An Integrated Open Framework for Heterogeneous MPSoC Design Space Exploration

Federico Angiolini[1], Jianjiang Ceng[2], Rainer Leupers[2], Federico Ferrari[1], Cesare Ferri[1], and Luca Benini[1]

[1]Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna, 40136 Bologna, Italy

[2]Institute for Integrated Signal Processing Systems, RWTH Aachen University, 52056 Aachen, Germany

ABSTRACT

In recent years, increasing manufacturing density has allowed the development of Multi-Processor Systems-on-Chip (MPSoCs). Application-Specific Instruction Set Processors (ASIPs) stand out as one of the most efficient design paradigms and could be especially effective as SoC computing engines. However, multiple hurdles which are hindering the productivity of SoC designers and researchers must be solved first. Among them, the difficulty of thoroughly exploring the design space by simultaneously sweeping axes like processing elements, memory hierarchies and chip interconnect fabrics. We tackle this challenge by proposing an integrated approach where state-of-the-art platform modeling infrastructures, at the IP core level and at the system level, meet to provide the designer with maximum openness and flexibility in terms of design space exploration. [1]

1. INTRODUCTION

To face increasing architectural design complexity in deep-submicron technology nodes, the reuse-centric paradigm is a natural solution. However, this approach still poses significant design challenges. First of all, general-purpose IP blocks lend themselves very well to quick parallel deployment in a Multi-Processor System-on-Chip (MPSoC), but often do not provide enough performance when running complex user applications, such as multimedia streaming or floating point computation. In fact, depending on the application, dedicated IP blocks could deliver much higher efficiency thanks to task-optimized circuitry. This observation leads to ASIPs (Application Specific Instruction set Processors), *i.e.* to IP cores stemming from the architecture of general-purpose processors but with an instruction set comprising at least some custom instructions optimized to accelerate the task at hand. If designed with state-of-the-art CAD toolchains, ASIPs can provide most of the advantages of dedicated IPs but reduce development time by several times [18] and maintain flexibility.

ASIPs alone cannot unfortunately be a full answer to the SoC design woes. In fact, understanding the performance issues in a multicore system brings the challenge to a new level. One axis of exploration involves memory hierarchies, where both the partitioning among local (*e.g.* caches) and higher-latency memories and the partitioning among private and shared buffers have to be investigated. Another critical point is the interconnection fabric, which must have the lowest possible silicon real estate requirements while managing to comply with the bandwidth and latency requirements of a multiprocessor system. The assessment of such compliance is a non-trivial task, because the fabric is loaded with both explicit (*e.g.* inter-processor messages) and implicit (*e.g.* cache refills) traffic.

It is crucial to notice that the interaction of the above mentioned subsystems (processing, memory, communication) is complex, and therefore that they are not very prone to standalone analysis. For example, the move from general-purpose IP cores to ASIPs with a highly parallel task-specific execution engine is likely to generate more stress for the memory and interconnection subsystems, which may not be able to cope with it. In this case, computing resources would be wasted. On the other hand, a low-latency design with large caches and a fast interconnect could be overkill if the IP cores were not properly sized. Even worse, caching policies may exhibit different and unpredictable performance depending on how specific ASIP instructions are implemented.

These simple examples highlight the need for a thorough exploration of the design alternatives as a way to find global optimal design points. However, this approach generates a huge design space. With traditional CAD development tools, which emphasize the optimization of just one component (*e.g.* the IP core or the cache memory), many feedback loops are required, slowing down design space exploration. Therefore, a *simultaneous* analysis approach, namely a *virtual platform*, should be devised to reduce feedback loops and to improve productivity.

SoCs are the central topic of a huge body of research. Many tools have been developed to explore the SoC design space and to tackle SoC development in the quickest possible way. Usually, two major families of tools can be easily recognized: academic and industrial. They differ in many respects, the main difference being conceptual and related to the different purpose they serve. Research tools are usually open in nature, and experimentation is encouraged and welcome; but not easy. Documentation is minimal, user interfaces are hard to use, and the do-it-yourself approach to problem solving is dominant. In stark contrast, industrial tools support a variety of useful development and verification features, but the typical expected design flow calls for deploying pre-designed and pre-verified blocks, configuring some parameters, maybe adding one or two custom blocks, and testing. This kind of flow is efficient, but does not encourage research and exploration: IP blocks are shipped in encrypted form and their internal architecture cannot be explored or extended.

