SoC Infrastructures for Predictable System Integration

Pieter van der Wolf and Jeroen Geuzebroek
Synopsys
Eindhoven, The Netherlands
pieter.vanderwolf@synopsys.com

***Abstract*—Advanced SoCs integrate a diverse set of system functions that pose different requirements on the SoC infrastructure. Predictable integration of such SoCs, with guaranteed Quality-of-Service (QoS) for the real-time functions, is becoming increasingly challenging. We present a structured approach to predictable integration based on a combination of architectural principles and associated analysis techniques. We identify four QoS classes and define the type of QoS guarantees to be supported for the two classes targeted at real-time functions. We then discuss how a SoC infrastructure can be built that provides such QoS guarantees on its interfaces and how network calculus can be applied for analyzing worst-case performance and sizing of buffers. Benefits of our approach are predictable performance and improved time-to-market, while avoiding costly over-design.**

Keywords - SoC infrastructure; system integration; real-time; predictability; Quality-of-Service; network calculus;

I. INTRODUCTION

Many consumer systems, like digital TVs or mobile phones, employ Systems-on-Chip (SoCs) that integrate a diverse set of system functions [1][2]. Example system functions are audio processing, video processing, connectivity, and a (multi-core) host CPU executing a variety of software applications. Integration of such wide array of system functions is becoming increasingly challenging as complexity increases and stringent requirements on performance, cost, power consumption, and real-time behavior have to be satisfied. For example, it should be verified that the execution of the complex software stacks on the host CPU never results in a missed deadline for the real-time audio and video processing functions. Moreover, for many consumer systems it needs to be validated that the requirements are met not just for one use case, but for tens or hundreds of different use cases.

A key issue in building such SoCs is that the system functions have different requirements on the SoC infrastructure, i.e. interconnect and memory, that binds them together. For example, some functions, like audio or video processing [3], have real-time requirements, while other functions do not. Some functions, like a host subsystem, are latency critical, while other functions are more latency tolerant. Predictable integration of such diverse set of system functions, with guaranteed Quality-of-Service (QoS) for the real-time functions, is particularly challenging when cost efficiency demands sharing of resources like interconnect and memory.

A specific source of unpredictability is the access to off-chip DRAM, in particular for bandwidth-hungry multimedia applications for the cost-sensitive consumer market. Low cost demands a small number of DRAM devices and (therefore) a narrow interface. A rule of thumb for SoCs for e.g. digital TV is that the cost per 16 data pins is $2-3, due to an extra DRAM device, extra pins on the package, and additional silicon area for the memory controller. The DRAM interface is typically shared by many clients and arbitration must be performed. The interference of competing clients is hard to predict as the number of accesses (in a time window) and the associated addresses are often unknown. Further, DRAM performance is heavily impacted by the mapping of the addresses of the accesses to banks and rows and the penalties incurred (bank conflicts, read-write turnarounds, refreshes, etc.).

Over time the bandwidth for DRAM access has improved significantly as a result of higher data transfer rates for successive DRAM generations. Latency, however, has improved at a much slower pace. Hennessy and Patterson reported a 7% improvement per year in row access time in the 1990's [4] (revised to 5% in later editions) and the term "memory wall" was coined [5]. Over the last years, latency improvements have slowed even further, see TABLE I [6]. An improvement of about 3% per year is now more realistic.

TABLE I. LATENCIES FOR DRAM GENERATIONS

DRAM speed	CAS latency		RAS access time	
	Slow DRAM	Fast DRAM	(ns)	(cycles)
DDR2-667	15 ns	12 ns	45 ns	15
DDR2-800	15 ns	10 ns	45 ns	18
DDR2-1066	13.125 ns	11.25 ns	45 ns	24
DDR3-800	15 ns	12.5 ns	37.5 ns	15
DDR3-1066	15 ns	11.25 ns	37.5 ns	20
DDR3-1333	15 ns	10.5 ns	36 ns	24
DDR3-1600	13.75 ns	10 ns	35 ns	28

The relevant measures of DRAM latency, i.e. CAS latency (for slow/fast speed bins) and RAS access time, show only small improvements, measured in ns. When measured in cycles (of the memory interface clock) they show significant increases, implying that penalties for e.g. bank conflicts are increasing compared to the cycles spent on data transfer (for a given transaction size). As a consequence 1) the variability in memory access latencies is increasing and 2) high bandwidth efficiency is harder to achieve for faster DRAMs. The growing interference of competing clients and the increasing penalties in DRAM access make that memory access latency is getting more unpredictable over time. A further source of unpredictability is the SoC interconnect with its arbiters.

