A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms

Andreas Wieferink, Tim Kogel,
Rainer Leupers, Gerd Ascheid, Heinrich Meyr
Institute for Integrated Signal Processing Systems
Aachen University of Technology, Germany
wieferink,kogel,leupers,ascheid,meyr@iss.rwth-aachen.de

Gunnar Braun, Achim Nohl
CoWare, Inc.
San Jose, CA, USA
gunnar,achim@coware.com

Abstract

Current and future SoC designs will contain an increasing number of heterogeneous programmable units combined with a complex communication architecture to meet flexibility, performance and cost constraints. Designing such a heterogenous MP-SoC architecture bears enormous potential for optimization, but requires a system-level design environment and methodology to evaluate architectural alternatives. This paper proposes a methodology to jointly design and optimize the processor architecture together with the on-chip communication based on the LISA Processor Design Platform in combination with SystemC Transaction Level Models. The proposed methodology advocates a successive refinement flow of the system-level models of both the processor cores and the communication architecture. This allows design decisions based on the best modeling efficiency, accuracy and simulation performance possible on the respective abstraction level. The effectiveness of our approach is demonstrated by the exemplary design of a dual-processor JPEG decoding system.

1. Introduction

The emerging platform based design paradigm poses enormous challenges to conceptualize, implement, verify and program today's complex SoC designs. System architects are obliged to employ heterogeneous computational fabrics to meet conflicting requirements with respect to performance, flexibility and power efficiency. The complexity problem of increased heterogeneity also applies for the on-chip communication: already today numerous buses of different types are necessary to cope with the escalating data traffic [1]. In the near future we will even see full scale Networks-on-Chip to provide the required performance as well as Quality-of-Service (QoS) [2, 3].

In the predominant industrial practice, design of complex and programmable SoCs still follows the traditional flow, where the textual architecture specification phase is followed by sequential and mostly decoupled implementation of HW and SW parts. Finally processors, buses, memories and dedicated logic blocks are integrated on the HDL based implementation level. The high complexity and poor simulation speed on this level prohibits consideration of architectural tradeoffs, and integration of the embedded software has to wait even until the silicon is available. Due to the lack of early system integration, complex SoC's are either over-designed or fail to fulfill the specified performance requirements. Especially, the performance related to the SW part, like e.g. CPU load, impact of the RTOS, or SW response time, can hardly be analyzed without a cycle accurate simulation model.

Furthermore, the SW performance is heavily affected by the shared communication and memory architecture, thus an isolated consideration of a single processor may hide potential bottlenecks due to bus utilization and memory access latency. Such issues have to be addressed as early in the design flow as possible to prevent from late and costly changes of the architecture specification.

We propose an iterative *system level* processor/communication co-exploration methodology depicted in figure 1, where the Architecture Description Language (ADL) LISA [4] is used for the system-level description of processor architectures and SystemC [5] based cycle-accurate Transaction Level Modeling (TLM) [6] captures the communication architecture and further peripheral devices.

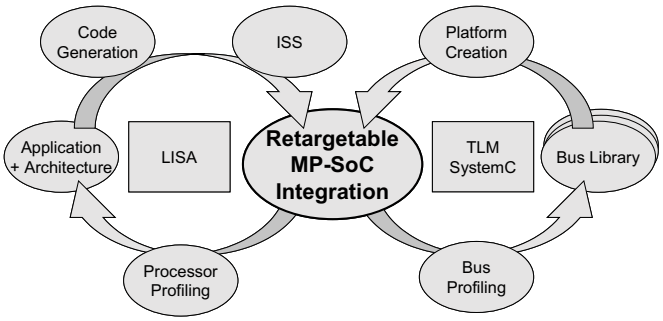

Figure 1. Co-exploration methodology

As depicted on the left hand side of figure 1, the enabling technology for our approach is the LISA based processor exploration environment, that provides capabilities for automatic generation of code generation tools (C-compiler, assembler, linker) as well as the Instruction-Set Simulator (ISS) and the profiling environment. As shown on the right hand side of figure 1, we rely on a SystemC library of existing bus architectures with standardized interfaces [7, 8], which can be easily assembled to a communication platform instance.

