Architectures and Modeling of Predictable Memory Controllers for Improved System Integration

Benny Akesson, Kees Goossens
Eindhoven University of Technology
{k.b.akesson, k.g.w.goossens}@tue.nl

***Abstract*—Designing multi-processor systems-on-chips becomes increasingly complex, as more applications with real-time requirements execute in parallel. System resources, such as memories, are shared between applications to reduce cost, causing their timing behavior to become inter-dependent. Using conventional simulation-based verification, this requires all concurrently executing applications to be verified together, resulting in a rapidly increasing verification complexity. Predictable and composable systems have been proposed to address this problem. Predictable systems provide bounds on performance, enabling formal analysis to be used as an alternative to simulation. Composable systems isolate applications, enabling them to be verified independently.**

Predictable and composable systems are built from predictable and composable resources. This paper presents three general techniques to implement and model predictable and composable resources, and demonstrates their applicability in the context of a memory controller. The architecture of the memory controller is general and supports both SRAM and DDR2/DDR3 SDRAM and a wide range of arbiters, making it suitable for many predictable and composable systems. The modeling approach is based on a shared-resource abstraction that covers any combination of supported memory and arbiter and enables system-level performance analysis with a variety of well-known frameworks, such as network calculus or data-flow analysis.

***Index Terms*—predictability; composability; memory controller; memory patterns; real-time; SDRAM; arbitration; latency-rate servers**

I. Introduction

The complexity of Systems-on-Chip (SoC) is increasing, as a growing number of independent applications are integrated on a single chip. These applications are mapped on heterogeneous multi-processor platforms with distributed shared memory hierarchies that must strike a good balance between performance, cost, power consumption and flexibility [1]–[4]. The platforms exploit an increasing amount of application-level parallelism by enabling concurrent execution of more and more applications. This results in a large number of *use-cases*, which are different combinations of concurrently running applications [5]. Some applications have *real-time requirements*, such as a minimum throughput of video frames per second, or a maximum latency for processing those video frames. Real-time requirements are often verified by system-level simulation at the end of the design process.

To reduce cost, platform *resources*, such as processors, hardware accelerators, interconnect, and memories, are shared between resource *requestors*, being the processing elements to which applications are mapped. However, resource sharing causes *interference* between requestors, making the timing-behaviors of applications inter-dependent. This requires that all applications in a use-case are verified together, which results in three problems with respect to verification. 1) The number of use-cases *increases rapidly* with the number of applications, forcing industry to reduce coverage and verify only a subset of use-cases that have the toughest requirements [3], [6]. 2) The verification of a use-case cannot begin until all applications it comprises are available. 3) Use-case verification becomes a *circular process* that must be repeated if an application is added, removed, or modified [7]. Together these three problems contribute to making the integration and verification process a dominant part of SoC development in terms of both time and money [1], [7], [8].

We address the verification problem using two complexity-reducing concepts: *predictability* and *composability* [9]. Predictable systems *bound interference* between applications. This enables performance bounds to be provided, such as upper bounds on latency or lower bounds on throughput. Applications in predictable systems can hence be independently verified using formal performance analysis frameworks. The benefit of formal performance verification is that conservative performance guarantees can be provided for all possible combinations of initial states of resources and arbiters, all input stimuli, and all concurrently executing applications. The drawback is that formal approaches require performance models of the software, the hardware, and the mapping [10], [11], which are not always available.

Composability offers a complementary divide-and-conquer approach to verification. Applications in a composable system are completely isolated and cannot affect each other's temporal behaviors. This means that the *interference experienced by an application is independent of others*. Note that with these definitions predictability does not imply composability or the other way around [9]. Composable systems address the verification problem in the following four ways [10]: 1) Applications can be verified in isolation, resulting in a linear and non-circular verification process [7]. 2) Simulating only a single application and its required resources reduces simulation time compared to complete system simulations. 3) The verification process can be incremental and start as soon as the first application is available. 4) Intellectual property (IP) protection is improved, since the verification process no longer requires the IP of independent software vendors to be shared. These benefits reduce the complexity of simulation-based verification, making it a feasible option with a larger number of applications and inputs. Predictability and composability both solve important parts of the verification problem and provide a complete solution when combined.

