Processor/Memory Co-Exploration on Multiple Abstraction Levels

Gunnar Braun, Andreas Wieferink
Oliver Schliebusch, Rainer Leupers, Heinrich Meyr
Integrated Signal Processing Systems
Templergraben 55, 52056 Aachen, Germany
gunnar.braun@iss.rwth-aachen.de

Achim Nohl
LISATek Inc.
190 Sandhill Circle, Menlo Park, CA
achim.nohl@lisatek.com

Abstract

Recently, the evolution of embedded systems has shown a strong trend towards application-specific, single-chip solutions. As a result, application-specific instruction set processors (ASIP) are more and more replacing off-the-shelf processors in such systems-on-chip (SoC). Along with the processor cores, heterogeneous memory architectures play an important role as part of the system. According to last year's ITRS [5], in 2004 about 70 percent of the chip area will be made up of memories. As such architectures are highly optimized for a particular application domain, processor core and memory subsystem design cannot be apart, but have to merge into an efficient design process. In this paper, we present a unified approach for processor/memory co-exploration using an architecture description language. We show an efficient way of considering instruction set and memory architecture during the entire exploration process. Finally, we illustrate the feasibility of our approach with a real-world case study.

1. Introduction

One of the key factors for a successful design of application-specific instruction set processors (ASIP) is an efficient architecture exploration phase. The objective of the architecture exploration is to reduce the huge design space in order to find the best-suited architecture for a given application under a number of constraints, such as performance, power consumption, chip size, and flexibility. Although there are a number of analytical approaches, large parts of the design space exploration still have to be carried out by simulating alternative architecture implementations. It becomes obvious that the design methodology and simulation performance have a significant impact on the efficiency of the exploration process, hence, on the quality of the architecture implementation and the design time. Particularly, the ability to vary the abstraction level of the underlying architecture model, that means, the refinement from an initial functional specification to an RT-level implementation, is of extreme importance to avoid expensive turn-around times.

In order to address the demands of the ASIP design process, so-called architecture description languages (ADLs) have been established with the objective to close the gap between purely functional data-flow models and implementation models in a hardware description language (HDL) like VHDL or Verilog. Few ADLs allow architecture exploration on *multiple abstraction levels*, that is, the designer is enabled to thoroughly explore the processor's instruction set before elaborating alternative implementations of the microarchitecture. However, a flexible choice of the abstraction level is not only a requirement for the specification of the processor core, but also for the memory attached to it.

ASIPs are highly optimized for the performance of a particular task. This is achieved by employing application-tailored instruction sets, special-purpose functional units, and dedicated hardware resources with irregular data paths. As the design of the memory subsystem has a major impact on performance, power consumption, and chip size, the memory architecture has to be designed to match the application's needs as much as the processor core does. However, this cannot be achieved by employing conventional one- or two-level memory hierarchies as found in most general-purpose processors, but leads to very application-specific, heterogeneous memory architectures. Furthermore, the essential tight interaction between processor core and memory subsystem forbids a decoupled design of both.

This leads to the following conclusions: first, the architecture exploration must go along with the memory subsystem exploration, and second, the design language must support a flexible and efficient way of modeling heterogeneous memory architectures on different levels of abstraction.

This work presents a unified approach for an ADL-based architecture/memory co-exploration on multiple abstraction levels. Our approach allows to explore ASIP core *and* memory subsystem, starting from a functional ISA model without implementation details down to a cycle-accurate microarchitecture model. Along with the refinement of the processor model, the abstraction level of the memory model is also lowered – from an initial assumption of ideal memories via functional models down to cycle-accurate simulation models with the same temporal and structural accuracy as the processor model. We will show that the ability to model on abstract level has a number of benefits, such as short turn-around times, less modeling effort, and high simulation performance. This allows to explore a much larger part of the design space before evaluating alternative microarchitecture implementations on cycle-accurate level.