978-3-9810801-7-9/DATE11/©2011 EDAA

Correct integration of real-time functions demands that the processing elements executing these functions obtain a guaranteed QoS from the SoC infrastructure in order to meet their deadlines under worst-case conditions. Current practice is that SoC infrastructures typically do not provide such QoS guarantees and (hence) a structured approach to the integration of real-time functions is lacking. As a consequence, designers must resort to over-dimensioning (e.g. wider/faster DRAM interfaces, larger on-chip memories, bigger/faster CPUs) and extensive simulation to validate all the use cases of the SoC, with the risk that a corner case is overlooked. As complexity increases, this approach results in costly over-design, costly design iterations, and increased risk.

This paper presents a structured approach to the predictable integration of a variety of system functions into SoCs for advanced applications. Our approach is based on a combination of architectural principles and associated analysis techniques. After discussing related work in the next section, we present a classification of QoS requirements for system functions in Section III. In Section IV we present a format for specifying traffic requirements of system functions. The architectural principles and analysis techniques for SoC infrastructures that enable predictable integration are discussed in Section V. Section VI demonstrates the benefits of the approach through a case study and conclusions are drawn in Section VII.

II. Related Work

A number of research activities have focused on achieving predictable integration by means of dedicated architectures. In [7] a SoC architecture template is proposed that enables data flow analysis for real-time stream processing functions. Also, work has addressed dedicated architectures for SoC infrastructure elements like interconnect [8] or memory controllers [9]. These solutions take a rigorous approach by employing arbitration schemes that explicitly bound the interference by competing clients. Other work has aimed for predictable integration by defining techniques for modeling and (worst-case) analysis that can be applied to existing SoC architectures. Examples are [10] and [11], which apply network calculus to derive worst-case bounds on memory access performance.

Network calculus is a mathematical framework to derive worst-case bounds on latency, backlog and throughput. It was pioneered by Cruz [12] for the analysis of communication networks. Stiliadis and Varma [13] extended this work and introduced the Latency-Rate server as an abstract network element. LeBoudec [14] further developed the network calculus theory and based it on Min-Plus algebra. The basic elements in this algebra are arrival curves and service curves. Arrival curves bound traffic flows. Service curves describe output guarantees of network elements. Using arrival curves and service curves, the maximum backlog and maximum delay can be determined.

III. QoS Requirements

The system functions to be integrated in a SoC have different requirements on the SoC infrastructure. In Figure 1 we present a classification that helps to define the types of QoS that need to be supported by the SoC infrastructure.

	Latency critical	Latency tolerant
No real-time guarantee	Low latency clients ▪ Host CPU ▪ GFX LL	Best effort clients ▪ Connectivity ▪ Peripherals BE
Real-time guarantee	RT low latency clients ▪ Audio (on CPU) AL	RT streaming clients ▪ Video LR

Figure 1. Classification of QoS requirements.

First, we distinguish between clients that are latency critical and clients that are latency tolerant. For example, processing on a CPU is typically latency critical, as the latency leads to stalls of the CPU that have a direct impact on the performance of the system function. Therefore, memory access latency should be low, i.e. in the range of tens to a few hundred ns for fetching a cache block from DRAM. Function-specific processing elements can often be more latency tolerant. Video processing, for example, can in many cases be made latency tolerant using input and output buffers. While memory accesses are in flight, the video processing can continue processing using the data/room in the input/output buffers. In this way latencies of several µs can be accommodated. It is often important to make as many clients as possible latency tolerant, as this enables truly low latencies for latency-critical clients.

