients to Memory Channels

The optimal algorithm for mapping clients to memory channels presented in the previous section may not be scalable for future systems in terms of computation time, as the number of variables and constraints increases drastically with the problem size, i.e. 17 variables and 39 constraints are added for every new client. Hence, we devised a fast heuristic algorithm to map clients to the memory channels that minimizes total memory bandwidth allocated to the clients while considering the client requirements.

Our heuristic algorithm consists of two main steps: (1) *Sorting clients*: We perform a two-step sorting approach. In the first step, we find the minimum number of channels to which each client must be interleaved to meet its latency requirements. Then we create a list of communicating client groups with at least one client in the group requiring more than one memory channel. Note that a client group may have one or more clients. The clients requiring more than one memory channel have tight latency requirements, and hence, we need to map them first to minimize over-allocation of rate. This is because we need to minimize the number of channels to which clients with low latency requirements are mapped to minimize over-allocation of rate, as explained in Section 3.3.1. However, the probability of them being mapped to multiple memory channels are high when the other clients have used most of the bandwidth available in the memory channels.

The second step consists of creating a list of communicating clients sorted in ascending order of their latency requirements. The minimum latency of a client in a group is selected for the sorting. By mapping clients to the memory channels in order of this list, the over-allocation of rate is reduced. The two lists in this step are concatenated to create a single list. (2) *Mapping to the memory channels*: The clients are mapped to memory channels using a first-fit algorithm, which allocates them one by one from the sorted list to the first available channel(s) with enough resources (bandwidth and memory capacity) to satisfy the client requirements. During the mapping process, a configuration process is invoked for each client to determine the interleaving granularity and the rate that needs to be allocated in each channel. This is because, according to Equation (3.5), a higher rate than the requested rate may need to be allocated depending on the latency requirement and the interleaving granularity. Note that whenever we are computing the allocated rate in this algorithm, we consider the discretization of rate that happens when it is finally converted to TDM slots. We proceed by discussing the two steps in detail.

Sorting Clients

As we concluded in Section 3.3.1, we need to minimize the number of channels to which a client is interleaved to minimize the allocated bandwidth. Since we use a first-fit algorithm for mapping clients to memory channels, the last clients are more prone to be interleaved across multiple memory channels during the mapping process, because the available bandwidth and memory capacity in the channels keep reducing. We must hence start mapping the clients that might end up having a larger over-allocation of rate if interleaved across multiple channels.

When a client is interleaved across a number of memory channels, n_{Ch_c}, as expressed by Equation (3.5), the amount of over-allocation of the required rate increases when its latency requirement is lower and the request size, N, is larger. Note that with a larger request size, the discretization of rate increases and the completion latency gets longer. Hence, we map clients with lower latency requirement and larger request sizes first. Since it is hard to sort the clients based on two parameters, i.e. request size and latency requirement, we perform a simple two-step sorting approach:

1. We find the minimum number of channels to which each client must be interleaved to meet their latency requirements. If the request size of a client is large such that its latency requirement (in service cycles), $\hat{L}_c^{sc}$, cannot be satisfied in a single memory channel even after allocating a rate of 100%, it must fundamentally be interleaved across multiple memory channels. The minimum number of channels, $\check{n}_{Ch_c}$, to which the request needs to be interleaved is given by:

$$\check{n}_{Ch_c} = 2^{\lceil \log_2(q_c/\hat{L}_c^{sc}) \rceil} \tag{4.20}$$

In the above equation, $q_c/\hat{L}_c^{sc}$ is rounded to the upper power of two since we need to allocate service units in powers of two for logical-to-physical address translation according to our method presented in Section 3.3.3. When the request size and the number of service units in each memory channel is a power-of-two, the number of channels to which the request is interleaved must also be a power of two to meet the worst-case latency requirement of the client. Consider an example scenario in which $q_c = 8$ service units and $\hat{L}_c^{sc} = 3$ service cycles. In this case, if we consider $\lceil q_c/\hat{L}_c^{sc} \rceil = 3$ channels and with an allocation of the service units of 4, 2 and 2 in each memory channel, respectively, the latency requirement of 3 service cycles cannot be met since it would take 4 service cycles even with $\check{\rho}'_c = 1$. Hence, we need 4 memory channels to successfully map with 2 service units allocated to each memory channel. This means that our heuristic distributes the number of service units, and thereby also the rate, to all memory channels equally when a client is interleaved across multiple memory channels. Note that the optimal method presented in Section 4.3.2 does not have the restriction of interleaving to the number of channels in powers of two, which is supported by our multi-channel memory controller as well.

We make a list of communicating client groups with each group consisting of at least one client requiring more than one memory channel, i.e. $\check{n}_{Ch_c} > 1$. Note that a client group may consist of a single client that does not have any communication requirements. We need to map these clients first because $\check{n}_{Ch_c} > 1$ indicates a lower latency and a larger request size, which must be mapped first to avoid a larger over-allocation of rate. The order of mapping of clients from this list does not matter because we find $\check{n}_{Ch_c}$ for each client after allocating 100% of bandwidth available in each channel, and hence a client with $\check{n}_{Ch_c} > 1$ uses most of the bandwidth of the $\check{n}_{Ch_c}$ memory channels. This means two client groups belonging to this list cannot be mapped to the same set of channels.

2. The remaining client groups with clients requiring $\check{n}_{Ch_c} = 1$ are sorted (using a quick-sorting algorithm) according to the ascending order of the average of the worst-case latency requirements of the clients in each group. This is because the amount of over-allocation of rate increases as the latency requirements get tighter according to Equation (3.5).

Finally, we append the above two lists in-order to make a single list consisting of groups of clients. Mapping of clients from this list to the memory channels is presented in the next section.

Mapping to Memory Channels

The client groups are picked one by one from the sorted list in order and a configuration process, shown in Algorithm 3, is used to find the number of service units,

i.e. interleaving granularity, and the rate that must be allocated in each channel for the number of channels, $\check{n}_{Ch_c}$, to which each client in the group needs to be interleaved. The interleaving granularity, $N_{m,c}$, in every channel is determined by dividing the request size by the number of channels to which the request needs to be interleaved (line 1). Note that $N_{m,c}$ will always a power of two since q_c and $\check{n}_{Ch_c}$ are always powers of two. For the interleaving granularity in each channel, $N_{m,c}$, the new rate $\rho''_{m,c}$ is computed such that it satisfies the latency requirement $\hat{L}_c^{sc}$ by solving Equation (2.3)(line 2). Since the ceiling functions from the latency equation are removed, we added 2 to make the computation conservative. The rate required by the client, $\rho_{m,c}^{req}$, in the channel is then maximum of its required rate and the newly computed rate (line 3). The required rate is divided equally across the number of channels to which the client needs to be interleaved, since we distribute the number of service units evenly among the channels. Note that this approach may not be optimal for clients with high latency requirements, as it might be possible to allocate different rates in different channels without over-allocating the rate. Finally, the allocated rate, $\rho'_{m,c}$, is computed considering discretization of the required rate (line 4).

Algorithm 3 Find interleaving granularity and allocated rate of a client.

Input: Min. number of channels interleaved $\check{n}_{Ch_c}$, request size q_c, worst-case latency $\hat{L}_c^{sc}$ and bandwidth $\check{b}_c$ requirements, worst-case gross bandwidth of a memory channel b_m^{gross}, TDM frame size f.
Output: Number of service units $N_{m,c}$ and rate $\rho'_{m,c}$ allocated to each channel.

1: $N_{m,c} = \frac{q_c}{\check{n}_{Ch_c}}$
2: $\rho''_{m,c} = \frac{(f-\hat{L}_c^{sc}+2)+\sqrt{(f-\hat{L}_c^{sc}+2)^2+4\cdot f\cdot N_{m,c}}}{2\cdot f}$
3: $\rho_{m,c}^{req} = max\left(\frac{\check{b}_c}{b_m^{gross}\cdot\check{n}_{Ch_c}}, \rho''_{m,c}\right)$
4: $\rho'_{m,c} = \frac{\lceil f\times\rho_{m,c}^{req}\rceil}{f}$

Finally, the client is assigned to the number of channels among the set of channels that satisfy its memory capacity and bandwidth requirement using a first-fit algorithm. Note that the memory capacity requirement is divided equally among the channels to which the client is interleaved. When a client needs to be interleaved across multiple memory channels, the algorithm searches among channel combinations of the specific number of required channels. If there are no such number of channels that can satisfy the requirements, $\check{n}_{Ch_c}$ is increased to the next power of two. The configuration process is invoked again to determine the new interleaving granularity and the allocated rate in each channel, and the mapping of clients to the memory channels is repeated. To determine the optimal frame size, the whole mapping process is repeated with different frame sizes, from the lowest value of one to a sufficiently large value. Finally, the successful mapping

with the lowest total allocated rate is selected that satisfies the condition that the sum of rates allocated to all clients in each channel is less than or equal to one.

4.3.4 Algorithm Computational Complexity and Optimality

For a system consisting of $|C|$ clients and $|M|$ memory channels, finding the minimum number of channels for all clients takes $\mathcal{O}(C)$ time, sorting the client groups using a quick sort algorithm $\mathcal{O}(C^2)$ time units (the number of groups will be equal to the number of clients in the worst-case), and mapping each client in C to a memory channel after searching for resource availability in $|M|$ memory channels with all $(\log_2(M)+1)$ possible values of $\check{n}_{Ch_c}$ (i.e. different power of two combinations with M channels) $\mathcal{O}(C \times M \times (\log_2(M)+1))$. In total, our heuristic algorithm takes $\mathcal{O}(C + C^2 + F \times C \times M \times (\log_2(M)+1))$ time, since the mapping process needs to be repeated until an upper bound F on the frame size. Since the number of clients and the frame size will typically be larger than the number of memory channels, i.e. $C \geq M$ and $F \geq M$, respectively, the time complexity of our heuristic algorithm can be expressed as $\mathcal{O}(C^4)$.

Our heuristic algorithm always interleaves to a number of memory channels in powers of two. Hence, we divide the request size, and thereby also the rate, equally when a client is interleaved across multiple memory channels, which is optimal for clients with tight latency requirement as we have seen in Section 4.3.3. However, when the latency requirement of a client is relaxed, its request can be split in different powers of two and allocated to different channels with different rates (according to its bandwidth requirement) at the same time meeting its latency requirement. We do not consider extending the heuristic to support interleaving across a number of memory channels that is not in a power of two for two reasons: (1) This work primarily focuses on mapping of firm real-time clients with tight latency requirements. (2) When request sizes are not evenly distributed across memory channels, the complexity of the mapping process increases since we need to check the bandwidth and memory capacity availability in all possible combinations of memory channels. We evaluate the impact of this restriction on the mapping success ratio of our heuristic algorithm in the experimental section presented next.

4.3.5 Optimal, Heuristic and Existing Mapping Algorithms - Performance Comparison

In this section, we evaluate the mapping success ratio of our two proposed mapping algorithms, optimal and heuristic (presented in Sections 4.3.2 & 4.3.3, respectively), and two existing mapping algorithms, *First-fit* and *Interleave-all*. The First-fit is a basic bin-packing algorithm that picks one client at a time and maps to one of the first memory channels that has enough resources (bandwidth and

memory capacity) available to meet the client requirements. The First-fit algorithm does not interleave memory requests across multiple memory channels, and hence, we used this algorithm to evaluate the benefits of interleaving across multiple memory channels, since our heuristic algorithm is based on first-fit which interleaves memory clients across memory channels. Note that the First-fit algorithm do not consider communicating clients into account. The Interleave-all algorithm maps every client to all memory channels available by distributing the number of service units and rate evenly among all channels. This is the traditional method for mapping in multi-channel memories and has the advantage that only a single Channel Controller is required for all the memory channels [132]. With these algorithms, we span the extreme ends of the design space from no interleaving to full interleaving, and that our solution is configurable within this space. Furthermore, we compare the computation time of our optimal and heuristic algorithms in this section.