In the present paper, we will propose a methodology to integrate pre-existing standalone CAD tools in a complete virtual platform, therefore paving the way for faster and more thorough analysis of the available architectural choices. We will explore some of the alternative ways to implement such an integration, defining two wrapping policies aimed at giving different emphasis to the cache design. Subsequently, we will apply our methodology to state-of-the-art CAD tools, such as the commercial LISATek suite [6] and the academic MPARM environment [9]. These tools respectively focus on the seamless development of ASIPs, and on the analysis of system-level issues such as multiprocessor performance and communication assist facilities. Therefore, we feel they are the perfect complement to each other: LISA custom designed IPs can be quickly deployed in the platform environment and en-

[1]This work has been partially supported by the ARTIST2 EU Network of Excellence on Embedded Systems. Authors from Bologna University acknowledge financial support by Semiconductor Research Corporation (SRC) under contract no. 1188 and by STMicroelectronics.

3-9810801-0-6/DATE06 © 2006 EDAA

able further system level exploration. Moreover, both tools adopt a SystemC simulation backbone, which enables a clean integration. Our virtual platform aims at the sweet spot between the industrial and academic approaches. While its LISATek roots guarantee industrial-grade development and debugging facilities, all of the platform code (LISA processing blocks and SystemC interconnects and memories) can be modified at any time for research purposes. Open-source software support is also provided for the required hardware abstraction layers. The result is an open platform where the architecture of each hardware module can be changed, and which is easily extensible by adding new models. As an example of the MPSoC exploration enabled by our platform, we will show cycle-accurate simulations of heterogeneous platforms where cores interact on the interconnect competing for shared resources.

The present work is organized as follows. Section 2 summarizes previous work in the field. Section 3 and Section 4 provide a discussion of the LISATek and MPARM tools we used during our study. Section 5 is about the integration process to get a unified prototyping platform, and Section 6 shows examples of results which can be achieved thanks to this effort. Eventually, Section 7 draws conclusions about our work and proposes future extensions.

2. RELATED WORK

As previously mentioned, many industrial and academic MPSoC virtual platforms have been proposed. Among the industrial ones, Synopsys CoCentric System Studio [22], CoWare ConvergenSC [5], the ARM RealView MaxSim [3] and others [15, 28] spring to mind as some of the most known. They all share a plug-and-play approach of licensed IP blocks whose models are provided in encrypted form. As a result, if the internal architecture is to be investigated and optimized (*e.g.* with the addition of custom instructions or data lanes), alternative open blocks must be written by hand, taking much of the appeal of IP portfolios away. Academic tools are much more heterogeneous in nature, as they are often built with limited resources to test some specific MPSoC aspect, typically the IP cores alone. The SimpleScalar [19] framework stands out for its feature set, but is essentially a single-processor model with an unclear scalability path towards multiprocessor systems. Many other projects exist [27, 24, 25, 26], but their scope seems currently to be too limited for full MPSoC exploration.

A subspace of MPSoC design is covered by existing ASIP design tools. Tools available today can be roughly categorized into three categories, Architecture Description Language (ADL) driven, template architecture based, and predefined component library based. Within the first category, there are tools like EXPRESSION [10], archC [17] or CHESS [23]. However, little information is publicly available about their usage in a heterogeneous MPSoC simulation environment. The work described in this paper uses the LISATek tool suite, which is ADL driven, too. Unlike the ADL driven approach, a partially configurable processor is used as template by the tools in the second category, where Tensilica [4] is a popular representative. ASIPMeister [8] has a predefined library of processor micro-architecture components. So, it falls into the third category.

3. THE LISATEK DESIGN PLATFORM

The LISATek processor design platform is built around the LISA 2.0 ADL [7]. Figure 1 shows the processor design flow supported by LISATek. From a processor model written with the LISA 2.0 ADL, a set of processor development tools such as instruction-set simulator, C-compiler, assembler, and linker are automatically generated to support architecture exploration. A graphical user front-end is also available for software debugging and profiling purposes. Moreover, RTL hardware models in the most popular hardware description languages, VHDL, SystemC and Verilog, can also be generated from the LISA model for hardware implementation. With the LISA platform, the ASIP development time can be greatly reduced compared to the traditional manual approach. Design efficiency is achieved through high degree of automation.

4. THE MPARM ENVIRONMENT

The MPARM [9, 16] environment is designed to investigate the system-level architecture of MPSoC platforms. To be able to fully assess system performance, a cycle-accurate modeling infrastructure is put into place.