978-3-9810801-7-9/DATE11/©2011 EDAA

Several predictable and/or composable systems have been proposed, such as Time-Triggered systems [7], Loosely Time-Triggered systems [12], METERG systems [13], Virtual Private Machines [14], PRET Machines [15], the MERASA platform [16], and our CompSoC [10] platform. Predictable and composable systems are built from predictable and composable resources. The CompSoC platform is a multi-processor system with predictable and composable processor tiles [17], memory controllers [18], network-on-chip [19], and operating system [20] with dynamic voltage and frequency scaling [21].

This paper presents *three general techniques* to model and implement predictable and composable shared resources. 1) *Combining resources and arbiters, each with predictable behaviors*, resulting in a predictable shared resource for any combination of resource and arbiter. 2) Turning the predictable shared resource into a composable shared resource by *always emulating worst-case interference* from other requestors, making their temporal behaviors independent of each other's actual behavior. This implementation of composability is complementary to conventional techniques based on static-scheduling and time-division multiplexing (TDM) and more suitable for certain resources. 3) Using a *shared-resource abstraction* that enables system-level performance analysis of the controller with several well-known frameworks, such as network calculus [22] and data-flow analysis [23], for any combination of supported resource and arbiter. The techniques are demonstrated in the context of the CompSoC memory controller, which supports both SRAM and DDR2/DDR3 SDRAM, and a wide range of arbiters. The memory controller architecture, the underlying techniques to achieve predictability and composability, and the modeling approach are all general and useful in other predictable and composable systems. Although both the memory controller architecture and modeling approach are general and support several memory types, we focus the discussion on SDRAM, since these memories have three important characteristics that make the implementation of predictability and composability challenging. 1) The latency of requests and the bandwidth offered by the memory are *highly variable* and depend on other requestors [24]. 2) Some memory requestors are *latency critical* and require low latency to reduce the number of stall cycles on the processor [4], [25]. 3) For cost reasons, SDRAM bandwidth is a *scarce resource* that must be efficiently utilized [4], [25], [26].

This paper is organized as follows. Section II reviews related work on predictable and composable SDRAM controllers. Section III then explains why SDRAM memories are challenging to use in real-time systems. The architecture of our predictable and composable memory controller is then presented in Section IV, followed by a discussion on our shared-resource abstraction in Section V. Lastly, the paper is concluded in Section VI.

II. Related work

Most SDRAM controllers are either statically or dynamically scheduled, depending on the type of systems they target. Statically scheduled controllers, such as [27], execute precomputed schedules of SDRAM commands that have been computed at design time. These controllers are predictable, since the latency and bandwidth provided to a requestor during a use-case can be bounded at design time by analyzing the schedule. These controllers are furthermore composable, since the schedule is executed in the same manner regardless of the behavior of the applications. However, the predictability and composability of these controllers comes at the expense of flexibility. The precomputed schedules limit the applicability to applications whose memory behavior can be exactly specified at design time, which is not the case for more dynamic input-dependent applications. Furthermore, many schedules have to be computed and stored, as the number of use-cases grows rapidly with the number of applications [5].

Dynamically scheduled memory controllers, on the other hand, schedule SDRAM commands at run-time based on available requests. These controllers target high efficiency and flexibility to fit in high-performance systems with dynamic applications. Several of these controllers feature sophisticated mechanisms to reduce latency or improve efficiency. Examples involve preference for requests that target open rows in the memory banks [28]–[32], or that fit with the current direction (read/write) of the data bus [30]–[34]. The problem with these controllers is that the interaction between all these mechanisms is complex, making the controllers unpredictable. A predictable dynamically scheduled controller is presented in [35]. However, this approach is limited to systems that are performance monotonic [13], meaning that local reductions in execution time cannot result in longer overall execution times. It is shown in [36] that this property does not hold for all systems. Furthermore, no hardware implementation is provided for this controller. Neither of the mentioned dynamically scheduled memory controllers is composable. Traditional approaches to dynamic composable resource sharing are based on TDM [10]. This approach is very inefficient for resources with highly variable latency, such as SDRAM, especially in presence of latency-critical requestors [9].