The rest of this paper is organized as follows. Section 2 presents related work. The modeling language and underlying memory model is introduced in section 3. In section 4,

1530-1591/03 $17.00 © 2003 IEEE

an exemplary exploration process illustrates the feasibility of our approach. Finally, section 5 concludes this work and gives an outlook on future research topics.

2. Related Work

Related work can be separated into two areas, architecture description languages and memory exploration, modeling, and simulation.

The former category comprises an extensive amount of work about design space exploration for processor-based embedded systems, in both academia and industry. Examples are ISDL [10], EXPRESSION [16], MIMOLA [17], nML [6], ARC [1], Target [18], and Tensilica [19]. While these approaches capture various aspects of the processor architecture and address different abstraction levels, no previous approach except EXPRESSION has explicit support for the modeling of memory architectures.

The EXPRESSION language is one of few architecture description languages that allow for processor/memory co-exploration. Besides the ability to model the processor core on microarchitecture level, the memory subsystem can be described by choosing from predefined memory models for DRAM, SRAM, caches, etc. and describing the interconnectivity as a netlist. However – in contrast to this work – the EXPRESSION language only supports cycle-accurate memory modeling without the capability to refine the description from abstract to microarchitecture level. Furthermore, no results on simulation efficiency have been published so far.

Poseidon Technologies [4] offers a memory architecture exploration tool, MemBrain, which is based on an extensible architecture description language XADL. However, similar to EXPRESSION, only cycle-accurate modeling is supported.

Besides the above approaches, many publications can be found in the area of memory exploration, modeling, simulation, and synthesis. Dinero-IV [9] is a popular memory simulator written in C, which is capable of modeling arbitrary deep cache hierarchies. The simulator takes a memory trace as input and generates memory profiling data. Similar approaches are Active-Memory [13], MemSpy [14], and Tycho [12]. However, all these simulators are decoupled from the processor design process, and do not allow the modeling of very heterogeneous memory architectures. A good overview of existing work about memory simulation and synthesis can be found in [8].

In addition to the above mentioned work, there are a number of companies in the area of embedded memories. Denali Software [2] offers configurable memory models written in C and hardware description languages. Although inevitable for synthesis, configurability is generally very limited, and the stand-alone models contradict the need for a unified processor/memory architecture design process.

The architecture description language LISA [7] is capable of modeling processor architectures on different levels of abstraction, e.g. on instruction or cycle accuracy. In this work, we pick up with our previous research, and introduce extensions that allow for processor/memory co-exploration with LISA, hence presenting a unified approach for the design of processor-based, embedded systems.

3. Memory Exploration using LISA

The language for instruction set architectures (LISA) has been designed for the description of instruction set processor architectures. LISA belongs to the class of so-called *mixed structural/behavioral* ADLs, that is, a model description is composed of a structural part describing the processor resources such as registers, memories, pipelines, and a behavioral part reflecting the processor's instruction set including instruction encodings, assembly syntax, functional behavior, and timing. LISA is capable of modeling the processor architecture on different abstraction levels regarding the hardware structure as well as time. That means, a purely functional model of the instruction set can be refined in structure by e.g. adding a pipeline or i/o interfaces, and by increasing temporal accuracy, e.g. by changing the granularity from instructions to clock cycles.

However, LISA is not yet capable of modeling non-ideal memories, that means, there is no notion of latencies, caches, or memory interconnects. These limitations have been overcome within this work by adding language support for a flexible and intuitive description of arbitrary memory architectures. The necessary extensions can be separated into two major parts, first, the description of the different types of memories and their interconnections, and second, the description of the memory access in the instruction/operation behavior description.

3.1. Memory Modules

Each memory present in the architecture has to be defined in the model description by providing memory type, data type, size, and a number of (partly) optional parameters depending on the memory type. Predefined memory types are *bus*[1], *cache*, *ram*, and *write buffer*, but user-defined modules can be easily added as long as they are present in the form of C(++) code or libraries. The parameters available for configuration of the predefined modules are shown in table 1. Additionally, all memories can be configured for read-only or read-write access.

