Analyzing On-Chip Communication in a MPSoC Environment

Mirko Loghi[†] loghi@sci.univr.it
Federico Angiolini[‡] fangiolini@deis.unibo.it
Davide Bertozzi[‡] dbertozzi@deis.unibo.it
Luca Benini[‡] lbenini@deis.unibo.it
Roberto Zafalon[#] roberto.zafalon@st.com

[†]Dipartimento di Informatica - University of Verona
Strada Le Grazie, 15 37134 Verona, Italy
[‡]Dipartimento di Elettronica, Informatica e Sistemistica (DEIS) - University of Bologna
Viale Risorgimento, 2 40134 Bologna, Italy
[#]Advanced System Technology - Research & Innovation - STMicroelectronics
Via C. Olivetti, 2 20041 Agrate Brianza (MI), Italy

Abstract

This work focuses on communication architecture analysis for multi-processor Systems-on-Chips (MPSoCs), and it leverages a SystemC-based platform to simulate a complete multi-processor system at the cycle-accurate and signal-accurate level. These features allow to stimulate the communication sub-system with functional traffic generated by real applications running on top of a configurable number of ARM processors. This opens up the possibility for communication infrastructure exploration and for the investigation of its impact on system performance at the highest level of accuracy. Our simulation environment proved capable of a detailed comparative analysis between two industry-standard communication architectures, under realistic workloads and different system configurations, pointing out the impact of fine grained architectural mismatches on macroscopic performance differences.

1. Introduction

As technology scales toward deep sub-micron, the integration of a complete system consisting of a large number of IP blocks on the same silicon die is becoming technically feasible. In this context, the communication sub-system of these complex Systems-on-Chip (SoCs) is increasingly critical for system performance, and therefore represents a key component to be investigated during architecture definition and tuning.

Most current designs are based on shared communication resources (busses) due to their low cost. Unfortunately, scalability is limited by serialization for multiple bus access requests. As the number of IP blocks to be interconnected increases, on-chip micro-networks of interconnects (or Networks-on-Chips, NoCs) [1] can provide an adequate solution [1, 15, 2]. Such a widening design space emphasizes the need for a thorough exploration of interconnection choices, including bus designs of varying complexity and alternative topologies [13].

This paper proposes a complete platform for analysis and trade-off exploration of on-chip communication architectures, and provides a complete case study of practical interest: namely, a detailed comparative analysis, under a number of different architectural configurations, of two industry-standard communication infrastructures: AMBA Advanced High Performance Bus (AHB) from ARM and STBus interconnect from ST Microelectronics.

Our simulation platform for multi-processor systems generates functional traffic for the communication architecture by means of real applications running on top of a scalable number of ARM processors. The whole system is simulated (communication initiators, private as well as shared memories, synchonization devices) at the cycle-accurate and signal-accurate level, thus minimizing the degrees of approximations with respect to real MPSoCs. This allows a realistic performance analysis of on-chip interconnects: their performance can be accurately assessed for different classes of applications (communication versus computation-dominated traffic), software architectures (stand-alone versus operating system supported applications) and system configurations (cache size, external memory latency, etc.).

With respect to previous work, our approach targets high accuracy MPSoC simulation, in that it relies on functional traffic instead of fixed execution traces, or statistic traffic generators, or analytical models. In this way, dynamic effects such as interaction among traffic sources can be taken into account. Experimental results demonstrate that subtle protocol mismatches and middleware-induced behavior are indeed responsible for macroscopic performance differences.

This paper is structured as follows. Section 2 discusses previous work. Section 3 describes the hardware and software architecture of our simulation platform. Section 4 provides some technical insights about the features of the communication architectures we implemented. Section 5 details the experimental results with respect to different benchmarks. Finally, conclusions are drawn in Section 6.

