A Real-Time Multi-Channel Memory Controller and Optimal Mapping of Memory Clients to Memory Channels

MANIL DEV GOMONY, Eindhoven University of Technology, The Netherlands
BENNY AKESSON, Czech Technical University in Prague, Czech Republic
KEES GOOSSENS, Eindhoven University of Technology, The Netherlands

Ever increasing demands for main memory bandwidth and memory speed/power trade-off led to the introduction of memories with multiple memory channels, such as Wide IO DRAM. Efficient utilization of a multi-channel memory as a shared resource in multi-processor real-time systems depends on mapping of the memory clients to the memory channels according to their requirements on latency, bandwidth, communication and memory capacity. However, there is currently no real-time memory controller for multi-channel memories, and there is no methodology to optimally configure multi-channel memories in real-time systems. As a first work towards this direction, we present two main contributions in this article: 1) A configurable real-time multi-channel memory controller architecture with a novel method for logical-to-physical address translation. 2) Two design-time methods to map memory clients to the memory channels, one an optimal algorithm based on an integer programming formulation of the mapping problem, and the other a fast heuristic algorithm. We demonstrate the real-time guarantees on bandwidth and latency provided by our multi-channel memory controller architecture by experimental evaluation. Furthermore, we compare 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. We show that an optimal solution 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. Finally, we demonstrate configuring a Wide IO DRAM in a High-Definition (HD) video and graphics processing system to emphasize the practical applicability and effectiveness of this work.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems — *Real-Time and Embedded Systems*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Multi-channel memories, Memory controller, Optimal mapping, Heuristic algorithm

ACM Reference Format:
Gomony, M.D., Akesson, B., and Goossens, K., 2014. A Real-Time Multi-Channel Memory Controller and Optimal Mapping of Memory Clients to Memory Channels. *ACM Trans. Embedd. Comput. Syst.* X, X, Article X (January 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In heterogeneous multi-processor platforms, main memory (off-chip DRAM) is typically a shared resource for cost reasons and to enable communication between the

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, CA505 BENEFIC, CA703 OpenES, ARTEMIS-2013-1 621429 EMC2, 621353 DEWI, and by the European social fund within the framework of realizing the project "Support of inter-sectoral mobility and quality enhancement of research teams at Czech Technical University in Prague", CZ.1.07/2.3.00/30.0034.
Corresponding authors address: Gomony, M.D., Faculty of Electrical Engineering, Eindhoven University of Technology; email: m.d.gomony@tue.nl

DOI: http://dx.doi.org/10.1145/0000000.0000000

processing elements. Such platforms run several applications with firm and soft real-time requirements [Kollig et al. 2009; van Berkel 2009; Melpignano et al. 2012], and moreover, the firm real-time applications impose strict worst-case requirements on main memory performance in terms of bandwidth and/or latency [van der Wolf and Geuzebroek 2011; Steffens et al. 2008]. These requirements must be guaranteed at design time to reduce the verification effort, which is made possible using real-time memory controllers [Paolieri et al. 2013; Akesson and Goossens 2011a; Reineke et al. 2011; Shah et al. 2012; Bayliss and Constantinides 2012; Wu et al. 2013; Li et al. 2014; Kim et al. 2014] that bound the memory access time by employing predictable arbiters, such as Time Division Multiplexing (TDM) and Round-Robin. Moreover, real-time memory controllers can be analyzed using shared resource abstractions, such as the Latency-Rate ($\mathcal{LR}$) server model [Stiliadis and Varma 1998] which can be used in formal performance analysis based on e.g., network calculus [Cruz 1991] or data-flow analysis [Sriram and Bhattacharyya 2000].

Memories with multiple physical channels and wide interfaces, such as Wide IO DRAMs [JEDEC], are essential to meet the main memory *power/bandwidth* demands of future real-time systems [Gomony et al. 2012]. 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 benefit average-case performance [Sancho et al. 2010; Cabarcas et al. 2010; Nikara et al. 2009]. In addition to request sizes, firm real-time memory clients in real-time multiprocessor platforms come with different requirements on memory bandwidth, latency, communication and memory capacity 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* [Lin and Brandt 2005]. The optimal mapping of the memory clients to the memory channels for efficient memory bandwidth utilization results in different granularities at which the memory requests from each of the clients are interleaved in each channel, which requires a configurable memory controller and logical-to-physical address translation logic. Currently, there is no configurable real-time memory controller architecture for multi-channel memories and there is no methodology to map firm real-time memory clients to memory channels, meeting their bandwidth, latency, communication and memory capacity requirements.

This article presents two of our main contributions: 1) A real-time multi-channel memory controller architecture, with a new programmable *Multi-Channel Interleaver* and a novel method for logical-to-physical address translation that enables interleaving of a memory request in different sizes across any number of memory channels. 2) Two design-time methods to determine the optimal mapping of memory clients to the memory channels considering their requirements on bandwidth, latency, communication and memory capacity. The first method is an optimal algorithm based on an integer programming formulation of the mapping problem, and the second a fast heuristic algorithm. We demonstrate by experimentation that our multi-channel memory controller satisfies the real-time requirements of the memory clients. Furthermore, we experimentally compare the computation time and mapping success ratio of the optimization problem formulation in a solver and the heuristic algorithm against two existing mapping algorithms. Finally, we demonstrate configuring a multi-channel Wide IO DRAM for a High-Definition (HD) video and graphics processing system using our approach.

In the remainder of this article, Section 2 reviews the related work, Section 3 gives an introduction to state-of-the-art real-time memory controllers and the $\mathcal{LR}$ server model. In Section 4, we introduce our proposed multi-channel memory controller architecture, including our method for logical-to-physical address translation. The two

proposed methods to map memory clients to memory channels, the one based on integer programming formulation and the heuristic algorithm are presented in Section 5 & 6, respectively. In Section 7, we present both the experimental evaluation of the multi-channel memory controller architecture and the performance evaluation of our two mapping methods. Section 8 then presents a case study of configuring a Wide IO DRAM in an HD video and graphics processing system, and finally we conclude in Section 9.

2. RELATED WORK

Among the previous related work, some exploit the benefits of interleaving data across multiple memory channels. [Aho et al. 2009; Nikara et al. 2009; Zhu et al. 2002; Cabarcas et al. 2010] proposed interleaving data across the memory channels such that all channels are accessed by a single transaction to improve average-case performance. Similarly, [Casini 2008] proposed splitting the traffic within a logical address region across multiple memory channels to improve average-case performance by reducing average latency. Dynamic mechanisms for efficient data placement to reduce average memory access latency in a system comprising multiple memory controllers is proposed by [Awasthi et al. 2010]. 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 focus on memory controller architectures and logical-to-physical address translation for multi-channel memories. [Zhang et al. 2010] proposed a parallel-access mechanism in which two separate DDR Finite State Machines (FSM) are used to control eight memory channels of a 3D-DRAM. The proposed architecture by [Loi and Benini 2010] 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. The multi-channel NAND flash memory controller by [Ou et al. 2011] uses a dynamic mapping strategy by using a mapping table that stores the logical-to-physical address translation, and a crossbar switch for routing traffic across multiple memory channels. Also, the multi-channel memory controller architecture by [Bouquet 2000] routes an incoming request to any of the memory channels using a crossbar. [Zhang et al. 2012] presented an 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 [Paolieri et al. 2013; Akesson and Goossens 2011a; Reineke et al. 2011; Shah et al. 2012; Bayliss and Constantinides 2012; Wu et al. 2013; Li et al. 2014; Kim et al. 2014], 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.

In our previous work [Gomony et al. 2013], we presented a high-level architecture of a real-time multi-channel memory controller and an optimal method for mapping memory clients to memory channels based on an integer programming formulation of the mapping problem. As an extension of our previous work, in this article, we present the detailed architecture of the multi-channel memory controller including its experimental evaluation using a SystemC prototype implementation. We extend our optimization problem formulation to determine an optimal frame size for a TDM arbiter. In addition, we present a fast heuristic algorithm to map memory clients to the channels including its performance comparison with the optimal method and two existing mapping algorithms.