Second, we distinguish between clients that do not need a real-time (RT) guarantee and clients that need a real-time guarantee to meet some hard deadline. An example of the first category is a host CPU, which we like to run as fast as possible, but there is no hard deadline. Video processing, on the other hand, can have hard deadlines, e.g. when it drives a display that periodically needs a new video frame. Even if we add buffering for latency tolerance, it still demands a guaranteed QoS from the SoC infrastructure in order to assure that the input buffers never underflow and the output buffers never overflow.

Based on these distinctions we define four QoS classes, as shown in Figure 1. The key to predictable integration is to have these QoS classes explicitly supported by the SoC infrastructure, complemented by associated techniques for static analysis. In our approach to predictable integration, each system function is assigned the appropriate QoS classes on its interfaces (see Figure 5). The approach further comprises 1) the specification of the traffic of the system functions, 2) the use of a SoC infrastructure that supports the defined QoS classes, and 3) analysis techniques for verifying that the traffic requirements of the real-time system functions are satisfied by the QoS guarantees of the SoC infrastructure. The latter also includes the sizing of buffers. The QoS guarantees protect the real-time functions from interference of competing clients and support their predictable integration.

For the RT Low Latency QoS class we propose that guarantees take the form of an upper bound on the average latency for memory accesses over a particular window. An example system function that may exploit such Average-Latency (AL) guarantee is frame-based audio processing on a CPU, with deadlines in the ms range. For this kind of audio processing it can typically be validated that deadlines are always met as long as the average memory access latency over a certain window is below a particular value. Having a SoC infrastructure that can be configured / programmed to provide

the AL type of QoS guarantee is the key to efficient and correct integration of such audio processing function.

For the RT Streaming QoS class we propose that guarantees take the form of a Latency-Rate (LR) guarantee. This implies that a throughput (ρ) is guaranteed after an initial latency (θ), as shown in Figure 2.

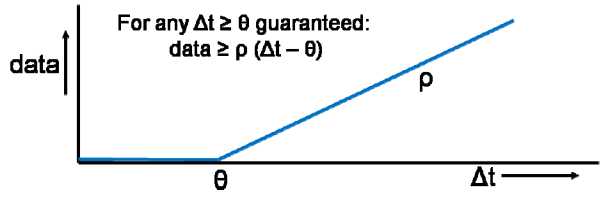

Figure 2. Latency-Rate guarantee.

In order to provide such guarantee, the SoC infrastructure needs to behave as a Latency-Rate server [13]. A Latency-Rate server provides a LR guarantee during a busy period, i.e. as long as the data rate requested by the client is at least ρ. Note that a LR guarantee specifies a lower bound on the service provided (i.e. the minimum amount of data accepted / supplied by the SoC infrastructure) for any window of arbitrary size within a busy period. The SoC infrastructure needs to offer LR guarantees at the interfaces to the processing elements executing RT Streaming functions. An example system function that may exploit such QoS guarantee is a video function that streams data to / from memory. If we know the data traffic requirements of the video function, we can exploit the LR guarantees to size input / output buffers such that they never underflow / overflow. The buffer sizing is discussed in more detail in Section V.

The QoS classes for the non-real-time clients are Low Latency (LL) and Best Effort (BE).

IV. Traffic Specification

The starting point for the design and programming of a SoC infrastructure is the specification of the traffic of the different system functions that need to be served. Such specification is used as input for determining the type of infrastructure, the size of buffers, and the choice of arbiters and arbiter settings.

TABLE II. Traffic Parameters

Parameter	Short Description
Bandwidth	The bandwidth requested by the client
Burstiness	The number of individual transactions that are issued in a burst
Burstiness Gap	The gap (time) between consecutive transactions within the same burst
Transaction Size	The size of individual transactions
Read Ratio	The ratio between the number of read transactions and the total number of transactions
Addressing	The addressing relation between consecutive transactions
$Trans^{Max}$	Maximum number of outstanding transactions
QoS Class	The Qos Class of the traffic
Latency	The requested average (AL) or maximum (LR / θ) latency for the traffic

In this section we propose a structured format for specifying this traffic. TABLE II lists the important traffic parameters.