1530-1591/04 $20.00 (c) 2004 IEEE

2. Related work

While methodologies for modelling and evaluation of architectures of processing elements are well known, modelling of an extensive range of on-chip communication interconnects and their integration into a single simulation environment combining processing elements and on-chip communication is still an active research area. [13] proposes a hierarchical modeling framework where new communication architectures can be developed by means of a library of reusable components.

Efficient mapping of system communication requirements on target communication architectures is becoming an integral part of any system design flow. In [7] a two step systematic exploration of the communication architecture design space is addressed. Similarly, the work in [5] aims at selecting the communication infrastructure that best meets application communication requirements.

The enabling technology for communication optimization is system-level performance analysis. To this purpose, several approaches have been proposed:
(i) The entire system can be simulated using models of the components and their communication at different levels of abstraction [9, 4].
(ii) Static system performance estimation techniques including models of the communication time between system components. Time estimates are usually either optimistic (ignoring dynamic effects such as bus contention) [3] or pessimistic (assumption of worst case scenario) [14].
(iii) Intermediate approaches, deriving set of traces from an initial cosimulation of the system (assuming abstract data transfers), and forwarding them to an analysis tool that, for a specified communication architecture, comes up with system performance estimates [7].

A limitation of previous approaches is that performance of communication architectures is derived under non-realistic workloads. Traditionally, parameterized statistic traffic generators are used [6, 16, 8], that, in spite of their generality, prevent designers from assessing performance in presence of real-life workloads and make it difficult to account for dynamic effects such as bus contention. The work in [11] goes in this direction.

Our work is based on a simulation environment that models all hardware and software components of a multiprocessor system at a high level of accuracy, while at the same time providing sufficient simulation speed to run significant applications. We stimulate the communication subsystem with functional traffic, to provide realistic performance estimates and statistics, referred to different classes of applications.

3. Multiprocessor simulation platform

3.1. Hardware architecture

The architecture of our platform (see Fig.1) is representative of a large class of homogeneous MPSoC platforms. It is composed of (i) a configurable number of 32-bit ARM processors (in this paper, four), (ii) their private memories, (iii) a shared memory, (iv) a hardware interrupt module, (v) a hardware semaphore module, (vi) the 32-bit interconnection among them all. This interconnection can be an AMBA AHB bus or an STBus arbitrary topology, resulting in different versions of the platform.

Processor cores are modeled by means of an adapted version of a GPL-licensed ARM *Instruction Set Simulator* (ISS) called SWARM [12] and written in C++. Since all of the hardware devices mentioned above, including the interconnection layer, are coded in SystemC, we embedded the ISS into a SystemC wrapper.

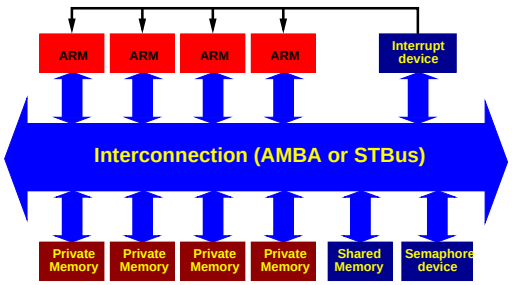

Figure 1. The platform architecture

The platform instantiates several memory devices, which can be used as private or shared memories. Their latency can be configured to explore interconnection performance under several conditions. To compare AMBA AHB and STBus in an unbiased fashion (see Section 4 for more details), memories were kept as simple as possible and did not feature any kind of internal buffering: every access, even when part of a burst, requires the same number of cycles.

The interrupt device allows processors to send interrupt signals to each other. This hardware primitive is needed for interprocessor communication and is mapped in the global adressing space. For an interrupt to be generated, a write should be issued to a proper address of the device. The semaphore device is also needed for the synchronization among the processors; it implements *test-and-set* operations, the basic requirement to have semaphores.

3.2. OS and benchmarks

We ported the RTEMS operating system [10] to our platform. Our choice was motivated by the fact that RTEMS is a lightweight OS for embedded systems, but it offers at the same time good support for multiprocessing, and provides native calls for communication and synchronization in such multiprocessor environments.