3. BACKGROUND

This work relies on existing single-channel real-time memory controllers to bound the execution time of a memory transaction, and uses the $\mathcal{LR}$ server model as the shared resource abstraction to derive bounds on service provided by predictable arbiters. Hence, we introduce them in this section.

3.1. Real-time memory controllers

State-of-the-art real-time memory controllers [Paolieri et al. 2013; Akesson and Goossens 2011a; Reineke et al. 2011; Shah et al. 2012; Bayliss and Constantinides 2012; Wu et al. 2013; Li et al. 2014; Kim et al. 2014] bound the execution time of a memory transaction by fixing the memory access parameters of a transaction, such as burst length and number of read/write commands, at design time. These parameters define the *access granularity* of the memory controller, which defines the amount of data read/written from/to the memory per request. For a fixed access granularity, real-time memory controllers 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 transaction. Also, the worst-case bandwidth offered by a memory for a fixed access granularity can be computed [Akesson and Goossens 2011b]. In this article, we refer to a memory transaction of a fixed size as a *service unit*, and the time taken to serve such a service unit is a *service cycle*. The service cycle for a read and a write transaction can be different and depends on the memory device and the memory controller.

3.2. $\mathcal{LR}$ servers

Latency-Rate ($\mathcal{LR}$) servers [Stiliadis and Varma 1998] is 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) [Katevenis et al. 1991], Deficit Round-Robin (DRR) [Shreedhar and Varghese 1996], and priority-based arbiters with a rate-regulator, such as Credit-Controlled Static Priority (CCSP) [Akesson et al. 2008] and Priority Based Scheduler (PBS) [Steine et al. 2009]. The $\mathcal{LR}$ abstraction enables modeling of many different arbiters, and is compatible with a variety of formal analysis frameworks, such as data-flow analysis [Sriram and Bhattacharyya 2000] or network calculus [Cruz 1991].

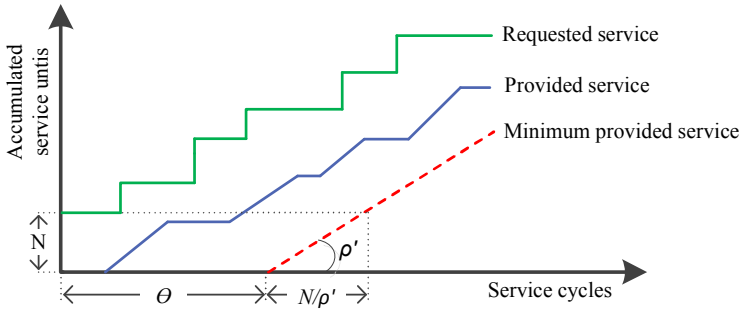

Fig. 1. Example service curves of a $\mathcal{LR}$ server showing service latency (Θ) and completion latency (N/ρ').

Using the $\mathcal{LR}$ abstraction, a lower linear bound on the service provided by an arbiter to a client or *requestor* can be derived. In this article, we use the term *requestor* to denote a memory client that requests access to a memory resource with certain bandwidth and latency requirements. Figure 1 shows example service curves of a $\mathcal{LR}$ server.

The requested service by a requestor at a time consists of one or more service units, indicated on the y-axis of the figure. The minimum service provided to the requestor is the service guaranteed by the $\mathcal{LR}$ abstraction, which depends on two parameters, namely the *service latency* Θ and the *allocated rate* ρ' (bandwidth). The *service latency* is intuitively 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 [Akesson and Goossens 2011b]. After a request consisting of N service units is scheduled to be served, it receives service at the allocated rate ρ' and it hence takes N/ρ' *service cycles* to finish serving the request, called the *completion latency* of the requestor. The *worst-case latency* $\hat{L}$ (in service cycles) of a requestor is the total time taken by its request of size N service units at the head of its request queue to get served in the worst-case [1], which is the sum of the service latency and the completion latency, given by Equation (1). The advantage of this approach is that it can be applied to other arbiters belonging to the class of $\mathcal{LR}$ servers by changing the expression for Θ.

$$\hat{L} = \Theta + \lceil N/\rho' \rceil \tag{1}$$

This work considers a TDM arbiter with continuous slot allocation as an example of a $\mathcal{LR}$ server, but our approach generally applies to other $\mathcal{LR}$ servers with linear expression for Θ, such as a TDM arbiter with distributed slot allocation. As mentioned before, the service cycle duration for a read and write can be different. Since a requestor 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. Note that it is shown in [Goossens et al. 2013] that the service cycle for read and write transactions can be made equally long with negligible loss in the guaranteed memory bandwidth. Figures 2(a) & 2(b) show a TDM frame of size f with requestor R allocated to two slots using continuous and distributed slot allocation strategies, respectively. Here, R gets a rate $\rho' = 2/6$, since two out of six slots are allocated to R. The service latency (Θ) of R is 4 and 2 for continuous and distributed TDM, respectively, because of the interference from other requestors that occupy the remaining set of TDM slots. In terms of rate ρ' and/or frame size f, the service latencies of continuous and distributed TDM are given by $\Theta = f \times (1 - \rho')$ and $\Theta = f/(f \times \rho') - 1$, respectively, as shown in [Akesson and Goossens 2011b].

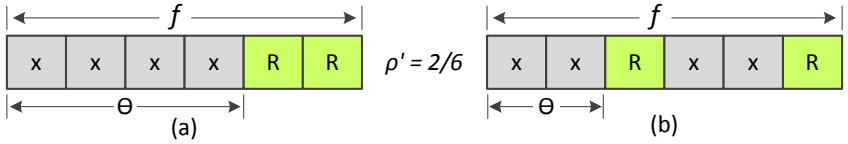

Fig. 2. Example TDM frame of size f showing service latency (Θ) of requestor R with its slots allocated using (a) continuous and (b) distributed allocation strategies.

Hence, for a TDM arbiter with a frame size f, the worst-case latency of a requestor with an allocated rate of ρ' for continuous and distributed TDM is given by Equations (2) & (3), respectively, in which both service latency and completion latency are rounded up to make the bound conservative.

$$\hat{L} = \lceil f \times (1 - \rho') \rceil + \lceil N/\rho' \rceil \tag{2}$$

$$\hat{L} = f/\lceil f \times \rho' \rceil - 1 + \lceil N/\rho' \rceil \tag{3}$$

[1] For simplicity, we do not consider requestors 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 [Stiliadis and Varma 1998].

4. MULTI-CHANNEL MEMORY CONTROLLER FOR REAL-TIME SYSTEMS

In this section, we present our proposed real-time multi-channel memory controller architecture. We start this section with an analysis of the impact of 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 multi-channel memory controller which is based on the conclusions from the analysis, followed by a novel method for logical-to-physical address translation.