A major advantage is the free choice of the desired data type of the memory blocks, in other words, the memory model is not limited to byte- or word-wise organized memories. Although less frequently appearing in practice, some very application-specific architectures with extremely tight constraints on code size employ program memories with bit-widths which are not a multiple of eight [11]. In these cases, it is possible to use a bit data type provided by LISA language in order to define e.g. 21-bit instruction memories. This allows the definition of a very compact instruction set without wasting memory for each stored instruction.

[1] Although the bus cannot be understood as a memory, it is part of the memory model, since it establishes the interconnects among different memories and processor.

Memory type	Parameters
CACHE	line size, number of lines, associativity, write allocation policy, write back policy, replacement policy, read/write latency, block size, subblock size, endianess
RAM	size, page size, read/write latency for normal, burst, and page mode access, endianess, block size, subblock size, number of banks
BUS	transfer block/subblock size, address type, latencies
WRITEBUFFER	line size, block size, subblock size, flush policy, write latency, endianess

Table 1. Memory Module Parameters

```
MEMORY_MAP
  {
    BUS(pbus), RANGE(0x0800000,0x087ffff) -> icache[(31..2)];
    BUS(dbus), RANGE(0x0200000,0x020ffff) \
                                -> banked1[(1..0)][(31..2)];
    BUS(dbus), RANGE(0x0300000,0x030ffff) \
                                -> banked2[(19..18)][(17..0)];
  }
```

Figure 1. Sample Memory Mapping Scheme

3.2. Interconnectivity

The interconnectivity of the instantiated memories and the processor core is established by specifying the *next level module(s)* for each memory component. Each cache, bus, or buffer *sees* only the next level memory it can access. Buses play a particular role, since they can connect to more that one memory, hence, each attached memory is identified by an address space unique for that particular bus.

The connection between processor and memories is established by assigning separate address ranges from the processor's address space to the defined memories components. A *memory map* as part of the LISA model describes how memory addresses are mapped onto the physical addresses of the respective memories. A sample address mapping scheme is shown in figure 1.

The LISA code excerpt shows mappings for three different address ranges. The first line defines a mapping of the address range specified by the `RANGE` keyword onto a memory *icache*. The parameter in square brackets following the memory name describes the actual address translation through a bit mask. The range `(31..2)` indicates that the last two bits of the address are ignored for addressing *icache*, or, each four consecutive addresses refer to the same memory block. This is a common setup for byte-wise addressable memories with a block size of 32 bits. Furthermore, the memory is attached to a bus *pbus*.

The second and third line of the example show two common address mapping schemes for banked memories, *block addressing* and *interleaved addressing*. In the second mapping, the two least significant bits of an address are used to address the first dimension of the two-dimensional memory *banked1*. As the first dimension selects the memory bank, this is an interleaved address mapping where consecutive addresses refer to different banks. In the block addressing scheme in the third line, bits 18 and 19 are used to select the bank, i.e. coherent blocks of addresses are mapped onto the same memory banks.

The description contained in the memory map allows the modeling of most address mappings commonly in found in embedded processors. However, it is not possible to formally describe complex virtual address translation as performed in processors with a memory management unit (MMU) (e.g. MIPS and some ARM implementations). In such cases, the address translation has to be described by means of a custom C(++) function.

3.3. Memory Interface

While the description of memories and their interconnects determines the structural abstraction level, the temporal accuracy is defined by how the memory is accessed from the model of the processor core. Two memory interfaces are provided to access data from the defined memories, a *functional* and a *cycle-accurate* interface.

3.3.1 Functional Interface

The functional memory interface allows basic access to the memory subsystem. It is made up of only two methods, a *read* and a *write* function. Both accept a number of parameters for specifying access mode (e.g. burst) and requested block or subblock. The requested memory access is performed immediately, and the accumulated latency is returned (in case of success).