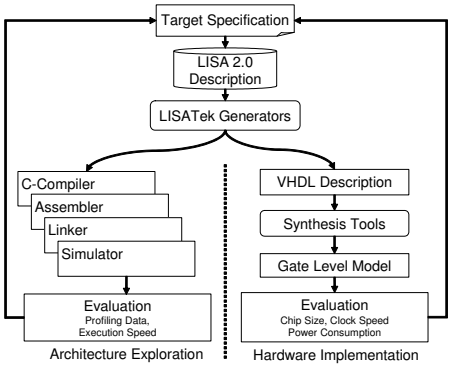

Figure 1: LISATek Based ASIP Design Flow.

MPARM is a plug-and-play platform based upon the SystemC [12] simulation engine, where multiple IP cores and interconnects can be freely mixed and composed. At its core, MPARM is a collection of component models, comprising processors, interconnects, memories and dedicated devices like DMA engines. The user can deploy different system configuration parameters by means of command line switches, which allows for easy scripting of sets of simulation runs. A thorough set of statistics, traces and waveforms can be collected to analyze performance bottlenecks and to debug functional issues. To take into account other crucial design variables, power models for many of the MPARM components are supplied. Frequency and voltage scaling can be realized at runtime thanks to dedicated programmable registers.

Before the work that we are presently showing, MPARM featured a choice of several IP cores to be used as system masters. These models were mostly taken from the open source or academic domain, and while spanning over a range of architectures, they typically modelled pre-existing industrial general purpose processors with little to no possibility of modifying the supported instruction set and architecture. This work will extend the MPARM infrastructure to seamlessly host LISATek devices, therefore giving the designer the freedom to deploy custom IP blocks and to quickly test the performance of architectural variants.

MPARM provides extensive facilities to study the performance of alternative memory hierarchies. Three layers of memory devices are defined: (1) on-tile, strongly coupled to the processor, e.g. caches and ScratchPad Memories (SPMs); (2) on-chip, attached to the system interconnect; (3) off-chip, driven by a DRAM memory controller. In addition, to analyze interprocessor communication behaviour, memories can be defined as private or shared, and a cache snooping mechanism is provided.

In terms of interconnect, MPARM provides a wide choice, spanning across multiple topologies (shared buses, bridged configurations, partial or full crossbars, Networks-on-Chip or NoCs) and both industry-level fabrics (AMBA AHB and AXI [2], STBus [21]) and academic research architectures (×pipes [20]).

On top of the hardware platform, MPARM provides a port of the uClinux [13] and RTEMS [11] operating systems, several benchmarks from domains such as telecommunications and multimedia, and libraries for synchronization and message passing.

5. INTEGRATING ASIP CORES IN A MULTIPROCESSOR SYSTEM

As previously discussed, we believe in the importance of creating a unified virtual prototyping environment, where ASIPs can be tested along with the other SoC subsystems to provide a full assessment of alternative design choices. For this reason, we evaluated the feasibility of embedding LISATek core models within the MPARM platform, devising proper interfacing and wrapping methods.

5.1 The LISATek Simulation Interface

As mentioned in Section 3, the LISATek processor design flow is based on LISA 2.0 processor models. Since the creation of the models exceeds the topic of this paper, it will not be discussed it here. Given a LISA model, the LISATek tool is able to generate instruction-set simulators for the processor under design. Typically, the generated simulator is directly used by the debugger in form of a dynamic library. However, a compiled static simulator library is also generated, and specifications exist to integrate it into the system environment. In our case, the system environment would be the MPARM. All the core models generated by the LISATek suite, regardless of the nature of the ASIP at hand, have the same interface. The interaction is based upon four key pillars:

- The simulated core can be cycled by calling specific functions. If the processor is modelled in an instruction-accurate fashion, then the generated model can be stepped on an instruction basis. On the other hand, a model derived from a cycle-accurate LISA description can be stepped on both instruction and cycle basis.
- Core-initiated communication (*e.g.* reads, writes) is performed through a specific Application Programming Interface (API), which is discussed below. It is the task of the external program to provide an implementation of said API.
- System-initiated communication (*e.g.* interrupts), if any, can be forwarded to the core when cycling it, and therefore on a fine-grain cycle-by-cycle basis, by proper flipping of extra pins. Of course the LISA core model must be made aware of the meaning of these extra pins to take proper action.
- An external LISATek Debugger tool can be interfaced to the core via the IPC (Inter-Process Communication) mechanism. The external program must simply invoke the Debugger with proper references; subsequently, the LISATek model and the Debugger interact autonomously.