We then developed benchmark tasks running on top of RTEMS and requiring heavy bus activity. The first benchmark performs independent matrix multiplications at each processor, and does not require interprocessor communication. Operands are stored in the private memories of the processors.

A second benchmark implements a pipeline of matrix multiplications. Each processor executes a matrix multiplication between an input matrix and a private operand matrix, then feeds its output to the logically following processor. The platform receives a continuous flow of input matrices and produces a continuous flow of output matrices. Every core follows a fixed execution pattern: (i) copies an input matrix from shared space to private space; (ii) multiplicates it with a matrix already in private space; (iii) copies the resulting matrix back to shared space. During all of the process, interrupt and/or semaphore slaves are queried to keep synchronization.

Additionally, in order to capture the impact of the operating system on system performance, we rewrote the benchmarks in standalone C/Assembler code. We had to choose different synchronization techniques for our OS and non-OS pipelined benchmarks. RTEMS-based code uses OS queues to exchange matrices between processors, while C/Assembler code simply uses predefined memory areas, and achieves synchronization by directly accessing

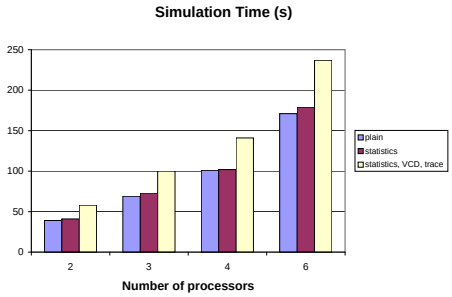

Figure 2. Simulator performance with pipelined matrix multiplications

system semaphores. Semaphore checking is performed by polling. Therefore, although functionally equivalent, some low-level differences exist between benchmarks: RTEMS code implicitly uses both semaphores and interrupts, while C/Assembler code relies on a polling mechanism.

3.3. Code development and analysis support

We developed tools and support libraries to support fast development and debugging of new applications and benchmarks on our platform. This is key for establishing a solid and flexible simulation environment.

Application code, either RTEMS-based or not, can be easily compiled with standard GNU cross-compilers. Scripts and makefiles fully automate the process of building for an SMP platform. Simple function calls, provided by support libraries of the simulator, allow flexible performance profiling: statistics can be collected during OS boot, application execution, or critical sections of algorithms. The output of the simulator can be configured to be very instructive, including (i) statistics about processor and interconnection performance (see Section 5 for more details), (ii) VCD waveforms of all bus signals, and (iii) traces of memory accesses performed by every core.

Debug functions include a built-in debugger, which allows to set breakpoints, execute code step-by-step and inspect memory content; it is additionally capable of dumping the full internal status of the execution cores. When testing applications written without underlying OS support (i.e., no native I/O calls are available), messages and status information can still be easily provided to the user by means of pseudo-instructions.

Simulation accuracy and flexibility have to be traded-off with simulation speed. However, Fig.2 shows that our platform, despite being signal-accurate and cycle-accurate, is fast and usable. The chart depicts simulation performance with the AMBA AHB interconnect, as a function of the number of processors and of the requested output statistics. When running on a Pentium® 4 2.26 GHz workstation, execution times of OS-PIP (the most resource demanding benchmark), including RTEMS boot and ten full pipeline iterations, stayed always below four minutes even when simulating a six-processor system with all possible statistics and debug information enabled. Always in this six-processor scenario, 62,000 to 86,000 CPU cycles could be simulated per second, depending on the choice of outputs. As the chart shows, the overhead of statistics collection on execution times is almost negligible; a more significant impact is associated with VCD waveforms and memory traces, due to substantial hard disk activity (about 300 MB of data in the six-processor benchmark). By disabling such outputs, or reducing their scope (trimming the amount of VCD traces to be recorded), speed-ups of around 25% could be achieved.