Experimental Setup

The experimental setup consists of the optimization problem model implemented in the CPLEX optimization tool [7], implementation of our proposed heuristic, the First-fit and Interleave-all algorithms in C++, for a TDM arbiter. For a fair comparison with the heuristic, the First-fit and Interleave-all algorithms are also run with different TDM frame sizes to determine the optimal frame size with the lowest over-allocation of rate (after discretization) and which satisfies the condition that the sum of rates allocated to all clients in each channel is less than or equal to one. Note that we did not include communication requirements for clients in the use-cases to be fair against the First-fit algorithm, which does not consider communication groups and including them in reality could further reduce the performance of the algorithm. For the implementation of the optimization problem for a TDM arbiter in CPLEX, we substitute its worst-case latency expression given by Equation (2.3) in Equation (4.8) of Constraint 1. Since CPLEX does not support decision variables in the denominator, such as ρ' in Equation (2.3), we multiply the equation by ρ' and the constraint hence becomes quadratic, as expressed by Equation (4.21), making it a Quadratic Constrained Quadratic Problem (QCQP). The two ceiling functions had to be removed to make the problem linear, and hence the service latency and the completion latency are approximated as $f \times (1 - \rho'_{m,c}) + 1$ and $N_{m,c}/\rho'_{m,c} + 1$, respectively, to make the computation conservative.

$$\forall m \in M, c \in C : f \times \rho'^2_{m,c} - \rho'_{m,c} \times (f - \hat{L}_c + 2) - N_{m,c} \geq 0 \tag{4.21}$$

To compare the performance of the optimal method and the heuristic under different scenarios, we used a synthetic use-case generator, which generates memory clients according to a normal distribution function with latency requirements in the range 1-10 μs, bandwidth requirements 1-1000 MB/s and request sizes

64-512 B. We selected these ranges since they cover the following different traffic classes of real memory clients: clients with low average latency requirements, such as LCD controllers and CPUs [125], clients with medium latency requirements, such as H.264 video decoders [12], and clients with relaxed latencies, which includes a wide variety of clients with low and high bandwidth requirements, e.g., graphics processing [125], input processors [123]. We do not consider memory capacity requirements since we did not have sufficient data to define the range for the different traffic classes. We used a 4-channel WideIO 200 MHz DRAM with a service unit size of 64 B as the target memory.

Mapping Success Ratio

Using our proposed heuristic and the First-fit and Interleave-all algorithms, we performed mapping with 200 different use-cases, which are feasible according to the optimal algorithm, with different number of clients (5-25) and requirements. Note that we had to limit the number of use-cases to 200 due to the computation time of the optimal algorithm. In all algorithms, we set an upper bound of 100 for the frame size considering the long computation time of the optimal algorithm, but we assume that this is sufficiently large for our use-cases. The mapping success ratio of the proposed heuristic, First-fit and Interleave-all algorithms normalized to the success ratio of the optimal method is shown in Figure 4.3. Also, the figure shows the average over-allocation of rate for the three algorithms. It can be seen that our proposed heuristic algorithm has the highest mapping success ratio of 93%, followed by the First-fit 81% and Interleave-all with 68%. The Interleave-all has the highest over-allocation of bandwidth as expected. Our heuristic algorithm failed to find a valid mapping for about 7% of the use-cases consisting mainly of clients with relaxed latency requirements. The mapping failed for those use-cases when the total required bandwidth by all clients is more than 95% of the maximum bandwidth capacity of all channels. As we have already seen in Section 4.3.4, our heuristic algorithm distributes the rate evenly across the channels to which a client is interleaved. Since the heuristic algorithm does mapping based on a first-fit algorithm, it could fail for one of the last clients to be mapped (with relaxed latency requirements) which could be allocated with different rates in different channels according to the slack bandwidth available in each channel in order to meet its bandwidth requirement. For all use-cases we considered, our heuristic algorithm allocated clients with tight latency requirements to the same number of channels as the optimal algorithm and hence both of have the same amount of over-allocation of bandwidth.

The First-fit algorithm failed in 19% of use-cases since it did not interleave clients that could have been successfully mapped by interleaving across multiple channels. The Interleave-all algorithm failed for 33% of the use-cases, since it over-allocates a much larger amount of bandwidth than required to meet the latency requirements of clients with tight latency requirements. Figure 4.4 shows the increase in over-allocated bandwidth by the Interleave-all algorithm (of 100

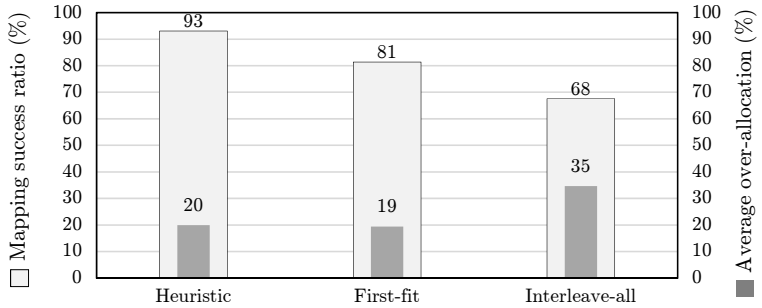


Figure 4.3: Mapping success ratio of the heuristic, First-fit and Interleave-all algorithms normalized to the optimal method. Also is shown the average over-allocation of rate for all the three algorithms.

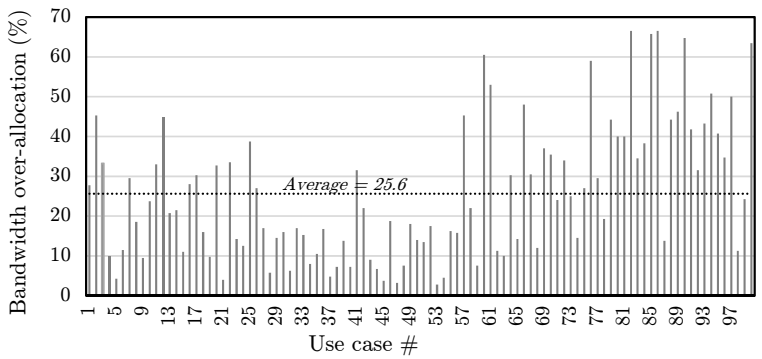


Figure 4.4: Increase in bandwidth over-allocation of the Interleave-all algorithm compared to the optimal algorithm for different use-cases.

feasible mappings) compared to the heuristic and optimal algorithms. Hence, *the traditional approach of interleaving every memory client across all memory channels available is not an efficient method in real-time multi-processor platforms.*

To summarize, we have seen that our heuristic algorithm outperforms the First-fit and Interleave-all algorithms in terms of mapping success ratio. Note that the front-end for First-fit and Interleave-all algorithms could be simpler (lower area and power consumption) as a Channel Selector would not be required. However, the cost in the worst-case latency due to the Channel Selector is one clock cycle, which is negligible. We have shown that both the traditional approaches of interleaving memory request to all the memory channels and not interleaving at all does not efficiently utilize the memory bandwidth and that an alternative solution in the middle is warranted. In the next section, we evaluate the trade-off between algorithm computation time and mapping success ratio of our optimal and heuristic algorithms.

Computation Time Comparison

The computation time of the optimal algorithm in CPLEX and the heuristic algorithm with different number of clients and for different number of memory channels are shown in Table 4.2. Note that the solver takes a significant amount of time to search because of the large design space of the optimization problem. Hence, the time taken by CPLEX shown in this table is for finding the first optimal solution and this is observed from the solutions found by the tool at different time instants until it terminates normally. We considered up to 16 memory channels, as it will be a valid multi-channel memory configuration in the near future [6]. It can be seen that the heuristic algorithm runs much faster (the First-fit and Interleave-all algorithms also run in less than a second) than the optimization tool, and is required to scale to future needs. To summarize, we have seen that our solver-based method finds an optimal solution within few seconds to about 2 hours for small to medium-size systems. However, mapping large future systems require the heuristic algorithm and this comes at a reduction of approximately 7% in success ratio of the mapping.

Table 4.2: Tool vs Heuristic - computation time

Channels	*Clients*	*CPLEX*	*Heuristic*
4	25	7 mins	$<$ 1 sec
	50	25 mins	
	100	2 hrs	
8	25	4 hrs	$<$ 1 sec
	50	1 day	
	100	$>$ 2 days	
16	25	$>$ 3 days	$<$ 1 sec
	50		
	100		

4.4 Case Study: High-Definition Video and Graphics Processing System

In this section, we present a case study where we demonstrate the proposed design-flow for bandwidth-efficient memory subsystem design by applying it to a High-Definition (HD) video and graphics processing system. First, we derive memory subsystem requirements for the video processing system, and then show the design of the memory subsystem.

4.4.1 HD Video and Graphics Processing System Requirements

A HD video (1080p) and graphics processing system with a Unified Memory Architecture (UMA) is shown in Figure 4.5. This system is based on the industrial systems from [123] and [125] combined to create a suitable load for a modern multi-channel memory. The Input Processor (IP) receives the encoded video stream, the Video Engine (VE) decodes the video, the GPU performs post-processing (e.g. video overlay) and finally, the HDLCD Controller (HDLCD) sends the screen refresh by copying all the data to the frame buffer. In addition, the host CPU require memory access to perform system-dependent activities [125]. The GPU and CPU requirements are based on [125], and the IP requirements on [123]. The VE and HDLCD requirements are computed considering the requirements for HD video with a resolution of 1920 × 1080, 8 bpp and 30 fps [28].

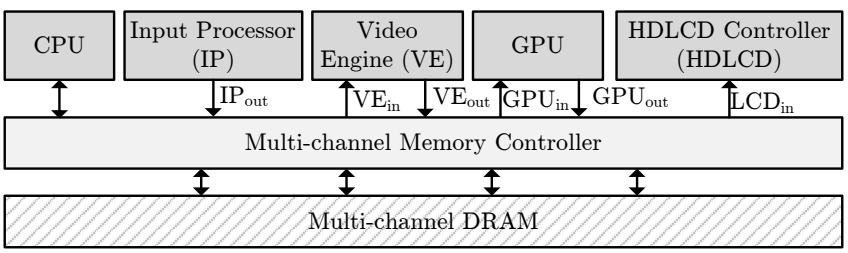


Figure 4.5: Memory-centric architecture of a HD video and graphics processing system.

The Input Processor (IP) receives an H.264 encoded YUV 4:2:0 video stream with a resolution of 720 × 480, 12 bpp, at a frame rate of 30 Hz [123], and writes to memory (IP$_{out}$) at 15.6 MB/s. The VE generates traffic for reading the compressed video and reference frames for motion compensation (VE$_{in}$), and decoder output (VE$_{out}$). The motion compensation is the most bandwidth demanding traffic, and requires at least 769.8 MB/s to decode the video samples at a resolution of 1920 × 1080, 8 bpp, at 30 fps [60, 28]. The bandwidth requirement to output the decoded video image is 93.3 MB/s. These worst-case bandwidth requirements must be satisfied to meet the sufficient bandwidth requirements over a frame period. We consider the request sizes of IP and VE as 64 B and 128 B, respectively.

The bandwidth requirement of the GPU depends on the complexity of the frame to be processed. Assuming processing requirements of 50 MB/frame in the worst-case, the GPU needs on average a bandwidth of 1500 MB/s [125]. The GPU traffic can be split into the pixels read by GPU for processing (GPU$_{in}$) and the frame being rendered by the GPU (GPU$_{out}$). The GPU does not know the complexity of a frame in advance, and hence it completes processing the frame at its maximum rate and remains idle until the next frame. We consider GPU$_{out}$ to have a firm requirement in order to meet the deadline of 16.6 ms for every

frame (for 60 Hz screen refresh). Hence, GPU_{out} needs a bandwidth of at least 248.8 MB/s. GPU_{in} is then allocated a bandwidth of 1251.2 MB/s. We assume a GPU cache-line size of 256 B for the request size. Note that we consider a large request size for the GPU cache line to demonstrate channel interleaving.