While all of these items were implemented during our work, the most interesting for discussion here is the API for core-initiated communication [29]. In a system environment, this LISATek API is the communication interface between the core and the external resources. It must be implemented by the external platform and passed to the processor simulator during system initialization. In addition to some control functions, the API is mainly composed of eight data-related calls:

```
int read(AType addr, DType *data, int n, ...);
int write(AType addr, DType *data, int n, ...);
int request_read(AType addr, DType *data, int n, ...);
int request_write(AType addr, DType *data, int n, ...);
int try_read(AType addr, DType *data, int n, ...);
int could_write(AType addr, DType *data, int n, ...);
int dbg_read(AType addr, DType *data, int n, ...);
int dbg_write(AType addr, DType *data, int n, ...);
```

Three sets of calls, each of which constituting a sub-interface, can be distinguished. The first two calls represent the *blocking* sub-interface: they are based on the assumption that a non-cycle-accurate LISA core may be attached to a cycle-accurate external module. In this case, communication requests which can not be serviced immediately should yield control to the simulator, freezing the caller for as many cycles as needed to complete the transaction. As a result, no concurrent activity can be performed in the LISATek core if a transaction is pending.

The calls from the third to the sixth implement the *non-blocking* sub-interface; it is vital when designing cycle-accurate cores. The `request_read()` or `request_write()` functions are initially invoked; control is always returned. Subsequently, `try_read()` or `could_write()` can be invoked at each clock edge to try to carry the pending transaction on. The return status can be a negative acknowledge (*e.g.* if wait states are needed), but since control is always returned, the core is free to perform other tasks in background, such as shifting its pipeline.

The last two calls of the API are the *debug* sub-interface. Their purpose is to provide an instant reaction, bypassing any wait states. While of course this is not a realistic assumption for a physical system, the calls are extremely useful for debug purposes, such as monitoring or manipulating the content of an external memory while executing a benchmark. They are also useful to load the contents of a memory during the reset cycle.

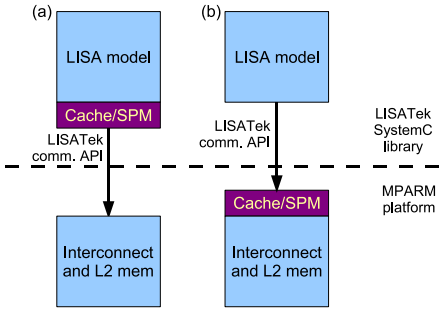

Figure 2: Possible placements of the L1 memory: (a) tightly coupled with the IP core, (b) as a system component.

The implementation of these function calls depends completely on the communication method used in the system; *e.g.* if the simulator needs to work with a system modelled at the RTL level, then the API must be implemented to translate the resource requests into RTL signals. In our case, the implemented API will translate the requests into SystemC signals which can be understood by the MPARM platform. Since MPARM is a cycle- and signal-accurate platform, implementing the first two sub-interfaces was straightforward. The third was supported by directly interfacing with the data arrays which hold the contents of simulated memories. In case caches were present, the implementation was tuned so to take them into account (*e.g.*, writing data to both the cache and the external memory when using write-through policies, and just to the cache when using write-back; or maybe only to the external memory in case of a write miss).

5.2 L1 Memory Placement Strategies

The LISA language makes no assumptions about how to model memory hierarchies. The language allows the specification of cache subsystems, but also permits the implementation of a flat memory array to which all accesses should be directly made. A typical LISA ASIP model is likely to take the second route, for at least two reasons: (i) implementing a complex cache controller is time-consuming, (ii) it is not very meaningful to accurately model a cache if there is no accurate model of the delays associated to an external memory.

The LISATek API mentioned in Section 5.1 is transparent to the presence of caches. In fact, the API can be the *outer* interface of a cache layer, to handle refills and writebacks, as well as the *inner* interface, used by the processor to query a cache controller. Figure 2 illustrates the alternatives.

When integrating the LISATek processor models within MPARM, a choice had to be made regarding the most suitable L1 memory placement strategy. The alternatives were to develop the L1 memories together with each processor, therefore using the LISATek communication API among caches and MPARM; or to develop the L1 memories as an MPARM block, and using the API interface to drive them. Both paradigms allow for cycle-accurate modeling. Tightly coupling the L1 memory to the IP core has the advantage of allowing for arbitrarily complex interactions among the two components. Instead, an external module has the obvious advantage of reuse, where a single cache controller can be seamlessly used by any IP core.

