Copyright Notice

Institute of Communication Networks and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47, D-70569 Stuttgart, Germany
Phone: ++49-711-685-68026, Fax: ++49-711-685-67983
Email: mail@ikr.uni-stuttgart.de, http://www.ikr.uni-stuttgart.de

Beyond Performance: Secure and Fair Memory Management for Multiple Systems on a Chip

Carlos Macián
Institute of Communication Networks
and Computer Engineering
University of Stuttgart
Pfaffenwaldring, 47
D-70569 Stuttgart, Germany
Email: macian@ikr.uni-stuttgart.de

Sarang Dharmapurikar
Applied Research Lab
Washington University
One Brookings Drive
St. Louis, MO 63130
Email: sarang@arl.wustl.edu

John Lockwood
Applied Research Lab
Washington University
One Brookings Drive
St. Louis, MO 63130
Email: lockwood@arl.wustl.edu

***Abstract*— Developments in VLSI technologies create the possibility of hosting several independent (sub) systems in a single chip. There is a need to share a number of resources, especially off-chip resources, which creates new constraints in the design process. Although performance is still a key constraint, sharing implies that secure access to those resources and QoS guarantees are needed. In this paper, an architecture is presented that achieves the goals listed above. The Embedded Hardware Manager acts as a middleware between the applications and the resources, taking the role of resource manager and security agent. The results show that it can prevent resource misuse and undue information peeking or even altering while maintaining individual QoS guarantees. At the same time, high performance is still achieved.**

I. Introduction

In recent years, the evolution in chip size, on-chip memory and gate density opened the door to the integration of whole systems on a single chip (SoC). At present, the sustained development in VLSI technology allows to go one step further: The advent of Multiple-Systems-on-a-Chip (MSoC). State-of-the-art chips can easily support several embedded microprocessors and a number of specialized hardware modules concurrently. And yet, new constraints have to be taken into account in this environment: The number of I/O pins grows at a slower pace than gate density, provoking a mismatch in I/O bandwidth and processing power. Further, multiple systems (or subsystems) will have to share those pins. An ef£cient way of sharing access to external resources is needed.

In this context, the way in which interfaces and the access to off-chip resources are managed becomes critical for the successful introduction of MSoCs. The emphasis so far has been mainly in £nding ways to extract the maximum possible performance (both in terms of bandwidth and delay) out of those resources. While performance is clearly a must, it is not suf£cient. When several independent systems are sharing a chip, two other criteria become critical: Security and QoS. As with multitasking operating systems on a CPU, it is unacceptable that a malicious (or simply misbehaving) component could block the memory interface for the entire chip, or that it could ¤ood the communication channels with data, that ignores the SoC arbitration mechanism. Or even that it would try to access reserved regions in off-chip memory that were granted to other systems. QoS is also paramount: Different systems will have different requirements in terms of delay sensitivity, performance, bandwidth, etc. Classical approaches tend to share those resources fairly among competing systems. That might not be enough in this new scenario, in which a more complex scheduling may be needed to acknowledge the different requirements of the systems.

In this paper, an architecture realizing those goals is presented: It manages access to off-chip resources in a secure way and enforces a complex QoS policy, while still providing good performance, which we believe to be mandatory for future MSoCs. The focus of this work is on access to off-chip DRAM, but our architecture is completely general (see Section III). The rest of the paper is organized as follows: In Section II related work is reviewed. Section III describes the architecture and its main elements. In Section IV a set of tests and results are presented and Section V concludes the paper.

II. Related Work

The issue of memory management has been widely studied in the literature, but generally only in the context of throughput optimization. Security and QoS issues in shared access have often been neglected. McKee et al. have presented a general classi£cation of techniques for throughput maximization through request ordering in [1]. Different page interleaving schemes have been implemented for reducing the cache con¤icts and cache misses [2] [3]. A comprehensive set of algorithms for memory request reordering has been presented by Rixner et al. [4] and McKee [5]. Some of the other general purpose SDRAM controllers have been discussed in [6]. Other studies have tried to explore more effective ways to organize data into memory [7] or memory itself [8]. Although these works cover most general techniques to minimize access latency, no special attention has been given to the problem of QoS or multi-module memory requests.

In [9] the issue of memory access sharing is considered, but only in the context of on-chip RAM. Furthermore, only a very basic scheduling scheme, based on £xed priorities, is used.