4.1. Multi-channel memories and $\mathcal{LR}$ servers

Compared to a single-channel memory, multi-channel memories give us the opportunity to interleave memory requests at different granularities across the memory channels. When the memory request of a requestor 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 requestor with a required rate (bandwidth) ρ' *increases* when the number of channels to which its request is interleaved increases. This counter-intuitive result is shown in Equation (4), which shows the worst-case latency for a TDM arbiter in each memory channel, assuming the required rate ρ' and the total number of service units N in a memory request are distributed evenly across the number of channels to which the request is interleaved, n_{Ch}. It can be seen that the service latency increases with n_{Ch}, however, the completion latency remains constant [2]. 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.

$$\hat{L}' = \left\lceil f \times \left(1 - \frac{\rho'}{n_{Ch}}\right) \right\rceil + \left\lceil \frac{N/n_{Ch}}{\rho'/n_{Ch}} \right\rceil = \left\lceil f \times \left(1 - \frac{\rho'}{n_{Ch}}\right) \right\rceil + \left\lceil \frac{N}{\rho'} \right\rceil \tag{4}$$

Interleaving a memory request to more than one memory channel is unavoidable under the following four conditions: (1) When the latency requirement of a requestor cannot be met in a single channel even after allocating a 100% bandwidth ($\rho' = 1$) to the requestor. This could happen with larger request sizes as can be seen in Equation (2), if the request size is large such that even after allocating a 100% bandwidth ($\rho' = 1$) of a channel, it does not meet its latency requirement, it must be interleaved across multiple channels with an over-allocated rate. (2) When the bandwidth requirement of a requestor could not be satisfied with the available bandwidth in a single memory channel. (3) When the memory capacity requirements cannot be met with the capacity available in a memory channel. (4) When a requestor needs to communicate with another requestor 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 requestors with different request sizes and diverse requirements on bandwidth, latency, communication and memory capacity, the optimal mapping of requestors to the memory channels for minimal bandwidth utilization results in different degrees of interleaving across the memory channels for each requestor. This implies that the existing methods, in which all requestors are interleaved in the same fashion to the memory channels are not always optimal. Hence, we need a programmable memory controller architecture that can be config-

[2]This conclusion is valid for all other $\mathcal{LR}$ arbiters as well since they all have the rate term, ρ', which will always gets split across channels and the completion latency remains constant. This is evident from their worst-case latency equations [Akesson and Goossens 2011b].

ured to interleave memory requests of a requestor to any number of available memory channels at different granularities.

4.2. Real-time multi-channel memory controller architecture

To enable mapping of memory requests from memory clients to memory channels at different granularities, our multi-channel memory controller 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 proposed multi-channel memory controller, shown in Figure 3, consists of a *Multi-Channel Interleaver*, and a Channel Controller in each memory channel. The Channel Controller can be any state-of-the-art real-time memory controller [Paolieri et al. 2013; Akesson and Goossens 2011a; Reineke et al. 2011; Shah et al. 2012; Bayliss and Constantinides 2012; Wu et al. 2013; Li et al. 2014; Kim et al. 2014] employing any $\mathcal{LR}$ arbiter. We use a Multi-Stage Crossbar that connects each requestor to every Channel Controller. The Multi-Channel Interleaver consists of an *Atomizer*, *Channel Selector (CS)* and a *Sequence Generator* connected to each memory requestor. The Multi-Channel Interleaver has separate *request* and *response* paths for each requestor.

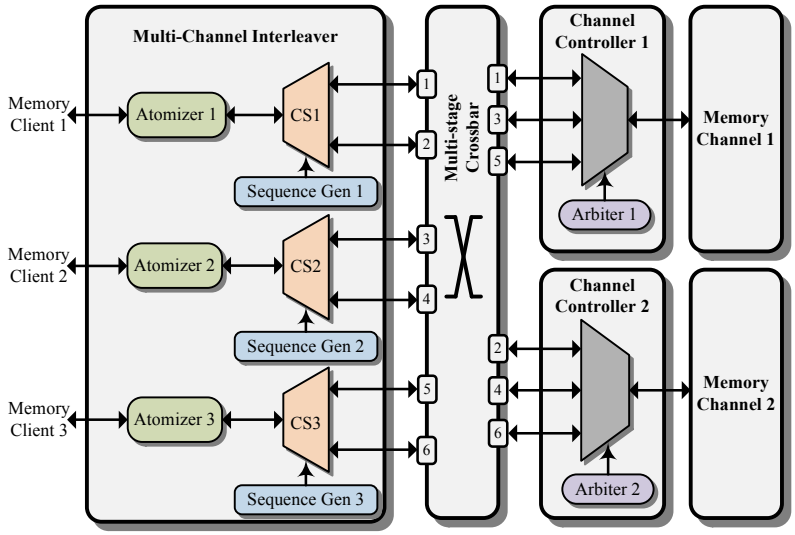

Fig. 3. High-level view of real-time multi-channel memory controller architecture showing three memory clients and two memory channels. The *Atomizer* splits a memory request in to smaller service units and the *Channel Selector (CS)* routes these service units to the different memory channels according to the configuration in the Sequence Generators.

A detailed architecture of the Channel Selector showing both request and response paths is shown in Figure 4. 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 ensures non-blocking delivery of service units (write data) to the different memory channels (assuming no input buffers in the Channel Controllers). 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 until all of the responses from the different channels have arrived,

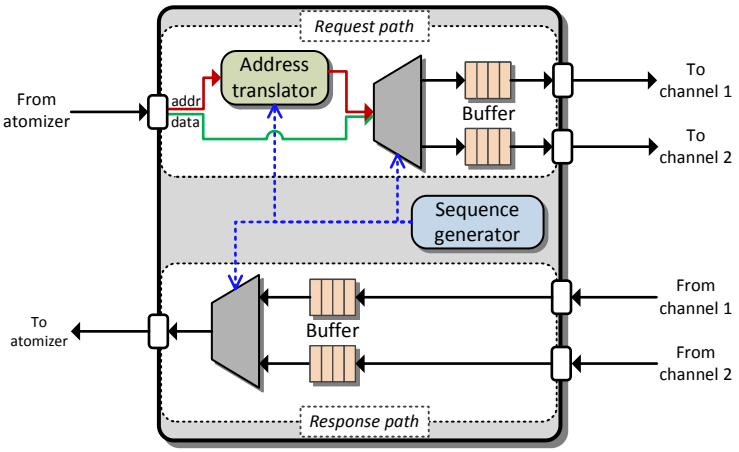

Fig. 4. Detailed architecture of the Channel Selector, showing request and response paths. The Sequence Generator routes the service units to the respective memory channels to which they are mapped after performing the logical-to-physical address translation. The responses from different channels are buffered in the response path before forwarding the complete response to the Atomizer.

and then the Atomizer forwards the complete response to the requestor. Hence, the size of the output and input buffers (for write data and read response, respectively) should be equal to the maximum number of outstanding transactions of a requestor times its request size (assuming same transaction size for all requests from a requestor). Since, the Sequence Generators, Arbiters and Atomizers can be configured at design-time, this architecture enables all possible connections of a requestor to any of the memory channels with any level of interleaving, and different rate allocated to each requestor in each channel.

4.3. Logical-to-physical address translation

As discussed before, an optimal mapping of requestors to memory channels could result in each channel allocated a different number of requestors and different memory capacities allocated to the requestors 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 requestor R1 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 5a and 5b illustrate the logical and physical views of the memory, respectively. Assuming a service unit size of 64 B, every request from the requestor consists of four service units. Figure 5b 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 keep the lookup

table size to a minimum, we propose a method to compute the logical address in each channel, expressed by Equation (5). Note that $Request\ size$ is in service units.

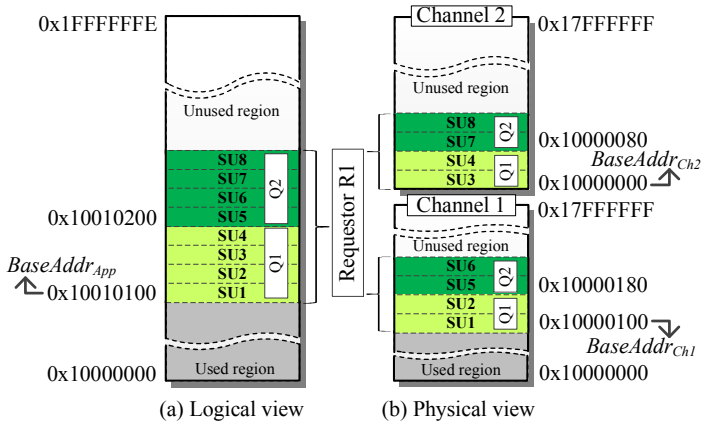

Fig. 5. Example memory map showing requestor R1 allocated to two memory channels, with every request Q1 and Q2 interleaved across the two memory channels.

$$ReqAddr_{Ch} = (ReqAddr_{App} - BaseAddr_{App}) \gg \log_2(Request\ size/N_{Ch_n}) + BaseAddr_{Ch_n} \quad (5)$$

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 requestor in each channel is proportional to the number of service units of its request allocated to the 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 power of two [3].