As further discussed in section 3, both system level modeling languages advocate a successive refinement flow: LISA fosters instruction and cycle accurate modeling of arbitrary programmable cores. On the other hand, appropriate SystemC TLM communication models feature near cycle accu-

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

rate *transaction layer* interfaces as well as fully cycle accurate *transfer layer* interfaces [7].

The major contribution of this paper is to provide a *retargetable* integration of arbitrary LISA based processor models with SystemC based communication platform models. The corresponding co-exploration methodology advocates a joint top-down refinement and an iterative profiling driven optimization of heterogeneous Multi-Processor SoC's.

The rest of the paper is organized as follows: After discussing the related work in section 2, we give an introduction to system-level modeling of processor and communication architectures in section 3. The main body of the paper in section 4 presents the successive refinement flow for joint processor/communication optimization. A case study of a complex dual processor JPEG decoding system is presented in section 5. Finally, section 6 concludes this work and gives a short outlook on future research topics.

2. Related Work

Traditionally, mixed HW/SW-systems are not integrated and co-simulated until the synthesizable Register Transfer Level (RTL) description of the complete HW part is available [9]. Connecting multiple Instruction-Set Simulators (ISS) to an HDL simulator via Inter Process Communication (IPC) enables verification of HW/SW interfaces. However, the low simulation speed is prohibitive for architectural exploration and embedded SW development.

Communication models on multiple levels of abstraction for early processor integration have been proposed since a long time to overcome the simulation performance and modeling efficiency bottleneck [10, 11]. This principle is now leveraged by the TLM paradigm [6] by providing standardized, system level bus-interfaces with different levels of abstraction [7, 8]. Based on the SystemC language as the emerging EDA standard for system-level design, SystemC TLM communication models are supported by a new generation of Electronic System Level (ESL) SoC design tools [12, 13].

Recent research projects in both academia [14] as well as industry [15] predominantly propose system level MP-SoC integration according the component-based design principle. This bottom-up approach enables rapid composition of complex platforms from basic IP components, but, on the other hand, imposes major drawbacks with respect to flexibility and accuracy: First, the fixed set of available Common-Of-The-Shelf (COTS) processor IP limits the potential for joint optimization of the communication and processor architecture. Second, pure IP integration frameworks rely on Instruction Set Simulators, which need to be furnished with Bus Function Models (BFMs) to create a bus cycle accurate behavior. As already proven by Guerra et al. [16], BFMs fail to provide full cycle accuracy for complex pipelined architectures.

To overcome these drawbacks, we propose a top-down refinement flow, where the processor architecture is jointly optimized with the communication architecture and successively refined to full cycle accuracy. Such a methodology can only be realized with processor exploration frameworks based on an ADL like ISDL [17], EXPRESSION [18], nML [19], MIMOLA [20]. However, to our knowledge none of the existing frameworks provides any support for retargetable MP-SoC integration at the system level.

3. Abstraction Levels for Processor and On-Chip Communication Models

As shown in the table below, functionality for a System-on-Chip can be modeled on a multitude of abstraction levels. An abstraction level is determined by the accuracy of communication, data, time and software modeling. Our system level co-exploration methodology is addressing the bold faced abstraction levels, where the application SW is executed on the target processor model. In this section, we first describe the abstraction levels relevant for our approach separately for processor modules and the intermodule communication, before we introduce our unique retargetable integration technology.

communic. accuracy	data accuracy	timing accuracy	software accuracy
TLM	packets	un-timed	functional specification
		timed	
	bytes/ words	**transaction**	**instr. accurate ISS**
		transfer	**cycle accurate ISS**
RTL	bitvectors	cycle	HDL processor model

3.1. Processor Model

Using the LISA ADL, processor architectures can be modeled on two major levels of abstraction: instruction accurate (IA) and cycle accurate (CA) models.