Bandwidth specifies the long-term average bandwidth of the corresponding traffic flow. Bandwidth is an important parameter for arbiter programming and buffer sizing. For video flows, the average bandwidth can be derived from resolution information, frame rate, bits per pixel, etc. For host CPU's, bandwidth can be specified assuming a typical average latency as CPUs tend to stall on memory requests and run faster, requesting more bandwidth, when the latency is low.

Burstiness and Burstiness Gap define how bursty the traffic of a flow behaves. *Burstiness* specifies the number of transactions that are issued as a burst of transactions, where each transaction is issued *Burstiness Gap* ns after the previous one. A burst results in a peak bandwidth that the SoC infrastructure should be able to serve. A trade-off can be made between over-allocating bandwidth, i.e. allocate a higher bandwidth than the long-term average bandwidth, and the size of a traffic-shaping buffer.

An example is provided in Figure 3, where client A issues 4 transactions in a burst, while client B does not. Although on average (time window 8000ns) the number of issued transactions is the same, the traffic of client A results in a higher peak bandwidth. This results in a higher traffic upper bound for client A. A higher traffic upper bound translates into higher buffer requirements, as described in the next section.

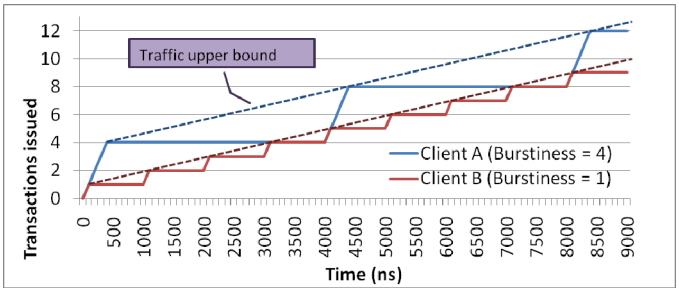

Figure 3. Impact of burstiness on traffic upper bound.

Transaction Size defines the size in bytes of individual transactions. The transaction size impacts DRAM efficiency, as small transactions have a higher protocol overhead (relative to the actual data transfer) compared to large transactions.

Read Ratio specifies the ratio between the number of read transactions and the total number of transactions (read + write) issued by a client. For sizing read and write buffers in the SoC infrastructure it is important to not only know the bandwidth and latency requirements, but also the amount of data flowing in each direction (read/write). If the read ratio varies significantly over time, one may choose to define separate traffic flows for read and write traffic.

Addressing defines the address relation between consecutive transactions. Especially for memory controllers the addressing has a significant impact on the efficiency at which traffic can be served. In case consecutive transactions have consecutive addresses, the address mapping (both at the clients and at the memory controller) can be set up such that

DRAM accesses run linearly through the rows of the banks. When all traffic flows use such a linear addressing, the DRAM traffic can be equally balanced over all banks, limiting the occurrence of bank conflicts and bank congestion. In case of more random addressing, bank balancing becomes harder. Further, page hits are more likely to occur for linear addressing than for random addressing.

$Trans^{Max}$ specifies the maximum number of outstanding transactions a traffic client supports. This number has a significant impact on the throughput that can be obtained. In case a client supports only one outstanding transaction, it cannot issue a second transaction until the first one has completely finished. The latency of each transaction heavily limits the number of transactions that can be issued within a certain time window, thereby limiting the throughput

In case a client supports two outstanding transactions, it can already issue a second transaction before the first one has finished. However, also the SoC infrastructure must support two outstanding transactions. On the other hand, the SoC infrastructure should not be over-dimensioned to support more outstanding transactions than will ever be issued by the clients.