The HDLCD is latency critical [123], and continuously scans the frame buffer at a constant rate. For an uncompressed 1080p 60 Hz display at 32 bpp, the HDLCD requires at least 248.8 MB/s to output a frame every 33.3 ms. Note that each rendered frame is displayed twice by the LCD controller. For a LCD burst size of 256 B, the latency requirement would be 1028.8 ns, which is equal to 205 clock cycles for a 200 MHz memory. The CPU is cache-based and has a cache line size of 64 B [125]. We allocate a bandwidth of 150 MB/s to the latency-sensitive system-dependent bandwidth requirements of the CPU [125]. A summary of system requirements is shown in Table 4.3. The clients that need to communicate are assigned the same group number g.

Table 4.3: Memory subsystem requirements for the HD video processing system

Client	$\check{b}_c$ (MB/s)	$\hat{L}_c^{cc}$ (cycles)	s_c (B)	g_c
IP_{out}	15.6	-	64	1
VE_{in}	769.8	-	128	1
VE_{out}	93.3	-	128	2
GPU_{in}	1251.2	-	256	2
GPU_{out}	248.8	205	256	3
LCD_{in}	248.8	205	256	3
CPU	150	-	64	4
Total	2777.5			

4.4.2 Demonstration of Design-Flow

Given that we have derived the different client requirements for the HD video and graphics processing system, we now proceed to apply the proposed design-flow to demonstrate its effectiveness on the derived use-case.

According to our methodology for memory type selection and configuration presented in Section 4.1.2, we first compute the peak bandwidth of all the considered memory types and select those memories that provide a peak bandwidth greater than or equal to the total gross bandwidth requirements of all the clients. In this case study, we consider the mobile memories shown in Table 4.1. Note that for simplicity, we consider memory types on a single die, although it is possible to combine multiple memories forming several memory channels, wider interfaces, or multiple ranks. The computed peak bandwidth of all memory types are shown in Table 4.4, sorted in ascending order of bandwidth. As we can see from Table 4.3, the total worst-case requirements by all the clients is 2777.5 MB/s. Hence, the memory types from LPDDR2-533-x32 onwards are selected for computation of worst-case gross bandwidth.

Table 4.4: Peak bandwidth of different memories.

Memory	*Operating frequency (MHz)*	*Interface width (bits)*	*Number of channels*	*Burst length (BL)*	*Data rate*	*Peak Bandwidth (MB/s)*
LPDDR	133	16	1	8	2	532
LPDDR	208	16	1	8	2	832
LPDDR	133	32	1	8	2	1064
LPDDR2	333	16	1	8	2	1332
LPDDR	208	32	1	8	2	1664
LPDDR2	533	16	1	8	2	2132
LPDDR2	333	32	1	8	2	2664
LPDDR2	533	32	1	8	2	4264
LPDDR3	667	32	2	8	2	10672
LPDDR3	800	32	2	8	2	12800
WideIO	200	128	4	4	1	12800
WideIO	266	128	4	4	1	17024
WideIO2	400	64	4	8	2	25600
WideIO2	533	64	4	8	2	34112

We computed the gross bandwidth of the selected memory types for different service unit sizes according to the guidelines for the configuration of memory map presented in Section 4.2.1. Then, for all the service unit sizes that have worst-case gross bandwidth larger than or equal to the total client bandwidth requirements, which is 2777.5 MB/s, the aggregate bandwidth requirement is computed using the algorithm presented in Section 4.2.2. The computed worst-case gross bandwidth (b_m^{gross}) for the different service unit sizes u and their aggregate rate requirement (b_u^{agg}) for different memories are shown in Table 4.5. Note that the minimum service unit size of Wide IO is 64 B.

In this case-study, we select memories in the order of increasing bandwidth until a successful configuration is found. Note that in reality this order can be based on other factors as well, such as the memory price. It can be seen that LPDDR2-533-x32 provides the required total client bandwidth of 2777.5 MB/s with larger service unit sizes, starting from 256 B onwards. However, its aggregate bandwidth requirement with larger service unit sizes, i.e. from 256 B onwards, is much higher than the gross bandwidth offered by the memory making them an invalid option. This is because of the over-allocation of bandwidth required due to poor data efficiency of the service unit, mainly due to the clients of request sizes 64 B and 128 B. We do not show the results for larger service unit sizes than 512 B as the negative impact of data efficiency increases further with larger service unit sizes. Note that in general it is never beneficial to select service unit sizes larger than the largest client request size since the loss in bandwidth due to data-efficiency is much higher than the gain in gross bandwidth. Hence, we conclude that LPDDR2-533-x32 cannot satisfy the client requirements.

Now we consider LPDDR3-1333-x32, which is a faster memory compared to

Table 4.5: Gross bandwidth of different memories and the required aggregate bandwidth after considering data efficiency.

Memory	*Bandwidth (MB/s)*	*32 B*	*64 B*	*128 B*	*256 B*	*512 B*
LPDDR2-533-x32	b_m^{gross}	445.5	888.3	1765.9	3177.6	3569.4
	$\check{b}_u^{agg}$	-	-	-	5020.6	8388.0
LPDDR3-1333-x32	b_m^{gross}	916.9	1828.7	3636.9	6541.1	8697.0
	$\check{b}_u^{agg}$	-	-	3000.4	4251.7	8392.6
LPDDR3-1600-x32	b_m^{gross}	933.8	1862.7	3705.6	6891.2	10161.6
	$\check{b}_u^{agg}$	-	-	3020.1	4272.5	8434.1
WideIO-SDR-200-x128	b_m^{gross}	-	3393.6	6356.9	10158.0	11283.0
	$\check{b}_u^{agg}$	-	2799.8	2971.9	4215.6	8377.6
WideIO-SDR-266-x128	b_m^{gross}	-	3762.1	7517.0	12288.6	15012.1
	$\check{b}_u^{agg}$	-	2802.8	2988.0	4301.0	8406.8
WideIO2-DDR-800-x64	b_m^{gross}	-	2959.8	5903.7	11735.8	19259.1
	$\check{b}_u^{agg}$	-	2811.8	2996.1	4254.2	8377.7
WideIO2-DDR-1066-x64	b_m^{gross}	-	3100.2	6181.7	12288.4	22371.2
	$\check{b}_u^{agg}$	-	2821.2	3013.6	4300.9	8445.1

LPDDR2-533-x32 and has two memory channels. Note that the gross bandwidth of a multi-channel memory is computed by multiplying the gross bandwidth of a single channel with the number of channels since we assume the same service unit size in all channels. The gross bandwidth of LPDDR3-1333-x32 is more than twice of LPDDR2-533-x32 for the same access granularity size because the former memory is faster and has twice the number of channels. We can see that the aggregate bandwidth requirements for LPDDR3-1333-x32 are lower than the gross bandwidth of service unit sizes 128 B, 256 B and 512 B. Hence, for these access granularities, we performed mapping and configuration of arbiter using our optimal and heuristic algorithms presented in Section 4.3.2 and Section 4.3.3. However, none of the service unit size configurations provided a successful mapping with either algorithm. The same results hold for the faster memory, LPDDR3-1600-x32.

Next, we considered WideIO-SDR-200-x128, which is a single data-rate memory with four memory channels. WideIO-SDR-200-x128 provides sufficient gross bandwidth for all service unit size configurations, 64 B, 128 B, 256 B and 512 B. We performed mapping for all these service unit size configurations. The 64 B configuration failed to provide a successful mapping because there was insufficient bandwidth to meet all requirements. Also, the 512 B configuration failed to map due to its poor data efficiency. However, we got successful mappings with service unit sizes 128 B and 256 B with slack bandwidth of 2118.4 MB/s and 4126.6 MB/s, and TDM frame sizes of 6 and 8, respectively. The optimal algorithm in a solver took about 10 seconds and the heuristic algorithm less than a second to find the successful mappings. The mapping results for service unit sizes 128 B and 256 B are shown in Tables 4.6 and 4.7, respectively. Note that we obtained the same

mapping results with both the optimal and heuristic algorithms.

Table 4.6: Mapping of clients to WideIO-SDR-200-x128 - allocated service units ($N_{m,c}$) & rates ($\rho'_{m,c}$) for a service unit size of 128 B

Client	*Channel 1*		*Channel 2*		*Channel 3*		*Channel 4*	
	$N_{1,c}$	$\rho'_{1,c}$	$N_{2,c}$	$\rho'_{2,c}$	$N_{3,c}$	$\rho'_{3,c}$	$N_{4,c}$	$\rho'_{4,c}$
IP_{out}	0	0	0	0	1	0.17	0	0
VE_{in}	0	0	0	0	1	0.50	0	0
VE_{out}	0	0	1	0.17	0	0	0	0
GPU_{in}	0	0	2	0.83	0	0	0	0
GPU_{out}	2	0.33	0	0	0	0	0	0
LCD_{in}	2	0.33	0	0	0	0	0	0
CPU	1	0.33	0	0	0	0	0	0
Total	5	0.99	3	1.00	3	0.67	0	0.00

Table 4.7: Mapping of clients to WideIO-SDR-200-x128- allocated service units ($N_{m,c}$) & rates ($\rho'_{m,c}$) for a service unit size of 256 B

Client	*Channel 1*		*Channel 2*		*Channel 3*		*Channel 4*	
	$N_{1,c}$	$\rho'_{1,c}$	$N_{2,c}$	$\rho'_{2,c}$	$N_{3,c}$	$\rho'_{3,c}$	$N_{4,c}$	$\rho'_{4,c}$
IP_{out}	0	0	0	0	1	0.13	0	0
VE_{in}	0	0	0	0	1	0.63	0	0
VE_{out}	0	0	1	0.13	0	0	0	0
GPU_{in}	0	0	1	0.50	0	0	0	0
GPU_{out}	1	0.375	0	0	0	0	0	0
LCD_{in}	1	0.375	0	0	0	0	0	0
CPU	1	0.25	0	0	0	0	0	0
Total	3	1.00	2	0.63	2	0.76	0	0.00

It can be seen that, IP_{out} & VE_{in}, VE_{out} & GPU_{in} and GPU_{out} & LCD_{in}, are allocated to the same memory channel to enable communication between them. The total worst-case bandwidth allocated to all clients with 128 B and 256 B service unit sizes are 4238.4 MB/s and 6031.2 MB/s, respectively. This means that the over-allocated rate is 1266.9 MB/s and 1815.6 MB/s. The over-allocation is primarily due to the tight latency requirement of GPU_{out} & LCD_{in}, and secondarily due to the discretization of the rate when the bandwidth requirements are converted to TDM slots.

Given that we have determined the mapping of memory clients to memory channels, we show how to configure our multi-channel memory controller architecture proposed in Section 3.3. Let us consider that we need to configure the multi-channel memory controller for the four-channel WideIO-SDR-200-x128 memory for a service unit size of 256 B, i.e. with the mapping results of Table 4.7. At first, the arbiter in each memory channel needs to be set with a frame size of 8 as we assume the same frame size in all memory channels. Then, for each

client c, each of those arbiters need to be allocated with the rates according to the values of $\rho'_{m,c}$ in Table 4.7. For example, clients GPU_{out} & LCD_{in} needs to be allocated with 3 slots each in Channel 1 corresponding to a rate of 0.375. Finally, the sequence generator of each client needs to be configured to route its service units to the different channels according to the values of $N_{m,c}$. For example, the sequence generators of GPU_{out} & LCD_{in} need to be configured such that their service units are only forwarded to Channel 1.

To summarize, we have demonstrated our proposed design-flow for the bandwidth-efficient design of memory subsystem for the HD video and graphics processing system. We performed mapping of the memory clients, using both our optimal and heuristic algorithms, considering their latency/bandwidth requirements, request sizes and/or communication requirements, for minimal total bandwidth allocated to the clients. We found successful mapping with WideIO-SDR-200-x128, with a slack bandwidth of 1815.6 MB/s, that could be allocated to soft/non-real time clients in the system to improve their performance. Note that memory capacity requirements of the clients generally also impact the interleaving of requests across channels, which we did not consider in this case study

4.5 Summary

Bandwidth-efficient design of memory subsystems in real-time systems is challenging as there are several design choices to be made, such as memory interface width and operating frequency, memory map, interconnect interface width and operating frequency, and mapping of memory clients to the memory channels, while satisfying the memory client requirements. In this chapter, we defined the problem statement and then presented a structured design-flow for bandwidth-efficient design of memory subsystem in real-time systems. Moreover, we have presented the main steps in the design-flow in detail, such as configuration of memory map parameters, computation of aggregate client bandwidth, and mapping of memory clients to memory channels using optimal and heuristic algorithms.