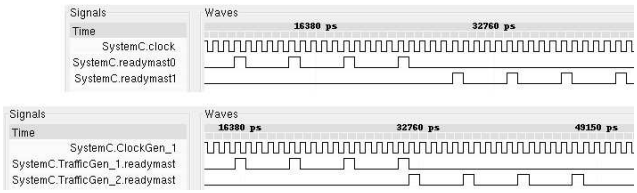

Figure 3. AMBA, STBus memory transfers (4 wait states)

4. Bus architectures and protocols

4.1. AMBA AHB

Three distinct buses are actually defined within the AMBA specification. AMBA Advanced High Performance Bus (AHB) is the most performing one and was selected for our analysis, even though we chose to omit support for some of its features (*e.g.* locked transfers and some advanced burst techniques, which are not supported by ARM cores).

AMBA AHB is based upon a traditional shared bus topology and exploits pipelining to maximize performance. It features distinct data and address/control buses. Transfers are composed of an *address phase* and of a *data phase*, and the address phase of a new transfer overlaps with the data phase of the previous transfer. This allows maximum throughput while imposing light timing requirements upon the slaves, but also increases latency: a minimum of three clock cycles are needed to complete a transfer, two spent in the arbitration and addressing phase and one (plus wait states) in the data phase. In our simulations, we assumed one wait cycle for the slaves, so minimum latency amounted to four cycles.

It is important to stress here that AMBA AHB does support bursts, but it treats them as streams of single transactions; bursts are simply a way of arbitrating just once for multiple transfers, thereby reducing latency. This means that memories have no way of early detecting bursts and accordingly make use of prefetching or buffering.

According to the AMBA AHB specification, when one master owns the bus no other master can perform any transaction. This means that high-latency slaves can dramatically reduce bus performance. AMBA AHB provides two workarounds to this shortcoming. The first is a mechanism called ”split/retry transfer”: a high-latency slave can optionally decide to release the bus while preparing its response to a master-initiated transaction. However, this mechanism requires more complex slaves and arbiters and, for fast on-chip memory devices, access times are short enough to prevent advantageous use of split/retry transfers. A second way of improving bus usage is called ”early burst termination”: if the arbiter detects that the bus has been busy for too long, it can interrupt a burst transfer in progress and assign the bus to another master with pending bus access request. While allowing for better granularity of accesses, this approach does not actually hide latencies, since a preempted master has to compete for re-gaining bus access and to resume the suspended transfer.

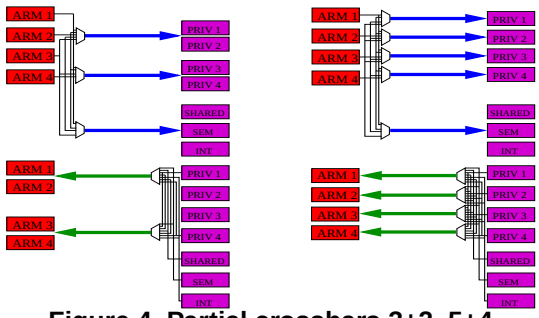

Figure 4. Partial crossbars 3+2, 5+4

4.2. STBus

STBus is a flexible communication architecture developed by STMicroelectronics. Its specifications define three different protocols; the simplest is called *type 1* and supports simple load/store operations, *type 2* adds more complex transfers, pipelining and split transactions, and finally *type 3* adds out-of-order support. Our tests were based on type 3 protocol.

The topology of an STBus interconnect is also very flexible and can range from a simple shared bus, like AMBA AHB, to a full crossbar. We analyzed performance obtained from a variety of topologies.

STBus features two data communication channels, one from initiators (e.g., processors) to targets (e.g., memories and dedicated hardware) and the other in the opposite direction. This allows an initiator to send a request while a target is sending a response. This overlapping of transfers is a key performance enhancer. Fig.3, taken from our simulations, shows how STBus is able to speed up transactions by requesting new bursts while previous ones are still completing and thus having no idle cycles inbetween. The figure also shows that our simulator does allow detailed, cycle accurate tracing of all bus signals.