The use of the functional interface has a number of advantages compared to the cycle-accurate interface. Firstly, it is very simple to use within a functional, instruction-based model description, since the designer can just assume that data is available when requested. This prevents from modeling complex memory controllers, and allows to quickly establish a working model of the architecture. Obviously, this implies a certain temporal inaccuracy, however, operation timing is generally not of concern in this early design phase.

The second advantage is that very high simulation performances can be achieved with the functional interface. This is due to the fact that memory simulation only takes place when an access is performed. That means, the memory simulator does not have to store a state of progress or to perform a request queue management due to the fact that each memory access is self-contained. As a consequence, this obsoletes the need for synchronous memory simulation (as required for cycle-accurate memory simulation).

Finally, the simplicity of the interface is extremely valuable for the integration of proprietary memory components. For instance, a C(++)-based bus implementation with a customized protocol is easily included by embedding the bus model into an interface wrapper. Once the model obeys the functional interface, it can be employed as any of the predefined modules, i.e. instantiated, wired, and accessed.

In summary, the use of the functional memory interface allows a quick iteration cycle, since changes are carried out within minutes, and, due the high simulation performance, profiling results showing latency cycles, cache hit/miss rates, and bottlenecks in the memory architecture are quickly obtained.

However, functional memory simulation is not appropriate to exploit parallelism, that is, parallel or pipelined mem-

ory architectures. Therefore, once a processor/memory architecture is found that roughly meets the initial design criteria, the model can be refined to a cycle-accurate model, which is used to collect the desired profiling data of the microarchitecture.

3.3.2 Cycle-accurate Interface

Compared to the functional interface, cycle-accurate memory access requires a *request-based* interface, that means, each access must be *requested* first before the actual transaction can take place. Therefore, the cycle-accurate interface provides separate methods for sending access requests to memory and eventually receiving data (in case of a read operation).

As data might not be available until several clock have passed after the request, the memory possibly adopts many internal states of progress while processing the request(s). Therefore, a *synchronous* simulation of the memory architecture is mandatory. A state transition function as part of the memory interface, which is called for each simulated clock cycle, serves this purpose.

The application of the cycle-accurate memory interface has a number of consequences for the processor model. In contrast to functional memory simulation, it is now necessary that the processor model accounts for unavailability of memory resources, for instance, by stalling program execution until data becomes available. On the other hand, cycle-accurate modeling allows to use different pipeline stages for initiation and completion of a data transfer, a common practice for hiding memory latencies. Again, this stresses the importance of processor/memory co-exploration, since the choice of the memory affects the design of the pipeline, and vice versa.

In summary, the migration from function to cycle-accurate memory access results in a model much closer to the hardware, and thus allows a further, deeper exploration of the architecture. On cycle-accurate level, pipelines or buffers might introduced, or separate buses might be chosen for instruction and data memory. It would be very difficult of measure the effects of these modifications on functional level.

3.4. Memory Simulator

The memory simulator is embedded into an instruction set simulator, which is automatically constructed from the LISA architecture description. In order to ease the exploration process, most of the parameters shown in table 1 can be changed during simulator run-time in our retargetable, graphical debugger frontend. Besides accumulating profiling data, the frontend visualizes the modeled memory architecture graphically as a block diagram.

4. Case Study

In order to illustrate the suggested architecture/memory co-exploration process, this section shows how the presented approach can be applied to tailor a processor's microarchitecture and memory subsystem to match a particular application. We will demonstrate how the memory architecture exploration process benefits from very fast functional simulation, and how the found architectural alternatives can be analyzed in detail afterwards using cycle-accurate models.

```
MEMORY_MAP
 {
  BUS(stdbus), RANGE(0x00000000,0x000fffff) -> prog[(31..2)];
  BUS(stdbus), RANGE(0x00100000,0x001fffff) -> data[(31..2)];
  BUS(stdbus), RANGE(0x40000000,0x40800000) -> heap[(31..2)];
  BUS(stdbus), RANGE(0x7fff8000,0x7fffffff) -> stack[(31..2)];
 }
```