To maximize memory bandwidth utilization, we presented design guidelines for the selection of the memory type (i.e. interface width and operating frequency) and configuration of its memory map based on worst-case analysis of memory types of different configurations across and within generations. The basic idea of our design guidelines is to maximize memory bank parallelism up to four-active command window constraint before increasing the burst size in the same bank.

We presented an optimal method based on an integer programming problem formulation and a fast heuristic algorithm to map memory clients to the memory channels and configure the arbiters in each channel, while minimizing the total bandwidth allocated to the clients. We show that for a use-case scenario consisting of 4 memory channels and up to 100 memory clients, the optimal algorithm in a solver can find a mapping in 2 hours, and our heuristic in less than 1 second. Our heuristic algorithm finds an optimal solution in less than 1 second with up

to 16 memory channels, which clearly outperforms the solver in terms of scaling for future needs. This comes at a cost of 7% reduction in successfully mapped use-cases, which is significantly lower than the failure ratios 19% and 33% of two existing mapping algorithms. Also, by comparing with our proposed approaches, we show that the traditional approaches of not interleaving memory request to any of the memory channels or interleaving across all the channels are not efficient in terms of memory bandwidth utilization.

Finally, we demonstrated the effectiveness of our proposed design-flow step by step on a real use-case scenario by configuring the memory subsystem in a HD video processing system.

Chapter 5

Related work

In this thesis, we present scalable memory subsystem architectures and design methodologies for bandwidth-efficient memory subsystem design in real-time systems. We discuss the related work in this chapter. Section 5.1 introduces the previous works on memory interconnect architectures and Section 5.2 discusses related work on the co-design of memory interconnect and memory controller. Section 5.3 presents the state-of-the-art approaches of interleaving requests across multi-channel memories and multi-channel memory controllers. Finally, Section 5.4 discusses previous works on design methodologies for memory subsystem design.

5.1 Memory Interconnect Architectures

As discussed in Section 1.1.4, existing memory interconnect architectures implementing predictable arbitration policies can be classified into centralized and distributed according to the implementation of the arbitration policy. The centralized implementations of predictable arbitration policies consisting of a tree of multiplexer stages for priority resolution among the clients presented in [118, 39, 88, 20] are not scalable in terms of clock frequency. This is because the number of logic gates in the critical path for multiplexing increases with the number of clients, restricting their maximum synthesizable frequency. Although the scalability issue is addressed using distributed implementation with global arbitration, i.e. globally-arbitrated, using TDM in Network-On-chip (NoC) [55, 45, 113], TDM is not suitable when the clients have diverse bandwidth and latency requirements. For example, the clients with low latency and bandwidth requirements often need

to be allocated more than their required bandwidth to meet their latency requirements, which is not desirable when memory bandwidth is scarce. NoCs using priority-based packet switching [117] provide real-time guarantees, but the buffering of packets in every router stage increases the memory access latency, area and power consumption, as we have seen in Section 3.4.

Memory trees with distributed implementation of several local arbiters, i.e. locally-arbitrated, consisting of 2-to-1 multiplexer stages connected in a tree-like structure and each stage having a RR arbiter are presented in [43, 107], and a similar binary arbitration tree using a First Come First Serve (FCFS) policy in [110]. In the hybrid arbitration tree [50], a combination of RR and TDM arbitration policies were used in which the latency-sensitive clients are scheduled with TDM and bandwidth-demanding clients with work-conserving RR to improve their average-case performance. However, cascading multiple independent arbitration stages leads to larger area and power usage due to the buffering of memory requests at every arbitration stage. Note that the requests are not dropped and rescheduled in the arbitration stages of locally arbitrated interconnects, unlike GAMT.

To summarize, existing centralized implementation of arbitration policies are not scalable in terms of clock frequency with number of clients and the locally-arbitrated distributed implementations suffer from long latencies and large area and power usage due to the buffers in the local arbitration stages. On the other hand, existing distributed memory interconnects using global TDM arbitration (buffer-less) are not suitable for clients with diverse requirements. Also, there is no re-configurable architecture supporting different arbitration policies.

Our proposed generic distributed globally-arbitrated memory tree (GAMT) can be configured to operate as five different arbitration policies and supports work-conservation. The experimental results in Section 3.4 show that GAMT outperforms the centralized implementations by more than four times in terms of clock speed and over 51% and 37% in terms of area and power consumption, respectively, for a given bandwidth.

5.2 Co-Optimization of Memory Interconnect and Memory Controller

Related work on co-optimization of memory interconnect and memory controller can be classified into those that provide guarantees on real-time requirements to the clients and those that focus on improving the average-case performance of the system, respectively. Traditionally, power/performance optimized tree topologies using bus-based interconnects and predictable arbitration policies [50, 40] were used to provide real-time guarantees while accessing a shared memory. Since bus-based interconnects were not scalable in larger SoCs, statically scheduled TDM [51, 57, 97, 122, 112, 130, 11] and priority-based NoCs [116] were introduced. Although a tree topology using TDM NoC, *channel trees* [55], has been

proposed for accessing shared resources to reduce the worst-case latency using a fully pipelined response path, it was not optimized for area or power consumption. Optimization of a TDM NoC in terms of area and power consumption was done in [120] by coupling it with a memory interface and removing the buffers and flow control by implementing a Direct Memory Access (DMA) access table inside a NI. However, the approach is specific for DMA clients and is not applicable for clients with diverse requirements on bandwidth and/or latency. At the application level, DRAM-aware mapping of application tasks to the processing nodes exploiting bank-level parallelism [74] has been proposed, and memory-centric scheduling approach in which statically computed TDMA schedules are made for each core to access the memory system to avoid memory access contention, allowing applications to be verified in isolation[133]. However, those were not optimized in terms of area and/or power consumption.

Other related work that focus on improving the average-case performance of the memory clients include a memory-centric NoC design that explores the benefits of a dedicated NoC for shared DRAM access by funneling the traffic from different processing cores of different data-bus widths to the memory using optimized width converters [114]. Another memory-centric NoC design includes a connectionless tree topology NoC for shared memory access that multiplexes multiple clients to one bus master are proven to reduce average latency and hardware cost as opposed to a general connection-oriented NoC [110]. At the architecture level, DRAM traffic-aware NoC routers [65] and network interfaces [37] exploit bank-level parallelism and minimize bus turnaround time by grouping read and write transactions to improve memory utilization. Also, memory controllers that interacts with the NoC and make command scheduling decisions based on the congestion information from the NoC [79] improves the average-case performance, but do not provide guarantees on bandwidth and/or latency to the clients.

To the best of our knowledge, there exists no previous work that optimizes the interconnect with the memory controller in terms of area, power consumption and performance, while providing real-time guarantees on bandwidth and/or latency to the clients.

Our proposed Coupled Memory Interconnect (CMI) architecture can be used to couple a globally-arbitrated memory interconnect with a real-time memory controller and configure the interconnect parameters for minimal area and/or power consumption. As shown in Section 3.4, coupling the NoC and memory controller using our approach saves over 45% in guaranteed latency, 24% and 20% in area, 27% and 19% in power consumption, for two different NoC types and with different DRAM generations, for a system consisting of 16 memory clients.

5.3 Multi-Channel Memory Access

Among the previous work related to accessing multi-channel memories, some exploit the benefits of interleaving data across multiple memory channels. [12, 101, 137, 31] proposed interleaving data across the memory channels such that all channels are accessed by a single transaction to improve average-case performance. Similarly, [32] proposed splitting the traffic within a logical address region across multiple memory channels to reduce average latency. Mechanisms for efficient data placement to reduce average memory access latency in a system comprising multiple memory controllers is proposed by [24]. However, all of them focus on the improvement of average-case performance, and do not consider providing guarantees on bandwidth and latency to firm real-time applications.

The rest of the related work focuses on memory controller architectures and logical-to-physical address translation for multi-channel memories. [135] proposed a parallel-access mechanism in which two separate memory controllers are used to control eight memory channels of a 3D-DRAM and the architecture by [85] has every processing element allocated to its own local DRAM channel with a memory controller, and a custom crossbar is used to route incoming traffic from other processing elements. Also, the multi-channel NAND flash memory controller by [103] and the multi-channel memory controller architecture by [29] uses a crossbar switch for routing traffic across multiple memory channels with a mapping table that stores the logical-to-physical address translation. [134] presented a memory controller architecture for fine-grained DRAM access of memory chips in a DIMM by grouping them in logical sub-ranks of different interface widths and accessing them concurrently. However, neither of the aforementioned memory controller architectures provide any firm performance guarantees and hence they cannot be used for formal verification of firm real-time clients. Conversely, even though there are real-time memory controllers that provide bounds on memory performance [105, 13, 108, 115, 25, 131, 82, 62], they do not consider multi-channel memories and interleaving memory requests across multiple memory channels, i.e., they do not support an efficient mapping of memory clients to memory channels, which could lead to larger design costs.

To summarize, presently there is no real-time memory controller with a suitable logical-to-physical address translation method for multi-channel memories that allows memory requests of clients to be interleaved across different memory channels with different interleaving granularities. Our proposed multi-channel memory controller (MCMC) architecture in Section 3.3 with the novel method for logical-to-physical address translation can interleave memory requests across multiple memory channels at different granularities. Moreover, MCMC provides real-time guarantees on memory bandwidth and latency to firm real-time clients.

5.4 DRAM Subsystem Design Methodology

Related work on design methodologies for DRAM subsystems focus on memory selection and configuration and arbiter configuration. A design-space exploration of DRAM system-level parameters is performed in [63] and [36], although the design choices are targeting non-real-time systems. However, some of their conclusions, such as exploiting bank-level parallelism and selecting the correct transaction size to reduce latency, hold true for any system. In contrast, analysis on DRAMs is performed with real-time memory controllers that provide bounds on worst-case bandwidth and latency in [105, 13, 108, 115, 25, 131, 82, 62]. However, none of these explores different memory configurations to determine the optimal operating points, nor do they specifically consider memories for mobile devices including 3D-stacked DRAMs.

In [12] and [101], an average-case analysis of off-chip multi-channel memories is performed to evaluate the performance of multiple memory channels offered by 3D-stacked DRAMs. However, none of these works compare the real-time performance of mobile memories across generations. [34] compares different memory architectures for mobile devices, and a comparison of parallel interface DRAMs, such as LPDDR2 and Wide-IO, is made with serial interface memories in terms of bandwidth and power consumption. A methodology is proposed in [76] to select the memory configuration for a mobile device based on storage capacity, throughput, latency, power, cost and thermal concerns. However, none of these work consider providing bounds on bandwidth to real-time applications.

Optimal configuration of TDM arbitration policy has been studied by many in the past [109, 56, 86, 19]. However, none of them consider mapping of service units to the multiple memory channels while allocating the rates in each channel. To the best of our knowledge, there is no prior work that proposes a methodology for the bandwidth-efficient design of memory subsystem in real-time system, which involves selection and configuration of the memory and mapping of the memory clients to the memory channels.

Our proposed design-flow performs a bandwidth-efficient design of memory subsystems in real-time systems considering various client requirements on memory bandwidth, latency, communication, and capacity. In addition, the design-flow performs mapping of clients to the memory channels using two different methods, one an optimal algorithm based on integer program formulation of the problem and the other, a heuristic algorithm.

5.5 Summary

Existing centralized memory interconnect architectures are not scalable as the critical path in the priority resolution logic increases with the number of clients while the locally-arbitrated distributed memory interconnects incur large area, power and latency overhead due to their decoupled arbitration stages. Moreover,

the existing globally-arbitrated distributed memory interconnects support only TDM, which is not suitable in reconfigurable systems when the client requirements are dynamic and diverse. Our proposed generic distributed globally-arbitrated memory tree (GAMT) can be configured with five different arbitration policies and can be run up to four times faster compared to centralized architectures.