STBus features fast arbitration, and this makes it possible to complete single read transfers in just two cycles, versus the three needed by AMBA - one cycle for arbitration/sending addresses and one for receiving data. When inserting a wait state, the minimum latency becomes of three cycles.

Due to the simple burst protocol supported by AMBA, as outlined above, in our tests we chose to always use extremely simple memories without any buffering and prefetching; this reduces the performance of STBus, but allows a fair comparison of bus performance.

5. Quantitative analysis

In this section, we will present two different types of quantitative analysis enabled by our simulator. The first is a *performance comparison* amongst five interconnections: AMBA AHB (AMBA), STBus configured as a shared bus (ST-BUS), STBus set up as a full crossbar (ST-FC), and two additional STBus partial crossbar topologies ST-32 and ST-54 (see Fig.4). These interconnects will be tested with the four benchmarks described in section 3.2: matrix multiplications performed independently by each processor and in pipeline, with and without an underlying OS (OS-IND, OS-PIP, ASM-IND and ASM-PIP respectively). All these results were measured with 8 kB ARM caches and with 1 wait state memories.

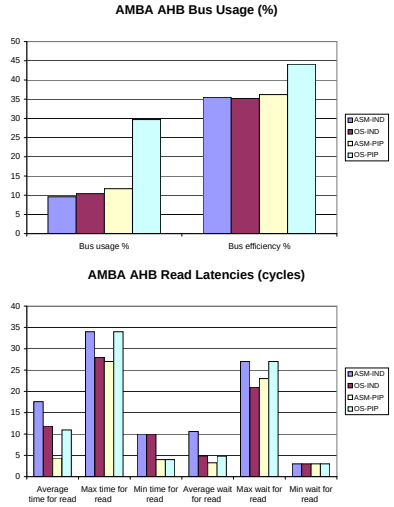

Figure 5. AMBA AHB analysis

The second type of analyis is an *architectural design space exploration*. Based on the most meaningful benchmark (OS-PIP), we will explore performance in presence of different system parameters like cache size, memory latencies and compiler optimizations.

To better describe the metrics that our simulator can measure, we refer to Fig.5(a).A and 5(b).B, both regarding AMBA. Fig.5(a).A shows statistics about bus usage during the execution of the four benchmarks. Two groups of results are reported: the first one details absolute bus usage, *i.e.* data transfers over total execution time, while the second expresses data transfers over the time during which the interconnect is busy. The first set of results is indicative of overall *bus congestion*, and shows that three benchmarks put relatively light pressure on system interconnect (around 10%), while OS-PIP is much more demanding, due to larger memory footprint and worse memory locality, which increase the amount of bursts for cache refills.

The second set of metrics is instead more representative of *bus efficiency*; higher values are to be desired. The chart shows that the best efficiency is achieved in OS-PIP, because the high amount of traffic in this benchmark increases the exploitation of AMBA bus pipelining. It is important here to remember that, due to the pipelined nature of AMBA, every access keeps the interconnect busy for a relatively long time, even though ownership may only refer to either the data or address bus. When pipelining is not stressed due to infrequent accesses, this fact heavily reduces apparent efficiency without significantly improving throughput.

Fig.5(b).B reports latencies for read accesses. Six groups of results are shown; the first three ones refer to the number of cycles needed to complete a transaction, while the last three the number of wait cycles before the arrival of the first datum. For each of these sets, average, maximum and minimum times are provided. Since read accesses can be bursts or single, completion times are not strictly correlated to wait times. Minimum waiting times are three cycles, as expected from the specifications (see Section 4). Minimum completion times depend on the benchmark; OS-PIP and ASM-PIP access synchronization slaves (semaphores *etc.*) by means of single read transactions, which can be completed in one cycle after the waiting time (four cycles total). OS-IND and