Figure 2. ARM Model Memory Map

4.1. Overview

For our experiments, we chose a LISA processor model of an ARM processor core as a basis for the exploration process. The model is an instruction-accurate implementation of the ARMv4 instruction set as found in ARM7 processor cores, that is, only the functional behavior of the processor's instructions is modeled, without any information about instruction timing. Furthermore, the model contains five ideal memories connected to the processor core via a bus *stdbus*. The memories are organized in 32 bit blocks with byte-wise addressing. Figure 2 shows the memory map of the model. The model has been verified against ARM's simulator with several applications from the domains dsp, networking, and image processing.

Unfortunately, ideal memory is still hard to find in real life, hence, the model has been modified to account for non-ideal memories. As a reference for the results of the following exploration process, we assumed all five memory blocks to be off-chip DRAMs supporting page and burst mode access, with an access latency of 20 (processor) clock cycles for normal access, 10 for page mode access, and 7 for burst mode access. Furthermore, we assumed a page size of 4 blocks. This setup corresponds to the first configuration in table 2.

As a target application, we chose a JPEG-2000 compression algorithm encoding a 397x377 sized bitmap. Both the C source code of the algorithm and the sample bitmap are freely available from the JPEG group's web page [3]. The source code has been compiled and linked using the software development tools delivered with the ARM Developer Studio (ADS) available from ARM. The resulting binary executable has been run on the simulator generated from the LISA model. All the simulations have been carried out on a 1.6 GHz (Athlon-based) desktop PC running SuSE Linux version 8.

The objective of the following case study is to find an optimum microarchitecture implementation of the ARM instruction set for running the JPEG algorithm. Here, *optimum* refers to the implementation which gives the minimum number of overall clock cycles for encoding the bitmap.

4.2. Functional Simulation

The use of the functional memory model together with the abstract, instruction-accurate ARM model allows a very fast simulation of the complex JPEG-2000 application. Due to the application of our *just-in-time cache-compiled simulation technique* [15], the entire JPEG encoder can be simulated within a few minutes. This enables the simulation of many design alternatives in a short amount of time.

The different memory architectures we considered are shown in table 2. As can be seen from the table, we successively added caches for the different memories, or, in case of the stack memory, replaced the DRAM by an on-chip SRAM with single-cycle access. The latter design choice is self-evident, since the stack memory area is of only 32KB size, however, most frequently accessed (besides program memory). Similarly, all other design decisions from memory configuration one to six are based on profiling results obtained from instruction set simulation.

The performance of the different memory architectures is presented in figure 3. For each configuration, the middle bar shows the number of total latency cycles due to memory access. The leftmost bar shows the (constant) number of executed instructions as a comparison. The chart points out that the number of latency cycles could be reduced drastically by roughly one order in magnitude (configuration 5).

Figure 3 also shows that no further optimization of the memory architecture can be evaluated on this level of abstraction. In configuration 5, we almost achieved the optimum state that the number of memory accesses equals the number of memory latency cycles, i.e. close to an average of a single clock cycle per memory access[2]. If we assume a non-pipelined ARM implementation with an average *CPI* (cycles per instruction) ratio of 1 (without the memory references), we can estimate the total number of clock cycles n_{cycle} from the number of executed instructions n_{instr} and the number of overall memory latency cycles n_{lat} as

$$n_{cycle} = n_{instr} \cdot CPI + n_{lat} = n_{instr} + n_{lat}$$

Applying this to configuration 5 leads to a total execution time of 665 million clock cycles, which is still about 2.7 cycles per instruction. Further optimization can only be achieved by introducing parallelism, however, the effects can only be measured by cycle-accurate simulation.

4.3. Cycle-accurate Simulation

For further performance evaluation, we refined the instruction set model of the ARM to a cycle-accurate, microarchitecture model with a three-stage pipeline. The pipeline consists of fetch, decode, and execute stage, which allows the parallelize program memory read and execution of an ALU, control, or load/store operation. The implementation of the necessary modifications for cycle-accurate memory access took about one week.