Most of the previous work on the optimization of memory interconnect focus only on average-case performance improvement. On the other hand, existing distributed memory interconnects providing real-time guarantees are decoupled from the memory controller, i.e. they run at unsynchronized clock frequencies. This requires a bus-based memory interconnect with a predictable arbitration policy in front of the memory controller, which increases the area usage, power consumption and the worst-case latency of a memory request. Our proposed coupled memory interconnect (CMI) removes the additional arbitration point in front of the memory controller by generating the clock frequencies for the memory interconnect and the memory controller from the same clock source and aligning their clock edges at their service cycle boundaries. CMI has lower area, power and latency compared to a decoupled architecture.

Previous solutions for multi-channel memory access either interleave memory request across all the memory channels and/or are suitable only for the average-case performance improvement. Moreover, existing multi-channel memory controller architectures do not provide any real-time guarantees on memory performance to the clients. Our multi-channel memory controller (MCMC) architecture with a novel logical-to-physical address translation allows memory requests to be interleaved across different memory channels at different interleaving granularities and different rates allocated to them.

Existing design methodologies for selecting and configuring a DRAM device are not suitable for real-time systems as they do not consider providing real-time guarantees to the clients. On the other hand, previous work on optimal arbiter configuration do not consider a multi-channel memory. Our proposed design-flow performs bandwidth-efficient design of memory subsystem in real-time systems. The design-flow includes methodologies and algorithms for memory selection, configuration, and optimal mapping of clients to the memory channels.

Chapter 6

Conclusions and future work

In heterogeneous multi-processor platforms for real-time systems, main memory, i.e. the off-chip DRAM, is shared between multiple memory clients for cost and communication reasons. Due to the continuous demand for higher memory bandwidth of applications, memories operating at high frequencies and/or with multiple memory channels, i.e. multi-channel memories, are introduced. State-of-the-art real-time memory subsystems consist of a real-time memory controller and a memory interconnect with a predictable arbitration policy that multiplexes requests arriving from different clients to the memory controller. Existing memory subsystems cannot support faster memories and are not suitable for multi-channel memories. This is because the memory interconnect cannot be synthesized at higher clock frequencies with a large number of clients and the memory controller does not support interleaving of memory requests across multiple memory channels of a multi-channel memory. Additionally, existing memory interconnects are decoupled from the memory controller, which consumes large area and power consumption and increase the memory request latency. On the other hand, while designing a memory subsystem for a real-time system, there are several design choices that need to be made, such as the memory type selection, memory controller configuration and mapping of memory clients to memory channels in a multi-channel memory. With an increasing number of applications being integrated into multi-processor platforms for real-time systems, the design complexity of such platforms is increasing as well. Currently there exist no design methodologies for faster and efficient design of memory subsystems in real-time

systems.

In the remainder of this chapter, Section 6.1 summarizes the different contributions presented in this thesis and Section 6.2 discusses some promising future work.

6.1 Conclusions

This section summarizes our main contributions presented in this thesis, i.e. the generic globally-arbitrated memory tree (GAMT), the coupled memory interconnect (CMI), multi-channel memory controller (MCMC) and the design-flow for bandwidth-efficient design of memory subsystems in real-time systems.

6.1.1 Globally-Arbitrated Memory Tree

To address the scalability issue with the present memory interconnects, we proposed a generic, distributed and globally-arbitrated memory tree (GAMT) that can be configured with five different arbitration policies that have been proposed for shared memory access in real-time systems. Moreover, GAMT optionally supports work-conservation for the different arbitration policies. GAMT consists of pipelined 2-to-1 multiplexers connected in a tree-like structure with dedicated accounting and priority assignment (APA) logic per client at the leaves of the tree and the memory controller at the root. The APA logic keeps track of the eligibility status of a client to get scheduled and all the clients schedule requests according to a single global schedule. During a scheduling interval, the APA logic sets a priority level on the priority lines and the priorities of all the clients gets resolved at the multiplexers, i.e. each multiplexer grants access to the client with the highest priority. Hence, the request with the highest priority in a scheduling interval reaches the memory controller and the remaining requests are dropped. The dropped requests are then rescheduled during the next scheduling interval. Due to the distributed APA and the priority resolution, GAMT can be run up to four times faster compared to traditional centralized architectures. Moreover, it has performance gain over 51% and 37% in terms of bandwidth/power and bandwidth/area trade-off, respectively. Note that GAMT consumes more power than centralized implementations due to the larger switching activity.

6.1.2 Coupled Memory Interconnect

We presented a coupled memory interconnect (CMI) architecture that can be used to couple a globally-arbitrated memory interconnect, such as TDM NoCs and GAMT, to the memory controller without the decoupling buffers and the additional bus-based interconnect in front of the memory controller, thereby reducing area usage, power consumption and worst-case latency. The basic idea is to generate the clocks for the memory interconnect and the memory controller

from the same clock source such that the clock edges are aligned at the interconnect and memory service cycle boundaries. The service unit size is made equal in both interconnect and in the memory controller. We show that by coupling the TDM NoC and memory controller using our proposed architecture minimizes the guaranteed latency by 45%, area usage by 24% and 20%, and power consumption by 27% and 19%, for two different TDM NoC types, for a system consisting of 16 memory clients.

6.1.3 Multi-Channel Memory Controller

For efficient utilization of a multi-channel memory, we proposed a multi-channel memory controller (MCMC) architecture that consists of a dedicated channel selector for each client, which routes the service units to the different memory channels according to the configuration in a sequence generator. For each memory channel, there is a dedicated channel controller with a memory interconnect employing a predictable arbitration policy that multiplexes the requests arriving from the different channel selectors. We proposed a novel method for logical-to-physical address translation that allows interleaving memory requests across the different channels with different interleaving granularities. Finally, we demonstrated the real-time guarantees on bandwidth and latency provided by our multi-channel memory controller architecture by experimental evaluation.

To summarize, a completely scalable memory subsystem for real-time systems can be realized using the MCMC architecture for the multi-channel memory with each channel having a GAMT coupled with the channel controller using our CMI architecture.

6.1.4 Design-Flow for Bandwidth-Efficient Memory Subsystem Design

Our proposed design-flow for the bandwidth-efficient design of memory subsystems in real-time system perform the memory type selection, memory controller configuration and mapping of clients to the channels, while considering the real-time requirements of the clients. The design-flow consists of four main steps. In the first step, the memory types whose peak bandwidth is greater than or equal to the gross bandwidth requirement of all the clients are selected. Then, the worst-case gross bandwidth for the different service unit sizes are computed according to our proposed design guidelines in the second step. In the third step, the aggregate bandwidth requirement of all clients for the different service unit size configurations is computed considering the client requirements and specifications. Then, those service unit sizes are selected with gross bandwidth that satisfies the aggregate bandwidth requirement. Finally, for all selected service unit size configurations, the clients are mapped to the memory channels using one of two proposed algorithms. One is an optimal algorithm based on an integer programming formulation of the mapping problem and the other a fast heuristic

algorithm. We compared the performance of the mapping problem formulation in a solver and the heuristic algorithm against two existing mapping algorithms in terms of computation time and mapping success ratio. Our analysis show that the traditional approaches of interleaving memory requests across all memory channels and no interleaving at all results in poor memory utilization. We show that a valid mapping can be found in 2 hours using the solver and in less than 1 second with less than 7% mapping failure using the heuristic for realistically sized problems. Also, we show that our heuristic algorithm finds a valid mapping in less than 1 second with up to 16 memory channels, which clearly outperforms the solver in terms of scaling for the future needs. Finally, we performed a case-study of designing the memory subsystem in a High-Definition (HD) video and graphics processing system to emphasize the practical applicability and effectiveness of our design-flow.

6.2 Future work

6.2.1 Multi-Channel Memory Controller for Mixed Time-Criticality Systems

Multi-channel memories, such as WideIO DRAMs, are essential to meet the bandwidth/power demands of future mobile systems. In this thesis, we considered only firm real-time memory clients for the design and configuration of the multi-channel memory controller. However, there can be a mix of firm and soft real-time applications in a real-time system. Hence, different architectural choices for the multi-channel memory controller for improving the average-case performance of soft real-time applications, while providing real-time guarantees to the firm real-time applications need to be explored. The different architectural choices include selection of different arbiter types, memory controller configurations in different memory channels and selection of interleaving schemes. In this work, we assumed the same arbiter type and channel controller configuration in all memory channels. However, it might be beneficial to have different arbiter types in different memory channels as the client requirements are quite diverse. Moreover, there will be an impact on the optimal mapping of memory clients to the channels if different service unit sizes are considered in different channels as the clients come with different request sizes.

6.2.2 Real-Time Host Controller for HMC

Hybrid Memory Cube (HMC) [6] is a new DRAM architecture using Through-Silicon-Via (TSV) technology that sets new standard for memory performance, power consumption and cost. HMC consists of 16 memory channels including a configurable channel (memory) controller in the base logic layer for each memory channel. Moreover, every channel is made accessible via each IO link using a

crossbar switch in the logic layer. To access a memory channel using the crossbar switch, a communication protocol as specified by the standard needs to be followed. The main challenge in this work is to devise a predictable host controller for the HMC that can interleave memory requests across the memory channels, while accessing the channels using the crossbar. The access time for a memory channel needs to be bounded in order to provide real-time guarantees to the clients. As the crossbar allows connections to any memory channel via an IO link using a communication protocol, the access time of a memory channel will vary depending on the IO link through which the request is sent. Hence, bounding the memory access time may not be straightforward. Moreover, the mapping of the clients to the memory channels will have an impact on the memory access time.

6.2.3 Heterogeneous Multi-Channel Memory Subsystem

In this thesis, we considered a homogeneous multi-channel memory and channel controller configuration, i.e. all memory channels are of same memory type and memory controller configuration. However, it could be beneficial to have different memory types and configurations in the memory channels. For example, memories with wider interfaces and low operating frequencies are more efficient compared to memories with narrow interfaces and higher operating frequencies [48]. On the other hand, memories with narrow interface width that provide smaller access granularities are suitable for clients with small request sizes for better data-efficiency. Also, different service unit sizes in each channel according to the client request sizes might have an impact on the data-efficiency. Hence, our proposed mapping algorithms need to be extended to support heterogeneous multi-channel memories and channel controller configurations. Also, the multi-channel memory controller architecture needs to be extended to support a heterogeneous memory subsystem.

6.2.4 Heterogeneous GAMT Operation

As we have seen in Section 3.1, our proposed generic and globally-arbitrated memory tree (GAMT) can be configured to run as one of five different arbitration policies. The GAMT architecture allows the different clients to be run with different arbitration policies at the same time as well, which could provide certain benefits. For example, some clients can have non-work-conserving TDM and be isolated from other clients, while others use work-conserving priority-based scheduling. In this case, an unused slot of a client running with TDM can be used by a backlogged client running with a priority-based arbiter, which improves its average-case performance. However, a detailed timing analysis needs to be performed to determine the impact on real-time guarantees. In addition to using different arbitration policies for different clients, each client can be switched between multiple arbitration policies at run-time. For example, this could be beneficial for soft real-time applications that has dynamic service requirements [77].

However, the switching from one arbitration policy to another might take some time, which might impact the real-time guarantees of the clients. The impact of running multiple arbitration policies and safe switching between arbitration policies needs to be analyzed and the potential benefits needs to be evaluated experimentally.

6.2.5 Network-on-Chip Based Memory Tree for a Multi-Channel Memory

In this thesis, we proposed a memory interconnect per memory channel in a multi-channel memory. This means, for a large number of clients, we need to use a distributed memory interconnect, such as GAMT or a TDM NoC. When using the TDM NoC, it must be connected in a tree-like structure and coupled with the channel controller. However, instead of using a NoC tree per memory channel, it must be possible to use the NoC in other topologies while maintaining the coupling with the memory controller. In this case, the NoC must be able to route the traffic to the memory channel according to the interleaving configured in the channel selector. Such a NoC based memory tree may have lower area and power consumption compared to the tree topology currently proposed. This requires a methodology to be devised as well to configure the TDM slots in the routers of the NoC.

Bibliography

[1] http://www.dramexchange.com/.

[2] SimpleScalar 2.0. http://www.simplescalar.com.