After careful consideration, we went for the second alternative. While the LISATek communication API seems to be flexible enough to support all of the relevant core/cache interactions, thus making it less useful to develop caches inside of each core, we found that the reuse capability, given an equal development time, allows the shared cache module to support more features (different associativity levels, write-back *vs.* write-through policies, snooping capabilities, power optimizations and models), thus becoming more suitable for performance assessment.

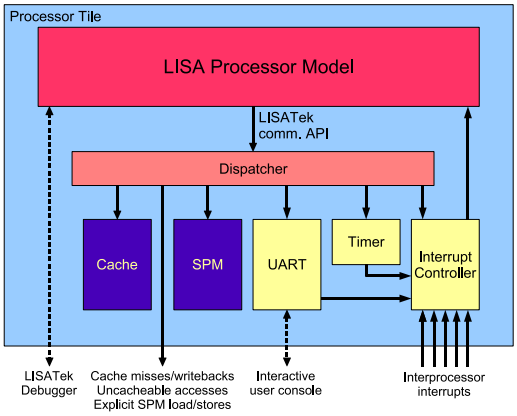

Figure 3: The scheme of a *processor tile*.

While this subsection mostly mentioned cache memories, it is worth stressing that we also made it possible to instantiate an SPM next to (or in place of) them. Since a large body of research exists on how to exploit SPMs to improve embedded system efficiency [1], this adds a further useful degree of freedom for architectural exploration.

5.3 Core-Associated Devices

When developing a shared MPARM block to handle the L1 memory, we also found it useful to cluster other functionality at the same layer. The end result is a *processor tile*, comprising IP cores and the most tightly coupled components (Figure 3). Namely, we developed (i) a timer device, (ii) an emulated serial port, (iii) a simple interrupt controller. The first component is vital if attempting to port an operating system. The second is very useful for debugging purposes; placing it next to IP cores, instead of in a shared location accessible to all system processors, has the advantage of allowing for independent input/output, and prevents debug traffic from spilling onto the system interconnect where it could pollute performance statistics. Finally, the interrupt controller is both a requirement of the other two devices and a crucial component to develop efficient synchronization mechanisms in multiprocessor systems. The controller is externally attached to a set of system-level wires which convey inter-core interrupts. On the IP core side, we implemented a simple interrupt handshaking protocol where the value of interrupt registers is copied on some LISATek core pins which are polled every cycle by the core to take proper action. The interrupt controller is memory mapped, to let the core reset the pending interrupt flags and configure the masking status.

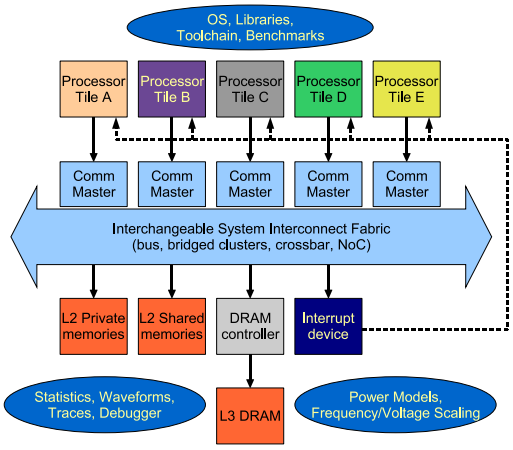

Figure 4: The overall system architecture.

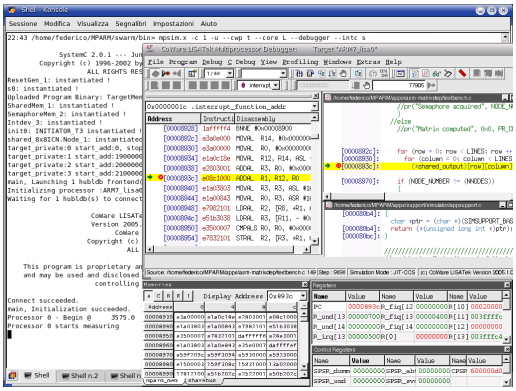

Figure 5: A system simulation screenshot.

5.4 The Resulting System Architecture

The MPARM architecture is layered, to flexibly accommodate for different master devices and interconnect models. As can be seen in Figure 4, the IP core tiles talk to a master device, which is in charge of handling any arbitration and/or routing phases required by the specific underlying fabric. The tile/master interface can either be MPARM-custom or comply with the OCP 2.0 [14] specifications.