The last two traffic parameters, QoS class and Latency, are present for integration purposes. *QoS class* specifies the QoS class of the traffic. It is used to program arbiters within the SoC infrastructure. *Latency* specifies the targeted average latency for AL clients or the maximum initial latency (θ) for LR clients. It is used to program arbiters in such a way that the SoC infrastructure satisfies the latency requirements.

V. SoC Infrastructure Architecture and Analysis

For predictable integration we need to define and analyze a SoC infrastructure such that it meets the traffic requirements of the system functions, including the associated QoS guarantees. Specifically for the real-time system functions we need to perform a *worst-case* analysis to ensure that the QoS guarantees are met under all circumstances, e.g. for all use cases as well as for peak loads during use cases. As we explained in Section I, simulation-based approaches impose increased cost and/or risk. Therefore we advocate static analysis, employing formal models and techniques that allow us to analyze the worst-case performance.

In order to apply static analysis, first the SoC infrastructure needs to be modeled. Specifically, the arbiters in interconnect and memory controllers need to be modeled in order to analyze the worst-case delay and throughput for the specified traffic flows. An important requirement is that the modeling yields tight bounds on the worst-case behavior, to avoid costs of over-design. For example, if analysis for the LR clients yields an overly conservative initial latency (θ), then extra area must be spent on input / output buffers which is not needed for correct operation.

If interconnect and memory controllers have not been designed with these modeling requirements in mind, then it is typically very hard to derive a model of the SoC infrastructure that offers acceptable performance bounds. This is specifically true for (multi-ported) DRAM controllers, which need to offer high bandwidth efficiency while providing low latency to latency-critical clients like CPUs. In order to achieve this, they typically schedule DRAM commands dynamically. Commercial multi-ported DRAM controllers often use complex arbitration schemes, including mechanisms like (static) priorities (to support latency-critical clients), ageing and budgeting (to avoid starvation and provide control for QoS), and reordering of DRAM commands (to avoid penalties like bank conflicts or read-write turnarounds). However, these controllers are very hard to analyze, let alone derive an acceptable bound on worst-case behavior. In terms of our QoS classes they support the non-real-time classes LL and BE. A key issue is that the service obtained by a client is often highly dependent on the actual traffic of other clients. We expect that this situation will deteriorate further as the number of clients and their bandwidth requirements increase and, as motivated in Section I, DRAM access becomes increasingly unpredictable. Åkesson [9] explicitly addresses predictability for memory controllers and offers LR guarantees at the ports.

The way forward is to design memory controllers and interconnect with requirements for QoS in mind. In our approach, a multi-ported memory controller allows QoS requirements to be explicitly programmed per port. That is, for each port the QoS class can be programmed, with explicit AL and LR guarantees for ports of the RT Low Latency and RT Streaming classes, respectively. See Figure 4.

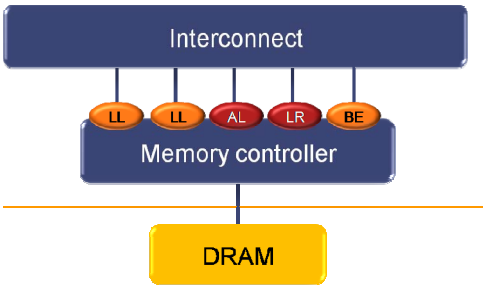

Figure 4. Multi-ported memory controller with QoS properties at its ports.

The memory controller schedules requests from clients in such a way that the AL / LR guarantees for the subset of ports with real-time clients are satisfied, thereby resolving the unpredictability of DRAM access for these clients. In addition, it tries to offer low latency to the non-real-time Low Latency clients. Best Effort clients are served with the lowest priority.

In typical situations the LL ports are served with the highest priority, yielding low latencies for the latency-critical clients. The other ports are served with a lower priority in between the LL requests. The memory controller explicitly accounts the service provided on its AL / LR ports. When an account indicates that a port is in danger of being underserved, this port takes priority over the LL ports in order to ensure that its guarantee is met. The explicit accounting of service provided on the AL / LR ports is the key to providing guarantees independent of the DRAM penalties encountered, and protects the real-time clients from interference by competing clients. For realistic systems, the LL ports are served with the highest priority for about 90% of time, to be overruled by AL / LR ports only infrequently. Therefore, this solution strikes a good balance for offering both QoS guarantees and low latency. In addition, the programming of this memory controller is extremely simple as the required QoS guarantees can be programmed directly. No extensive tweaking of (priority) settings is required.