Commercial products, like the QoS-aware memory controller from Denali Software Inc., are available [10]. However, since the architectures are proprietary, underlying algorithms are unknown. Better documented is the work by Sonics Inc. [11]. Their goals and approach resemble the design presented here, but no details about their scheduler are disclosed. Furthermore, they limit the possible QoS strategies to three (minimum delay, minimum guaranteed bandwidth and best effort), while our design offers additional ¤exibility and per-¤ow guarantees, not per-class. The data available for memory usage seems to con£rm that the work presented here clearly achieves higher ef£ciency.

III. EHM Architecture

The Embedded Hardware Manager (EHM) regulates the access to, and sharing of, off-chip resources in a MSoC. Figure 1 presents the basic architecture.

Every system or application present in the chip has access to the off-chip resources exclusively through the EHM. In this way, bypassing the security and scheduling mechanisms is prevented. As an interface between the applications and the EHM a fully compliant version of the Open Core Protocol (OCP) interface [12] has been chosen. The advantages of that interface were many: It is an open standard, thus not tying us to any vendor-speci£c solution. Furthermore, its characteristics are well adapted to the needs of a system on chip: It is an extremely ¤exible interface thanks to its many optional signals, which allow to adapt to a vast range of system behaviors. It is general-purpose, thus allowing the interconnection of a wide range of applications (microprocessors, memory controllers and peripherals). It allows bursts of different sizes and is a point-to-point interface, thus eliminating any possibility of interference in accessing the EHM among the different applications on the chip. A detailed description of the OCP speci£cation can be found in [12]. Since the EHM design is a general, platform- and technology-independent resource manager, it can also support other standard, non-proprietary interfaces, that could further ease the integration of third-party products, like the Wishbone [13] and AMBA [14] interfaces.

Every application thus multiplexes access to all shared resources on the OCP interface. How that multiplexing is done is application speci£c.

The Embedded Hardware Manager is composed of a set of Service Managers (Svc Mgr), one per application. Every Svc Mgr controls access to all shared resources (memory, I/O, etc) for one application. To that end, it is divided in a set of Resource Usage Managers (RUMs), one per shared resource. Every RUM implements the QoS and security policy for its application and shared resource, as will be described in Section III-A for the DRAM case. Obviously, an application's QoS and security requirements vary among resources: E.g., a computing-intensive application like image processing might need a lot of memory bandwidth and less I/O capacity, while encryption barely uses memory access but requires packet processing at wire speed, thus intensively using I/O channels. Furthermore, the capabilities and access patterns of every resource are also different: Single-access (as in SRAM) vs bursts (as in DRAM), single burst (as in memory access) vs multiple bursts (packet transfer), etc. Every RUM is therefore optimized for its speci£c resource. The emphasis of this paper being on memory management, the role of the SDRAM RUM will be explained in more detail.

RUMs control the transfer of requests to the corresponding resource, responses being handled by the General Responder. Requests are locally acknowledged by the OCP slave, in order not to block the OCP interface, which otherwise would have to wait for the response (e.g. data coming from a read request to SDRAM) before being freed for subsequent transactions. This provides a certain multiplexing gain in interface usage. Whenever a response is ready to be transfered to the application, it is sent to the General Responder, which acts as a multiplexer, serializing responses coming from different resources, potentially simultaneously. Although the present implementation serves every resource in a round-robin fashion, the General Responder could assign priorities to them, in case that certain requests had more stringent delay requirements.

Access requests are passed to the Resource Controller, which polls every Svc Mgr following a certain strategy. For the SDRAM Controller, two different strategies have been implemented to give access to memory. The £rst and simplest is a round-robin scheduler, that grants access to the bus to every Svc Mgr in turn. Our second implementation tries to optimize memory usage by dividing time in so-called *epochs*. Requests being served in an epoch will be rearranged to better utilize memory and minimize the overhead associated with DRAM management.

The next sections describe the modules in more detail.

A. SDRAM RUM

The SDRAM Resource Usage Manager (RUM) regulates access to the data interface according to the security and QoS policies chosen. For that, it performs two functions: It controls resource usage and also checks the validity of memory addresses being accessed. Figure 2 presents the basic architecture.

