Architecture and Optimal Configuration of a Real-Time Multi-Channel Memory Controller

Manil Dev Gomony*, Benny Akesson†, and Kees Goossens*
*Eindhoven University of Technology, The Netherlands
†CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal

***Abstract*—Optimal utilization of a multi-channel memory, such as Wide IO DRAM, as shared memory in multi-processor platforms depends on the mapping of memory clients to the memory channels, the granularity at which the memory requests are interleaved in each channel, and the bandwidth and memory capacity allocated to each memory client in each channel. Firm real-time applications in such platforms impose strict requirements on shared memory bandwidth and latency, which must be guaranteed at design-time to reduce verification effort. 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.**

This paper has four key contributions: (1) A real-time multi-channel memory controller architecture with a new programmable *Multi-Channel Interleaver* unit. (2) A novel method for logical-to-physical address translation that enables interleaving memory requests across multiple memory channels at different granularities. (3) An optimal algorithm based on an Integer Linear Program (ILP) formulation to map memory clients to memory channels considering their communication dependencies, and to configure the memory controller for minimum bandwidth utilization. (4) We experimentally evaluate the run-time of the algorithm and show that an optimal solution can be found within 15 minutes for realistically sized problems. We also demonstrate configuring a multi-channel Wide IO DRAM in a High-Definition (HD) video and graphics processing system to emphasize the effectiveness of our approach.

I. 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 processing elements. Such platforms run several applications with diverse real-time requirements [1], and moreover, the firm real-time applications impose strict worst-case requirements on main memory performance in terms of bandwidth and/or latency [2] [3]. These requirements must be guaranteed at design-time to reduce the verification effort, which is made possible using real-time memory controllers [4]–[6] that bound the memory access time by employing predictable arbiters, such as TDM and Round-Robin. Real-time memory controllers can be analyzed using resource abstractions, such as the Latency-Rate ($\mathcal{LR}$) server model [7].

Memories with multiple physical channels and wide interfaces, such as Wide IO DRAMs [8], are essential to meet the main memory *power/bandwidth* demands of future real-time systems [9]. In multi-channel memories, the bandwidth allocated to firm real-time memory clients to meet their latency requirements depends on the mapping of clients to the memory channels and the granularity at which the memory requests are interleaved in each channel, i.e., the *interleaving granularity*. The allocated bandwidth must be minimal so that the slack bandwidth available can be allocated to the soft real-time clients in the system, which improves their average-case performance. However, for optimal memory bandwidth utilization, there is currently no methodology to map memory clients to memory channels and to determine the interleaving granularity, and the bandwidth and memory capacity allocated to each memory client in each channel. Also, there is no real-time memory controller architecture for multi-channel memories that can be programmed with the optimal configuration.

This paper has four contributions: (1) A real-time multi-channel memory controller architecture, shown in Figure 1, with a new programmable *Multi-Channel Interleaver* and each channel controlled by an existing real-time memory controller. (2) A novel logical-to-physical address translation method that enables interleaving of a memory request in different sizes across any number of memory channels. (3) An optimal algorithm based on an Integer Linear Program (ILP) formulation to map memory clients to memory channels considering their communication dependencies, and to configure the memory controller for minimum bandwidth utilization. (4) We experimentally evaluate the run-time of the optimal algorithm, and we also demonstrate configuring a multi-channel Wide IO DRAM for a High-Definition (HD) video and graphics processing system using our approach.

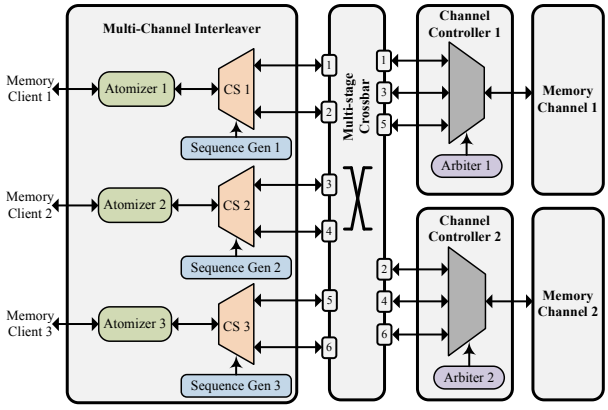

Fig. 1. High-level view of real-time multi-channel memory controller architecture showing three memory clients and two memory channels. The *Atomizer* chops a memory request in to smaller sub-units and the *Channel Selector (CS)* routes these sub-units to the different memory channels according to the configuration in the Sequence Generators.