In contrast, this paper presents a hybrid memory controller that combines aspects of both statically and dynamically scheduled approaches. The controller implements the three techniques explained in Section I, resulting in a predictable and composable controller that is *more flexible* than traditional predictable designs and extends composable service to support *any combination of application, predictable resource, and predictable arbiter*. The memory controller architecture and the techniques to implement and model predictable and composable resources are general and useful in other predictable and composable systems, such as [7], [12]–[16].

III. SDRAM overview

SDRAM memories are challenging to use in systems with real-time requirements because of their internal architecture. An SDRAM memory comprises a number of banks, each containing a memory array with a matrix-like structure, consisting of rows and columns [37]. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row. As an example, a 512 megabit DDR2-800 [38] chip with a word width of 16 bits has 4 banks, each with 8192 rows containing 1024 word-sized elements.

The behavior of an SDRAM memory is determined by the sequence of *SDRAM commands* that are communicated from the memory controller to the memory device. These

commands tell the memory to *activate* (open) a particular row in the memory array, to *read* or *write* a burst to/from an open row, or to *precharge* (close) an open row and store its contents back into the memory array. There is also a *refresh* command that charges the capacitors of the memory elements to ensure that the contents of the memory array are retained. Scheduling SDRAM commands is not a trivial task, since there is a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are minimum delays between issuing particular SDRAM commands, such as two activates, or an activate and a read or a write.

The SDRAM architecture makes the execution time of requests highly variable for three reasons. 1) A request targeting an open row can be served immediately, while it otherwise first needs the current row to be closed and the required row to be opened. 2) The data bus is bi-directional and requires several cycles to switch from read to write and vice versa. 3) The memory must occasionally be refreshed before executing the next request. The impact of these factors may cause the execution time of an SDRAM burst to vary by an order of magnitude from a few clock cycles to a few tens of cycles.

IV. Memory controller architecture

This section presents our memory controller and discusses the techniques employed to implement predictability and composability. The architecture of the memory controller, shown in Figure 1, is divided into a *front-end* and a *back-end*. The front-end is independent of memory technology and contains components to implement predictable and composable resource sharing. The back-end interfaces with the actual memory device and makes it into a predictable resource. The back-end is hence different for different types of memories, such as SRAM and SDRAM, as indicated in Figure 1. The components in the architecture are discussed in more detail throughout this section. We begin in Section IV-A by explaining how to make an SDRAM behave like a predictable resource. Section IV-B then explains how we use this predictability to provide composable service.

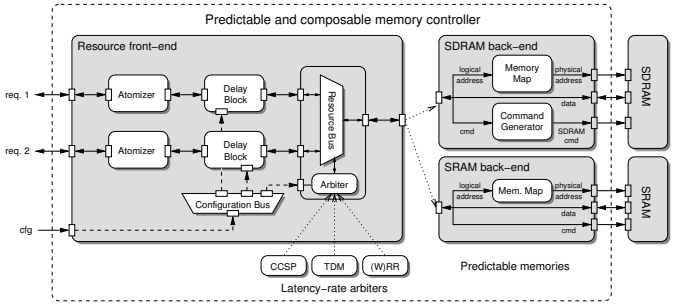

Fig. 1. Architecture template of predictable and composable memory controller, supporting two requestors.