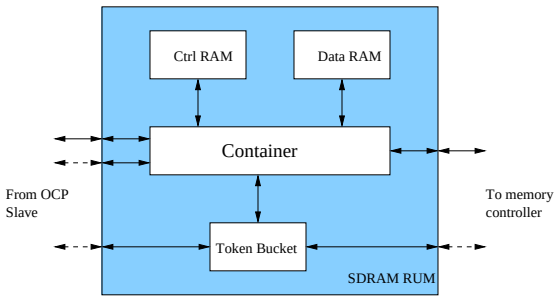

Fig. 2. SDRAM RUM Architecture

Since access to memory will be granted according to a scheduler, it can not be guaranteed that a request will be answered immediately. Hence, some form of intermediate

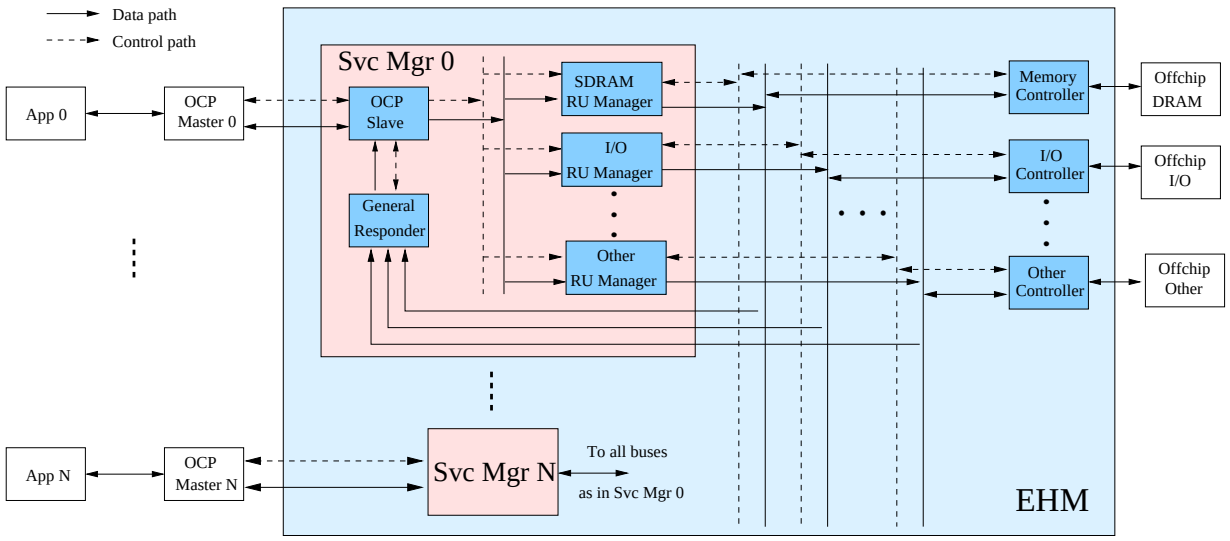

Fig. 1. Embedded Hardware Manager Architecture

buffering is needed. The SDRAM Resource Usage Manager performs that function. It can be seen as a queue storing the write and read requests, plus the corresponding data in the case of a write.

To control resource usage, a scheduler has been used that represents a good compromise between ease of implementation, performance and effectiveness. Especially important is that the scheduler will not allow large jitters in the access to the bus. In packet schedulers, for example, the time unit is large: It is the time needed to send a whole packet to the network. But in memory access, the time unit is much smaller, since it is only the time needed to send a data burst, which is typically at least an order of magnitude shorter than a packet. Hence, delaying a burst the equivalent of a packet transmission time might be unacceptable for delay-sensitive applications. That eliminates many current schedulers used in networking. To set an upper bound in delay and jitter and yet allow the burstiness typical of memory access, a combination of token buckets are used. There is a Token Bucket for every application. Its parameters are chosen to re¤ect the bandwidth and burst size accorded to that application and thus represent our mechanism to grant different QoS to different applications, according to their needs. Nevertheless, the maximum time that an application can use the bus is limited by the maximum burst size. Once exhausted, the Memory Controller grants access to the next application with a pending request. In this way, the maximum delay is known and bounded to (N-1)*MBS, where N is the number of applications and MBS represents the maximum time allowed to transmit the maximum burst size on the OCP.