Now that we have a solid foundation with a memory controller that offers true QoS guarantees at its ports, the next step is to build a SoC infrastructure that offers QoS guarantees at the interfaces to the system functions. See Figure 5.

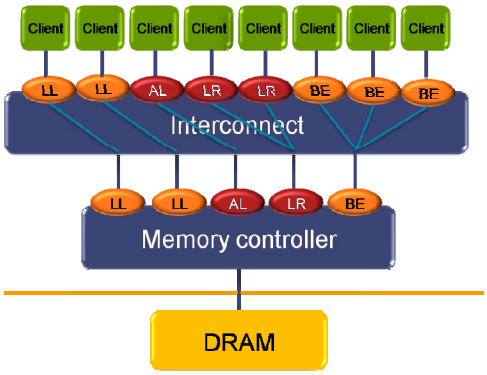

Figure 5. SoC infrastructure offering QoS guarantees to system functions.

For this purpose we need an interconnect that provides bandwidth guarantees as well as bounds on request and response latencies. An example of an interconnect that has these properties is the Aethereal NoC [15]. Using the bounds on the request and response latencies, the latency value of an AL guarantee at the client interface can be related to the latency value of the corresponding AL guarantee at the memory controller. Similarly the θ-values of LR guarantees at client interface and memory controller can be related. If a LR port of the memory controller is shared by multiple LR clients, then the interconnect must allow the ρ-value of the LR guarantee at the memory controller port to be distributed to ρ-values at the client side. The TDM-based arbitration in the Aethereal NoC allows such bandwidth allocation per client.

The LR guarantees at the client side can be used to size input / output buffers so that they never underflow / overflow. Figure 6 illustrates the buffer size calculation using the traffic upper bound derived for the system function and the LR-guarantee offered by the SoC infrastructure.

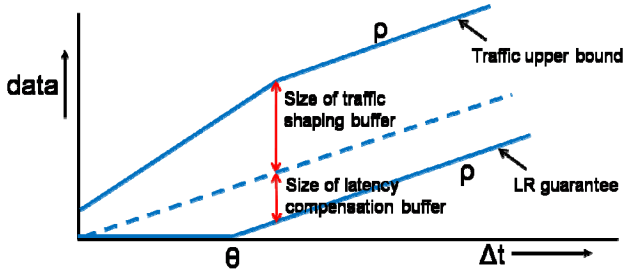

Figure 6. Buffer sizing for LR-clients.

First, the buffer is used as a traffic shaping buffer, in order to regulate the peak bandwidth of the client down to the average bandwidth (ρ). Second, the buffer is used for latency compensation, which requires a buffer of size ρ * θ. Note that it is guaranteed that a client never stalls for a full / empty buffer as long as the client stays within its traffic bound. Stalls may occur if the client traffic goes outside the traffic bound. Careful definition of the traffic bounds is therefore important.

The buffer sizing can be used in two ways. If the client buffer size is fixed, then a LR-guarantee, i.e. ρ and θ, can be derived for the SoC infrastructure. Conversely, using a LR-guarantee of the SoC infrastructure, the size of the buffer at the client side can be determined. The latter approach may be used to optimize buffer sizes across different SoCs, in particular for high bandwidth clients, like video functions.

VI. Case Study

We present a case study in which multiple clients with different QoS requirements share a common memory interface. The case study comprises heavy traffic requirements such that the memory controller cannot serve all traffic initiated by the clients and, hence, has to decide which client(s) to under-serve. This allows us to evaluate the performance and QoS properties for a peak load in the system.