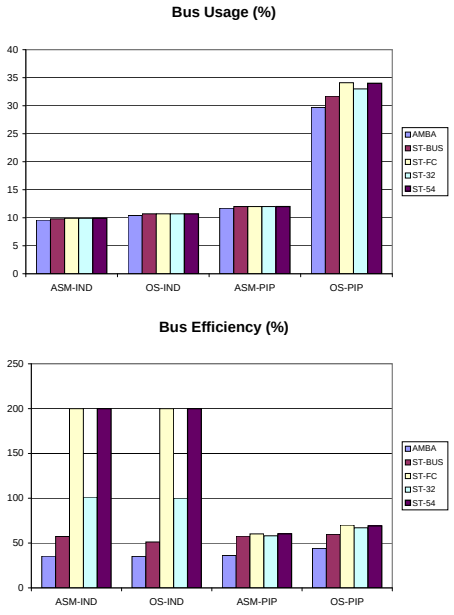

Figure 6. Bus traffic analysis

ASM-IND instead only perform reads when there is a need for cache line refills, and thus all reads are bursts; this means seven cycles added to wait states, totaling ten cycles. Average times reflect what seen in bus usage statistics; highest latencies can be observed for OS-IND, ASM-IND (all reads are bursts) and OS-PIP, which exhibits both a high percentage of bursts and heavy interconnection usage. Finally, maximum times are an interesting metric for worst-case analysis: 34 cycles might be necessary to complete a read transfer.

It is possible to extract similar statistics for writes. However, since ARM does not support burst writes, accesses are always single and latencies are much more aligned, only depending on bus traffic.

5.1. Interconnection comparison

Fig.6.A and 6.B compare bus traffic on interconnections. Overall bus traffic is definitively comparable, since it mostly depends on benchmark features. Bus efficiency, in contrast, is higher for STBus. Since transfers are composed of one wait state followed by a single datum, efficiency could be estimated to be 50%; STBus however is always above that threshold, because, even in its shared bus topology, it has the ability to hide some wait states via its dual request/response channels (refer to Fig.3). AMBA efficiency instead is always below 50%; this is because AMBA AHB appears to be busy not only when its data bus is, but also when its address bus is, which reduces the percentage of actual transfer cycles within the length of a transaction. Efficiency would lie at 50% only if there was a continuous stream of bus transfers filling the AMBA AHB pipeline, which is not the case here, especially in benchmarks with low traffic. ST-FC, ST-32 and ST-54 are able to boost dramatically bus efficiency, since they allow more transfers in parallel. Since accesses to shared devices (shared memory, sempahores, interrupt module) are serialized anyway, the advantage in ASM-PIP and OS-PIP is still relatively small; but in OS-IND and ASM-IND, where all accesses are to private memories, crossbars significantly outperform other schemes. ST-32 achieves 100% efficiency (two memories can be accessed at a time), while ST-54 and ST-FC hit 200% (four memories at a time). It is evident that crossbars behave best when data access is local and no destination conflicts arise.