In the remainder of this paper, Section II reviews the related work, Section III gives an introduction to state-of-the-art real-time memory controllers and the $\mathcal{LR}$ server model. In Section IV, we introduce our proposed multi-channel memory controller architecture, including a method for logical-to-physical address translation. We present the formulation of our optimal algorithm in Section V, and evaluate its run-

978-3-9815370-0-0/DATE13/©2013 EDAA

time in Section VI. Section VII 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 VIII.

II. Related work

Among the previous related works, some exploit the benefits of interleaving data across multiple memory channels. In [10] and [11], data is interleaved across the memory channels such that all channels are accessed by a single transaction to improve average-case performance. Similarly in [12], the traffic within a logical address region is split 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 in [13]. 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 previous related works focus on memory controller architectures and logical-to-physical address translation for multi-channel memories. In [14], a parallel-access mechanism is proposed in which two separate DDR Finite State Machines (FSM) are used to control 8 memory channels of a 3D-DRAM. The proposed architecture in [15] 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 in [16] uses a dynamic mapping strategy by using a mapping table that stores the logical-to-physical address translation, and a crossbar switch is used for routing traffic across multiple memory channels. Also, the multi-channel memory controller architecture proposed in [17] routes an incoming request to any of the memory channels using a crossbar. In [18], an architecture is presented 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 are predictable or perform logical-to-physical address translation for requests interleaved with different interleaving granularities. Even though there are real-time memory controllers that provides bounds on memory performance [4]–[6], they do not consider multi-channel memories and interleaving data across multiple channels.

To summarize, presently there is no real-time multi-channel memory controller and no logical-to-physical address translation method for multi-channel memories. Also, there is no structured methodology to determine the optimal mapping and number of memory channels to which a memory request needs to be interleaved, the interleaving granularity, and the bandwidth allocated in each channel, for optimal memory bandwidth utilization in real-time systems.

III. Background

This work relies on existing single-channel real-time memory controllers to bound the memory response time, 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.

A. Real-time memory controllers

State-of-the-art real-time memory controllers [4]–[6] bound the execution time of a memory transaction by fixing the memory access parameters, such as burst size and number of read/write commands, at design-time. These parameters define the *access granularity* of the memory controller. When the access granularity is fixed for a memory device, the worst-case execution time of a read/write transaction can be computed from the worst-case timing behavior provided by the memory data-sheet. Also, the worst-case bandwidth offered by a memory for a fixed access granularity can be computed [19]. In this paper, we refer to a memory transaction of a fixed size as a *service unit*, and the time taken to serve a service unit is *service cycle*. The service cycle for a read and a write transaction can be different and depends on the memory device.

B. $\mathcal{LR}$ servers

Latency-Rate ($\mathcal{LR}$) servers are a general model to capture the worst-case behavior of various scheduling algorithms or arbiters in a simple unified manner [7], 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, and priority based arbiters with a rate-regulator. The $\mathcal{LR}$ abstraction enables modeling of many different arbiters, and is compatible with a variety of formal analysis frameworks, such as data-flow or network calculus.

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 paper, we use the term *requestor* to denote a memory client that requests access to a memory resource with certain bandwidth and latency requirements. Figure 2 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. 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).

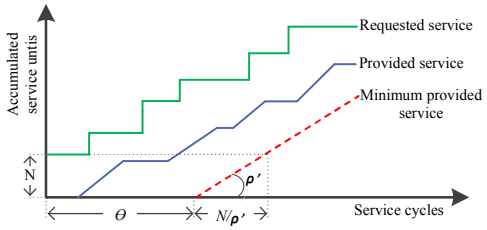

Fig. 2. Example service curves of a $\mathcal{LR}$ server showing service latency and completion latency.