The memory controller implements the accounting-based arbitration explained in the previous section. It supports the LL, BE, and RT Streaming QoS classes. Priorities can be assigned to ports of the LL/BE classes to control the arbitration among them. Ports of the RT Streaming class are programmed with the required LR guarantee. The simulations have been run on the actual RTL of a 32-bit memory controller that takes into account all (timing) penalties of the attached DRAM devices, in this case two 1Gbit DDR2-800 x16 devices.

For this case study, clients include a CPU with low latency requirements, clients with best effort traffic and latency-tolerant video clients with real-time requirements. The memory controller has been configured with four ports. The SoC interconnect maps the CPU traffic to a LL port. All best effort traffic is mapped to a BE port. The traffic of the video clients is mapped to two other ports, P1 and P2. All this traffic must be served in order to meet the real-time requirements of the video clients. The amount of traffic of the video clients is twice the amount requested on the LL or BE port.

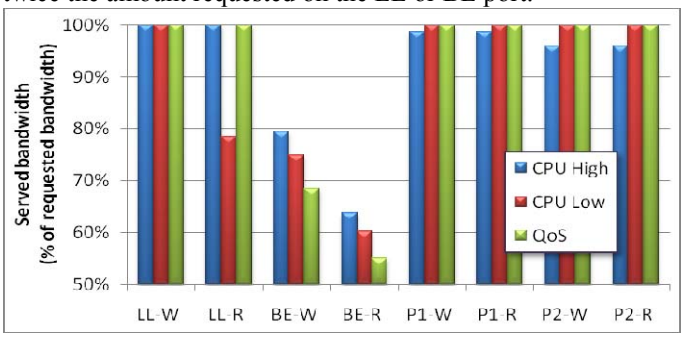

Figure 7. Served bandwidth on ports of memory controller.

The performance is evaluated for three different arbitration schemes of the memory controller. Figure 7 and Figure 8 show the served read (-R) and write (-W) bandwidth, respectively the average read latency of first data arrival, for the different arbitration schemes. In all arbitration schemes the BE port has the lowest priority as it requires no guarantees at all.

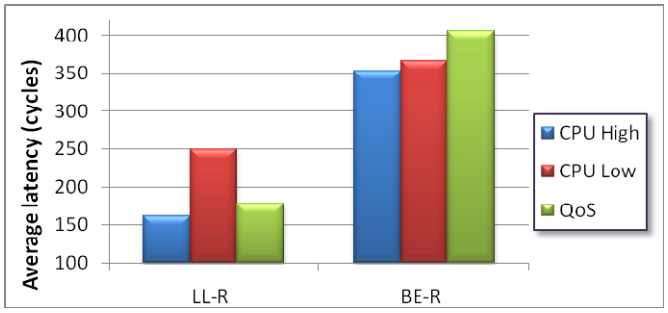

Figure 8. Latency for read traffic on LL and BE ports of memory controller.

In the first experiment the memory controller is programmed with a fixed priority scheme. The LL port has the highest priority, to satisfy the low latency needs of the CPU. The ports P1 and P2 are programmed at a priority lower than the LL port but higher than the BE port. The results, labeled CPU High in Figure 7 and Figure 8, show that the CPU traffic indeed has a relatively[1] low latency. However, not all traffic of the video clients at ports P1/P2 is served and hence real-time requirements are not met.

In the second experiment the ports P1/P2 are fixed to the highest priority, to assure that the video clients meet their real-time requirements. The LL port has a lower priority, but still higher than the BE port. The corresponding results, labeled CPU Low in Figure 7 and Figure 8, indeed show that all traffic of the video clients is served. However, the CPU traffic gets a huge latency penalty.

In the last experiment we deploy the accounting-based arbitration described in section V. The ports P1/P2 are programmed with the required LR guarantees. Initially the LL port with the CPU traffic has the highest priority. As soon as the memory controller detects that the LR guarantees for the video clients are at risk, it dynamically raises the priorities of the LR ports above the LL port priority. When the memory controller detects that the LR guarantees of the video clients are no longer at risk, it again lowers the priorities of the LR ports. The results, labeled QoS in Figure 7 and Figure 8, show that all traffic of the video clients is served, while the impact on the CPU latency is limited.