From figure 3, it can be observed that the introduction of the pipeline gives a *CPI* ratio of 1.38 (including memory references) for configuration 5. The total cycle count now also considers latencies caused by control hazards, hence, is no longer an estimate of the architecture performance. This cycle-accurate model allows the exploration of further techniques for increasing performance, e.g. latency hiding.

In configuration 6, we added an application-specific write buffer between data cache and DRAM. The buffer has been taken from a VLIW multimedia processor, where it serves two purposes, the buffering of entire cache lines and of single blocks or subblocks. This has two major advantages. First, the cache can quickly write *victim* lines, i.e. cache lines that have been evicted from the cache, and sec-

[2]Memory architectures like this are often found in application domains where performance is far more important than power consumption or chip size, e.g. gigabit networking.

Config. 1		
program	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
data	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
heap	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
stack	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
Config. 2		
program	cache	assoc=1, size=256KB, linesize=4, ltcy=1
data	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
heap	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
stack	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
Config. 3		
program	cache	assoc=1, size=256KB, linesize=4, latency=1
data	DRAM	ltcy=20, page_ltcy=10, burst_ltcy=7
heap	DRAM	ltcy=20, page_ltcy=10, burst_ltcy=7
stack	SRAM	size=32KB, latency=1
Config. 4		
program	cache	assoc=1, size=256KB, linesize=4 bytes, ltcy=1
data	DRAM	ltcy=20, page_ltcy=10, burst_latency=7
heap	cache	assoc=4, size=256KB, linesize=4, ltcy=1, write_back
stack	SRAM	size=32KB, ltcy=1
Config. 5		
program	cache	assoc=1, size=256KB, linesize=4, ltcy=1
data	cache	assoc=1, size=256KB, linesize=4, ltcy=1, write_back
heap	cache	assoc=4, size=256KB, linesize=4, ltcy=1, write_back
stack	SRAM	size=32KB, latency=1
Config. 6		**(only for cycle-accurate simulation)**
program	cache	assoc=1, size=256KB, linesize=4, ltcy=1
data	cache	assoc=1, size=256KB, linesize=4, ltcy=1, wr_thrgh.
	writebuf.	size=64B, linesize=4, ltcy=1, victim, flush-on-valid
heap	cache	assoc=4, size=256KB, linesize=4, ltcy=1, write_back
stack	SRAM	size=32KB, latency=1