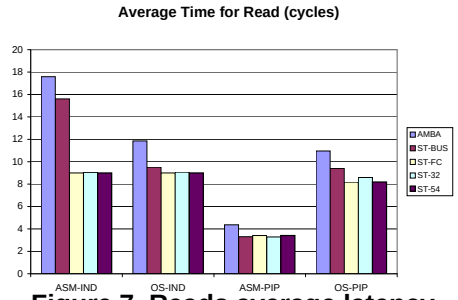

Figure 7. Reads average latency

Fig.7 shows average completion latencies in read accesses. STBus is faster and exhibits lower latencies. ST-BUS has an edge of one to about two cycles over AMBA, mostly due to arbitration (which ST-BUS always performs one cycle faster), and in part also to the ability of sometimes hiding one wait state. Once more, crossbars show a substantial advantage in OS-IND and ASM-IND benchmarks; ST-FC and ST-54, where no conflict on private memories ever arises, both achieve the minimum theoretical latency of nine cycles (all reads are bursts). ST-32 trails immediately behind ST-FC and ST-54 in these benchmarks, with rare conflicts which do not occur sistematically because execution times shift among conflicting processors. OS-PIP still shows significant improvement for crossbar designs. ASM-PIP, in contrast, puts ST-BUS at the same level of crossbars, and sometimes the shared bus even proves slightly faster. This can be explained with the continuous semaphore polling performed by this (and only this) benchmark; while crossbars may have an advantage in private memory accesses, the resulting speedup only gives processors more opportunities to poll the semaphore device, which becomes a bottleneck. Unpredictability of conflict patterns can then explain why a simple shared bus can sometimes slightly outperform crossbars.

We want to stress again that our research is not focused on proving that one interconnect or one topology is better than another, especially since the benchmarks highlight that designs having the highest cost in terms of silicon area outperform the others. Our interest is in understanding performance differences, and the best way to do that is to analyze as accurately as possible bus traffic and interactions. The tool we developed, in contrast to previous approaches, makes no assumptions on traffic and describes interconnect performance in detail, while at the same time allowing thorough exploration of the design space.

5.2. Architectural exploration

The chart in Fig.8 describes the impact of the presence of RTEMS on interconnection performance. Bus traffic spikes up for OS-PIP with respect to ASM-PIP; this is due, as mentioned above, to the larger memory footprint and worse locality of the OS-PIP benchmark. Average latencies for read accesses, not depicted here, show a similar pattern; heavier bus traffic increases them, by up to 150% in the AMBA case.

Fig.9.A shows total execution time of the OS-PIP benchmark, in scenarios having different cache and memory la-

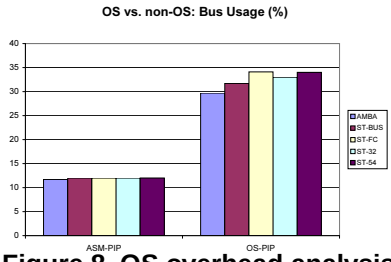

Figure 8. OS overhead analysis

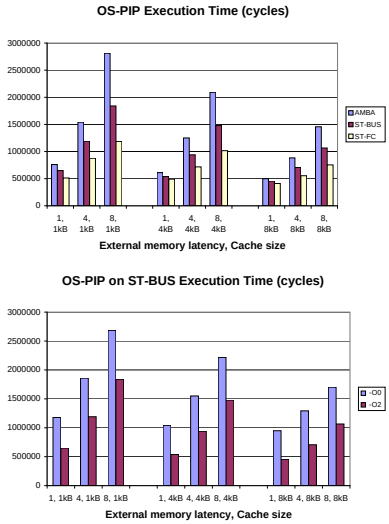

Figure 9. Performance as a function of system variables

tency settings. Cache sizes of 1 kB, 4 kB and 8 kB and memories with 1, 4 and 8 wait states are explored. Three interesting comparisons can be made by looking at this graph. The first is again an analysis of interconnection performance, this time as a function of different environments. As expected, STBus always exhibits an advantage, in that it cuts execution times from 9% to 35% with respect to AMBA (from 18% to 58% with ST-FC). ST-54 (not graphed) performs almost identically to ST-FC, while ST-32 (not graphed) once again trails behind other crossbars, but is still faster than both buses. When comparing more efficient interconnections to less efficient ones, gains are lowest when the traffic is lightest, *i.e.* with big caches and fast memories, but progressively increase with interconnection congestion.

The second analysis regards performance improvement due to cache size. A 4 kB cache can bring 4% to 26% speed-ups in execution time with respect to a 1 kB cache; with 8 kB, speed-ups range from 20% to 48%. The widest gaps, as expected, can be noticed with relatively slow interconnections and high latency memories.