These results show that a fixed priority based arbitration scheme cannot satisfy both the real-time requirements on one port and the low-latency requirements on another port. Either the real-time traffic is at risk or the CPU traffic gets a significant latency penalty. The more advanced accounting-based arbitration scheme is able to satisfy both real-time requirements and low latency requirements. High performance CPUs can obtain the best possible service, including support for peak loads, without putting the real-time clients at risk.

VII. CONCLUSIONS

This paper presents a structured approach to the predictable integration of a variety of system functions into SoCs for advanced applications. It is based on a combination of architectural principles and associated analysis techniques. We identify four QoS classes and define Average-Latency and Latency-Rate guarantees for the two classes targeted at system functions with real-time requirements. Key elements of our approach are 1) a format for traffic specification, 2) architectural elements (memory controller, interconnect) that explicitly support the defined QoS classes, and 3) static analysis of worst-case performance and buffer sizing applying network calculus.

A case study demonstrates that with our approach QoS guarantees can be provided to real-time clients while also offering low latencies to a latency-critical CPU. It also demonstrates that traditional priority-based arbitration without accounting either does not provide such guarantees or suffers from high latencies for the CPU.

Key benefits of our approach are predictable performance and improved time-to-market, while avoiding costly over-design to guarantee real-time behavior. These benefits extend to the system integrator that applies the SoC in a consumer product. For example, modifying application software executing on the host CPU will not erroneously result in violations of real-time constraints for audio / video functions, as the QoS guarantees for these functions remain valid.

[1] Due to the very high load on the memory controller, the absolute latencies are quite high.

REFERENCES

[1] P. Kollig, C. Osborne, T. Henriksson, "Heterogeneous multi-core platform for consumer multimedia applications", Proc. DATE, 2009.

[2] C.H. van Berkel, "Multi-core for mobile phones", Proc. DATE, 2009.

[3] P. van der Wolf and T. Henriksson, "Video processing requirements on SoC infrastructures", Proc. DATE, 2008.

[4] J.L. Hennessy and D.A. Patterson, "Computer architecture: a quantitative approach", Second edition, Morgan Kaufmann, 1996.

[5] W.A. Wulf and S.A. McKee, "Hitting the memory wall: implications of the obvious", Computer Architecture News, vol. 23, pp. 20-24, 1995.

[6] JEDEC Standard, DDR2/DDR3 SDRAM specifications, Documents JESD79-2F, JESD208, JESD79-3E.

[7] A. Hansson, K. Goossens, M. Bekooij, J. Huisken, "CoMPSoC: a template for composable and predictable multi-processor system on chips", ACM ToDAES, vol. 14, issue 1, Jan. 2009.

[8] K. Goossens, J. Dielissen, A. Radulescu, "The Aethereal network on chip: Concepts, architectures, and implementations", IEEE Design and Test of Computers, vol. 22, no. 5, pp. 414-421, 2005.

[9] B. Åkesson, K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration", Proc. DATE, 2011.

[10] T. Henriksson, P. van der Wolf, A. Jantsch, A. Bruce, "Network calculus applied to verification of memory access performance in SoCs", Proc. ESTIMedia 2007, pp. 21-26.

[11] J.P. Vink, K. van Berkel, P. van der Wolf, "Performance analysis of SoC architectures based on Latency-Rate servers", Proc. DATE, 2008.

[12] R.L. Cruz, "A calculus for network delay, Part I: network elements in isolation, and Part II: network analysis", IEEE Trans on Information Theory, vol. 37, no. 1, pp. 114-141, 1991.

[13] D. Stiliadis and A. Varma, "Latency-Rate servers: a general model for analysis of traffic scheduling algorithms", IEEE/ACM Transactions on Networking, vol. 6, no. 5, pp. 611-624, October 1998.

[14] J.-Y. LeBoudec, "Network Calculus", Springer Verlag, no. 2050, LCNS, 2001.

[15] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, M. Bekooij, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis", IET CDT, 2009.