Instruction accurate models comprise the full processor instruction set, and the generated processor simulator sequentially executes assembly instructions. Every instruction of a LISA processor model is composed from a number of (atomic) operations [1]. If such an operation needs to access the processor memory, in an instruction accurate model this is efficiently modeled via the *ideal* or the *functional* memory interface [21]. Both interfaces basically offer two methods to the processor: one for read and one for write access. To enable easy processor modeling on this abstraction level, these interface methods are expected to return successfully in any case. In a standalone processor simulator, the requested access is performed immediately. A functional memory model can additionally provide accumulated latency information for every access.

The instruction accurate model is generally not cycle accurate since pipeline effects are not considered at all. In contrast, cycle accurate processor models fully simulate the processor pipeline by assigning the individual operations of the instruction to the respective pipeline stages and by implementing an operation scheduling according to the processors execution scheme. Generally moving from an IA to a CA model implies a restructuring of the model. Fortunately, the memory interface does not need to be modified. In this first refinement step, memory access statements just are assigned to a suitable pipeline stage as a whole.

Unfortunately, real memories are non-ideal, thus will take at least one clock cycle to provide a requested data value in case of a read access. It is possible to hide at least portions of these latencies by distributing a memory access over several processor pipeline stages. To be able to model this correctly, the *cycle accurate* memory interface is applied. It offers separated methods for controlling the different phases of a single memory access, which then can be invoked from the respective processor pipeline stage. If the memory access latency is too long to be hidden completely, e.g. because of a cache miss, the processor can dynamically react by inserting stall cycles.

[1] This allows describing the behavior of a regular instruction set efficiently

3.2. Communication Modeling

As important as the SoC modules themselves is the communication between them. Possible communication topologies are point-to-point, bus based, or generic network-on-chip architectures [22]. The latter two can be modeled as dedicated communication modules, having point-to-point connections to the principal SoC modules.

The point-to-point communication between these generalized SoC modules can be modeled on different abstraction levels during the SoC design flow. The main distinction here is done concerning pin accuracy. Fully pin accurate communication models are generally also data and cycle accurate. The traditional RT-Level models communication on this level.

To cope with the complexity of current and future SoC designs, in the initial phases of the design flow, communication is modeled on a much higher level of abstraction. The *Transaction Level Modeling* (TLM) Communication waives modeling the pin wiggling in detail, but controls the inter module communication by *Interface Method Calls* (IMC) between the modules instead. This drastically increases simulation speed and modeling efficiency and still leaves full freedom concerning data and temporal accuracy by offering interface methods on the respective granularity level.

During early design phases, *transaction* TLM is applied, which uses more general IMC functions to initiate a full transaction by a single function call. The *transfer* TLM, in contrast, enables cycle accurate modeling by performing IMC function calls every clock cycle during an active communication. By that, every single transfer of the full transaction is explicitely invoked.

3.3. Retargetable Processor Integration

The previous sections, each on its own, represent prior art. The software module refinement flow outlined in section 3.1 is well described in [21]. It uses the generic LISA memory API, offering an ideal, a functional and a cycle accurate interface. On the SoC communication side, there are several publications about manually developing and refining TLM and RTL SoC modules in combination with their intercommunication [7, 8]. For that, bus model specific TLM APIs are applied.

This paper brings together the automatically generated processor simulators with the SoC bus models. To map the generic LISA API to the respective TLM bus API, we implemented a bus interface class library. For the available abstraction levels, the generic LISA memory API functions are implemented to map the memory accesses onto the TLM bus API the processor currently is interfacing to (Fig. 2).

The ideal LISA interface is needed for debugging purposes only and can directly be mapped to the respective debug interface of the bus model the processor is connected to.

The functional LISA interface has to make sure that all SoC modules stay synchronized to each other and also that the LISA behavioral model still can rely on successful memory accesses even in case of memory resources external to the respective processor module. Ensuring this, there are two types of bus module interfaces the LISA functional API call can be mapped to: the ideal (debug-) interface as well as the blocking (transaction TLM) interface (Fig. 2). The ideal interface performs the memory access without delays. In contrast, a blocking interface function call advances system simulation time as far as necessary to succeed with the function call. During this time, the respective processor simulator is suspended. In terms of a SystemC system simulation, the IMC implementation executes $wait()$ until the memory access actually is done. This includes time for bus arbitration, memory access request and memory device reply; in case of cache misses also for the full access for the next layer memory accesses.