The Container stores the requests and corresponding data in a small set of on-chip RAM. Access to memory is granted to these requests in a £rst-come-£rst-served basis. Storing requests is independent from actually delivering them to the Memory Controller. That process is controlled by the Token Bucket. Once a request has been made to the Memory Controller, the RUM must wait for a grant before the data can actually be delivered. Depending on the version of the Memory Controller, requests are served either in a round-robin fashion or according to a performance-maximizing strategy, as described in the next section.

Nevertheless, a malicious application might request bus usage to transmit a shorter burst and once granted, try to use up the maximum allowed time. To prevent this, the RUM keeps track of actual usage and discounts tokens accordingly. Once the limit has been reached, the transaction is aborted if it was not £nished by then. In this way, bus misuse is prevented.

The second task of the RUM is to map requested memory addresses to the address space granted to the application. For that purpose, a virtual memory scheme is used. Only the low bits of the address bus are taken into account by the RUM. It then pre£xes the high order bits according to a virtual address table and compares the resulting address with the last valid address for that application. If the requested position falls out of bounds, the request is dropped. In this way, peeking into memory or even altering its content is prevented.

The address range allocated to an application, together with the required QoS parameters (in the case of the SDRAM RUM, basically memory bandwidth), are calculated and agreed upon in the design phase, between the application developer and the memory management developer. If programmable hardware is used (FPGAs), it would be possible to dynamically change these values at compile time.

The implemented strategy is certainly not the only one possible and we are currently exploring alternatives. We nevertheless believe this choice to provide satisfactory results while being comparatively simple to implement.

B. Memory Controller Architecture

The Memory Controller depicted in £gure 4 is designed to maximize memory usage within an epoch. We de£ne an epoch as the time needed to serve all active requests from the SDRAM RUMs at the moment of polling. The controller will then rearrange those requests to minimize the overhead associated with DRAM operation, as described shortly, and serves them accordingly. Once £nished, a new epoch begins with a new polling phase.

DRAM ef£ciency is highly dependent on the access pattern. In particular the throughput depends on the burst length of the access, the address being accessed and the type of access (read/write). DRAM is partitioned in multiple banks to enable parallel access to the memory locations. Within each bank it is arranged in rows (pages) and columns. In case of successive memory requests on the DRAM, typically, there is very little delay involved between the two memory transactions if the second memory request operates on the same page as the previous request (page hit). There is a larger penalty if the second request operates on the same bank but different row than the one activated by the previous request (page miss). The penalty is smaller if the second memory request incurs a bank miss. In order to improve the ef£ciency of DRAM, some optimizations are required which can avoid the page miss, bank-miss and maximize the page-hits. The arbiter (memory controller) needs to permute the memory requests in such a way that it reduces the number of con¤icts and improves the throughput.

The memory controller developed for this project uses Micron SDRAM memory MT8LSDT864 [15] However the design principles discussed here apply to any generic SDRAM.

Figure 3 shows how two successive memory operations can be executed in different conditions for the SDRAM used. A simpli£ed representation of the timing diagram is shown where each box is equivalent to a clock cycle. Only the command and data bus are shown in the £gure, the upper row showing the commands.

The mnemonics and the color code for the commands are also illustrated in the £gure. The idle time in the two data transactions on the data bus is the reason for a loss in the SDRAM ef£ciency. There is a delay between two data transactions whenever there is a turn around from a read to a write operation or vice-versa. The penalty is larger when the following memory access incurs a page miss. All the transactions ending with SM in the £gure are page miss. The penalty is maximum when there is write to read turn around and a page miss too. In order to improve the ef£ciency of the DRAM, the page hits have to be favored and turn arounds and page miss have to be avoided.

The SDRAM controller is split up in a Request Selector and a Request Executer. The Request Selector logic takes the pending requests(one from each module) and selects a request among them which, if executed at the appropriate time, maximizes the SDRAM data bus utilization. It makes use of a brute force technique to £nd the best among the