[3] International Technology Roadmap for Semiconductors (ITRS): System Drivers, 2011. http://www.itrs.net.

[4] Wide I/O Single Data Rate Specification (JESD229), 2012. http://www.jedec.org.

[5] International Technology Roadmap for Semiconductors (ITRS), 2013. http://www.itrs.net.

[6] Hybrid Memory Cube (HMC), 2014. http://www.hybridmemorycube.org.

[7] IBM ILOG CPLEX Optimizer. http://www.ibm.com, 2014.

[8] Wide I/O 2 JEDEC Standard (JESD229-2), 2014. http://www.jedec.org.

[9] JEDEC Solid State Technology Association. http://www.jedec.com, 2015.

[10] Samsung. http://www.samsung.com, 2015.

[11] Tilera Corporation. http://www.tilera.com/, 2015.

[12] E. Aho, J. Nikara, P. Tuominen, and K. Kuusilinna. A case for multi-channel memories in video recording. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 934–939, 2009.

[13] B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, 2011.

[14] B. Akesson and K. Goossens. *Memory Controllers for Real-Time Embedded Systems*. Embedded Systems Series. Springer, first edition edition, 2011.

[15] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 251–256, New York, NY, USA, 2007. ACM.

[16] B. Akesson, A. Hansson, and K. Goossens. Composable Resource Sharing Based on Latency-Rate Servers. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 547–555, Aug 2009.

[17] B. Akesson, W. Hayes, and K. Goossens. Classification and Analysis of Predictable Memory Patterns. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 367–376, Aug 2010.

[18] B. Akesson, W. Hayes, and K. Goossens. Automatic Generation of Efficient Predictable Memory Patterns. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 177–184, Aug 2011.

[19] B. Akesson, A. Minaeva, P. Sucha, A. Nelson, and Z. Hanzalek. An Efficient Configuration Methodology for Time-Division Multiplexed Single Resources. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[20] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 3–14, Aug 2008.

[21] I. F. Akyildiz, D. M. Gutierrez-Estevez, and E. C. Reyes. The Evolution to 4G Cellular Systems: LTE-Advanced. *Phys. Commun.*, 3(4):217–244, Dec. 2010.

[22] R. Andoga, L. Fozo, and L. Madarasz. Digital Electronic Control of a Small Turbojet Engine - MPM 20. In *Intelligent Engineering Systems, 2008. INES 2008. International Conference on*, pages 37–40, Feb 2008.

[23] ARM Limited. *AMBA AXI Protocol Specification*, 2003.

[24] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[25] S. Bayliss and G. Constantinides. Analytical synthesis of bandwidth-efficient SDRAM address generators. *Microprocess. Microsyst.*, 36(8):665–675, Nov. 2012.

[26] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

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

[28] A. Bonatto, A. Soares, and A. Susin. Multichannel SDRAM controller design for H.264/AVC video decoder. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 137–142, 2011.

[29] C. Bouquet. Optimal Multi-channel Memory Controller System. Patent number: 6643746, 2000.

[30] S. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 396–407, Dec 2003.

[31] F. Cabarcas, A. Rico, Y. Etsion, and A. Ramirez. Interleaving granularity on high bandwidth memory architecture for CMPs. In *Embedded Computer Systems (SAMOS), International Conference on*, pages 250–257, 2010.

[32] P. Casini. SoC Architecture to Multichannel Memory Management Using Sonics IMT. White paper, 2008. Sonics, inc.

[33] K. Chapman. Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources. Application Note. In *http://www.xilinx.com*, 2012.

[34] Y. Choi, H. Jeong, and H. Kim. Future evolution of memory subsystem in mobile applications. In *Memory Workshop (IMW), 2010 IEEE International*, pages 1–2, May 2010.

[35] R. Cruz. A calculus for network delay. II. Network analysis. *Information Theory, IEEE Transactions on*, 37(1):132–141, 1991.

[36] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 62–71, 2001.

[37] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen. A low-latency and memory-efficient on-chip network. In *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pages 99–106, May 2010.

[38] B. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. de Massas, F. Jacquet, S. Jones, N. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[39] G. Dimitrakopoulos, C. Kachris, and E. Kalligeros. Scalable Arbiters and Multiplexers for On-FGPA Interconnection Networks. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 90–96, Sept 2011.

[40] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *Design Test of Computers, IEEE*, 18(5):21–31, Sep 2001.

[41] ISSCC Trends, 2013. http://isscc.org/doc/2013/2013_Trends.pdf.

[42] Freescale Semiconductor, Inc. *QorIQ P4080 Communications Processor Product Brief*, Document Number: P4080PB, Rev. 1, 09/2008 edition, 2008.

[43] J. Garside and N. Audsley. Prefetching across a shared memory tree within a Network-on-Chip architecture. In *System on Chip (SoC), 2013 International Symposium on*, pages 1–4, Oct 2013.

[44] M. D. Gomony, B. Akesson, and K. Goossens. Architecture and Optimal Configuration of a Real-Time Multi-Channel Memory Controller. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1307–1312, 2013.

[45] M. D. Gomony, B. Akesson, and K. Goossens. Coupling TDM NoC and DRAM Controller for Cost and Performance Optimization of Real-Time Systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2014.

[46] M. D. Gomony, B. Akesson, and K. Goossens. A Real-Time Multichannel Memory Controller and Optimal Mapping of Memory Clients to Memory Channels. *ACM Trans. Embed. Comput. Syst.*, 14(2):25:1–25:27, Feb. 2015.

[47] M. D. 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 *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 193–198, 2015.

[48] M. D. Gomony, C. Weis, B. Akesson, N. Wehn, and K. Goossens. DRAM Selection and Configuration for Real-Time Mobile Systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 51–56, 2012.

[49] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow. *SIGBED Rev.*, 10(3):23–34, Oct. 2013.

[50] K. Goossens, O. P. Gangwal, J. Roever, and A. P. Niranjan. Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends. In *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15. Kluwer, 2004.

[51] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 306–311. ACM, 2010.

[52] S. Goossens, T. Kouters, B. Akesson, and K. Goossens. Memory-map selection for firm real-time SDRAM controllers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 828–831, March 2012.

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

[54] P. Gumming. The TI OMAP Platform Approach to SoC. *Winning the SoC revolution: experiences in real design*, 2003.

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

[56] A. Hansson, K. Goossens, and A. Rădulescu. A Unified Approach to Constrained Mapping and Routing on Network-on-chip Architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pages 75–80, New York, NY, USA, 2005. ACM.

[57] A. Hansson, M. Subburaman, and K. Goossens. Aelite: A flit-synchronous Network on Chip with composable and predictable services. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 250–255, April 2009.

[58] N. Haron and S. Hamdioui. Why is CMOS scaling coming to an END? In *Design and Test Workshop, 2008. IDT 2008. 3rd International*, pages 98–103, Dec 2008.

[59] J. Henkel. Closing the SoC design gap. *Computer*, 36(9):119–121, Sept 2003.

[60] H. Hongqi, X. Jiadong, D. Zhemin, and S. Jingnan. High Efficiency Synchronous DRAM Controller for H.264 HDTV Encoder. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 373–376, 2007.

[61] H.Vuong. Mobile Memory Technology Roadmap. Mobile Forum, May 2013. http://www.jedec.org.

[62] Hyoseung Kim and de Niz, D. and Andersson, B. and Klein, M. and Mutlu, O. and Rajkumar, R. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154, April 2014.

[63] B. Jacob. A case for studying DRAM issues at the system level. *Micro, IEEE*, 23(4):44–56, July 2003.

[64] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[65] W. Jang and D. Pan. Application-Aware NoC Design for Efficient SDRAM Access. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(10), 2011.

[66] JEDEC Solid State Technology Association. *DDR SDRAM Specification*, JESD79C edition, 2003.

[67] JEDEC Solid State Technology Association. *DDR2 SDRAM Specification*, JESD79-2F edition, 2009.

[68] JEDEC Solid State Technology Association. *DDR3 SDRAM Specification*, JESD79-3E edition, 2010.

[69] JEDEC Solid State Technology Association. *JEDEC Low Power Double Data Rate (LPDDR) SDRAM Standard*, JESD209B edition, 2010.

[70] JEDEC Solid State Technology Association. *JEDEC Low Power Double Data Rate (LPDDR2) SDRAM Standard*, JESD209-2E edition, 2010.

[71] JEDEC Solid State Technology Association. *JEDEC Low Power Double Data Rate (LPDDR3) SDRAM Standard*, JESD209-3 edition, 2012.

[72] JEDEC Solid State Technology Association. *DDR4 SDRAM Specification*, JESD79-4 edition, 2014.

[73] JEDEC Solid State Technology Association. *JEDEC Low Power Double Data Rate (LPDDR4) SDRAM Standard*, JESD209-4 edition, 2014.

[74] X. Jin, N. Guan, Q. Deng, and W. Yi. Memory Access Aware Mapping for Networks-on-Chip. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 339–348, Aug 2011.

[75] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[76] A. Kahng and V. Srinivas. Mobile system considerations for SDRAM interface trends. In *System Level Interconnect Prediction (SLIP), 2011 13th International Workshop on*, pages 1–8, June 2011.

[77] S. Kang and H. Y. Yeom. Statistical Admission Control for Soft Real-time VOD Servers. In *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, SAC '00, pages 579–584, New York, NY, USA, 2000. ACM.

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

[79] D. Kim, S. Yoo, and S. Lee. A network congestion-aware memory subsystem for manycore. *ACM Trans. Embed. Comput. Syst.*, 12(4):110:1–110:18, July 2013.

[80] P. Kollig, C. Osborne, and T. Henriksson. Heterogeneous multi-core platform for consumer multimedia applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1254–1259. European Design and Automation Association, 2009.

[81] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330–335, 1997.

[82] Y. Li, B. Akesson, and K. Goossens. Dynamic Command Scheduling for Real-Time Memory Controllers. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[83] C. Lin and S. Brandt. Improving soft real-time performance through better slack management. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 12 pp.–421, 2005.

[84] A. Lofwenmark and S. Nadjm-Tehrani. Challenges in Future Avionic Systems on Multi-Core Platforms. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 115–119, Nov 2014.

[85] I. Loi and L. Benini. An efficient distributed memory interface for many-core platform with 3D stacked DRAM. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 99–104, 2010.

[86] Z. Lu and A. Jantsch. Slot allocation using logical networks for TDM virtual-circuit configuration for network-on-chip. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 18–25, Nov 2007.

[87] M. Bekooij and A. Moonen and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. In *Bits&Chips Symposium on Embedded Systems and Software*, 2007.

[88] M. Weber. Arbiters: design ideas and coding styles. In *Synopsys Users Group (SNUG)*, 2001.

[89] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142. ACM, 2012.

[90] B. Menhorn and F. Slomka. *Confirming the Design Gap*, volume 225 of *Advances in Intelligent Systems and Computing*. Springer International Publishing, 2013.

[91] MICRON DDR3 SDRAM Specification, 2006. Rev. M 08/14 EN.

[92] Micron. Low-Power Versus Standard DDR SDRAM Introduction. Technical Report TN-46-15, Micron Technology, Inc, 2007.

[93] Micron. Mobile Low-Power DDR SDRAM specifications, Rev. E 2/14 EN. www.micron.com, 2009.

[94] Micron. Mobile LPDDR2 SDRAM specifications, Rev. N 3/12 EN. www.micron.com, 2010.

[95] Micron. Mobile LPDDR3 SDRAM specifications, Rev. A 3/14 EN. www.micron.com, 2014.

[96] Micron Technology Inc. http://www.micron.com, 2015.

[97] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The Nostrum backbone-a communication protocol stack for Networks on Chip. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 693–696, 2004.

[98] O. Moreira, F. Valente, and M. Bekooij. Scheduling Multiple Independent Hard-real-time Jobs on a Heterogeneous Multiprocessor. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM.

[99] G. Morgan and A. Borg. Multi-core Automotive ECUs Software and Hardware Implications (ETAS Inc.), 2009. White paper.

[100] A. Nelson, K. Goossens, and B. Akesson. Dataflow formalisation of real-time streaming applications on a Composable and Predictable Multi-Processor SOC . *Journal of Systems Architecture*, 2015.