The cycle accurate LISA interface is mapped onto the transfer TLM API of the respective bus. It may be called non-blocking interface since none of these interface calls can "block" the processor, which means advance system simulation time (in any SoC module) before returning control back to the processor.

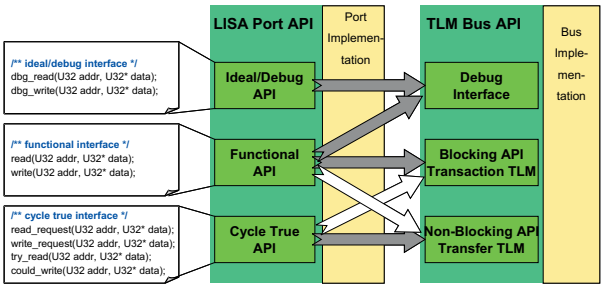

Figure 2. API mapping: LISA → TLM

4. The System Integration Design Flow

Having the above abstraction levels available for SoC modules as well as for their intercommunication, this section describes a 6-phase flow to successively refine the processor modules and their communication (Fig. 3). While keeping one side constant, the refinement of the other side can directly be verified for correctness.

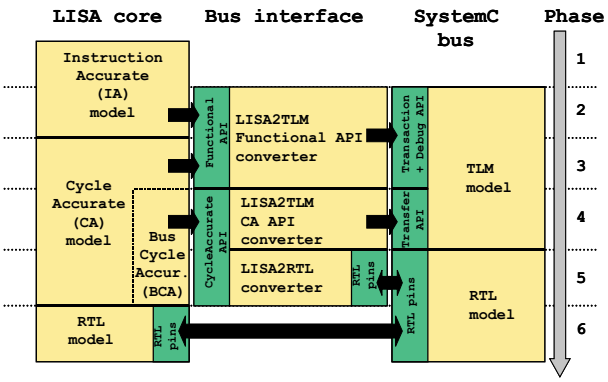

Figure 3. System integration flow phases

4.1. Phase 1: IA ASIP standalone

The flow starts with an instruction accurate processor module as outlined in [21]. The generated standalone processor simulator uses internal memory and bus resources and does not get any stimuli on its input pins. In the resource definition of a LISA model, the HW structure of the processor is defined. Several memory resources (RAM, ROM, cache, write-buffer) as well as buses can be freely instantiated and parameterized using pre-defined or user-defined models (Fig. 4). The resource models applicable here all implement the same generic LISA memory API.

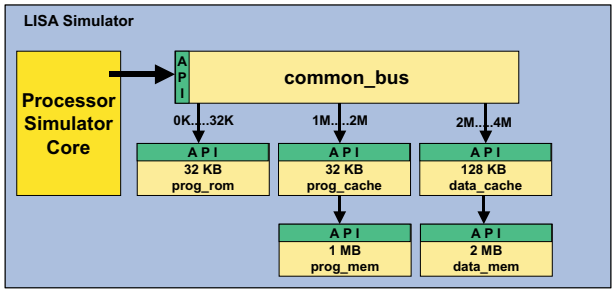

Figure 4. Standalone processor simulator

Additionally, a memory map is developed which defines the interconnections and bus address ranges of these modules. By that, the LISA behavioral description only needs to access the respective bus object, which contains the necessary intelligence to perform the address decoding and the communication with the selected slave module autonomously (Fig. 4).

Such an instruction accurate processor model can be developed quickly, but can only give a raw estimation of the number of cycles n_{cycles} consumed by a given application. With n_{instr} being the number of instructions executed by the application and n_{lat} being the latency accumulated by the memory accesses, the estimation is:

$$n_{cycles} = n_{instr} \cdot CPI_{int} + \sum n_{lat} \quad (1)$$

$$= n_{instr} \cdot CPI_{ext} \quad (2)$$

CPI_{int} is the average number of cycles per instruction depending only on control hazards in the pipeline itself, while CPI_{ext} also considers cycles additionally lost due to non-ideal memory accesses.