The *service latency* is the maximum time taken to schedule a request at the head of a requestor's request queue because of interfering clients and depends on the choice of arbiter and its configuration, e.g. allocated rate and/or priority [19]. 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 L^{max} (in service cycles) of a requestor is then the sum of the service latency and the completion latency, given by $L^{max} = \Theta + \lceil N/\rho' \rceil$.

This work considers a TDM arbiter as an example of a $\mathcal{LR}$ server. For a TDM arbiter with a frame size f and

consecutively allocated slots, the worst-case latency of a requestor with an allocated rate of ρ' is given by Equation (1). The service latency is $f \times (1-\rho')$ because of the interference from other requestors that occupy the remaining fraction of TDM slots. Both service latency and completion latency are rounded up to make the bound conservative.

$$L^{max} = \lceil f \times (1-\rho') \rceil + \left\lceil \frac{N}{\rho'} \right\rceil \quad (1)$$

IV. Multi-channel memory controller for real-time systems

We start this section with an analysis of the impact of interleaving data across multiple memory channels on the 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 real-time multi-channel memory controller architecture, followed by a method for logical-to-physical address translation.

A. $\mathcal{LR}$ servers and multi-channel memories

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*. 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 can be observed in Equation (2), 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 to the number of channels to which the request is interleaved nCh. It can be seen that the service latency increases with nCh, however, the completion latency remains constant. This conclusion is valid for all other $\mathcal{LR}$ arbiters as well and is evident from their worst-case latency equations [19]. Hence, when a requestor is interleaved across multiple memory channels, the latency requirement by the requestor might not be satisfied with its required rate ρ', and a higher rate than the required rate, i.e., *over-allocation* of rate might be required depending on its latency requirement.

$$L^{max'} = \left\lceil f \times (1 - \frac{\rho'}{nCh}) \right\rceil + \left\lceil \frac{N/nCh}{\rho'/nCh} \right\rceil \quad (2)$$

In a real-time system consisting of several memory requestors with diverse bandwidth/latency requirements, memory capacity requirements and request sizes, 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 configured to interleave memory requests of a requestor to any number of available memory channels at different granularities.

B. Real-time multi-channel memory controller architecture

The proposed multi-channel memory controller, shown in Figure 1, 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 [4]–[6] employing any $\mathcal{LR}$ arbiter. We use a Multi-stage Crossbar that connects each requestor to every Channel Controller. 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. The Multi-Channel Interleaver consists of an *Atomizer* and a *Channel Selector (CS)* and a *Sequence Generator* connected to each memory requestor. The Multi-Channel Interleaver has separate *request* and and *response* paths. In the request path, the Atomizer chops an incoming memory request into a number of service units, and the CS routes the service units to the different memory channels according to the configuration in the Sequence Generator. The response from the different memory channels arrive at different times. Hence, the incoming service units are buffered in the receive path until all service units from the different channels have arrived, and then the response is reconstructed by the Atomizer and sent back to the requestor. The CS also performs a logical-to-physical address translation for a requestor in each memory channel.

C. Logical-to-physical address translation

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. Figure 3a and 3b 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 4 service units. Figure 3b 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.

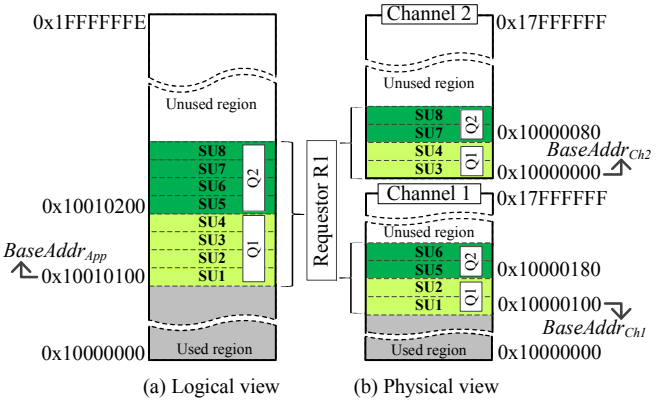

Fig. 3. Example memory map showing requestor R1 allocated to two memory channels, with every request Q1 and Q2 interleaved across the two channels.