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, presented in Section 5 & 6. We generate the base addresses for all the requestors mapped to each of the channels, $BaseAddr_{Ch}$, based on the memory capacity allocated to them.

Given that we have presented the multi-channel memory controller architecture that can be programmed with any mapping, we proceed with our two methods to map memory clients to the memory channels in the next two sections.

5. OPTIMAL METHOD FOR MAPPING MEMORY REQUESTORS TO MEMORY CHANNELS

This section presents an optimal method for mapping memory clients to memory channels based on an integer programming formulation of the mapping problem. 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.

5.1. System model

The set of memory channels is defined as $c \in C$, with each channel having a total memory capacity (in Bytes) given by $B^{ch}(c)$. The access granularity (in Bytes) of each

[3]The request sizes of most of the real-world memory requestors, such as CPUs, DSPs, LCD & DMA controllers are in the order of power of two [Steffens et al. 2008; Texas Instruments Inc.].

memory channel is given by AG, with a service cycle (in ns) given by SC^{ns} [4]. For each memory channel $c \in C$, the worst-case bandwidth (in MB/s) can be computed for a fixed access granularity AG (e.g. see [Akesson and Goossens 2011b]), and is given by $b^{ch}(c)$.

Consider a set of requestors denoted by $r \in R$, each with a worst-case latency requirement (in ns) given by $L^{ns}(r)$, minimum bandwidth requirement (in MB/s) given by $\check{b}(r)$, and a total memory capacity requirement (in Bytes) given by $\check{B}(r)$. Note that the *minimum rate required* by a requestor can be computed as the ratio of its minimum bandwidth requirement $\check{b}(r)$ and the worst-case bandwidth of a channel $b^{ch}(c)$. The worst-case latency of a requestor (in service cycles) in each channel $c \in C$ is given by $\hat{L_c}(r)$, and is defined as:

$$\forall r \in R : \hat{L_c}(r) = \lfloor L^{ns}(r)/SC^{ns}(c) \rfloor \tag{6}$$

The request size (in Bytes) of requests from a requestor $r \in R$ is given by $s(r)$. We assume a constant request size for all requests from a single requestor since it typically holds for the real-time requestors under consideration, such as CPUs, hardware accelerators, DMA and LCD controllers. The number of service units in each request is given by $q(r)$ and is defined by Equation (7). Since the request sizes, $s(r)$, and access granularity of a memory device, AG, is in the order of power of two, $q(r)$ will be a power of two.

$$\forall r \in R : q(r) = s(r)/AG \tag{7}$$

Each requestor $r \in R$ has an associated group number given by $g(r)$, which represents the communication dependency with other requestors, or in other words, requestors that need to communicate through shared memory are assigned the same group number. In the next section, we define the optimization problem statement and formulate it as an integer programming problem. A summary of the memory system and requestor parameters and their corresponding notations are given in Table I & II, respectively.

Table I. Memory system parameters

Parameter name	*Notation*
Set of memory channels	C
Memory capacity (in Bytes) of each memory channel	$B^{ch}(c)$
Access granularity (in Bytes) of each memory channel	AG
Service cycle duration (in ns)	SC^{ns}
Worst-case bandwidth (in MB/s) of each memory channel	$b^{ch}(c)$

Table II. Requestor parameters

Parameter name	*Notation*
Set of requestors	R
Worst-case latency requirement (in ns)	$L^{ns}(r)$
Minimum bandwidth requirement (in MB/s)	$\check{b}(r)$
Total memory capacity requirement (in Bytes)	$\check{B}(r)$
Worst-case latency (in service cycles) in each channel	$\hat{L_c}(r)$
Request size (in Bytes)	$s(r)$
Request size (in service units)	$q(r)$
Group number	$g(r)$

5.2. Optimization problem formulation

In this section, we present the formulation of the mapping problem as an integer programming problem. As mentioned before, we need to minimize the bandwidth allocated to firm real-time requestors to maximize the slack bandwidth, which improves the average-case performance of soft real-time requestors in the system. Hence, we define our optimization problem as follows: *Find the mapping of requestors to the memory channels, the number of service units allocated to those channels, N_c, and a rate, ρ'_c, for each requestor $r \in R$ in each memory channel $c \in C$, such that all requestor requirements are satisfied and the sum of rates allocated to all requestors is minimized.* The optimization problem is defined as:

[4]For simplicity, we assume the same access granularity in all memory channels.

$$\textit{Minimize:} \sum_{c \in C} \sum_{r \in R} \rho'_c(r) \tag{8}$$

Such that the following seven constraints are satisfied:

Constraint 1: The worst-case latency of each requestor $r \in R$ after allocation $\hat{L}'(r)$ must be less than or equal to its worst-case latency requirement $\hat{L}(r)$, and is defined as:

$$\forall r \in R: \ \hat{L}'(r) \leq \hat{L}(r) \tag{9}$$

The service units of every request of a requestor are allocated across the memory channels such that each requestor has a (Θ, ρ') pair per channel. The worst-case latency of a requestor $r \in R$ in each channel $c \in C$ is then given by $\hat{L_c}'(r)$, and is defined by Equation (10) [5], where $\Theta_c(r)$ is the service latency of a requestor in each channel.

$$\forall c \in C, r \in R : \hat{L_c}'(r) = \Theta_c(r) + \lceil N_c(r)/\rho'_c(r) \rceil \tag{10}$$

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

$$\forall c \in C, r \in R : \hat{L}'(r) = \max_{c \in C} \hat{L_c}'(r) \tag{11}$$

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

$$\forall c \in C, r \in R: \ \hat{L}(r) - \hat{L_c}'(r) \geq 0 \tag{12}$$

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

$$\forall c \in C : \sum_{r \in R} \rho'_c(r) \leq 1 \tag{13}$$

Constraint 3: The sum of rates allocated to each requestor $r \in R$ across all memory channels should be greater than or equal to its minimum required rate, defined by Equation (14).

$$\forall r \in R : \sum_{c \in C} \rho'_c(r) \geq \frac{\check{b}(r)}{b^{ch}(c)} \tag{14}$$

Constraint 4: The sum of service units $N_c(r)$ of each requestor $r \in R$ allocated across all memory channels must be equal to the total number of service units q(r) in every request from the requestor, defined as:

$$\forall r \in R : \sum_{c \in C} N_c(r) = q(r) \tag{15}$$

Constraint 5: The number of service units $N_c(r)$ of each requestor $r \in R$ allocated to each memory channel $c \in C$ must be a power of two. To formulate this as a linear constraint, we define two decision variables $b_c(r)$ and $N'_c(r)$ for each requestor in every channel. $b_c(r)$ is a binary decision variable defined by Equation (16) and $N'_c(r)$ can take a value in the range $0..\log_2[q(r)]$. Constraint 5 is then defined by Equation (17)

$$b_c(r) = \begin{cases} 1, & \text{if } N_c(r) > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

[5] For simplicity in presentation, we do not add the fixed delay which depends on the number of pipeline stages in the RTL implementation of the multi-channel memory controller architecture.

$$\forall c \in C, r \in R : N_c(r) = 2^{N'_c(r)} \times b_c(r) \tag{17}$$

Constraint 6: Each pair of communicating requestors, i.e. with the same group number $g(r)$, must be allocated to the same set of memory channels, and the number of service units of the requestors allocated in each channel must be proportional for data alignment since they may have different request sizes. To understand this, consider two communicating requestors R1 and R2, each with a request size of eight and four service units, respectively, interleaved across two memory channels. Assume that R1 issues a memory write request Q1 and R2 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 6, so that the ratio $Request\ size/N_{Ch_n}$ remains same for both requestors and results in coherent address translation according to Equation (5) [6].