The MPARM facilities allow the designer to flexibly instantiate complex platforms. Homogeneous as well as mixed processing tiles can be deployed, and selection among them is as simple as flipping a command line switch. Specific MPARM modules exist to handle addressing maps and to track simulation statistics, including cache hit rates, interconnect congestion and latencies, memory access patterns and (for the components for which a model is available) power consumption. In addition, the graphical LISATek Debugger can be launched to interactively inspect the status of each LISATek core, to set breakpoints and watchdogs, and to manually control the flow of execution.

6. EXPERIMENTAL RESULTS

In this section, we will demonstrate that we implemented a fully working and usable solution, and we will show a sample of the kind of analysis that can be performed on our combined platform.

In order to achieve the former objective, we implemented a LISA ARMv7 core, which is instruction-equivalent to another ARMv7 core that was already available in MPARM. Since the cycle accuracy of the core itself was not important for our purposes, we kept its model very simple (no pipeline) without any timing accuracy effort. The expected result was complete functionality of the system platform. This achievement is testified by the screenshot in Figure 5: the LISATek Debugger, in the foreground, is attached to a LISA core (currently paused on a breakpoint) that runs within MPARM. The console of the latter can be seen in the background. As a secondary result, by exercising the two ARMv7 implementations with the very same benchmark binary, we expected to find a perfect equivalence in the amount of memory accesses. Across several microbenchmarks and functional benchmarks from the multimedia and data encryption domains, including applications which leverage the RTEMS operating system, this result was indeed confirmed. On the other hand, we noticed a discrepancy of about 30% in the amount of execution cycles - which is perfectly normal due to the fact that the LISA ARMv7 model did not include a pipeline, while the MPARM one did. With LISA cores, we recorded up to 200k global simulated CPU cycles/second on an Athlon XP 2200+ machine with 512 MB of RAM.

Next, we prove the importance of being able to model the effect of memory hierarchies and interconnect congestion on system performance. The choice of an instruction-accurate ARM model does not prevent cycle-accurate exploration of the impact of the communication fabric. Figure 6 shows the execution time when a variable amount of IP cores, each of them independently performing the same benchmark, is attached to the interconnect. Each core

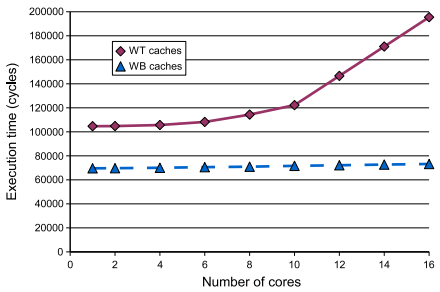

Figure 6: Performance *vs.* interconnect congestion.

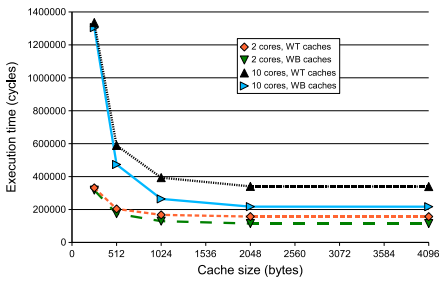

Figure 7: Performance *vs.* cache size.

executes an additional chunk of processing, therefore an increasing requirement of communication bandwidth is depicted (*i.e.* no parallelization). In absence of bus contention, execution times are expected to remain constant, as all cores operate simultaneously. A cache is interposed and configured with two alternative policies, namely Write-Back (WB; writes go to cache only, and are copied back in memory only when the cache line is evicted) and Write-Through (WT; writes always go to both cache and memory). The WB policy is clearly minimizing the amount of traffic which spills on the interconnect, but at the cost of additional complexity in the cache controller (*dirty bits* have to be tracked). The advantage of WB, which may not be fully clear when designing the IP core alone, is evident here. With WT caches, six or more processors are enough to congest an AMBA AHB interconnect, causing a progressive performance degradation. With WB caches, the amount of writes on the bus is drastically lower, and up to sixteen cores can be attached to the same fabric without significant bottleneck effects.

Subsequently, just by changing a command line parameter, we repeated the same experiment with varying cache sizes (Figure 7). As the chart shows, bigger caches help performance, but under low interconnect congestion (few IP cores on the bus and/or WB caches), their impact is much less than under high congestion.