As shown in Figure 3b, the service units of a memory request can end up in different physical addresses in each channel when interleaved across multiple memory channels. This is because the optimal mapping of requestors to the channels results in each channel mapped with different number of requestors with different memory capacities allocated. However, the application programmer must be able to view the entire memory space (including all memory channels) as a single continuous logical address space, as in Figure 3a, to avoid

explicit data partitioning and data movement while writing the application program. Hence, 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 (3).

$$ReqAddr_{Ch} = ((ReqAddr_{App} - BaseAddr_{App}) \gg (\log_2(Request\ size/N_{Ch_n}))) + BaseAddr_{Ch_n} \quad (3)$$

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 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 consider the number of service units allocated to each channel in the order of power of two, assuming request sizes to be a power of two.

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 an optimal algorithm for requestor mapping and allocation presented in Section V. 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. In the next section, we present an optimal method to map memory requestors to memory channels and configure the multi-channel memory controller.

V. Optimal method for requestor mapping and configuration in multi-channel memories

Given that we have presented a multi-channel memory controller architecture that can be programmed with an optimal configuration, we proceed with our method to determine the optimal configuration. First, we present a formal definition of our system and then our generic optimization problem formulation, which applies to any arbiter belonging to the class of $\mathcal{LR}$ servers.

A. System definition

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 a channel $c \in C$ is given by $AG(c)$, with a service cycle (in ns) given by $SC^{ns}(c)$. For each memory channel $c \in C$, the worst-case bandwidth (in MB/s) can be computed for a fixed access granularity $AG(c)$ (e.g. see [19]), and is given by $b^{ch}(c)$.

Consider a set of requestors denoted as $r \in R$, with a worst-case latency requirement (in ns) given by $L^{ns}(r)$, minimum bandwidth requirement (in MB/s) given by $b^{min}(r)$, and a total memory capacity requirement (in Bytes) given by $B^{req}(r)$. The worst-case latency requirement of a requestor (in service cycles) in each channel $c \in C$ is given by $L^{max}(r)$, and is defined as $\forall r \in R : L_c^{max}(r) = \lfloor L^{ns}(r)/SC^{ns}(c) \rfloor$. The request size (in Bytes) of requests from a requestor $r \in R$ is given by $s(r)$, and we assume a constant request size for all requests from a single requestor. The number of service units in each request is given by $q(r)$ and is defined as $\forall r \in R : q(r) = \lceil s(r)/AG \rceil$. 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 since they need to be able to access the same set of channels. In the next section, we define the optimization problem statement and formulate it as an ILP.

B. Optimization problem formulation

We define our optimization problem as follows: *Find the mapping of requestors to the memory channels, and the allocation of number of service units, N_c, and a rate, ρ'_c, for each requestor $r \in R$ in each memory channel $c \in C$, such that the sum of rates allocated to all requestors is minimized.* The optimization problem is defined as

$$\textit{Minimize:} \sum_{c \in C} \sum_{r \in R} \rho'_c(r) \quad (4)$$

Such that the following seven constraints are satisfied:

Constraint 1: The worst-case latency of each requestor $r \in R$ after allocation $L^{max'}(r)$ must be less than or equal to its worst-case latency requirement $L^{max}(r)$, and is defined as $\forall c \in C, r \in R : \ L^{max'}(r) \leq L^{max}(r)$. The service units of every request of a requestor are allocated across the memory channels such that each requestor has a (Θ, ρ) per channel. The worst-case latency of a requestor $r \in R$ in each channel $c \in C$ is then given by $L_c^{max'}(r)$, and is defined as $\forall c \in C, r \in R : L_c^{max'}(r) = \Theta_c(r) + \lceil N_c(r)/\rho'_c(r) \rceil$, where $\Theta_c(r)$ is the service latency of a requestor in each channel. 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 $\forall c \in C, r \in R : L^{max'}(r) = \max_{c \in C} L_c^{max'}(r)$. The non-linear *max* function is made linear to enable formulation as an ILP, and Constraint 1 is then defined as