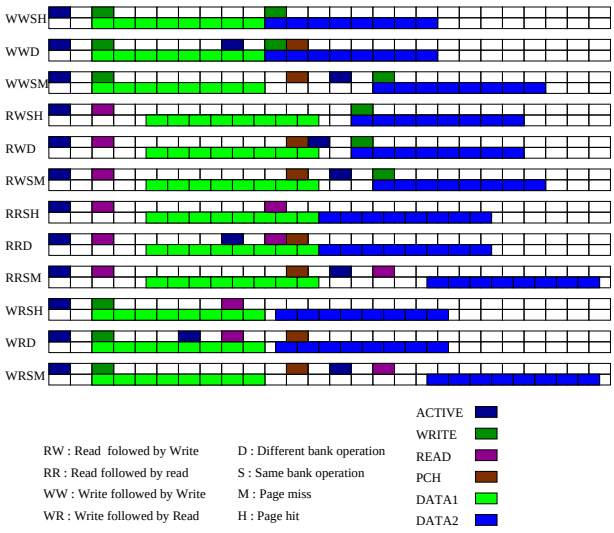

Fig. 3. Timing successive read and write operations in different conditions

pending requests. It compares the pending requests supplied to it against the current memory operation and decides which of the transactions shown in the £gure 3 the pending request corresponds to. The address of the pending request and the operation type (read or write) are compared against the address and the operation type of the current access. The result of the comparison indicates the transaction type that a memory request represents. The request which corresponds to the transaction with the least penalty is chosen and passed to the Request Executer.

The Request Executer logic executes the selected request at the appropriate time. It issues commands to the SDRAM at appropriate clock cycles in a state-machine fashion. Since the basic scheme allows two requests to have overlapping operations (termination of one request and the initiation of the next request) two state machines are required to control the SDRAM interface. If one state machine is busy executing commands for a request then the new request is loaded into the other state machine. If both state machines are busy, then the Request Executer refuses to accept any new request and the scheduling operation has to be repeated.

It is the epoch concept that ensures throughput optimization without compromising QoS. The order in which memory requests are enqueued depends only on the Token Buckets. This ensures that every application will get its accorded bandwidth. At the cost of some extra jitter, the memory controller rearranges the requests that the Token Buckets have enqueued to achieve better overall throughput. Since the request permutation occurs only at the beginning of every epoch, the additional delay that a request can suffer is bounded.

C. Architectural Properties

In summary, the architecture controls resource usage and provides protection of the system from misuse by (sub) modules. It ef£ciently uses the OCP interface, since bus requests can be placed while data operations are in progress,

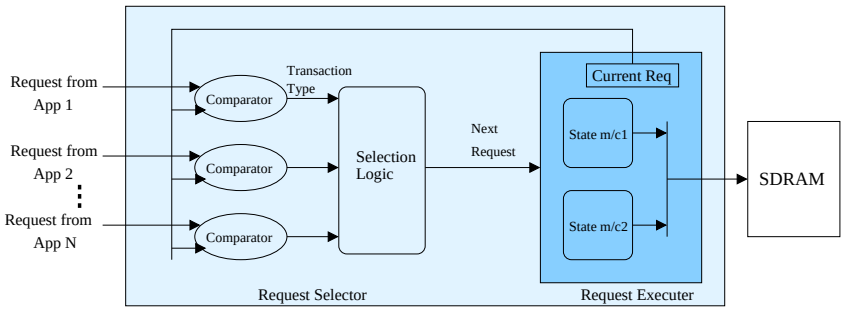

Fig. 4. Block diagram of the memory controller

hence making good use of request multiplexing. By using a general-purpose interface the details of the resource being accessed are hidden from the application. This simpli£es application design and increases reusability in the face of changing components. It also allows the same interface and overall architecture to manage access to different kinds of resources, such as communication channels, off-chip buses and CPU. The abstraction represented by the interface is nevertheless not so high so as to obscure or mislead the functioning of the resources. It is important to note that an ef£cient use of the hardware resources presupposes a detailed knowledge and control of their operation. Excessive abstraction usually implies a loss of ef£ciency and performance in exchange for easier handling. This architecture also delivers good performance and controlled, bounded delay, which is important for most applications, that are not completely delay-insensitive. Last but not least, the proposed design scales well with the number of applications. In fact, it would be very easy to automate the instantiation of Svc Mgrs depending on the number of applications present. By changing a small set of parameters, the required number of Svc Mgrs can be automatically instantiated. It is in fact the performance of the OCP and the memory which set a limit on the usefulness of sharing resources.