Since not all interconnect architectures have the same performance, we tested two shared buses using completely different protocols: AMBA AHB by ARM and STBus by STMicroelectronics. STBus is much more complex and offers more features, such as multiple simultaneous outstanding transactions. The results confirm this fact, as STBus is offering up to a 33% boost to overall system performance under high congestion. Performance is much closer when operating under light loads.

To further showcase possible design space scenarios, we created a mixed platform with one LISA ARMv7 core and one or more LISA FFT coprocessors. The latter devices were designed to optimally accomplish a specific task, namely a Fast Fourier Transform. Since they internally perform parallel computation (Figure 9), they feature high bandwidth requirements and contribute heavily to bus congestion. Figure 10 plots the latency, as seen by the ARM core, to complete bus transactions when increasing numbers of FFT cores are working in the background. A steep latency increase can be noticed, prompting the designer to quantify the amount of communication resources needed for the deployment of FFT coprocessors.

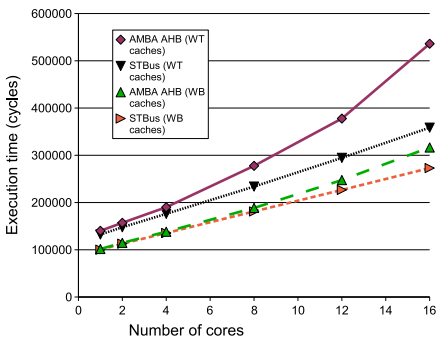

Figure 8: System performance with different interconnect fabrics.

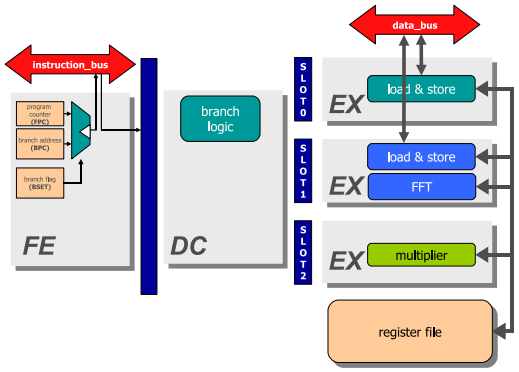

Figure 9: A 3-Slot VLIW FFT Processor.

To highlight how the availability of a full platform, including memory hierarchies and interconnect models, enables the study of non-trivial effects in MPSoC systems, we show in Figure 11 the polling behaviour of a DES encryption benchmark, where two control tasks (*initiator* and *terminator*) supply and collect chunks of raw data to a variable amount of parallel *worker* tasks that perform the actual encryption/decryption. We tested the system with one to six worker tasks, each running on a different LISA core. The worker tasks have to synchronize with the initiator and terminator tasks by semaphore polling before being able to exchange data chunks. The plot depicts the overall amount of system polling as a function of varying frequencies of polling executed by the initiator and terminator tasks. With few workers, the workload is very unbalanced (the initiator and terminator tasks have comparatively little to do) and configuring them for frequent polling

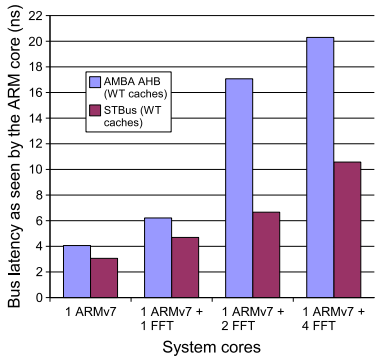

Figure 10: Bus latency of a mixed ARM + FFT platform.

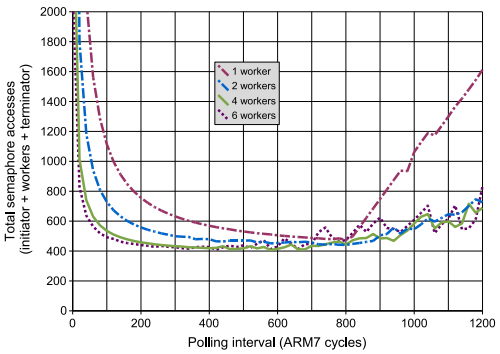

Figure 11: Polling behavior in the DES benchmark.