A. Predictability

The general technique behind our approach to predictability, corresponding to the first technique mentioned in Section I, is based on *combining resources and arbiters, each with predictable behaviors*. In the case of a memory, we require useful bounds on the offered bandwidth and the time to serve a scheduled request, since these characterize the worst-case behavior of an unshared memory. We refer to a memory satisfying this requirement as a *predictable memory*. A zero-bus-turnaround SRAM is an example of a trivially predictable memory, since random access is provided with constant single-cycle latency. Any off-the-shelf back-end for such a memory is hence useful with our approach. We also require a *predictable arbiter*, where the number of interfering requests that can be scheduled before a particular request is bounded. Combining a predictable memory and a predictable arbiter allows the maximum time to schedule a particular request to be computed, as later discussed in Section V, thus taking the effects of sharing the memory into account. Our approach is hence based on combining *independent analyses* of the memory and the arbitration. The strength of this approach is that it lets us design a general memory controller architecture and analysis, providing predictable service for *any combination of predictable memory and predictable arbiter*. First, we explain how our SDRAM back-end makes an unshared SDRAM behave like a predictable memory. We then proceed by discussing how to share the predictable resource among multiple requestors.

1) Predictable SDRAM back-end: Our memory controller uses a hybrid approach to SDRAM command scheduling that combines elements of statically and dynamically scheduled SDRAM controllers in an attempt to get the best of both worlds. The hybrid concept is based on *predictable memory patterns*, which are precomputed sequences (sub-schedules) of SDRAM commands. These patterns are dynamically combined at run-time by the command generator in the SDRAM back-end, based on the incoming requests. The memory patterns exist in five flavors: 1) read pattern, 2) write pattern, 3) read/write switching pattern, 4) write/read switching pattern, and 5) refresh pattern. The patterns are *automatically generated* [18] at design time based on the timing constraints of the particular SDRAM device and the bandwidth and latency requirements of the requestors. All requestors employ the same set of patterns, although the concept can be generalized to use different sets per requestor.

The patterns are created such that multiple read or write patterns can be scheduled in sequence. However, a switching pattern is required between a read and a write pattern, and vice versa. The refresh pattern is scheduled periodically and can be followed by either a read or a write pattern without a preceding switching pattern. The mapping from requests to patterns is illustrated in Figure 2.

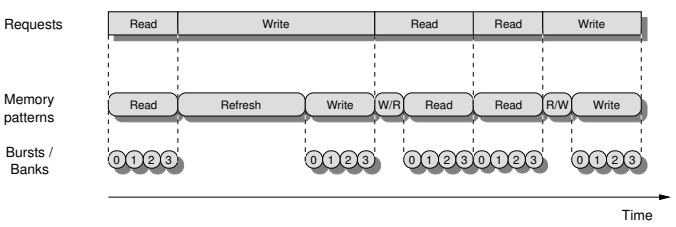

Fig. 2. Mapping from requests to patterns to SDRAM bursts.

The read and write patterns consist of a fixed number of SDRAM bursts, all targeting the same row in a bank. The bursts are issued to the different banks in sequence, since the data bus is shared between all banks to reduce the number of pins on the SDRAM interface. The fixed number of bursts is hence first sent to the first bank, then to the second, and so forth until all banks have been accessed, as shown in Figure 2. The reason to use bank-interleaving memory accesses in our controller is that they enable bank-level parallelism to be exploited by issuing activate and precharge commands to the banks during the intervals in which they are not transferring data. The read and write patterns are hence very efficient in terms of bandwidth, since it is possible to hide a significant part of the latency incurred by activating and precharging rows. This limits the overhead cycles incurred by always precharging a bank immediately after it has been accessed, which is known as a closed-page policy. We implement this policy, as it effectively removes the dependency on rows opened by earlier requests by returning the memory to a neutral state after every access. Removing this dependency between requests is a *key element* in our approach, since it *reduces the variation in the offered bandwidth and latency*, enabling tighter bounds on bandwidth and latency to be derived.

The main benefit of memory patterns is that they abstract SDRAM command scheduling to a higher level. This implies a reduction of state and constraints that have to be considered, making our approach easier to analyze than dynamically scheduled controllers. Memory patterns allow a lower bound on the offered bandwidth and the time to serve a request to be determined, since we know the length of each pattern, how much data they transfer, and how they are dynamically combined in the worst case [24]. The use of memory patterns hence gives our approach the predictability of statically scheduled memory controllers. It also has some properties of dynamically scheduled controllers, such as the ability to dynamically choose between read and write requests, and the use of run-time arbitration. This increases flexibility over traditional predictable designs.