IV. Results

The design was developed in VHDL and synthesized for the Field-programmable Port Extender (FPX) platform [16]. The FPX is an open platform that augments a network with re-programmable hardware. It enables new data-processing hardware to be rapidly developed, prototyped, and deployed over the Internet. Extensive simulations (several million memory accesses in all cases) were performed, changing the number of active applications as well as their bandwidth requirements. The goal was to measure the ef£ciency in memory usage as well as the enforcement of the bandwidth requirements. A set of simple greedy applications were run, which constantly and inde£nitely try to access memory, alternately to write and read. The alternation of writes and reads represents a very unfavorable pattern, since after every transfer a memory turnaround is needed. This makes the results very conservative and thus safe estimates. When several applications are active simultaneously, the described effect is not so critical, since the memory controller can rearrange the requests to better suit memory needs, although only within an epoch. The results of our simulations are summarized in Figure 5 as well as in Tables I, II, III and IV. In every case, we present the required and obtained memory bandwidth per application in two different conditions: under fair sharing and under an unequal pattern. Several bandwidth distributions were measured, we present only one for brevity. The results are consistent across all patterns.

Memory usage is de£ned as the number of clock cycles dedicated to data transmission divided by the total number of cycles. This number should ideally approximate 1, although that is impossible in practice due to the communication overhead and the scheduling effect. The results for one application are shown as reference. Since one application, being alone, produces a load equivalent to 40% of the total memory bandwidth, any number of applications equal or greater than 3 already offers a total load of over 100%. Hence, the tests also show the capacity of the design to safely deal with overload (320% for 8 applications).

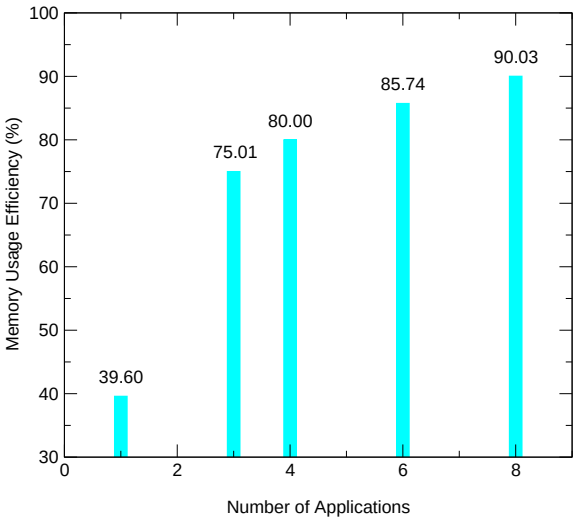

Fig. 5. Memory usage for different # applications

The number of applications varies from 3 to 8 to test the

App Nr	req (%)	obt (%)	req (%)	obt (%)
App0	33.3	33.3	40	40.0
App1	33.3	33.3	20	20.0
App2	33.3	33.3	40	40.0

TABLE I
REQUESTED AND OBTAINED BANDWIDTH: 3 APPS

App Nr	req (%)	obt (%)	req (%)	obt (%)
App0	25	25	20	19.6
App1	25	25	30	32.1
App2	25	25	20	19.6
App3	25	25	30	28.6

TABLE II
REQUESTED AND OBTAINED BANDWIDTH: 4 APPS

reactions of the system to an increasing sharing burden. Although not shown in the tables for clarity, as could be expected, the average and maximum delays, which are both bounded by the scheduler, increase with the number of applications. On the other hand, it can also be noticed, that the overall performance also increases (see Figure 5). This is due to the more ef£cient use of time, since overhead in queuing and scheduling requests can be partially overcome by other applications sending useful data meanwhile. As can be seen, it represents a fair approximation to the maximum capacity of the memory, thus proving the small cost of our architecture in terms of ef£ciency in exchange for a strong enforcement of QoS and security policies.

A second set of tests checked the capacity to deliver QoS in terms of individual bandwidth for the applications. As can

App Nr	req (%)	obt (%)	req (%)	obt (%)
App0	16.6	16.6	8.3	9.4
App1	16.6	16.6	25	24.5
App2	16.6	16.6	8.3	9.4
App3	16.6	16.6	8.3	9.4
App4	16.6	16.6	25	24.5
App5	16.6	16.6	8.3	9.4

TABLE III
REQUESTED AND OBTAINED BANDWIDTH: 6 APPS

App Nr	req (%)	obt (%)	req (%)	obt (%)
App0	12.5	12.5	30	30.6
App1	12.5	12.5	2	2.3
App2	12.5	12.5	2	2.3
App3	12.5	12.5	30	28.9
App4	12.5	12.5	2	2.3
App5	12.5	12.5	2	2.3
App6	12.5	12.5	30	28.8
App7	12.5	12.5	2	2.3

TABLE IV
REQUESTED AND OBTAINED BANDWIDTH: 8 APPS

be seen on the tables, in spite of having greedy applications, the limits set on the bandwidth were kept within reasonable limits in all cases, independently of the weights chosen.