$$\forall c \in C, r \in R : \ L^{max}(r) - L_c^{max'}(r) \geq 0 \quad (5)$$

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

$$\forall c \in C : \sum_{r \in R} \rho'_c(r) \leq 1 \quad (6)$$

Constraint 3: The sum of rates allocated to each requestor $r \in R$ across all memory channels should be greater or equal to its minimum required rate, defined by Equation (7). The minimum rate required by a requestor is the ratio of its minimum bandwidth requirement $b^{min}(r)$ and the worst-case bandwidth offered by a memory channel $b^{ch}(r)$.

$$\forall r \in R : \sum_{c \in C} \rho'_c(r) \geq \frac{b^{min}(r)}{b^{ch}(r)} \quad (7)$$

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

$$\forall r \in R : \sum_{c \in C} N_c(r) = q(r) \tag{8}$$

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 (9) and $N'_c(r)$ is in the range $0..\log_2[q(r)]$. Constraint 5 is then defined by Equation (10)

$$b_c(r) = \begin{cases} 1, & \text{if } N_c(r) > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

$$\forall c \in C, r \in R : N_c(r) = 2^{N'_c(r)} \times b_c(r) \tag{10}$$

Constraint 6: Each two 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. Two communicating requestors share the same physical address space for data sharing and they may have different request sizes. The number of service units of the two requestors allocated in each memory channel must be proportional, such that by dividing the logical address offset with the ratio of request size to the number of service units, as in Equation (3), results in the same physical address for both the requestors. For two communicating requestors r_i and r_j, the constraint is defined by Equation (11). The decision variable $N'_c(r)$ is the same one defined under Constraint 5.

$$\forall c \in C, r \in R, g(r_i) = g(r_j) : \\ N_c(r_i) \times 2^{N'_c(r_j)} = N_c(r_j) \times 2^{N'_c(r_i)} \tag{11}$$

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 as

$$\forall c \in C : \sum_{r \in R} \frac{N_c(r)}{q(r)} \times B^{req}(r) \leq B^{ch}(c) \tag{12}$$

Our generic optimization problem formulation can be used to model an optimization problem for any $\mathcal{LR}$ arbiter by using the worst-case latency derivation of the corresponding arbiter in Constraint 1. In the next section, we demonstrate modeling the optimization problem for a TDM arbiter and evaluate its run-time in an optimization tool.

VI. Optimization for a TDM arbiter

We modeled the optimization problem in the CPLEX optimization tool [20]. First, we substituted Equation (1) in Constraint 1. Since decision variables in the denominator of constraints are not supported by the tool, we multiply the equation by ρ', as it is in the denominator in Equation (1). The constraint hence becomes quadratic as expressed by Equation (13), making it a quadratic constrained integer problem. 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.

$$\forall c \in C, r \in R : \\ f \times \rho'_c(r)^2 - \rho'_c(r) \times (f - L^{max}(r) + 2) - N_c(r) \geq 0 \tag{13}$$

A. Run-time evaluation

We used a synthetic use-case generator that generates different classes of memory requestors to evaluate the run-time of the optimization problem in the tool. We considered three different classes of memory requestors: (1) Requestors with low average latency requirements (LL), such as LCD controllers and CPUs [21]. (2) Requestors with medium latency requirements (ML), such as H.264 video decoders [10]. (3) Requestors with relaxed latencies (RL), which includes a wide variety of requestors with low and high bandwidth requirements, e.g., graphics processing [21], input processors [3], etc. The bandwidth, latency and request size ranges of different traffic classes are shown in Table I.

