

A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation

Achim Nohl, Gunnar Braun,
Oliver Schliebusch, Rainer Leupers,
Heinrich Meyr
Integrated Signal Processing Systems
Templergraben 55, 52056 Aachen, Germany
nohl@iss.rwth-aachen.de

Andreas Hoffmann
LISATek Inc.
190 Sand Hill Circle
Menlo Park, CA 94025
andreas.hoffmann@lisatek.com

ABSTRACT

In the last decade, instruction-set simulators have become an essential development tool for the design of new programmable architectures. Consequently, the simulator performance is a key factor for the overall design efficiency. Based on the extremely poor performance of commonly used *interpretive* simulators, research work on fast *compiled* instruction-set simulation was started ten years ago. However, due to the restrictiveness of the compiled technique, it has not been able to push through in commercial products. This paper presents a new retargetable simulation technique which combines the performance of traditional compiled simulators with the flexibility of interpretive simulation. This technique is not limited to any class of architectures or applications and can be utilized from architecture exploration up to end-user software development. The work-flow and the applicability of the so-called *just-in-time cache compiled simulation* (JIT-CCS) technique will be demonstrated by means of state of the art real world architectures.

Categories and Subject Descriptors

I.6.3 [Simulation and Modeling]: Simulation Support Systems; I.6.3 [Simulation and Modeling]: Model Validation and Analysis; D.3.2 [Programming Languages]: Design Languages—LISA; C.0 [General]: Modeling of Computer Architecture

General Terms

Design, Languages, Performance

Keywords

Retargetable simulation, compiled simulation, instruction set architectures

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.
Copyright 2002 ACM 1-58113-461-4/02/0006 ...\$5.00.

1. INTRODUCTION

Instruction-set simulators are an integral part of a today's processor and software design process. Their important role within the architecture exploration, early system verification and pre-silicon software development phase is undisputable. However, the growing complexity of new programmable architectures has a negative impact on the simulation performance. Taking into account Moore's law, research work on a high performance simulation technique for instruction-set simulators was started in 1991 [10]. The so-called compiled simulation technique is rooted in the domain of RTL level hardware simulation and was applied successfully to instruction-set simulators. Nevertheless, the fast compiled simulation technique has not been able to succeed in commercial tools. This is due to the fact that the speed-up is expensively paid with restrictions and assumptions contradicting with the requirement of today's applications and architectures.

The major restriction for the utilization of compiled simulators is the requirement for static program code. This limits the compiled technique to a small class of digital signal processors (DSP). In contrast to typical DSP applications which are signal processing algorithms, micro-controller (μC) architectures usually run an operating system (OS). The significant characteristic of operating systems, run-time dynamic program code, conflicts with the limitation of compiled simulators. However, even for DSP architectures real-time operating systems are increasingly gaining importance. This paper presents a simulation technique which meets the requirements for both, high simulation speed and maximum flexibility. The so-called *just-in-time cache compiled simulation* (JIT-CCS) technique can be utilized within the architecture design as well as for end-user software development. The presented technique is integrated in the retargetable LISA processor design platform [2]. A generator back-end for the LISA 2.0 processor compiler has been developed, which automatically constructs a JIT-CCS simulator from a LISA machine description.

The rest of this paper is organized as follows: Section 2 presents related work concerning compiled simulation and result caching techniques. The restrictions of existing simulation techniques and their consequences on the applicability are worked out in section 3. Subsequently, section 4 describes the work-flow of the just-in-time cache compiled simulation technique. Finally, benchmark results for various state of the art architectures are presented in section 5.

2. RELATED WORK

Research work on instruction-set architecture simulation has been an active research topic since the early days of programmable architecture design. Within the scope of the EMBRA project [5] a high performance simulator for the MIPS R3000/R4000 processor has been developed. The objective is similar to the one presented in this paper – providing highest flexibility with maximum performance. Similar to FastSim [3] based simulators, the performance gain is achieved by dynamic binary translation and result caching. However, EMBRA is a non-retargetable simulator and restricted to the simulation of the MIPS R3000/R4000 architecture.