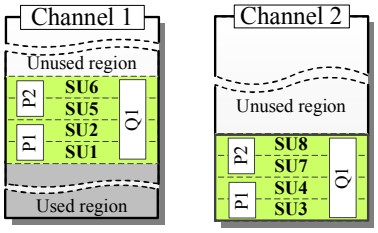

Fig. 6. Physical memory maps of two memory channels showing request Q1 of size eight service units aligned with requests P1 and P2 of size 4 service units each.

For two communicating requestors r_i and r_j, the constraint is defined by Equation (18). The decision variable $N'_c(r)$ is the same as defined under Constraint 5. This constraint ensures that for every non-zero number of service units of r_i allocated to a memory channel, $N_c(r_i)$, a corresponding number of service units in the order of power-of-two of requestor r_j, $2^{N'_c(r_j)}$ is allocated to the same channel, and vice versa. Also, it ensures that $N'_c(r_i)$ and $N'_c(r_j)$ are selected such that $N_c(r_i)$ and $N_c(r_j)$ are proportional. To understand this, consider our example with R1 and R2 as r_i and r_j, respectively. Assume that Constraint 5 assigns $N'_c(R1) = 2$ and $N'_c(R2) = 1$ resulting in $N_c(R1) = 4$ and $N_c(R2) = 2$, which satisfies Equation (18) since $4 \cdot 2^1 = 2 \cdot 2^2$. In contrast, for any non-proportional assignment by Constraint 5, say $N'_c(R1) = 4$ and $N'_c(R2) = 1$, Equation (18) will not be satisfied since $8 \cdot 2^1 \neq 2 \cdot 2^4$.

$$\forall c \in C, r_i, r_j \in R, g(r_i) = g(r_j) : N_c(r_i) \times 2^{N'_c(r_j)} = N_c(r_j) \times 2^{N'_c(r_i)} \tag{18}$$

Constraint 7: The total memory capacity of all requestors in each channel $c \in C$ must be less than or equal to the channel capacity $B^{ch}(c)$, defined by Equation (19). This constraint along with Constraint 4 ensures that the sum of the memory capacities allocated to a requestor in all memory channels is equal to its total memory capacity requirement.

$$\forall c \in C : \sum_{r \in R} \frac{N_c(r)}{q(r)} \times \check{B}(r) \leq B^{ch}(c) \tag{19}$$

[6]This constraint only ensures that the number of service units allocated in each channel are proportional. Furthermore, the service units of all communicating requestors must be aligned in each memory channel. As shown in Figure 6, the first four service units SU1-SU4 of R1 must be interleaved across two memory channels so that the response for the first read request from R2 contains four service units (data) from the continuous logical address space. To ensure this, the Sequence Generator in the multi-channel memory controller must be programmed accordingly.

Constraint 8: For every service unit allocated to a memory channel $c \in C$, there must be a corresponding rate allocated, and vice versa, defined by Equations (20) and (21). $b_c(r)$ is the same as in Constraint 5 and M is a constant with a value larger than the maximum rate, i.e., $M > 1$. Equation (20) ensures that when service units are allocated to a channel (according to Constraint 5), a corresponding rate is allocated in the channel. Equation (21) ensures that the rate is set to zero when there are no service units allocated to the channel.

$$\forall r \in R, \forall c \in C : (1 - b_c(r)) \times M + \rho'_c(r) > 0 \tag{20}$$

$$\forall r \in R, \forall c \in C : b_c(r) \times M - \rho'_c(r) \geq 0 \tag{21}$$

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 continuous 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 continuous TDM arbiter. In the worst-case latency derivation of continuous TDM (Equation (2)), we can see that for a given frame size, f, the rate that needs to be allocated depends on the latency requirement of a requestor 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, $\rho'_c(r)$, is the discretized rate for a given frame size, we define a decision variable, $\alpha_c(r)$, which can take a value between 0 and 1, and the allocated rate is then defined as $\rho'_c(r) = (\lceil f \times \alpha_c(r) \rceil)/f$. In essence, this constraints $\rho'_c(r)$ 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 requestors 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 requestors allocated in each memory channel, defined by Equation (22)

$$\forall c \in C : f \geq \sum_{r \in R} \lceil f \times \rho'_c(r) \rceil \tag{22}$$

6. A FAST HEURISTIC ALGORITHM FOR MAPPING REQUESTORS TO MEMORY CHANNELS

The optimal algorithm for mapping requestors to memory channels presented in the previous section may not be scalable for future systems in terms of algorithm computation time, as the number of variables and constraints increases with the problem size. Hence, we devised a fast heuristic algorithm for mapping requestors to memory channels that minimizes memory bandwidth utilization while considering the requestor requirements. Our heuristic algorithm consists of two basic steps: (1) *Sorting requestors*: We create a sorted list of requestors (in ascending order of their latency requirements) after finding the minimum number of channels to which each requestor needs to be interleaved. By mapping requestors to the memory channels in order from this list, over-allocation of rate is reduced. (2) *Mapping to the memory channels*: The requestors 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 requestor requirements. During the mapping process, a configuration process is invoked for each requestor to determine the interleaving granularity and the rate that needs to be allocated in each channel, since according to Equation (4), 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 which happens when it is finally converted to TDM slots. We proceed by discussing the two steps in detail.

6.1. Sorting requestors

As we concluded in Section 4.1, we need to minimize the number of channels to which a requestor is interleaved to minimize the allocated bandwidth. Since we use a first-fit algorithm for mapping requestors to memory channels, the last ones 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 requestors that might end up having a larger over-allocation of rate if interleaved across multiple channels.

When a requestor is interleaved across a number of memory channels, n_{Ch}, as expressed by Equation (4), the amount of over-allocation of the required rate increases when its latency requirement is lower and the request size, N, is larger. Hence, we map requestors with lower latency requirement and larger request sizes first. Since it is hard to sort the requestors 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 requestor must be interleaved to meet their latency requirements. If the request size of a requestor is large such that its latency requirement, $\hat{L}$, 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, $n_{\check{C}h}$, to which the request needs to be interleaved is given by:

$$n_{\check{C}h} = 2^{(\lceil \log_2(q/\hat{L}) \rceil)} \tag{23}$$

In the above equation, $q/\hat{L}$ is rounded to the upper power-of-two since we need to allocate service units in the order of power-of-two for logical-to-physical address translation according to our method presented in Section 4.3. 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 requestor. Consider an example scenario in which $q = 8$ service units and $\hat{L} = 3$ service cycles. In this case, $\lceil q/\hat{L} \rceil = 3$ and with an allocation of the service units of 4, 2 and 2 in each memory channel, respectively, the latency requirement of 3 service cannot be met. 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 requestor is interleaved across multiple memory channels. Note that the optimal method presented in Section 5 does not have the restriction of interleaving to the number of channels in the order of power-of-two.

We make a list of communicating requestor groups with each group consisting of at least one requestor requiring more than one memory channel, i.e. $n_{\check{C}h} > 1$. Note that a requestor group may consist of a single requestor which do not have any communication requirements. We need to map these requestors first because $n_{\check{C}h} > 1$ indicates a lower latency and a larger request size, which must be mapped first to avoid a larger over-allocation of rate. We do not sort this list based on the result of $\frac{q}{\hat{L}}$; moreover, we allocate each requestor group from this list to different memory channels. This is because, we find $n_{\check{C}h}$ for each requestor after allocating 100% of bandwidth available in each channel, and hence, a requestor with $n_{\check{C}h} > 1$ uses

most of the bandwidth of the $n_{\check{C}h}$ memory channels. This means two requestor groups belonging to this list cannot be mapped to the same set of channels.

(2) The remaining requestor groups with requestors requiring $n_{\check{C}h} = 1$ are sorted (using a quick-sorting algorithm) according to the ascending order of the average of the worst-case latency requirements of the requestors in each group. This is because the amount of over-allocation of rate increases as the latency requirements get tighter according to Equation (4).