[101] J. Nikara, E. Aho, P. Tuominen, and K. Kuusilinna. Performance analysis of multi-channel memories in mobile devices. In *System-on-Chip, 2009. SOC 2009. International Symposium on*, pages 128–131, 2009.

[102] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012.

[103] Y. Ou, N. Xiao, and M. Lai. A Scalable Multi-channel Parallel NAND Flash Memory Controller Architecture. In *Chinagrid Conference (ChinaGrid), 2011 Sixth Annual*, pages 48–53, 2011.

[104] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s):64:1–64:26, Mar. 2013.

[105] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s):64:1–64:26, Mar. 2013.

[106] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, 2002.

[107] A. Rahimi, I. Loi, M. Kakoee, and L. Benini. A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[108] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on*

Hardware/software codesign and system synthesis, CODES+ISSS '11, pages 99–108. ACM, 2011.

[109] J. Rosén, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60, Dec 2007.

[110] 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 *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*, DSD '12, pages 727–730, 2012.

[111] J. Sancho, M. Lang, and D. Kerbyson. Analyzing the trade-off between multiple memory controllers and memory channels on multi-core processor performance. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7, 2010.

[112] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 152–160, May 2012.

[113] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø. A Time-Predictable Memory Network-on-Chip. In *14th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 53–62, 2014.

[114] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. A DRAM Centric NoC Architecture and Topology Design Approach. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 54–59, July 2011.

[115] H. Shah, A. Raabe, and A. Knoll. Bounding WCET of applications using SDRAM with Priority Based Budget Scheduling in MPSoCs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 665–670, 2012.

[116] Z. Shi and A. Burns. Priority Assignment for Real-Time Wormhole Communication in On-Chip Networks. In *Real-Time Systems Symposium, 2008*, pages 421–430, Nov 2008.

[117] Z. Shi and A. Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 161–170, April 2008.

[118] E. S. Shin, I. V. J. Mooney, and G. F. Riley. Round-robin arbiter design and generation. In *Proceedings of the 15th International Symposium on System Synthesis*, ISSS '02, pages 243–248. ACM, 2002.

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

[120] J. Sparso, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a TDM-based Network-on-Chip. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1044–1047, March 2013.

[121] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000.

[122] R. Stefan, A. Molnos, and K. Goossens. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *Computers, IEEE Transactions on*, 63(3):583–594, March 2014.

[123] L. Steffens, M. Agarwal, and P. Wolf. Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 255–265, 2008.

[124] M. Steine, M. Bekooij, and M. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 37–44, 2009.

[125] A. Stevens. QoS for High-Performance and Power-Efficient HD Multimedia. ARM White paper, http://wwww.arm.com, 2010.

[126] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *Networking, IEEE/ACM Transactions on*, 6(5):611–624, 1998.

[127] Texas Instruments Inc. TMS320VC5505/5504 DSP Direct Memory Access (DMA) Controller.

[128] C. H. K. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265. European Design and Automation Association, 2009.

[129] P. van der Wolf and J. Geuzebroek. SoC infrastructures for predictable system integration. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, 2011.

[130] D. Wiklund and D. Liu. SoCBUS: switched network on chip for hard real time embedded systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, April 2003.

[131] Z. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 372–383, Dec 2013.

[132] Xilinx Inc. LogiCORE IP XPS Multi-channel External Memory Controller (XPS MCH EMC).

[133] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric Scheduling for Multicore Hard Real-time Systems. *Real-Time Syst.*, 48(6):681–715, Nov. 2012.

[134] G. Zhang, H. Wang, X. Chen, S. Huang, and P. Li. Heterogeneous multi-channel: Fine-grained DRAM control for both system performance and power efficiency. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 876–881, 2012.

[135] T. Zhang, K. Wang, Y. Feng, Y. Chen, Q. Li, B. Shao, J. Xie, X. Song, L. Duan, Y. Xie, X. Cheng, and Y. Lin. A 3D SoC design for H.264 application with on-chip DRAM stacking. In *3D Systems Integration Conference (3DIC), 2010 IEEE International*, pages 1–6, 2010.

[136] D. Zhu and C. Qian. Challenges in Future Automobile Control Systems with Multicore Processors. In *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems (CPS) from Components*, pages 17–18, Mar 2011.

[137] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain priority scheduling on multi-channel memory systems. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 107–116, 2002.

List of Abbreviations

APA Accounting and Priority assignment
CCSP Credit-Controlled Static Priority
CMI Coupled Memory Interconnect
CPU Central Processing Unit
DDR Double Data Rate
DMA Direct Memory Access
DRAM Dynamic Random Access Memory
DRR Deficit Round-Robin
DTL Device Transaction Level protocol
FIFO First In First Out
GAMT Globally Arbitrated Memory Tree
GPU Graphics Processing Unit
HD High-Definition
HDLCD High-Definition Liquid Crystal Display
JEDEC Joint Electron Device Engineering Council
$\mathcal{LR}$ Latency-Rate
LUT Look-Up Table
MCMC Multi-Channel Memory Controller
NoC Network-on-Chip
PBS Priority Based Scheduler
RR Round-Robin
SDR Single Data Rate
SoC System-on-Chip
SRAM Static Random Access Memory
TDM Time Division Multiplexing
UMA Unified Memory Architecture

List of Symbols

Notation	*Description*
AG	Access granularity
BC	Burst count
BI	Number of banks across which data is interleaved
BL	DRAM burst length (words)
C	Set of memory clients c
IW_i	Interface width of interconnect i
IW_m	Interface width of memory m
$\hat{L}_c^{ns}$	Worst-case latency requirement (in ns) of memory client c
$\hat{L}_{m,c}$	Worst-case latency (in service cycles) of a memory client c in a memory channel m
$\hat{L}_{i,c}$	Worst-case latency (in ns) of a memory client c in interconnect i
$\hat{L}_c^{sc}$	Latency requirement (in service cycles) of a memory client c
$\hat{L}_c^{ns}$	Latency requirement (in ns) of a memory client c
$\hat{L}_c^{cc}$	Latency requirement (in clock cycles) of a memory client c
M	Set of memory channels m
$N'_{m,c}$	Decision variable used to restrict the number of service units allocated to a memory client c in each memory channel m to a power of two
$N_{m,c}$	Number of service units of a memory client c allocated to a memory channel m
SU_m^{bytes}	Service unit size (in Bytes) of each memory channel m
SC_m^{cc}	Service cycle duration (in clock cycles) of memory m
SC_i^{cc}	Service cycle duration (in clock cycles) of interconnect i
SC_m^{ns}	Service cycle duration (in ns) of memory m
SC_i^{ns}	Service cycle duration (in ns) of interconnect i

Notation	*Description*
b_m^{gross}	Worst-case gross bandwidth (in MB/s) of memory m
$\check{b}_c$	Worst-case bandwidth requirement (in MB/s) of memory client c
$\check{b}_u^{agg}$	Aggregate bandwidth requirement of all memory clients (in MB/s) for a service unit size u
f	TDM frame size
f_i	Operating frequency (in MHz) of an interconnect i
f_m	Operating frequency (in MHz) of memory m
g_c	Group number of a memory client c
n_{hops}	Number of hops between source and destination NI
q_c	Request size of a memory client c (in service units)
s_c	Request size of a memory client c (in Bytes)
tRL	Minimum time between read command and the arrival of first data word on the data bus
$tRCD$	Minimum time between activate and read/write commands
$tRAS$	Minimum time between activate and precharge commands
$tFAW$	Four activate command window constraint
$tREFI$	Maximum time interval between two refresh operations
$tWTR$	Minimum time interval before a read operation after a write
$t_{m,c}$	Binary decision variable used to restrict the number of service units of a memory client c allocated in each memory channel m to a power of two
Θ_c $\Theta_{i,c}$	Service latency of a memory client c Service latency of a memory client c in interconnect i
$\alpha_{m,c}$	Decision variable used to ensure that the rate allocated to a memory client c in a memory channel m is discretized for a given frame size
δ_{ov}	Interconnect overhead (in clock cycles)
δ_m	Internal pipeline delay of the memory subsystem
δ_p	Fixed pipeline delay of the interconnect router
κ_m	Memory capacity (in Bytes) of a memory channel m
$\check{\kappa}_c$	Total memory capacity requirement (in Bytes) of a memory client c
ρ'_i	Rate allocated to a memory client in interconnect i
$\check{\rho}'_c$	Rate requirement of a memory client c
$\rho'_{m,c}$	Rate allocated to a memory client c in a memory channel m
σ_c	Allocated burstiness to a memory client c in a CCSP arbiter

About the Author

Manil Dev Gomony was born on 9 July 1980 in Trivandrum, India. He is currently working as a Researcher with the Bell Laboratories of Alcatel-Lucent. His research interests are application-specific processor architectures, memory systems, scheduling algorithms, and embedded software. Previously, he was a PhD student at the department of Electrical Engineering in Eindhoven University of Technology, Netherlands from 2011 to 2014. He received his Masters in Electrical Engineering with specialization in System-on-Chip design from Linköping University, Sweden in 2010, and Bachelor of Technology in Applied Electronics from University of Kerala, India in 2002. He has previous work experience in embedded software development in Robert Bosch India, where he designed and developed firmware for power management, device drivers and schedulers for multimedia and cyber-physical systems.

List of Publications

Journal Articles

[1] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow. *SIGBED Rev.*, 10(3):23–34, Oct. 2013. ***(Not covered in this thesis)***

[2] M. D. Gomony, B. Akesson, and K. Goossens. A Real-Time Multi-Channel Memory Controller and Optimal Mapping of Memory Clients to Memory Channels. *ACM Trans. Embed. Comput. Syst.*, 14(2):25:1–25:27, Feb. 2015.

Conference Papers

[3] M. D. Gomony, C. Weis, B. Akesson, N. Wehn, and K. Goossens. DRAM Selection and Configuration for Real-Time Mobile Systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 51–56, 2012.

[4] M. D. Gomony, B. Akesson, and K. Goossens. Architecture and Optimal Configuration of a Real-Time Multi-Channel Memory Controller. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1307–1312, 2013.

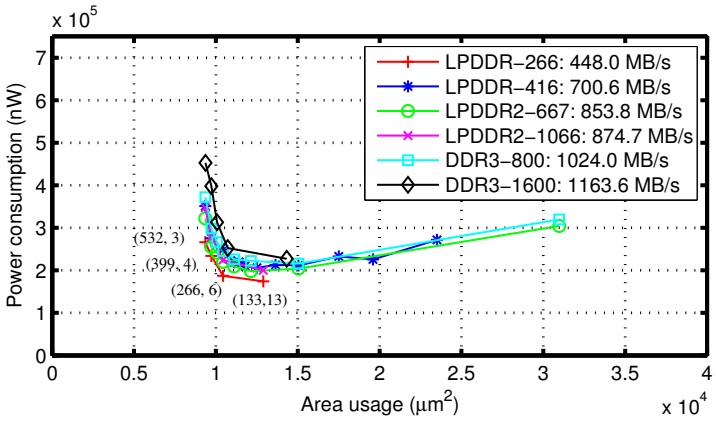
[5] M. D. Gomony, B. Akesson, and K. Goossens. Coupling TDM NoC and DRAM Controller for Cost and Performance Optimization of Real-Time Systems. In *De-*

sign, Automation Test in Europe Conference Exhibition (DATE), pages 1–6, 2014.

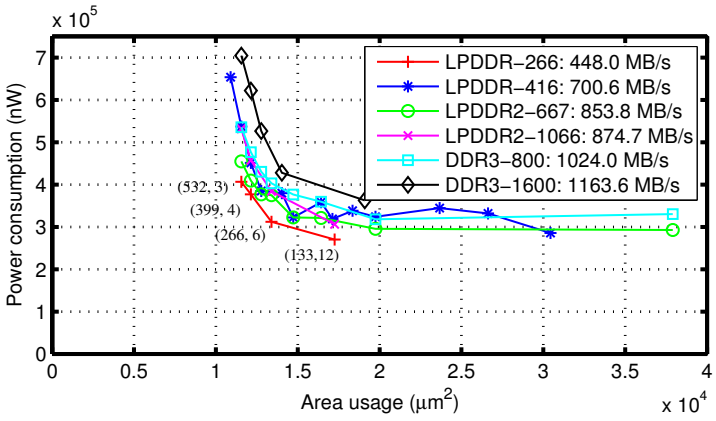
[6] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A Generic, Scalable and Globally Arbitrated Memory Tree for Shared DRAM Access in Real-time Systems. In Proceedings of the 2015 *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 193–198.