Table 2. Memory Configurations

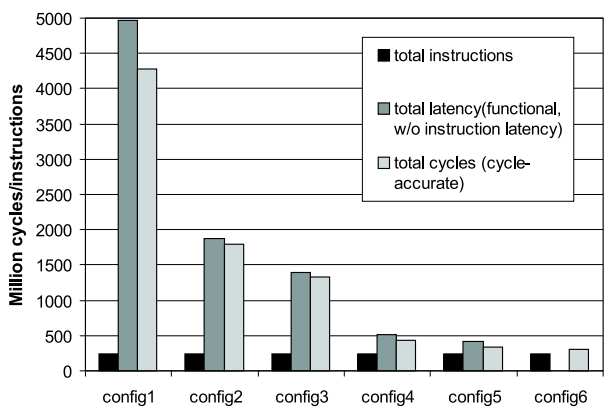

Figure 3. ARM JPEG Performance

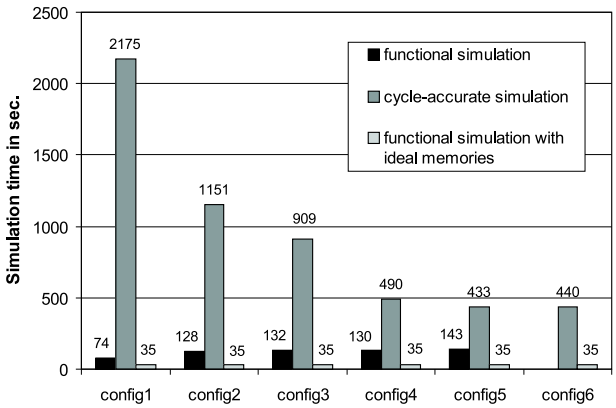

Figure 4. Simulation Time

ond, in case of a cache miss, the cache does not have to wait for the underlying DRAM to become available. Furthermore, the buffer tries to write entire lines to the RAM, and thus performs a write combining of consecutive cache misses. The write buffer has a customized flush policy preventing itself from getting full.

Figure 3 shows that the write buffer (configuration 6) gives an additional performance increase of 15%. Since the buffer increases the parallelism of the memory architecture, the cycle count reduction cannot be observed on functional level.

4.4. Comparison

Although cycle-accurate simulation has the major advantage of revealing the exact performance of the JPEG-2000 on the ARM, there are a number of drawbacks concerning the exploration process.

Cycle-accurate models are much more complex than instruction-accurate models. This holds true for both, the design of the processor core as well as the memory subsystem. The design effort for a cycle-accurate model is easily three times more than for an instruction-accurate model. The complexity is also reflected in simulation performance, as shown in figure 4. When migrating to cycle-accurate models, simulation performance is reduced by about one order in magnitude in average. As a conclusion, starting the exploration process on functional level allows the consideration and evaluation of many more architectural alternatives than it would on cycle-accurate level.

5. Summary

In this paper, we illustrated the need for architecture/memory co-exploration on multiple abstraction levels. We presented an extension of the architecture description language LISA, which allows for flexible modeling and simulation of heterogeneous memory architectures on instruction and clock cycle basis. Finally, we demonstrated the advantages of our approach with a real-world case study.

Future work will concentrate on the integration of more complex bus architectures (e.g. AMBA) and the utilization of the memory architecture description for compiler generation and HDL code synthesis.

References

[1] ARC Cores. *http://www.arccores.com*.
[2] Denali Software. *http://www.denali.com*.
[3] Official JPEG homepage. *http://www.jpeg.org*.
[4] Poseidon Technologies. *http://www.poseidontech.com*.
[5] International Technology Roadmap for Semiconductors. 2001 edition., 2001. Semiconductor Industry Association.
[6] 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.
[7] A. Hoffmann and T. Kogel and A. Nohl and G. Braun and O. Schliebusch and A. Wieferink and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.
[8] R. Beckmann and J. Herrmann. Using Constraint Logic Programming in Memory Synthesis for General Purpose Computers. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, 1997.
[9] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. *http://www.neci.nj.nec.com /homepages/edler/d4/*.
[10] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.
[11] T. Gloekler and S. Bitterlich. Power Efficient Semi-Automatic Instruction Encoding for Application Specific Instruction Set Processors. In *Proc. of the Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, May 2001.
[12] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California at Berkeley, 1987.
[13] A. R. Lebeck and D. A. Wood. Active Memory: A New Abstraction for Memory-System Simulation. In *Measurement and Modeling of Computer Systems*, pages 220–231, 1995.
[14] M. Martonosi, A. Gupta, and T. E. Anderson. Memspy: Analyzing Memory System Bottlenecks in Programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.
[15] A. Nohl, G. Braun, O. Schliebusch, A. Hoffmann, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proc. of the Design Automation Conference (DAC)*, 2002.
[16] P. Mishra and P. Grun and N. Dutt and A. Nicolau. Processor-Memory Co-Exploration driven by a Memory-Aware Architecture Description Language. In *Int. Conf. on VLSI Design*, Jan. 2001.
[17] R. Leupers. HDL-based Modeling of Embedded Processor Behavior for Retargetable Compilation. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, Sep. 1998.
[18] Target Compiler Technologies. *http://www.retarget.com*.
[19] Tensilica. *http://www.tensilica.com*.