Although bank-interleaving memory accesses allow us to bound the offered bandwidth, they come with three drawbacks. The first drawback is that continuously activating and precharging the banks increases power consumption compared to if only a single bank is used [26], [39]. The second drawback is that the memory is accessed with large granularity and hence requires large requests to be efficient [24]. An efficient access requires at least one SDRAM burst to every bank. A typical burst size for SDRAM is eight words and the number of banks is either four or eight. The minimum efficient request size for a 16-bit DDR2-800 memory with four banks is hence 64 B. For requests of this size, our solution guarantees a bandwidth of at least 1070 MB/s, corresponding to 66.8% of the peak bandwidth. However, if all requests are 32 B, only 33.4% the peak bandwidth can be guaranteed, since half of the data in the patterns is discarded. Efficiency bounds for more DDR2/DDR3 memories and request sizes are presented in [24]. It is also shown that these bounds are tight. The final drawback is that working with large requests in a non-preemptive manner also results in longer latencies [24]. Choosing bank-interleaving memory accesses hence prioritizes efficiency over latency and power.

Our approach is implemented as an SDRAM back-end, as shown in Figure 1. The back-end accepts a scheduled request, and translates the logical address into a physical address (bank, row, and column) using a bank-interleaved memory map. A command generator then issues the appropriate memory patterns and sends the SDRAM commands to the memory device. The implementation of the back-end is very light weight and has a small area foot print [40].

2) Predictable arbitration: After the previous section, we assume that we have a predictable memory, such as an SRAM or our SDRAM back-end based on predictable memory patterns, where useful bounds are known on both the offered bandwidth and the time to serve a request. Next, we consider the effects of sharing the predictable memory between multiple requestors using a predictable arbiter. There are many predictable arbiters described in literature, such as TDM and round robin (RR). However, most of these arbiters are unable to provide low latency to critical requestors, making them unsuitable for memory controllers. This problem is addressed by priority-based arbitration, although conventional static-priority scheduling is not starvation-free and cannot be used to build predictable or composable systems.

To address the issue of latency-critical requestors, we have developed a Credit-Controlled Static-Priority (CCSP) arbiter [41]. CCSP has been specially developed to arbitrate access to highly loaded SoC resources and has a small and fast hardware implementation [42]. The CCSP arbiter consists of a rate regulator and a static-priority scheduler. The rate regulator protects requestors by enforcing an upper bound on the provided service, according to an *allocated budget*, which is determined at design time. The static-priority scheduler schedules the highest priority requestor that is within its budget. The combination of rate regulator and static-priority scheduler makes the arbiter predictable, while still being able to satisfy the requirements of latency-critical requestors. A rate regulator creates a separation of concerns and makes it possible to bound the latency of a requestor in a static-priority scheduler without relying on the cooperation of higher priority requestors. However, to be completely robust, we also need to be independent of the sizes of scheduled requests to prevent a malfunctioning requestor from preventing access from others by issuing very large requests. We solve this problem using preemptive service, which is enabled by the *atomizer* [10]. The atomizer splits requests into *atomic service units*, referred to as atoms, which are served by the memory in a known bounded time. Large requests are hence chopped up in smaller pieces, ensuring that other requestors can access the resources within a bounded time. The size of the atoms are fixed and determined at design time. For an SRAM, the natural service unit is a single word, while it corresponds to the granularity of a read and write patterns for our pattern-based SDRAM controller. A benefit of using fixed-sized requests in the memory controller is that it simplifies the components in the architecture, resulting in a faster implementation. The advantage of adding the atomizer as a separate hardware block in front of the arbiter is that it effectively makes all predictable

arbiters preemptive on the granularity of atoms. This qualifies any existing predictable arbiter for use with our approach.

The combination of predictable memory and arbiter results in a predictable shared resource that enables formal performance analysis of applications, assuming the existence of a formal application model. However, some applications have behaviors that are too complex to accurately express in formal models, and have to be verified by simulation. To reduce the verification effort of these applications, our memory controller also provides composable service, as discussed next.