Although n_{instr} can be determined exactly already in this phase, n_{cycles} is still inaccurate. First, there is no information about CPI_{int}; a common estimation is the best case value $CPI_{int} = 1$. Second, the latencies n_{lat} caused by the memory accesses do not consider parallelism at all on this abstraction level. On the one side, parallelism within the processor may hide a portion of the latencies contributing to $\sum n_{lat}$. On the other side, in the real SoC system, the processor in general does not have exclusive access to its buses and memories as assumed in standalone simulation. The competition for resource access can drastically increase the latencies $\sum n_{lat}$.

4.2. Phase 2: IA ASIP → Transaction Bus

As soon as possible, a SoC module should be integrated into the system context. This allows, first, considering multi-processor and resource-sharing influences on access latencies early, as well as, second, verifying the processor in the final environment with realistic stimuli from outside.

This is possible by activating the newly developed TLM bus-interface capabilities when generating a LISA processor simulator. By that, every bus also gets a link to outside the LISA model where it can direct the read and write requests to (Fig. 5). Every resource connected to a bus can (dynamically) be configured to be modeled inside the LISA model (as it is the only option in standalone simulator), or its accesses to be directed to a port class forwarding the accesses to whatever has registered there.

A bus access invoked by a LISA operation first does the address decoding as before, but now the enhanced bus module

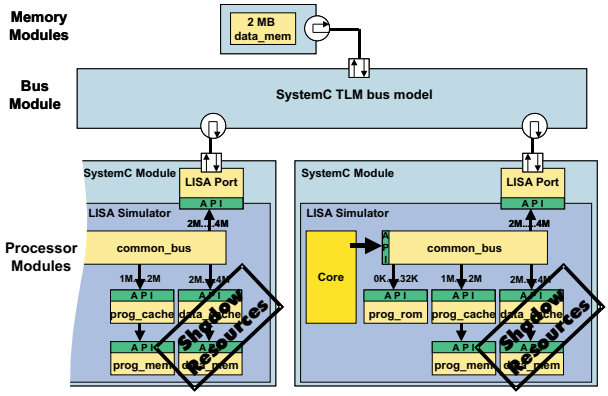

Figure 5. SystemC processor simulators with external data memory

first consults the identified internal module whether it is modeled inside the LISA model (e.g. for simulation performance reasons), or is a $shadow\ resource$, meaning the real resource is located outside the LISA simulator, thus the bus address is to be forwarded to the external port instead (Fig. 5). The bus port then translates the request to the environment whatever the instruction set simulator is embedded into (sect. 3.3).

Without modifying the processor model at all, redirecting the accesses to outside already gives a valuable upper and lower bound of the performance to be expected in system context. Referring to equation (1), the accuracy of the $\sum n_{lat}$ has significantly improved since shared resource and multi-processor issues are now considered. By mapping the accesses to the debug TLM interface (Fig. 2), a best case cycle count n_{cycles} is obtained since all additional latencies are neglected ($\sum n_{lat} = 0$). In contrast, when mapping the functional interface to the blocking (transaction) TLM interface, we get a rough worst case estimation for $\sum n_{lat}$ since the whole processor simulator is suspended during every memory access instead of allowing parallelism within the processor. Additionally, processor pipeline effects are not yet considered in both cases, thus we still assume the optimistic case $CPI_{int} = 1$.

However, this estimation is already relatively good and obtained almost 'for free', without modifying the instruction accurate processor model developed in section 4.1. In the following subsection, the processor model is refined for more accuracy, leaving the system environment as is.

4.3. Phase 3: CA ASIP → Transaction TLM Bus

Having identified instructions senseful to be implemented in the SoC module, the instruction accurate model can be refined to a cycle accurate model to gain some more detailed information about the timing behavior.