TABLE I
Traffic class specifications

Traffic	$L^{max}(\mu s)$	$b^{min}(MB/s)$	$s(B)$
LL	1-15	500-1000	64-1024
ML	15-30	150-500	64-1024
RL	30-100	1-1000	64-1024

Because of the large design space of the optimization problem (17 variables and 15 constraints for each requestor), the optimization tool takes significant amount of time to search through the entire design space. However, the time taken by the tool to find the first optimal solution is much less. This is observed from the solutions found by the tool at different time instants until it terminates normally. Since the tool does not automatically stop upon finding first optimal solution, we experimentally determined the maximum time it took to find the first solution for up to 100 seeds of use-cases. The results are shown in Table II for different number of requestors. It can be seen that the search can be terminated with a conservative time limit of 15 minutes in a worst-case scenario consisting of up to 100 requestors.

TABLE II
Worst-case run-time to find optimal allocation

Channels	*Requestors*	*First optimal*	*Complete run*
4	25	8.9 secs	3 hrs
	50	2.2 mins	10 hrs
	100	13.4 mins	2 days

VII. Case study: Configuring a Wide IO DRAM in a HD video and graphics processing system

In this section, we present the memory subsystem requirements for an HD video and graphics processing system, and then show configuring a 4-channel Wide IO SDR 200 MHz DRAM [8] device using our approach.

A. HD video and graphics processing system requirements

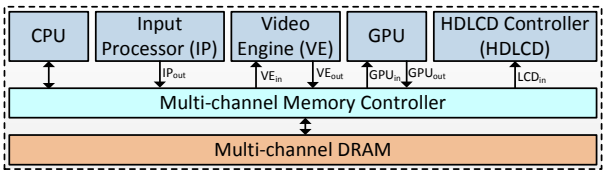

Fig. 4. Memory-centric architecture for HD video and graphics processing

A High-Definition video (1080p) and graphics processing system with a Unified Memory Architecture (UMA) is shown in Figure 4. This system is based on the industrial systems from [3] and [21] combined to create a suitable load for a modern multi-channel memory. The Input Processor (IP) receives an encoded video stream and writes to the memory. 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. The GPU and CPU requirements are based on [21], and the IP requirements on [3]. The VE and HDLCD requirements are computed considering the requirements for HD video with a resolution of 1920×1080, 8 bpp and 30 fps [22]. Due to lack of space, we do not show the derivation of the system requirements. A summary of the requirements is shown in Table III.

TABLE III
MEMORY SUBSYSTEM REQUIREMENTS

Requestor	b^{min} *(MB/s)*	L^{max} *(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}	500	102	256	3
LCD_{in}	500	102	256	3
CPU	150	-	128	4

B. 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 to meet the requirements of all requestors. We selected a service unit size equal to the access granularity of 64 B, since it is smaller than all request sizes in Table III, which allows interleaving of the memory requests across memory 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. We selected a frame size of 5 to meet the worst-case latency requirements of the HDLCD and GPU_{out} of 102 clock cycles, corresponding to 8 TDM slots. The configuration results found by the optimization tool are shown in Table IV.

TABLE IV
MAPPING OF REQUESTORS AND ALLOCATED SERVICE UNITS AND 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}	0	0	0	0	1	0.01	1	0.01
VE_{in}	0	0	0	0	1	0.4	1	0.4
VE_{out}	0	0	0	0	1	0.05	1	0.05
GPU_{in}	0	0	0	0	2	0.51	2	0.51
GPU_{out}	2	0.4	2	0.4	0	0	0	0
LCD_{in}	2	0.4	2	0.4	0	0	0	0
CPU	2	0.16	0	0	0	0	0	0
Total	6	0.96	4	0.8	5	0.97	5	0.97

It can be seen that the requestors GPU_{out} and LCD_{in} are interleaved across two memory channels to satisfy their latency requirements. Note that the rate allocated to each requestor is 0.8, i.e., 773.5 MB/s, which amounts to an over-allocated bandwidth of 273.5 MB/s. This relates to our conclusion in Section IV-A that increasing the degree of interleaving for a requestor may result in over-allocation of rate depending on its latency requirement. 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} also is 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. Since we know that GPU_{out} and LCD_{in} need a rate of 0.4 in each memory channel and GPU_{in} a rate of 0.51, the rate remaining in any of the single channels cannot satisfy the combined rate requirements of 0.82 by VE_{in} and IP_{out}. Hence, they are interleaved across two memory channels. CPU is not interleaved as its required rate can be satisfied with the rate available in a single channel.