Work on *retargetable* fast simulators using a machine description language was published within the scope of the FACILE project [4]. The simulator generated from a FACILE description utilizes the *Fast Forwarding* technique to achieve reasonably high performance. Fast forwarding is similar to compiled simulation and uses result caching of processor actions, indexed by a processor configuration code. Previously cached actions can be replayed directly in a repeated occurrence of a configuration. Due to the assumption that program-code is run-time static, dynamic program-code cannot be simulated. Furthermore, no results on the applicability of FACILE for VLIW or irregular DSP architectures have been published.

A retargetable tool suite which allows *cycle- and bit-true modelling* of pipelined processors is based on the EXPRESSION [1] language. Previous publications have shown its suitability for modelling real-world architectures like the Motorola DSP 56k or Texas Instruments TMS320C6000, however, no results are available on the performance of the generated simulators.

Retargetable *compiled* simulators based on an architecture description languages have been developed within the Sim-nML (FSim) [9], ISDL (XSSIM) [8] and MIMOLA [11] projects. Due to the simplicity of the underlying instruction sequencer, it is not possible to realize processor models with more complex pipeline control mechanisms like Texas Instruments TMS3206000 at a cycle accurate level. A further retargetable approach, which is based on machine descriptions in ANSI C, has been published by Engel and Fettweis [6]. However, only results for a single proprietary DSP architecture are available so far. Moreover, all of the presented compiled simulation approaches are qualified by the limitations that result from the compiled principle.

In summary, none of the above approaches combines retargetability, flexibility, and high simulation performance at the same time. The LISA language has proven to be capable of retargeting fast compiled instruction-set simulators for various real-world DSP architectures [2]. To finally overcome the flexibility restrictions of compiled simulators a novel simulation technique is presented in the following sections.

3. PROCESSOR SIMULATION

The broad spectrum of today’s instruction-set simulation techniques starts with the most flexible¹ but slowest interpretive technique. Compared to the interpretive technique, much higher simulation speed is achieved by compiled simulation, however, the gain is expensively paid with a loss of flexibility. This section critically compares the existing

¹Here flexibility refers to the ability to address all kinds of architectures and applications.

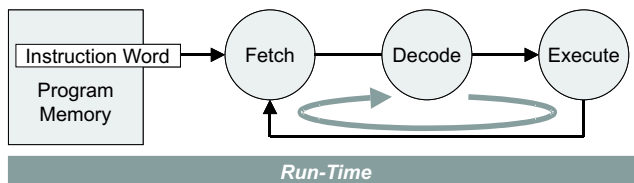


Figure 1: Interpretive Simulation Workflow

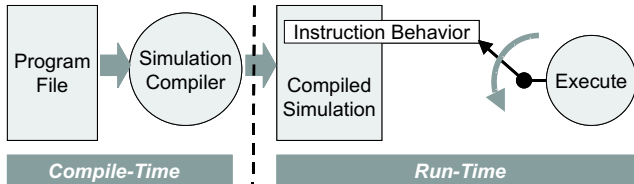


Figure 2: Compiled Simulation Workflow

simulation techniques. Especially, the impact of the trade-off between flexibility and performance on the suitability for different application domains is discussed.

3.1 Interpretive Simulation

An interpretive simulator is basically a virtual machine implemented in software, which interprets the loaded object code to perform appropriate actions on the host as shown in figure 1. Similar to the operation of the hardware, an instruction word is fetched, decoded, and executed at run-time (simulation loop), which enables the highest degree of simulation accuracy and flexibility. However, the straightforward mapping of the hardware behavior to a software simulator has major disadvantages. Unlike in real hardware, instruction decoding is a very time consuming process in a software simulator. Compared to the operation of the functional units, which can be easily transformed into equivalent instructions on the host, the decoder is characterized by control flow. Especially for today’s VLIW architectures the decoding overhead dominates.