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

6.2. Mapping to memory channels

The requestor groups are picked one by one from the sorted list in order and a configuration process, shown in Algorithm 1, 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, $n_{\check{C}h}$, to which each requestor in the group needs to be interleaved. The interleaving granularity, N, in every channel is determined by dividing the request size by the number of channels to which the request needs to be interleaved (line 2). Note that N will always be in the order of power-of-two since q and $n_{\check{C}h}$ are always power-of-two. For the interleaving granularity in each channel, N, the new rate ρ'_{new} is recomputed such that it satisfies the latency requirement $\hat{L}$ by solving Equation (2). Since the ceiling functions from the latency equation are removed, we added 2 to make the computation conservative. The rate required by the requestor in the channel is then maximum of its required rate and the newly computed rate (line 4). The required rate is divided equally across the number of channels to which the requestor needs to be interleaved, since we distribute the number of service units evenly among the channels. Finally, the allocated rate, ρ', is computed considering discretization of the required rate (line 5).

Algorithm 1 Find interleaving granularity and allocated rate of a requestor.

Input: Min. number of channels interleaved $n_{\check{C}h}$, request size q, worst-case latency $\hat{L}$ and bandwidth $\check{b}$ requirements, worst-case bandwidth of a memory channel b^{ch}, TDM frame size f.
Output: Number of service units N and rate ρ' allocated to each channel.

1: **procedure** CONFIGURE($n_{\check{C}h}, q, \hat{L}, b^{ch}, f, \check{b}$)
2: $N \leftarrow \frac{q}{n_{\check{C}h}}$
3: $\rho'_{new} \leftarrow \frac{(f-\hat{L}+2)+\sqrt{(f-\hat{L}+2)^2+4\cdot f\cdot N}}{2\cdot f}$
4: $\rho'_{req} \leftarrow max\left(\frac{\check{b}}{b^{ch}\cdot n_{\check{C}h}}, \rho'_{new}\right)$
5: $\rho' \leftarrow \frac{\lceil f\times\rho'_{req}\rceil}{f}$
6: **return** N, ρ'
7: **end procedure**

Finally, the requestor 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 requestor is interleaved. When a requestor 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, $n_{\check{C}h}$ 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 requestors 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 which satisfies the condition that the sum of rates allocated to all requestors in each channel is less than or equal to one.

6.3. Algorithm computational complexity and optimality

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

Our heuristic algorithm always interleaves to a number of memory channels in the order of power-of-two. Hence, we divide the request size, and thereby also the rate, equally when a requestor is interleaved across multiple memory channels, which is optimal for requestors with tight latency requirement as we have seen in Section 6.1. However, when the latency requirement of a requestor 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 the order of power-of-two for two reasons: (1) This work primarily focuses on mapping of firm real-time requestors 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.

7. EXPERIMENTS

We present two different experiments in this section: First, in Section 7.1, we demonstrate the real-time guarantees provided by our multi-channel memory controller previously presented in Section 4. Then in Section 7.2, we show the performance comparison between our two proposed methods for mapping memory requestors to memory channels, optimal and heuristic algorithm presented in Sections 5 and 6, respectively, and two existing mapping algorithms.

7.1. Multi-channel memory controller architecture evaluation

First, we present our experimental setup and then we proceed with a discussion of the simulation results.

7.1.1. Experimental setup. We implemented a cycle-accurate SystemC model of the multi-channel memory controller architecture using *Predator* [Akesson and Goossens 2011a] as real-time channel controllers attached to a Wide IO 200 MHz DRAM [JEDEC] memory model with each channel consisting of 4 banks and a data bus of 128-bits wide. TDM arbiters with continuous slot allocations, previously discussed in Section 3.2, are used as the $\mathcal{LR}$ arbiter in the channel controllers.

To demonstrate the guarantees provided by the multi-channel memory controller on worst-case latency and bandwidth, we consider two requestors: one with low latency requirements (R1) and the other with large worst-case bandwidth requirements (R2), which corresponds to real-time low-latency and streaming clients [van der Wolf and Geuzebroek 2011], respectively. We used the *mpeg2 decode* application from the MediaBench [Lee et al. 1997] benchmark suite applications as R1. To emulate R1, 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 [SimpleScalar] with a unified 64 KB and 128 KB L1 and L2 caches, respectively, 64 byte 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 is arrived back at the requestor without the impact of self-interference [7], we have configured the traffic generator with maximum one outstanding transaction such that R1 issues memory (read) 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 R2, 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 requestor is allocated to ensure that the requestor is always backlogged. The synthetic requestor 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 [Akesson and Goossens 2011b]). We selected the service unit size equal to the access granularity of 32 B, which takes 13 clock cycles to read and write to the memory (service cycle), and we choose this as the TDM slot size. We selected this service unit size since it is smaller than the request size (of 64 B) which gives us the flexibility of interleaving the memory requests across channels.

Bandwidth is computed by logging the time stamp at which each service unit is scheduled by the channel controller, and then counting the requests served by a channel controller. 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 latency of a transaction, we find the time difference between the time at which a read request arrives at the request buffer of the multi-channel memory controller until the complete response arrives back.

We measured the latency and bandwidth of R1 and R2, respectively, for different cases (discussed in the next section) and compared against the analytically computed worst-case bounds. The worst-case bandwidth of a requestor is computed as a fraction of the worst-case bandwidth provided by a channel using the fraction of TDM slots (rate) allocated to the requestor. We computed the worst-case latency bound using Equation (2), and also included the overhead due to the number of registers in the critical path or pipeline stages (9 clock cycles) in the hardware and one refresh duration (130 ns for Wide IO DRAM). We add only one refresh duration to the latency of a transaction, 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 200MHz 2 Gb device compared to the TDM frame size of 6 (= 390 ns), which we used in our experiments.

7.1.2. Simulation results. We need to evaluate the guarantees on latency and bandwidth provided by our multi-channel memory controller to R1 and R2, 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 in-

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

terleaving: 1) Neither requestor is interleaved across memory channels. 2) Only R1 is interleaved across two memory channels. 3) Only R2 is interleaved across two memory channels. 4) Both requestors are interleaved across memory channels. Figures 7 & 8 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 R1 and bandwidth of R2 during the first 200 μs of the simulation with their respective worst-case bounds.

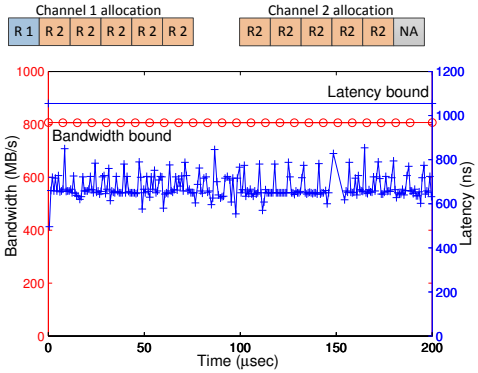

Fig. 7. *Case 3*: R2 is interleaved across two memory channels with a rate of 5/6 in each channel, and R1 is interleaved to one memory channel with a rate of 1/6.

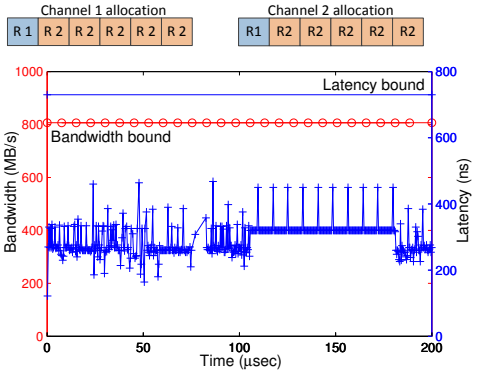

Fig. 8. *Case 4*: Both requestors R1 and R2 are interleaved across the two memory channels with a rate of 1/6 and 5/6, respectively, in each channel.