Despite of the restructuring of the model due to the pipeline presence, the memory interface is not yet modified. This model can be applied standalone (as done in [21]), but more interesting is applying this model again in the same system simulation context as before.

When applying the blocking interface again, every memory access, independent in which pipeline stage it is invoked, will block the whole model. The functional behavior still is correct, since every external memory access will guarantee to return the correct value.

Referring to equation (1) again, from this phase on, we obtain an accurate value for CPI_{int}. The only inaccuracy is caused by the term $\sum n_{lat}$ because still memory access concurrency is not modeled correctly.

4.4. Phase 4: CA ASIP → Transfer TLM Bus

To get more accurate information about the timing behavior, not only the processor itself, but also the communication to the outside has to be modeled cycle accurate (BCA = bus cycle accurate). As already proposed in [21], the LISA bus access has to be refined to employ the cycle accurate LISA memory interface instead of the functional one. The functional correctness first can be ensured in a standalone environment again, but as soon as possible it should be applied in the SoC environment again such that also multi-processor/multi-master influences and shared resources can be considered properly.

From now on, cycles that so far possibly have multiple times contributed to $\sum n_{lat}$ or CPI_{int} in equation (1) are properly considered in $n_{instr} \cdot CPI_{ext}$ (equation (2)). Thus, on this abstraction level simulation does not have temporal inaccuracy concerning the cycle count n_{cycles} any more.

4.5. Phase 5: CA ASIP → RTL Bus

Having fully reached cycle accurate abstraction level, it soon is desired also to switch to pin accurate models. We propose first replacing the TLM bus model with an RTL bus model. The LISA model then can be equipped with a LISA2RTL converter which translates the LISA memory API calls into the respective RTL pin wiggling instead of forwarding them to a TLM port. Thus, without modifying the LISA processor models, an RT-Level bus model can be introduced and verified with realistic pin true stimuli.

4.6. Phase 6: RTL ASIP → RTL Bus

Finally also the LISA processor needs to be replaced by an RTL processor model. A methodology to automatically generate an RTL model out of a LISA model has been presented in detail in [23] and is far beyond the scope of this paper.

5. Case Study

The outlined design flow has been applied to the development of a dual processor JPEG decryption system. The platform has been created and simulated in the CoWare ConvergenSC [13, 24] environment.

This section first describes the target application, as well as two alternative SoC topologies. To estimate the simulation performance and the accuracy of the more abstract models, we use the same application code for all comparisons.

5.1. JPEG decryption system

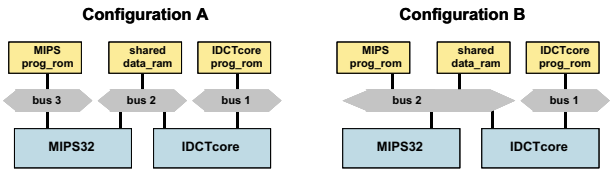

Figure 6. SoC topologies

The first four design and exploration steps have been verified and profiled on a JPEG-2000 decompression algorithm decoding a 150x100 sized bitmap. Both the C original source code of the algorithm and the sample bitmap are freely available from the JPEG group's web page [25]. We modified the code to run it jointly on two processor cores: First, a specially designed IDCTcore coprocessor for calculating the inverse discrete cosines transformation (idct), which is the algorithmic core of JPEG decryption. Second, a MIPS32 processor core performs the remaining, i.e., the control and file I/O tasks. Both processors share a common data memory, and each of them accesses a personal program memory. The bus topologies A,B considered in this section are shown in Fig. 6.

5.2. Simulation Performance

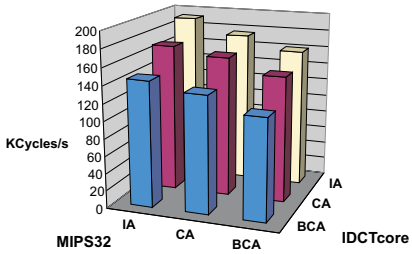

Figure 7. SoC simulation performance

The final SoC platforms A,B have been assembled and simulated on different levels of abstraction already during the exploration phase. As can be seen in Fig. 7 for topology A with three AMBA AHB bus instances, SoC processor modules having different accuracy levels [2] can freely be combined.