B. Composability

Composability implies that applications are temporally isolated [7]. The main problem with non-composable resources and arbitration is that they cause the time to serve a request to depend on other requestors. This might cause an application that satisfies its real-time requirements in isolation to miss deadlines after integration, due to contention for shared resources. This is addressed by the second technique presented in Section I. The key idea behind this technique is to make the system composable by delaying all signals sent to the requestor to *emulate maximum interference from other requestors* [43]. A requestor hence always receives the same worst-case service no matter what other requestors are doing, making their temporal behaviors independent. This particular approach to composability requires predictability, since it is not possible to delay signals to the worst case unless it is known and bounded. The major advantage of this approach is that is extends the use of composability beyond resources and arbiters that are inherently composable. Our approach is hence not limited only to zero-bus-turnaround SRAM controllers, but can capture the behavior of any predictable resource, such as our proposed SDRAM back-end based on predictable memory patterns. It furthermore supports any predictable arbiter, as opposed to being limited to TDM or static-scheduling, enabling service differentiation that increases the possibility of satisfying a given set of requestor requirements. A key benefit is that the approach does not have *any restrictions* on the applications. This ensures that all applications that cannot be formally verified can be verified independently by simulation with linear verification complexity.

Our approach to composability is implemented by the *delay block* [43], shown in Figure 1. The purpose of the delay block is to emulate worst-case interference from other requestors to provide a composable interface towards the atomizer. This makes the interface of the entire front-end composable, since the atomizer is not shared. The delay block is composable if all signals to the atomizer exhibit composable behavior, which implies that both the response data and the flow-control signals must emulate maximum interference. This is achieved by computing the latest possible time this information can be sent, which is possible for any predictable shared memory.

To provide composable service, a delay block needs information about the maximum interference that can be experienced by its requestor. This information is typically different for all requestors and changes between use-cases. A *configuration bus* is hence present in the architecture, as shown in Figure 1, that allows the worst-case interference to be (re-)programmed at run-time.

V. Shared-resource abstraction

To simplify system-level analysis, we use a shared-resource abstraction that captures the temporal behavior of many different resources and arbiter types. This corresponds to the third technique mentioned in Section I. We have chosen latency-rate servers [44] ($\mathcal{LR}$) as the shared-resource abstraction for our system. In essence, a $\mathcal{LR}$ server guarantees a requestor a minimum allocated bandwidth, ρ', after a maximum service latency, Θ, as shown in Figure 3. A $\mathcal{LR}$ server hence provides a lower bound on the amount of data that can be transferred during an interval, making it an abstraction of predictable service.

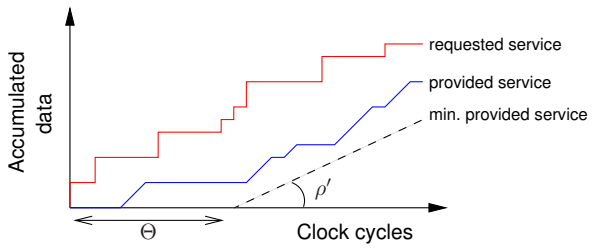

Fig. 3. The $\mathcal{LR}$ server shared-resource abstraction.

The $\mathcal{LR}$ server model is very general and applies to a wide range of shared resources, which is required to capture a complete system in a single model. The applicability of the $\mathcal{LR}$ model with respect to resources is very good, since it can be used with any predictable resource. Example uses of the model in literature involve modeling communication channels in busses [45] and networks-on-chips [46]. The model also supports a large number of arbiters. In theory, all predictable arbiters belong to the class of $\mathcal{LR}$ servers, since they guarantee that a request is scheduled within a maximum latency, making them starvation free. However, no arbiter truly belongs to the class until the service latency has been derived, which is difficult for complex arbiters. The arbiters that belong to the class of $\mathcal{LR}$ servers are hence a subset of the set of predictable arbiters. It is shown in [44] that many well-known arbiters, such as Weighted Round-Robin [47] (WRR), Deficit Round-Robin [48], and several varieties of Fair Queuing [49] are $\mathcal{LR}$ arbiters. Other examples of $\mathcal{LR}$ arbiters are TDM [45] and our priority-based CCSP arbiter [18].