3.2 Compiled Simulation

The objective of compiled simulation is to improve the simulation performance. Considering instruction-set simulation, efficient run-time reduction can be achieved by shifting time-consuming operations from the simulator run-time into an additional step before the simulation (compile-time). This step is performed by a tool called *simulation compiler* (see figure 2). Depending on architectural and application characteristics, the degree of compilation varies. All compiled simulators have in common that a given application is decoded at compile-time. Based on the results of the decoding phase the simulation compiler subsequently selects and sequences the appropriate host operations that are required to simulate the application. Since the time-consuming instruction scheduling is still performed at run-time (*dynamically scheduled*), *statically scheduled* [7] compiled simulators also move the instruction scheduling into the compilation phase. However, all compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is furthermore run-time static.

3.3 Importance of Flexibility

Many application domains are excluded from the utilization of compiled simulators. An overview of these domains is given in figure 3 and the details are discussed in the next paragraphs.

The integration of compiled simulators into embedded system environments is not possible, since the prime requirement, predictable program code, is not fulfilled when using external program memories.

Furthermore, applications with run-time dynamic program code, as provided by operating systems (OS), cannot be addressed by compiled simulators. Run-time changes of program code cannot be considered by a simulation compiler. Today’s embedded systems consists of multiple processor cores and peripherals which make an underlying OS indispensable. However, compiled simulators only allow the isolated simulation of applications which is not sufficient for the verification of a complete HW/SW system. Considering

Application Domain	Compiled Simulation Restriction
- System Integration - External Memories	Program code is not predictable before runtime.
- Operating Systems - Boot Loading	Dynamic program code is not allowed
- Multiple Instruction Sets	Run-Time switch between different instruction sets cannot be considered before simulation.
- Large Applications	Compiled simulation is qualified by an enormous memory usage.

Figure 3: Compiled Simulation Restrictions

novel architectural features, especially in the domain of low power architectures, multiple instruction-sets are widely used to reduce power and memory consumption. These architectures can switch to a compressed instruction-set at run-time. For instance the ARM core family provides a so-called "Thumb" instruction-set. This dynamic instruction-set switching cannot be considered by a simulation compiler, since the selection depends on run-time values and is not predictable.

Compiled simulation of large applications requires an enormous amount of memory (approx. 1000x, depending on the architecture) compared to interpretive techniques. As long as the host memory is big enough, the high memory consumption does not noteworthy decrease the performance. This is due to the fact that program execution concentrates on small local parts of the code, and thus cache effects are not noticeable. Of course for multi-processor simulation of embedded systems or processor arrays, the memory efficiency of the simulator becomes increasingly important.

Summarizing the above arguments the enormous performance gain of compiled simulators succumbs to their restrictiveness. This implies that the presented application areas are still dominated by the slow interpretive technique. But driven by the ever increasing complexity of applications, architectures, and systems the urgent requirement for a high performance simulation technique arises. This technique has to provide the maximum degree of flexibility together with a high performance.

4. JIT CACHE COMPILED SIMULATION

The *Just-In-Time Cache Compiled Simulation (JIT-CCS)* technique presented in this paper has been developed with the intention to combine the full flexibility of interpretive simulators with the speed of the compiled principle. The basic idea is to integrate the simulation compiler into the simulator. The compilation² of an instruction takes place

²In this context compilation does not refer to invoking a C compiler at run-time. The compilation process is discussed in section 4.2.

at simulator run-time, *just-in-time* before the instruction is going to be executed. Subsequently, the extracted information is stored in a simulation cache for the direct reuse in a repeated execution of the program address. The simulator recognizes if the program code of a previously executed address has changed and initiates a re-compilation. First of all, this method offers the full flexibility, as the interpretive technique does. However, it has to be investigated whether the performance of the compiled simulation can be achieved. The next paragraph discusses the theoretically achievable performance and gives a comparison to the compiled technique.

4.1 Performance

The total compiled simulation time $t_{app,cs}$ is equivalent to the total instruction execution time $t_{ex,total}$, which is the product of the instruction execution count n_{exec} and the average execution time per instruction $\tilde{t}_{ex,insn}$. For the just-in-time compiled simulation the total simulation time $t_{app,js}$ is additionally made up by the total compilation time $t_{comp,total}$. Under the assumption that the program code is constant the total compilation time only depends on the instruction count of the application n_{insn} and the average time for compiling a single instruction $\tilde{t}_{comp,insn}$.