On our AMD Athlon XP 2000+ equipped RedHat Linux 8 host, for any combination of SoC processor models, the simulation performance is between 100 and 200 kCycles per second. The formerly huge performance differences between the respective standalone processor simulators weaken drastically because their portion on the overall SoC simulation effort is lower; the SystemC scheduler and the TLM bus models also consume a significant - and constant - amount of simulation time. Additionally, due to the optimized JIT-CC [26] simulation technique, processor pipelines can be simulated quite efficiently, which lowers the abstraction level impact already on standalone simulation performance to less than one order of magnitude. The range is 1.2 ... 8.0 MCycles/s for the IDCTcore and 440 ... 1000 kCycles/s in case of the MIPS32.

5.3. Simulation Accuracy

To estimate the accuracy already available on the higher abstraction level models, we have taken the bus cycle accurate (BCA) SoC model (phase 4) as a reference, and compared it to the respective models on the two earlier phases (IA, CA). To get more compact and reliable accuracy information, in this section every SoC platform contains processor models of the same accuracy level only. Since the dual processor JPEG application does not run on a single processor due to the lack of the full set of stimuli, we do not take the standalone models of phase 1 into account for this comparison. Assumed there are no (avoidable) modeling errors, the cycle count of the phase 5 and phase 6 models should be the same as in phase 4, thus they are not considered here either.

[2] IA = phase 2; CA = phase 3; BCA = phase 4

The accumulated cycle count (in million cycles) necessary for decoding the given bitmap has been compared for the SoC models available in phase 2 to phase 4 for both SoC topologies and several *data_ram* access latencies. For this comparison, the *simple_bus* TLM model [5] has been chosen. The *prog_rom* latencies are always 1 cycle; the *data_ram* access latencies are as shown in the table below.

Config.		IA		CA		BCA
type	ltcy	ideal	block	ideal	block	(ref.)
A	1	8.49	11.22	9.15	11.40	9.20
A	5	8.49	25.25	9.15	25.46	19.98
A	20	8.49	68.85	9.15	69.00	53.36
B	1	8.49	11.87	9.15	12.05	11.85

As outlined in section 4, the IA and the CA level models allow a worst case and a best case estimation by mapping the processor requests to the ideal debug interface respectively the blocking TLM interface. As expected, the best case estimations are the same for all topologies and memory parameterizations since both are properties not considered. The ideal CA models deliver 8% higher cycle counts since pipeline hazards are correctly modeled in an $CPI_{int} > 1$ (see equation 1). This effect also applies to the respective worst case values. The latter reside on a higher level since both model types are suspended for the full duration of every memory access. Thus, these values highly depend on the memory latencies. The reference (BCA) model values are smaller since parallelism is modeled correctly only by them. For topology A, the 1 cycle latency can completely be hidden within the pipeline, thus the reference almost matches the ideal CA estimation. For topology B, this does not hold any more since the limited bus resources have to be shared between the two processors. As further can be seen in the table above, the ideal best case values can only give a better estimation if the memory latencies and the processor pipelines fit together very well; in most cases the blocking TLM interface delivers the more reliable data. Taking these values, the designer will be on the safe side.

For this example application, an instruction accurate processor model, which can be developed and verified within a few man weeks, already gives a relatively good estimation on the performance to be expected in system context. The CA processor model does not seem to increase the accuracy in SoC context drastically, but it is an important step between the migration from the IA to the BCA model, allowing full verification in system context. Such a profiling driven design of a BCA processor will take several months, depending if it is a simple co-processor like the IDCTcore or a complex MMU equipped processor like the MIPS32.

6 Summary and Outlook

In this paper, we illustrated the need for system level processor/communication co-exploration for complex SoC designs. We presented a bus interface technique, which allows connecting the processor designer's world and the SoC communication world very early in the design flow. Interfacing capabilities on multiple levels of abstraction down to RT-Level allow a successive refinement based on profiling data having the highest accuracy already possible. Finally, we showed the advantages of our approach on a real-world case study.