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. Memory capacity requirements by the requestors also impacts the interleaving of requests across channels, which we did not include in our case-study for the ease of presentation.

VIII. 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 algorithm to map memory requestors to the memory channels and configure the memory controller, while minimizing resource utilization. We show that for a realistic use-case scenario consisting of 4 memory channels and up to 100 memory requestors, an optimization tool can find the optimal mapping and configuration in less than 15 minutes. Finally, we demonstrated the effectiveness of our work in a real use-case scenario.

ACKNOWLEDGMENT

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 COBRA, ARTEMIS 100202 RECOMP, PT FCT, and NL STW 10346 NEST.

REFERENCES

[1] P. Kollig *et al.*, "Heterogeneous Multi-Core Platform for Consumer Multimedia Applications," in *Proc. DATE*, 2009.
[2] P. van der Wolf *et al.*, "SoC Infrastructures for Predictable System Integration," in *Proc. DATE*, 2011.
[3] L. Steffens *et al.*, "Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach," *Proc. ECRTS*, 2008.
[4] M. Paolieri *et al.*, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, 2009.
[5] B. Akesson *et al.*, "Architectures and Modeling of Predictable Memory Controllers-for Improved System Integration," in *Proc. DATE*, 2011.
[6] J. Reineke *et al.*, "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation," in *Proc. CODES+ISSS*, 2011.
[7] D. Stiliadis *et al.*, "Latency-rate servers: A general model for analysis of traffic scheduling algorithms," *IEEE/ACM Trans. Netw.*, vol. 6, no. 5, 1998.
[8] *Wide I/O Single Data Rate Specification*, JESD229 ed., JEDEC Solid State Technology Association, 2012.
[9] M. D. Gomony *et al.*, "DRAM Selection and Configuration for Real-Time Mobile Systems," in *Proc. DATE*, 2012.
[10] E. Aho *et al.*, "A Case for Multi-channel Memories in Video Recording," in *Proc. DATE*, 2009.
[11] Z. Zhu *et al.*, "Fine-grain Priority Scheduling on Multi-channel Memory Systems," in *Proc. HPCA*, 2002.
[12] P. Casini, "SoC Architecture to Multichannel Memory Management Using Sonics IMT," White paper, 2008, sonics, inc.
[13] M. Awasthi *et al.*, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proc. PACT*, 2010.
[14] T. Zhang *et al.*, "A 3D SoC Design for H.264 Application with On-chip DRAM Stacking," in *Proc. 3DIC*, 2010.
[15] I. Loi *et al.*, "An Efficient Distributed Memory Interface for Many-core Platform with 3D Stacked DRAM," in *Proc. DATE*, 2010.
[16] Y. Ou *et al.*, "A Scalable Multi-channel Parallel NAND Flash Memory Controller Architecture," in *Proc. ChinaGrid*, 2011.
[17] C. Bouquet, "Optimal Multi-channel Memory Controller System," Patent number: 6643746, United States Patent and Trademark Office (USPTO), 2000.
[18] G. Zhang *et al.*, "Heterogeneous multi-channel: Fine-grained dram control for both system performance and power efficiency," in *Proc. DAC*, 2012.
[19] B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems*, 1st ed., ser. Embedded Systems. Springer, 2011.
[20] "IBM ILOG CPLEX Optimizer," http://www.ibm.com, IBM Corporation.
[21] A. Stevens, "QoS for High-Performance and Power-Efficient HD Multimedia," ARM White paper, http://wwww.arm.com, 2010.
[22] A. C. Bonatto *et al.*, "Multichannel SDRAM Controller Design for H.264/AVC Video Decoder," in *Proc. SPL*, 2011.