is only a waste of interconnect bandwidth. As more workers are added, frequent polling becomes increasingly useful because more data chunks have to be distributed and collected per time unit. If the polling interval of the initiator and terminator becomes too wide, roles reverse, and it is the worker tasks which have to perform heavy polling before exchanging data. Therefore, the global polling amount increases again. The case with a single worker has the rightmost knee point (the control tasks are very lightly loaded, and can afford sparse semaphore checks) but the highest absolute polling amounts (the chance of hitting optimal synchronization points without much polling is very low). When the polling interval becomes large, all lines exhibit a shaky trend, because randomly missed synchronization points imply a long wait before the next semaphore release event and long strings of polling activity.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a CAD tool methodology which integrates a state-of-the-art ASIP toolchain within a full-featured virtual platform. The mix of these two environments enables SoC designers to sweep all axes of the configuration space by getting immediate feedback about the impact that one architectural feature has on the rest of the system. IP cores, memory hierarchies and interconnects are simultaneously under scrutiny. Effects which are difficult to predict, or at least to quantify, when operating from the IP core designer perspective alone can now be easily and thoroughly investigated.

Moreover, the designer is given a full choice of debug and inspection facilities, thanks to the respective strengths of the LISATek and MPARM worlds. Runtime debuggers with graphical interfaces, execution traces, waveforms and full statistics are available. The virtual platform offers some of the highlights typical of industrial-strength products together with an unmatched openness of the simulation models and a rich feature set. The low-level software stack (such as the required compilation toolchains) is part of the package.

Future work will leverage this technology to shed light on long-standing open problems, such as assessment of the performance of alternative hardware communication assists for MPSoCs and accurate analysis of the tradeoffs implied by the ASIP/coprocessor paradigm a the system level.

8. REFERENCES

[1] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 259–267, 2004.

[2] ARM Ltd. The Advanced Microcontroller Bus Architecture (AMBA) homepage. www.arm.com/products/solutions/AMBAHomePage.html.

[3] ARM Ltd. RealView MaxSim. www.arm.com/products/DevTools/MaxSim.html.

[4] N. Cheung, J. Henkel, and S. Parameswaran. Rapid configuration and instruction selection for an asip: A case study. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10802, Washington, DC, USA, 2003. IEEE Computer Society.

[5] CoWare Inc. ConvergenSC. www.coware.com/products/.

[6] CoWare Inc. LISATek. www.coware.com/products/.

[7] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[8] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai. Effectiveness of the ASIP design system PEAS-III in design of pipelined processors. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 649–654, New York, NY, USA, 2001. ACM Press.

[9] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004.

[10] P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 256–261, New York, NY, USA, 2001. ACM Press.

[11] Real-Time Operating System for Multiprocessor Systems (RTEMS). www.rtems.com.

[12] The SystemC initiative. www.systemc.org.

[13] The uClinux embedded linux/microcontroller project. www.uclinux.org.

[14] Open Core Protocol Specification, Release 2.0. www.ocpip.org, 2003.

[15] Prosilog. Magillem. www.prosilog.com.

[16] M. Ruggiero, F. Angiolini, F. Poletti, D. Bertozzi, L. Benini, and R. Zafalon. Scalability analysis of evolving SoC interconnect protocols. In *Proceedings of the 2004 International Symposium on System-on-Chip*, pages 169–172, 2004.

[17] M. B. Sandro Rigo, Guido Araujo and R. Azevedo. Archc: A systemc-based architecture description language. In *to appear in the 16th Symposium on Computer Architecture and High Performance Computing - Foz do Iguacu, Brazil*, October 2004.

[18] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, and A. Nohl. RTL processor synthesis for architecture exploration and implementation. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*, pages 156–160. IEEE, 2004.

[19] SimpleScalar LLC. SimpleScalar. www.simplescalar.com.

[20] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. D. Micheli. ×pipes Lite: A synthesis oriented design library for networks on chips. In *Proceedings of the 2005 Design, Automation and Test in Europe Conference (DATE'05)*, pages 1188–1193. IEEE, 2005.

[21] STMicroelectronics. The STBus interconnect. www.st.com.

[22] Synopsys Inc. System Studio. www.synopsys.org.

[23] Target Compiler Technologies N.V. The Chess/Checker Retargetable DSP Environment. www.retarget.com.

[24] The Liberty Research Group. The Liberty Simulation Environment. http://liberty.princeton.edu/.

[25] The MicroLib Community. MicroLib. http://microlib.org/.

[26] The SkyEye Community. SkyEye. www.skyeye.org.

[27] The SoCLib Partners. SoCLib. http://soclib.lip6.fr.

[28] Virtio. Virtual Platforms. www.virtio.com.

[29] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*, pages 1256–1263. IEEE, 2004.