Future work will concentrate on the RTL-near co-design phases as well as on automatic retargetability of the bus interface not only to the ISS, but also to the bus model.

References

[1] O. Ogawa, K. Shinohara, Y. Watanabe et al. A Practical Approach for Bus Architecture Optimization at Transaction Level. In *"Proc. Designers' Forum, Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2003.

[2] L. Benini, G. De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, pages 70–78, January 2002.

[3] E. Rijpkema, K.G.W. Goossens, A. Radulescu et al. Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2003.

[4] A. Hoffmann, T. Kogel, A. Nohl et al. A Novel Methodology for the Design of Application Specific Instruction-Set Processor Using a Machine Description Language. In *IEEE Transactions on Computer-Aided Design of Integrated Cicuits and Systems (TCAD) vol. 20 no. 11*, pages 1338–1354. IEEE, Nov 2001.

[5] SystemC initiative. *http://www.systemc.org*.

[6] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[7] A. Haverinen, M. Leclercq, N. Weyrich, D. Wingard. *White Paper for SystemC based SoC Communication Modeling for the OCP Protocol, http://www.ocpip.org/data/systemc.pdf*, 2003.

[8] A. Cochrane, C. Lennard, K. Topping et al. *AMBA AHB Cycle Level Interface (AHB CLI) Specification*, 2003.

[9] J. Notbauer, T. Albrecht, G. Niedrist,S. Rohringer. Verification and management of a multimillion-gate embedded core design. In *Proceedings of the Design Automation Conference (DAC)*, 1999.

[10] K. Hines, G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proceedings of the Design Automation Conference (DAC)*, 1997.

[11] J.A. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *Proceedings of the Design Automation Conference (DAC)*, 1997.

[12] CoCentric System Studio. *Synopsys, http://www.synopsys.com*.

[13] ConvergenSC. *CoWare, http://www.coware.com*.

[14] W. Cesario, A. Baghdadi, L. Gauthier et al. Component-Based Design Approach for Multicore SoCs. In *Proceedings of the Design Automation Conference (DAC)*, 2002.

[15] P. Paulin P. Magarshack. System-on-chip beyond the nanometer wall. In *Proceedings of the Design Automation Conference (DAC)*, 2003.

[16] L. Guerra, J. Fitzner, D. Talukdar et al. Cycle and Phase Accurate DSP Modeling and Integration for HW/SW Co-Verification. In *Proceedings of the Design Automation Conference (DAC)*, 1999.

[17] G. Hadjiyiannis, S. Devadas. Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, to appear in 2003.

[18] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Intl. Conf. on VLSI Design*, 2001.

[19] A. Fauth and J. Van Praet and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, Mar. 1995.

[20] R. Leupers. HDL-based Modeling of Embedded Processor Behavior for Retargetable Compilation. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, Sep. 1998.

[21] G. Braun, A. Wieferink, O. Schliebusch et al. Processor/Memory Co-Exploration on Multiple Abstraction Levels. In *DesignAutomation & Test in Europe (Date)*, Munich, March 2003.

[22] T. Kogel, M. Doerper, A. Wieferink et al. A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks. In *The First IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis*, Newport Beach (California USA), Oct 2003.

[23] O. Schliebusch, A. Chattopadhyay, M. Steinert et al. RTL Processor Synthesis for Architecture Exploration and Implementation. In *Conference on Design, Automation & Test in Europe (DATE) (accepted)*, Paris, France, Feb 2004.

[24] A. Wieferink, T. Kogel, A. Nohl et al. A Generic Toolset for SoC Multiprocessor Debugging and Synchronisation. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, The Hague (Netherlands), June 2003.

[25] Official JPEG homepage. *http://www.jpeg.org*.

[26] A. Nohl, G. Braun, A. Hoffmann et al. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proceedings of the Design Automation Conference (DAC)*, New Orleans, June 2002.