Each requestor in the memory controller has their own independent $\mathcal{LR}$ service guarantee, enabling independent performance analysis using formal methods. The bandwidth allocated to a requestor is easily determined by comparing the fraction of allocated service to the total available bandwidth, which is bounded by definition for predictable resources, such as our SDRAM back-end. In the general case, the service latency of a requestor is computed by multiplying the service latency of the chosen $\mathcal{LR}$ arbiter, corresponding to the maximum number of interfering requests, with the maximum time to serve a scheduled request, also known for any predictable resource. However, this results in very pessimistic service latencies for SDRAM memories where latencies are highly variable. We show in [24] how to reduce this pessimism by exploiting that only a small fraction of requests are affected

by refresh and that a relatively long write/read switch has to be followed by a shorter read/write switch.

A key benefit of the $\mathcal{LR}$ server abstraction is that it supports formal performance analysis using several well-known frameworks, such as network calculus [22] and data-flow analysis [23]. Both of these frameworks provide analysis tools that enable formal verification of real-time requirements and buffer sizing in systems comprising multiple resources shared by arbiters in the class of $\mathcal{LR}$ servers. The mathematical formalism of $\mathcal{LR}$ servers is designed to fit with the concept of service curves that are used to characterize applications in verification approaches based on network calculus, such as [25], [45], [50]. It also fits with data-flow analysis by using the data-flow component proposed in [51] that models the behavior of a $\mathcal{LR}$ server. This component enables the shared resource to be included in a data-flow graph that represents both the task graph of the application and the platform resources it uses in a single framework, as done in [46], [52].

VI. Conclusions

The verification complexity of real-time systems-on-chip is increasing. Predictable and composable systems have been proposed to address this problem, since they enable formal verification of real-time requirements and independent application verification by simulation, respectively.

This paper presents three general techniques to implement and model predictable and composable shared resources. 1) Combining resources and arbiters, each with predictable behaviors, resulting in a predictable shared resource for any combination of resource and arbiter. 2) Turning the predictable shared resource into a composable shared resource by always emulating worst-case interference from other applications, thus decoupling their actual behaviors. This approach extends composable service to support any combination of application, predictable resource, and predictable arbiter. 3) Using a shared-resource abstraction that enables system-level performance analysis of the controller with several well-known frameworks, such as network calculus or data-flow analysis. The techniques are demonstrated in the context of a general predictable and composable memory controller architecture, supporting both SRAM and DDR2/DDR3 SDRAM and a wide range of arbiters. The memory controller architecture as well as the techniques to implement and model predictability and composability are general and useful in many other predictable and composable systems.

References

[1] "International Technology Roadmap for Semiconductors (ITRS)," 2009.

[2] C. van Berkel, "Multi-core for Mobile Phones," in *Proc. DATE*, 2009.

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

[4] P. Kollig *et al.*, "Heterogeneous Multi-Core Platform for Consumer Multimedia Applications," in *Proc. DATE*, 2009.

[5] A. Hansson *et al.*, "Undisrupted Quality-Of-Service during Reconfiguration of Multiple Applications in Networks on Chip," in *Proc. DATE*, 2007.

[6] L. Steffens *et al.*, "Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach," *Proc. ECRTS*, 2008.

[7] H. Kopetz *et al.*, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, 2003.

[8] R. Saleh *et al.*, "System-on-chip: Reuse and integration," *Proc. IEEE*, vol. 94, no. 6, 2006.

[9] B. Akesson *et al.*, "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*. Springer, 2010, ch. 2.

[10] A. Hansson *et al.*, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, no. 1, 2009.

[11] E. A. Lee, "Absolutely positively on time: what would it take?" *IEEE Trans. Comput.*, vol. 38, no. 7, 2005.

[12] A. Benveniste, "Loosely time-triggered architectures for cyber-physical systems," in *Proc. DATE*, 2010.

[13] J. Lee *et al.*, "METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors," in *Proc. RTAS*, 2006.