Results on the success of the attacks described before (bus misuse and memory peeking) have not been included, for they always failed. As explained in section III-A, memory peeking or alteration is simply impossible, while bus misuse is immediately aborted and penalized.

V. CONCLUSIONS

Large Integrated Circuits enable hosting several independent systems on a single chip. The necessity to share a number of resources, especially off-chip resources (different kinds of memory, communication channels, CPUs), among those systems creates new constraints in the logic design process. Although performance is still a key constraint, sharing of resources creates the need for secure access to those resources as well as the ability to deliver QoS guarantees. Applications must be effectively isolated from each other in order to achieve a predictable behavior of the entire system on a chip.

An architecture has been presented that achieves the goals listed above. The Embedded Hardware Manager acts as a hardware layer between the applications and the resources, taking the role of resource manager and security agent. The results show that it can prevent resource misuse and information sharing violations while maintaining individual QoS guarantees. At the same time, high performance is still achieved.

REFERENCES

[1] S. A. McKee, W. Wulf, J. H. Aylor, R. H. Klenke, M. H. Salinas, S. I. Hong, and D. A. Weikle, "Dynamic access ordering for streamed computations," *IEEE Transaction on Computers*, vol. 49, no. 11, Nov. 2000.

[2] J. B. Carter, W. Hsieh, L. Stroller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, and R. Kuramkote, "Impulse: Building a smarter memory controller," in *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, Jan. 1999, pp. 70–79.

[3] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer con¤icts and exploit data locality," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, California, CA, Dec. 2000, pp. 10–13.

[4] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000, pp. 128–138. [Online]. Available: citeseer.nj.nec.com/rixner00memory.html

[5] S. A. McKee, "Hardware support for dynamic access ordering: Performance of some design options, Tech. Rep. CS-93-08, 9, 1993. [Online]. Available: citeseer.nj.nec.com/mckee93hardware.html

[6] C. Green, "Analyzing and implementing SDRAM and SGRAM controllers," *EDN Access*, Feb. 1998.

[7] C. Ykman-Couvreur, J.Lambrecht, D.Verkest, A. Nikologiannis, and G. Konstantoulakis, "System-level performance optimization of the data queueing memory management in high-speed network processors," in *Proceedings of DAC 2002*, New Orleans, Louisiana, 2002.

[8] R. Yan and S. C. Goldstein, "Mobile memory: Improving memory locality in very large recon£gurable fabrics," in *Proceedings of the IEEE symposium on Field-Programmable Custom Computing Machines*, napa, CA, 2002.

[9] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya, "Automatic generation of embedded memory wrapper for multiprocessor soc," in *Proceedings of DAC 2002*, New Orleans, Louisiana, 2002.

[10] Denali Software Inc., "Databahn product information," Available via Internet from `www.denali.com` as `/products_databahn.html`, 2001.

[11] W.-D. Weber, “Ef£cient shared DRAM subsystems for SoCs,” Available via Internet from `www.sonicsinc.com` as `/sonics/products/memmax/productinfo/docs/DRAM_Scheduler.pdf`, 2001.
[12] OCP International Partnership, “Open core protocol speci£cation, release 1.0,” 2001.
[13] Silicore Corp., “Speci£cation for the wishbone system-on-chip(soc) interconnection architecture for portable ip cores, revision b.2,” 2001.
[14] ARM, “Amba speci£cation, rev 2.0,” 1999.
[15] Micron Inc., “Small-outline SDRAM module MT4LSDT464(L)H, MT8LSDT864(L)H data sheet,” 1999.
[16] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, “Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX),” in *ACM International Symposium on Field Programmable Gate Arrays (FPGA’2001)*, Monterey, CA, USA, Feb. 2001, pp. 87–93.