Coupled Interconnect Architecture - Trade-offs

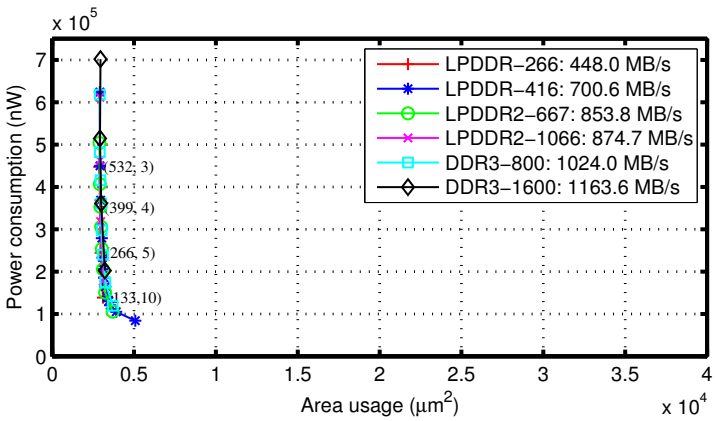
The area vs. power trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 16 B, 32 B, 64 B and 256 B are shown in this appendix.



(a) Aelite

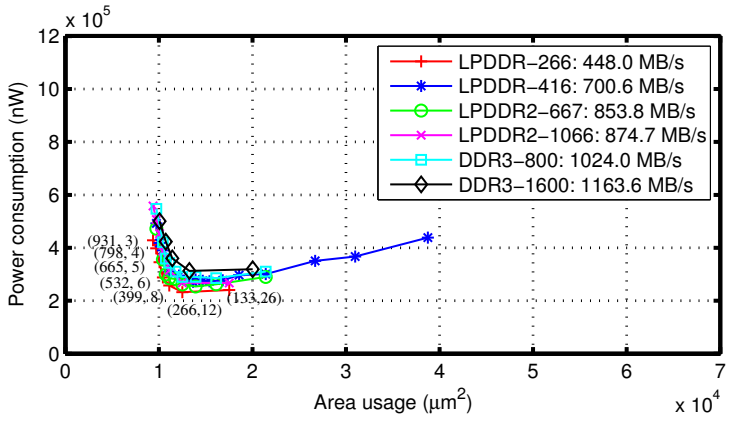


(b) Daelite

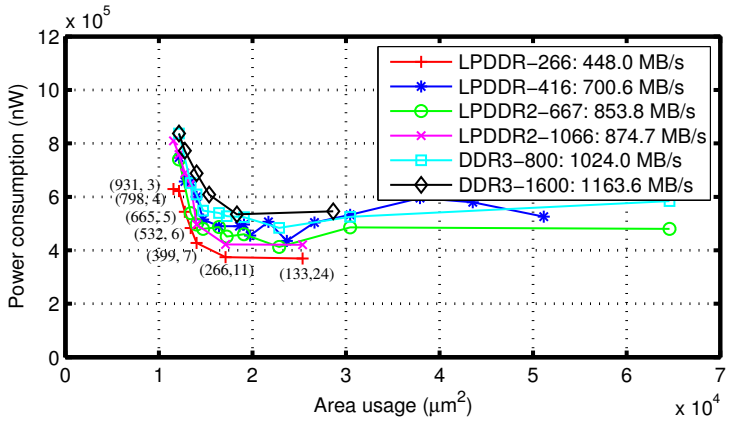


(c) GAMT

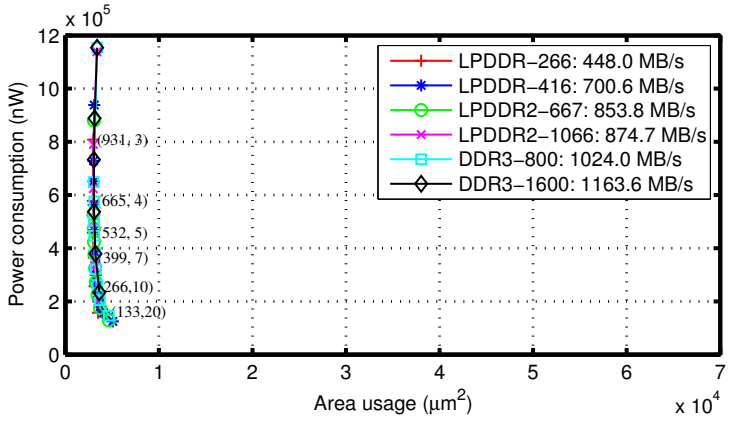
Figure E.1: Area vs. power trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 16 B. For clarity, only the (f_i, IW_i) combinations for LPDDR-266 are shown.



(a) Aelite

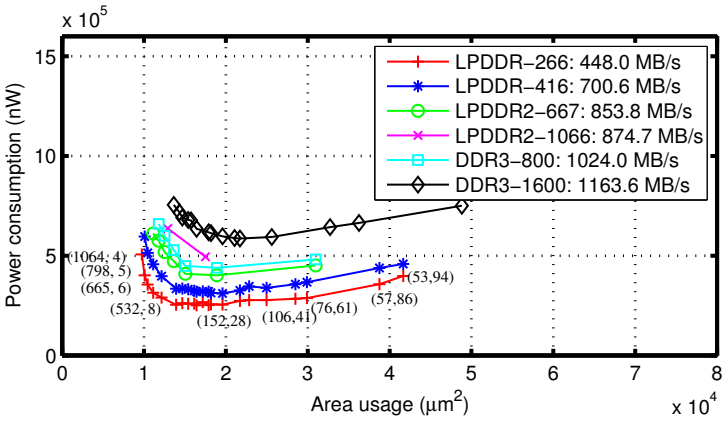


(b) Daelite

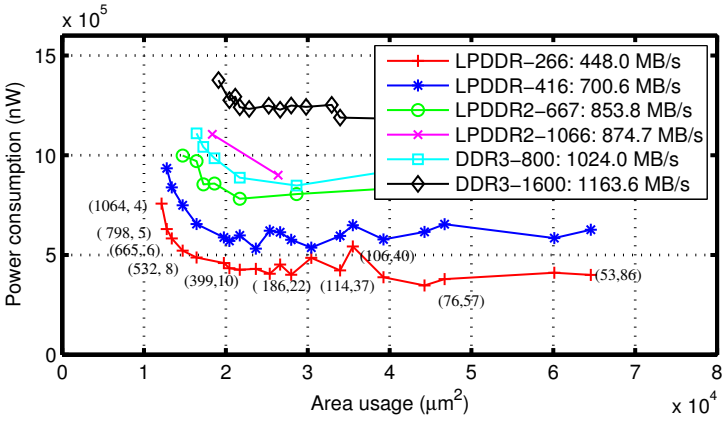


(c) GAMT

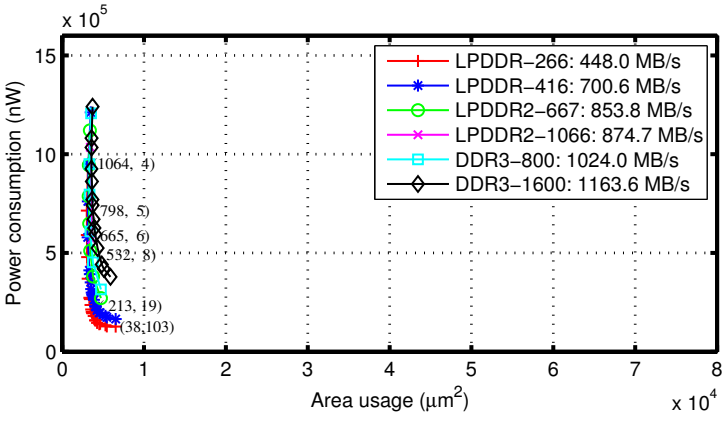
Figure E.2: Area vs. power trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 32 B. For clarity, only the (f_i, IW_i) combinations for LPDDR-266 are shown.



(a) Aelite

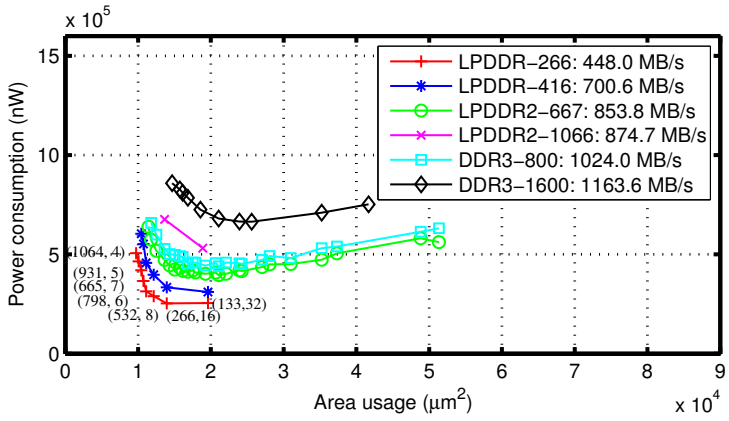


(b) Daelite

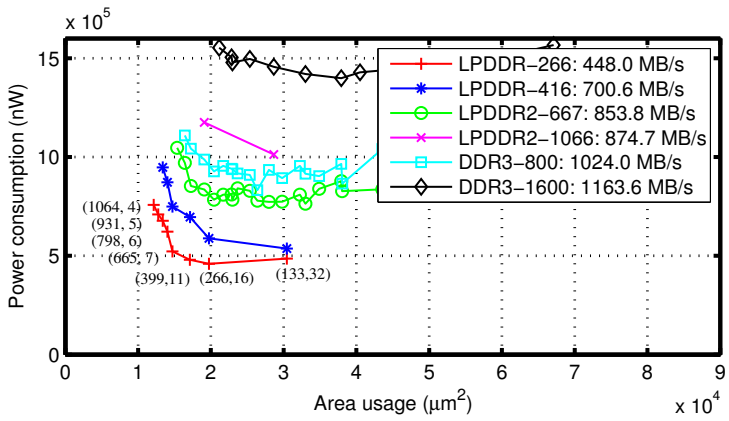


(c) GAMT

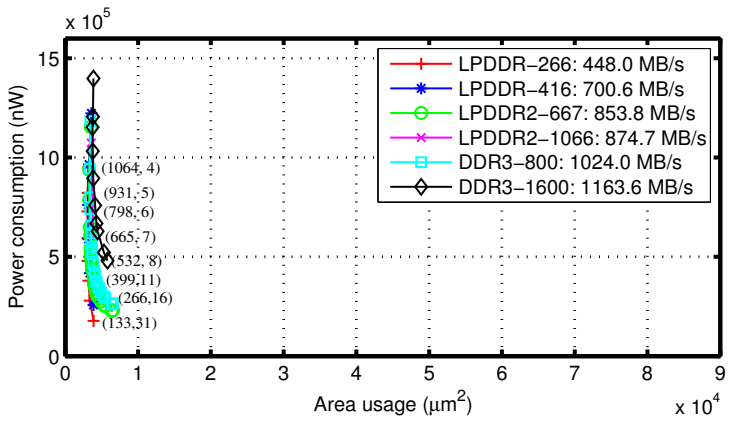
Figure E.3: Area vs. power trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 128 B. For clarity, only some of the (f_i, IW_i) combinations for LPDDR-266 are shown.



(a) Aelite



(b) Daelite



(c) GAMT

Figure E.4: Area vs. power trade-off of Aelite and Daelite NoCs, and GAMT coupled with different memories with a service unit size of 256 B. For clarity, only some of the (f_i, IW_i) combinations for LPDDR-266 are shown.