[14] K. J. Nesbit *et al.*, "Multicore resource management," *IEEE Micro*, vol. 28, no. 3, 2008.

[15] S. Edwards *et al.*, "The Case for the Precision Timed (PRET) Machine," in *Proc. DAC*, 2007.

[16] T. Ungerer *et al.*, "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability," *IEEE Micro*, vol. 30, no. 5, 2010.

[17] A. Molnos *et al.*, "A Composable, Energy-Managed, Real-Time MPSOC Platform," in *Proc. OPTIM*, 2010.

[18] B. Akesson, "Predictable and Composable System-on-Chip Memory Controllers," Ph.D. dissertation, Eindhoven University of Technology, 2010.

[19] K. Goossens *et al.*, "The aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proc. DAC*, 2010.

[20] A. Hansson *et al.*, "Design and Implementation of an Operating System for Composable Processor Sharing," *MICPRO*, 2011, Elsevier.

[21] K. Goossens *et al.*, "Composable dynamic voltage and frequency scaling and power management for dataflow applications," in *Proc. DSD*, 2010.

[22] R. Cruz, "A calculus for network delay. I. Network elements in isolation," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, 1991.

[23] S. Sriram *et al.*, *Embedded multiprocessors: Scheduling and synchronization*. CRC, 2000.

[24] B. Akesson *et al.*, "Classification and Analysis of Predictable Memory Patterns," in *Proc. RTCSA*, 2010.

[25] P. van der Wolf *et al.*, "SoC Infrastructures for Predictable System Integration," in *Proc. DATE*, 2011.

[26] D. Dunning *et al.*, "Tera-Scale Memory Challenges and Solutions," *Intel Technology Journal*, vol. 13, no. 4, 2009.

[27] S. Bayliss *et al.*, "Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search," in *Proc. FPT*, 2009.

[28] O. Mutlu *et al.*, "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers," *IEEE Micro*, vol. 29, no. 1, 2009.

[29] J. Shao *et al.*, "A burst scheduling access reordering mechanism," in *Proc. HPCA*, 2007.

[30] C. Macian *et al.*, "Beyond performance: Secure and fair memory management for multiple systems on a chip," in *Proc. FPT*, 2003.

[31] W.-D. Weber, *Efficient Shared DRAM Subsystems for SOCs*, Sonics, Inc, 2001.

[32] K. Lee *et al.*, "An efficient quality-aware memory controller for multimedia platform SoC," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 5, 2005.

[33] S. Heithecker *et al.*, "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements," in *Proc. DAC*, 2005.

[34] A. Burchard *et al.*, "A real-time streaming memory controller," in *Proc. DATE*, 2005.

[35] M. Paolieri *et al.*, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, 2009.

[36] T. Lundqvist *et al.*, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. RTSS*, 1999.

[37] B. Jacob *et al.*, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2007.

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

[39] "Calculating Memory System Power for DDR2," Micron Technology Inc., Tech. Rep., 2005, TN-47-04.

[40] B. Akesson *et al.*, "Predator: a predictable SDRAM memory controller," in *Proc. CODES+ISSS*, 2007.

[41] ——, "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, 2008.

[42] ——, "Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, 2009.

[43] ——, "Composable Resource Sharing Based on Latency-Rate Servers," in *Proc. DSD*, 2009.

[44] 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.

[45] J. Vink *et al.*, "Performance analysis of SoC architectures based on latency-rate servers," *Proc. DATE*, 2008.

[46] A. Hansson *et al.*, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET CDT*, 2009.

[47] M. Katevenis *et al.*, "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip," *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, 1991.

[48] M. Shreedhar *et al.*, "Efficient fair queueing using deficit round robin," in *Proc. SIGCOMM*, 1995.

[49] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proc. IEEE*, vol. 83, no. 10, 1995.

[50] T. Henriksson *et al.*, "Network calculus applied to verification of memory access performance in SoCs," in *Proc. ESTIMEDIA*, 2007.

[51] M. H. Wiggers *et al.*, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proc. SCOPES*, 2007.

[52] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. DAC*, 2007.