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 pessimistic bound. However, the worst-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 [Akesson and Goossens 2011b]. Note that R2 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 7 & 8, it can be seen that the average latency of R1 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 (4).

To summarize, we have demonstrated that the bounds on bandwidth and latency given to the requestors 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 varies 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 by the memory requestors.

7.2. 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 5 & 6, respectively), and two existing mapping algorithms, *First-fit* and *Interleave-all*. The First-fit is a basic bin-

packing algorithm that picks one requestor at a time and maps to one of the first memory channels that has enough resources (bandwidth and memory capacity) available to meet the requestor 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 requestors across memory channels. The Interleave-all algorithm maps every requestor 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 [Xilinx Inc.]. 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.

7.2.1. Experimental setup. The experimental setup consists of the optimization problem model implemented in the CPLEX optimization tool [CPLEX], implementation of our proposed heuristic, the First-fit and Interleave-all algorithms in C++, for a TDM arbiter, and a synthetic use-case generator. 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 (considering discretization of rate) and which satisfies the condition that the sum of rates allocated to all requestors in each channel is less than or equal to one. For the implementation of the optimization problem for a TDM arbiter in CPLEX, we substitute its worst-case latency expression given by Equation (2) in Equation (12) of Constraint 1. Since CPLEX do not support decision variables in the denominator, such as ρ' in Equation (2), we multiply the equation by ρ' and the constraint hence becomes quadratic, as expressed by Equation (24), 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 $f \times (1 - \rho'_c(r)) + 1$ and $N_c(r)/\rho'_c(r) + 1$, respectively, to make the computation conservative.

$$\forall c \in C, r \in R : f \times \rho'_c(r)^2 - \rho'_c(r) \times (f - \hat{L}(r) + 2) - N_c(r) \geq 0 \tag{24}$$

To compare the performance of the optimal method and the heuristic under different scenarios, we used a synthetic use-case generator, which generates memory requestors 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 requestors: requestors with low average latency requirements, such as LCD controllers and CPUs [Stevens 2010], requestors with medium latency requirements, such as H.264 video decoders [Aho et al. 2009], and requestors with relaxed latencies, which includes a wide variety of requestors with low and high bandwidth requirements, e.g., graphics processing [Stevens 2010], input processors [Steffens et al. 2008]. We do not consider memory capacity requirements since we did not have sufficient data to define the range for the different traffic classes. We considered a 4-channel Wide IO 200 MHz DRAM with an access granularity of 64 B and a worst-case bandwidth of 966.9 MB/s per channel (as in the previous section) for mapping the requestors.

7.2.2. Mapping success ratio. Using the heuristic, First-fit and Interleave-all algorithms, we performed mapping with 200 different use-cases [8], which are feasible according to the optimal algorithm, with different number of requestors (5-25) with different requirements in each use-case. In all algorithms, we set an upper bound of 100 for the frame size considering the long computation time of the optimal algorithm, but

[8]We had to limit the use-cases to 200 due to the long computation time of the optimal algorithm.

we assume that this is sufficiently large for our use-cases. The mapping success ratio of the heuristic, First-fit and Interleave-all algorithms, normalized to the success ratio of the optimal method is shown in Figure 9. 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 67%. Our heuristic algorithm failed to find a valid mapping for about 7% of the use-cases consisting mainly of requestors with relaxed latency requirements. The mapping failed for those use-cases when the total required bandwidth by all requestors is more than 95% of the maximum bandwidth capacity of all channels. As we have already seen in Section 6.3, our heuristic algorithm distributes the rate evenly across the channels to which a requestor is interleaved. Since the heuristic algorithm does mapping based on a first-fit algorithm, it could fail for one of the last requestors 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 requestors 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.

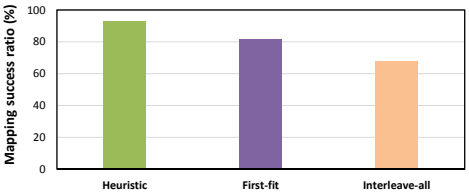

Fig. 9. Mapping success ratio of the heuristic, First-fit and Interleave-all algorithms normalized to the optimal method.

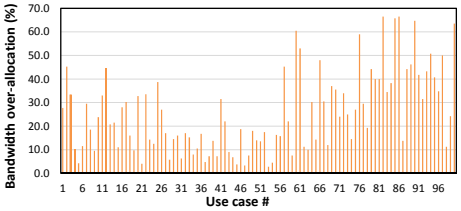

Fig. 10. Bandwidth over-allocation of the Interleave-all algorithm with respect to the optimal algorithm for different use-cases.

The First-fit algorithm failed for up to 19% of use-cases since it did not interleave requestors that could have been successfully mapped by interleaving across multiple channels. Note that we did not include communication requirements for requestors 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. 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 requestors with tight latency requirements. Figure 10 shows the over-allocated bandwidth by the First-fit algorithm (of 100 feasible mappings) with respect to the heuristic and optimal algorithms.

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 simple without the need of Channel Selector. However, the overhead in the worst-case latency due to the Channel Selector is one clock cycle which is negligible. Moreover, we saw that *the traditional approach of interleaving every memory requestor across all memory channels available is not an efficient method in real-time multi-processor platforms*. In the next section, we evaluate the trade-off between algorithm computation time and mapping success ratio of our optimal and heuristic algorithms.

7.2.3. Computation time comparison. The computation time of the optimal algorithm in CPLEX and the heuristic algorithm with different number of requestors and for different number of memory channels are shown in Table III. Note that the solver takes a significant amount of time to search through all solutions because of the large design space of the optimization problem. Hence, the time taken by CPLEX shown in this ta-

ble 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 [HMC]. 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, large size future systems require the heuristic algorithm to be analyzed in reasonable time and this comes at a reduction of approximately 7% in success ratio of the mapping.

Table III. Tool vs Heuristic - computation time

Channels	*Requestors*	*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		

8. CASE STUDY: CONFIGURING A WIDE IO DRAM IN A HD VIDEO AND GRAPHICS PROCESSING SYSTEM

In this section, we present a case study where we use the proposed multi-channel memory controller and mapping algorithm to configure a 4-channel Wide IO SDR 200MHz DRAM [JEDEC] device in a HD video and graphics processing system. First, we derive memory subsystem requirements for the video processing system, and then show the configuration of the multi-channel memory controller for the Wide IO memory device.

8.1. 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 11. This system is based on the industrial systems from [Steffens et al. 2008] and [Stevens 2010] 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. In addition, the host CPU and a Direct Memory Access (DMA) controller require memory access to perform system-dependent activities in UMAs [Stevens 2010]. For simplicity, we do not show the DMA block. The GPU and CPU requirements are based on [Stevens 2010], and the IP requirements on [Steffens et al. 2008]. The VE and HDLCD requirements are computed considering the requirements for HD video with a resolution of 1920×1080, 8 bpp and 30 fps [Bonatto et al. 2011].

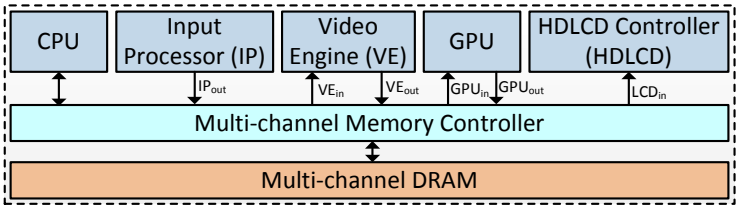

Fig. 11. Memory-centric architecture of a HD video and graphics processing system.

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 [Steffens et al. 2008], and writes to memory (IP_{out}) at 1 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 [Hongqi et al. 2007; Bonatto et al. 2011]. 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 transaction size of IP and VE as 128 B.

The bandwidth requirement by 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 [Stevens 2010]. 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} has as 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. Conservatively, we allocate a bandwidth of 1000 MB/s to GPU_{in}. In total, we allocate a worst-case guaranteed bandwidth of 1249.8 MB/s to the GPU, which is conservative compared to its average bandwidth requirement of 1500 MB/s [Stevens 2010]. We assume a GPU cache-line size of 256 B for the transaction size.