$$t_{app,cs} = t_{ex,total} \quad (1)$$

$$t_{app,js} = t_{ex,total} + t_{comp,total} \quad (2)$$

$$t_{ex,total} = \tilde{t}_{ex,insn} * n_{exec}$$

$$t_{comp,total} = \tilde{t}_{comp,insn} * n_{insn}$$

Based on the equations (1) and (2) the instruction throughput p_{cs} (3) and p_{js} (4) of the two simulation techniques can be deduced.

$$p_{cs} = \frac{1}{\tilde{t}_{ex,insn}} \quad (3)$$

$$p_{js} = \frac{1}{\tilde{t}_{ex,insn} + \tilde{t}_{comp,insn} * \frac{n_{insn}}{n_{exec}}} \quad (4)$$

In (5) it is shown that for a growing number of instruction executions the performance of the just-in-time compiled simulator converges to the performance of a compiled simulator.

$$\lim_{n_{exec} \rightarrow \infty} p_{js} = p_{cs} \quad (5)$$

The number of repeatedly executed instructions needed for a good convergence of the compiled simulation speed very well corresponds to the conditions provided by real-world applications. This is due to the fact that most programs behave according to the 90/10 rule: 90% of execution time is spent in 10% of the code. For instance the proportion $n_{exec}/n_{insn} = 256$ corresponds to a loop that is iterated 256 times. Together with the valid assumption that $\tilde{t}_{comp,insn} = 4 * \tilde{t}_{ex,insn}$, 98.5% of the compiled simulation performance is achieved. The validity of the assumptions made here is proved in section 5.

4.2 Just-in-Time Simulation Compiler

The complete design of the JIT-CCS is based on the integration of the simulation compiler into the simulator. In fact, the run-time compilation of instructions requires a new concept for the simulation compiler. Based on the separation of the simulation compiler and the simulator the following conditions are fulfilled for traditional compiled simulators: the instruction decode time is not critical since it does

not influence the simulation performance. Furthermore, the simulation compiler is allowed to generate C-code which is subsequently compiled by a C-compiler to build a compiled simulator. Obviously, this is not true for an integrated simulation compiler. Especially, the run-time call of a C-compiler is not practical.

In order to explain the work-flow of the just-in-time simulation compiler, some important characteristics of a LISA processor model have to be introduced. A LISA model is a mixed structural/behavioral description of a processor. The structural part keeps the definition of processor resources like registers, memories and pipelines. The processor's instruction-set including instruction-coding, assembly syntax, functional behavior, and timing is contained in so-called *LISA operations*. A single processor instruction can be composed by multiple LISA operations. The following example shows an excerpt of a LISA processor description for a simple **ADD** instruction. Beside the definition of the program memory and the register resources, two examples for LISA operations are given. The operation *ADD* implements the binary coding, assembly syntax, and the functional behavior of the processor instruction **ADD**. The instruction-set information of the operand registers *src1*, *src2*, and *dst* are referenced from the inferior LISA operation *Register*. The operation *Register* describes the binary coding, assembly syntax of a single register within the processor's register file. A reference to the respective resource is returned for the use in the functional behavior of operation *ADD*.

```

RESOURCE {
    PROGRAM_MEMORY byte8 prog_mem[0x0..0x1000];
    REGISTER word32 R[1..15];
}

OPERATION ADD {
    DECLARE { GROUP dst,src1,src2 = {Register}}
    CODING { 0b01011 0b0000 src1 src2 dst}
    SYNTAX { "ADD" dst "," src1 "," src2 }
    BEHAVIOR{ dst = src1 + src2}
}

OPERATION Register {
    DECLARE { LABEL index; }
    CODING { index=0bx[4]}
    SYNTAX { "R" index }
    EXPRESSION { R[index]}
}

```

The presented structure of LISA processor models enables the following procedure. The behavioral C code of all LISA operations is pre-compiled into C-functions which are part of the simulator. The JIT simulation compiler selects the appropriate operations, which are required to simulate an instruction, on the basis of the coding information. References to the selected C-functions are subsequently stored in the simulation cache. These references are utilized by the simulator to execute the instructions' behavior.