A third assessment that can be made is about memory latency impact on execution times. Increasing memory wait states from 1 to 4 slows down execution times from 35% to 104%, and 8 wait states even from 84% to 370%. This once more stresses the importance of fast memories, even though big caches and fast interconnections help somehow.

Finally, Fig.9.B highlights the impact of software optimization. The chart was computed comparing execution times on ST-BUS of the OS-PIP benchmark, when compiled with the -O0 (unoptimized) flag versus the usual -O2 (optimized). Compiler optimizations, thanks to smarter register allocation policies, cut down on external memory accesses, thus dramatically improving performance. In the ST-BUS case (other interconnections show similar speed-ups), optimized code achieves 31% to 52% lower execution times. In percentage, improvements are more noticeable when bus traffic is already low, *i.e.* in scenarios with big caches and fast memories.

6. Conclusions

In this paper we presented a MPSoC simulator designed to evaluate and compare interconnection architectures at a high level of accuracy; AMBA AHB and STBus were tested as an example of the simulator capabilities. STBus is more complex and powerful than AMBA, and benchmark results confirmed performance expectations. Our simulator proved capable of analyzing in detail similarities and differences between these architectures.

Further developments will extend the platform to other dedicated devices, such as a DMA controller, and to additional system cores, like StrongARM and PowerPC. Other interconnects, in addition to AMBA AHB and STBus, are being ported too; this includes AMBA AXI, multilayer AMBA AHB, and the $\times$pipes NoC. Finally, a linux port onto the platform is currently under consideration.

References

[1] L. Benini and G. Micheli. Networks on chips: a new soc paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[2] A. Brinkmann, J. Niemann, I. Hehemann, D. Langen, M. Porrmann, and U. Ruckert. On-chip interconnects for next generation systems-on-chips. *IEEE ASIC/SOC Conf.*, pages 212–215, September 2002.

[3] M. Gasteier and M. Glesner. Bus-based communication synthesis. *ACM Tran. Des. Autom. Electron. Syst.*, pages 1–11, January 1999.

[4] K. Hines and G. Borriello. Optimizing communication in embedded system cosimulation. *Int. Workshop on Hardware/Software Codesign*, pages 121–125, 1997.

[5] M. Kreutz, L. Carro, A. Zeferino, and A. Susin. Communication architectures for systems-on-chip. *Symposium on Integrated Circuits an Systems Design*, pages 14–19, September 2001.

[6] K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffi c performance characteristics of system-on-chip communication architectures, 2001.

[7] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *IEEE Trans. On CAD of Ics and Systems*, 20(6):768–783, June 2001.

[8] V. Lahtinen, E. Salminen, K. Kuusilinna, and T. Hamalainen. Comparison of synthesized bus and crossbar interconnection architectures. *ISCAS*, pages V433–V436, May 2003.

[9] J. Rowson and A. Sangiovanni-Vincentelli. Interface based design. *DAC*, pages 178–183, June 1997.

[10] RTEMS home page, http://www.rtems.com.

[11] K. K. Ryu, E. Shin, and V. J. Mooney. A comparison of fi ve different multiprocessor soc bus architectures. *EUROMICRO*, pages 202–209, September 2001.

[12] Software ARM, http://www.g141.com/projects/swarm/.

[13] S. M. Xinping Zhu. A hierarchical modeling framework for on-chip communication architectures. In *Proceedings of International Conference on Computer-Aided Design 2002*, November 2002.

[14] T. Yen and W. Wolf. Communication synthesis for distributed embedded systems. *Int. Conf. On CAD*, pages 288–294, November 1995.

[15] C. A. Zeferino, M. E. Kreutz, L. Carro, and A. A. Susin. A study on communication issues for systems-on-chip. *SBCCI*, pages 121–126, September 2002.

[16] Y. Zhang and M. Irwin. Power and performance comparison of crossbars and buses as on-chip interconnect structures. *Asilomar Conference on Signals, Systems and Computers*, 1:378–383, October 1999.