The HDLCD is latency critical [Steffens et al. 2008], 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 two times by the LCD controller. For a LCD DMA 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 device. The CPU is cache-based and has a cache line size of 64 B [Stevens 2010] . We allocate a bandwidth of 150 MB/s to the latency-sensitive system-dependent bandwidth requirements by the CPU and DMA [Stevens 2010]. The summary of system requirements are shown in Table IV. The requestors that need to communicate are assigned the same group number g.

8.2. Configuring the Wide IO DRAM

For the Wide IO SDR 200 MHz device with 4 memory channels, we selected an access granularity of 64 B in each channel that provides a worst-case bandwidth of 966.9 MB/s. This configuration provides sufficient guaranteed bandwidth of 3867.6 MB/s (4×966.9 MB/s) to meet the requirements of all requestors with 2511.7 MB/s considering the 35% slack for over-allocation. We selected a service unit size equal to the access granularity of 64 B, since it is smaller than most of the request sizes in Table IV, which allows interleaving of the memory requests across channels. For the service unit size of 64 B, it takes 13 clock cycles to perform a read or write operation (service cycle), and hence we choose this as the TDM slot size. Our optimization problem formulation in CPLEX took about 10 minutes and the heuristic algorithm less than a second to find a valid mapping of requestors to the memory channels with a frame size of 10. Both the methods provided the same mapping result, shown in Table V.

It can be seen that the requestors GPU_{out} and LCD_{in} are allocated to a single memory channel. Note that the required rate by each of those requestors is 0.25 and the rate allocated is 0.5 i.e., $0.5 \times 996.9 = 483.4$ MB/s, which amounts to an over-allocated bandwidth of 241.7 MB/s. The over-allocation primarily is due to its tight latency requirement and secondarily due to the discretization of the rate. Hence, interleaving them across memory channels could result in much larger over-allocation of rate as we

Table IV. Memory subsystem requirements

Requestor	$\check{b}$ *(MB/s)*	$\check{L}$ *(cycles)*	*s (B)*	*g*
IP_{out}	1	-	128	1
VE_{in}	769.8	-	128	1
VE_{out}	93.3	-	128	2
GPU_{in}	1000	-	256	2
GPU_{out}	248.8	205	256	3
LCD_{in}	248.8	205	256	3
CPU	150	-	64	4
Total	2511.7			

Table V. Mapping of requestors - allocated service units & rates

Requestor	*Channel 1*		*Channel 2*		*Channel 3*		*Channel 4*	
	N_1	ρ_1'	N_2	ρ_2'	N_3	ρ_3'	N_4	ρ_4'
IP_{out}	2	0.10	0	0	0	0	0	0
VE_{in}	2	0.80	0	0	0	0	0	0
VE_{out}	0	0	0	0	1	0.10	1	0.10
GPU_{in}	0	0	0	0	2	0.60	2	0.60
GPU_{out}	0	0	4	0.50	0	0	0	0
LCD_{in}	0	0	4	0.50	0	0	0	0
CPU	0	0	0	0	0	0	1	0.20
Total	4	0.90	8	1.00	3	0.70	4	0.90

have described in Section 4.1. GPU_{in} is interleaved across two memory channels, since its bandwidth requirement of 1 GB/s cannot be satisfied in a single channel. VE_{out} is also interleaved across the same set of channels as GPU_{in}, since they communicate and hence belong to the same group. However, over-allocation of rate is not required for GPU_{in} and VE_{out} because of their relaxed latency requirements, but for the discretization of their rates. The remaining requestors are not interleaved across memory channels as their bandwidth and latency requirements are satisfied by mapping to a single memory channel. From the above discussion, it is clear that the traditional approach of always interleaving all memory requestors across all memory channels does not work for this use-case.

To summarize, the requests from the requestors are interleaved across memory channels at different granularities depending on their latency/bandwidth requirements, request sizes and/or communication requirements, for optimal memory bandwidth utilization. In total, we have a slack bandwidth of 483.4 MB/s which could be allocated to soft/non-real time requestors in the system which improves their performance. Memory capacity requirements of the requestors generally also impact the interleaving of requests across channels, however, the memory capacity requirements by these requestors in our use case were not large enough to impact the mapping results. Hence, it is clear that for optimal memory bandwidth utilization, we need a configurable memory controller architecture that enables interleaving of memory requests across memory channels in different granularities and can be allocated with different rates in different memory channels.

9. CONCLUSIONS

Shared multi-channel memories in multi-processor platforms for real-time systems are tedious to configure and verify. As a first work in this direction, we presented a real-time multi-channel memory controller architecture that can interleave memory requests across multiple memory channels at different granularities. We also 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 multi-channel memory controller, while minimizing bandwidth utilization. We show that for a use-case scenario consisting of 4 memory channels and up to 100 memory requestors, a solver can find an optimal mapping in 2 hours, and our heuristic tool in less than 1 second. Also, we show that 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 the 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. Finally, we demonstrated the effectiveness of our work in a real use-case scenario by configuring a Wide IO DRAM controller in a HD video processing system.

In this work, we have evaluated the conservativeness of the real-time guarantees provided to the firm real-time clients. In the future, we would like to evaluate the average-case performance gain of the soft real-time clients in the system by allocating them the slack bandwidth after mapping the firm real-time clients.

REFERENCES

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

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

B. Akesson and K. Goossens. 2011b. *Memory Controllers for Real-Time Embedded Systems* (first edition ed.). Springer.

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

M. Awasthi, D.W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. 2010. 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)*. ACM, 319–330.

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

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

C. Bouquet. 2000. Optimal Multi-channel Memory Controller System. Patent number: 6643746. (2000).

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

P. Casini. 2008. SoC Architecture to Multichannel Memory Management Using Sonics IMT. White paper. (2008). Sonics, inc.

CPLEX. IBM ILOG CPLEX Optimizer. http://www.ibm.com.

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

M.D. Gomony, B. Akesson, and K. Goossens. 2013. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*. 1307–1312.

M.D. Gomony, C. Weis, B. Akesson, N. Wehn, and K. Goossens. 2012. DRAM selection and configuration for real-time mobile systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 51–56.

S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. 2013. A Reconfigurable Real-Time SDRAM Controller for Mixed Time-Criticality Systems. In *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013*.

HMC. http://www.hybridmemorycube.org.

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

JEDEC. Wide I/O Single Data Rate Specification. http://www.jedec.org.

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

H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. 2014. Bounding memory interference delay in COTS-based multicore systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS 14)*.

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

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

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

C. Lin and S.A. Brandt. 2005. Improving Soft Real-Time Performance Through Better Slack Management. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. 12 pp.–421.

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

D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. 2012. 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)*. ACM, 1137–1142.

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

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

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

J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. 2011. 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)*. ACM, 99–108.

J.C. Sancho, M. Lang, and D.K. Kerbyson. 2010. 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*. 1–7.

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

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

SimpleScalar. http://www.simplescalar.com.

S. Sriram and S. S. Bhattacharyya. 2000. *Embedded Multiprocessors: Scheduling and Synchronization* (1st ed.). Marcel Dekker, Inc., New York, NY, USA.

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

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

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

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

Texas Instruments Inc. TMS320VC5505/5504 DSP Direct Memory Access (DMA) Controller. http://www.ti.com.

C. H. (Kees) van Berkel. 2009. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. 1260–1265.

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

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

Xilinx Inc. LogiCORE IP XPS Multi-channel External Memory Controller (XPS MCH EMC). http://www.xilinx.com.

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

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. 2010. A 3D SoC design for H.264 application with on-chip DRAM stacking. In *3D Systems Integration Conference (3DIC), 2010 IEEE International*. 1–6.

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

Received month year; revised month year; accepted month year