4.3 Simulation Cache

The just-in-time simulation compiler stores the compiled data within a so-called simulation cache. The simulation cache is indexed by the non-ambiguous instruction address. For simplification purpose the cache size is considered to be unlimited first. Every time an instruction is going to be executed, the just-in-time simulation compiler looks up the cache entry that corresponds to the current program address. Before the previously cached data is used to accelerate the simulation, the validity is verified. Therefore,

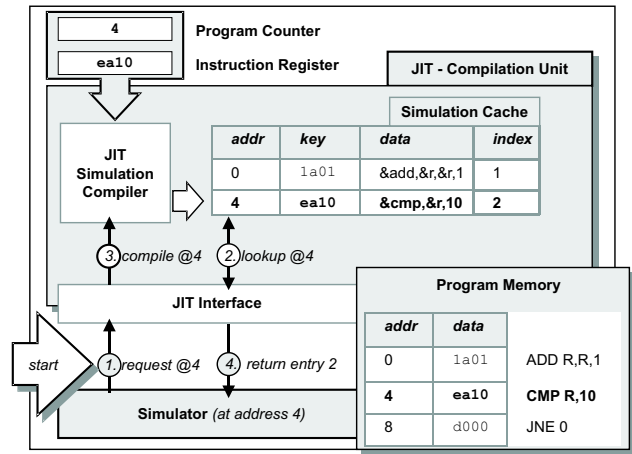


Figure 4: JIT-CCS Workflow

each cache entry contains a key, the instruction word, which is compared to the simulator's instruction register contents. If an instruction word change is detected, re-compilation is initiated at run-time. Often, processors with multi-word instructions, or parallel instructions within a VLIW bundle, require that the key is made up by more than one instruction register. Figure 4 illustrates the workflow of the JIT-CCS. The simulator is going to execute the instruction at address 4. Therefore the compiled data for the instruction at address 4 (1) is requested. The simulation cache management looks up the address in the simulation cache (2) and indicates that this address has not been compiled before. Therefore the JIT compiler is invoked (3) which stores the data together with the instruction word **ea10** in the cache. Eventually a reference to the table entry is returned to the simulator (4).

The presented workflow of the JIT compiled simulator shows that the major disadvantage of the compiled simulation technique, the requirement for fixed and predictable program code, does not apply for the JIT compiled simulator. However, another important parameter has to be considered – the application size. The size of the program code must be assumed to be run-time dynamic. Since the cache has a run-time static size, which is usually smaller than the application size, a replacement strategy is needed. Furthermore, a unique but not biunique mapping function has to be defined which describes the assignment of program memory addresses to cache addresses (hash-function). The cache key which has been initially used to identify changed program code, is now additionally employed to detect cache collisions. These collisions result from the fact that multiple program memory addresses share a single entry in the cache.

An important concern is the minimization of the cache administration overhead. Since the instruction compile time $\tilde{t}_{comp,insn}$ defines the penalty for a cache miss, the overhead for the cache administration has to be significantly smaller. To fulfill this constraint, a simple one-level cache with a direct address mapping was chosen (see figure 5). Each program memory address corresponds to exactly one entry in the cache. When a cache collision occurs the respective entry is directly overwritten by the new instruction. The direct address mapping assures a minimum cache miss rate ϵ_{miss} for spatially coherent code. Loops, the time critical parts of a program, directly benefit from this characteristic if the loop kernel fits into the cache.

The convergence of the JIT-CCS (*just-in-time cache compiled simulator*) instruction throughput to the performance

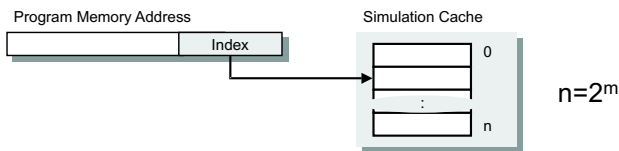


Figure 5: Direct Mapping

of the compiled simulator is now significantly characterized by the simulation cache miss rate ϵ_{miss} as shown in equation (6) (disregarding the cache administration). The results presented in section 5 will demonstrate that already small caches, which are sufficient to store the most time-critical loop of an application, achieve a very good performance approximation of the traditional compiled technique.

$$p_{jsccs} \approx \frac{1}{\bar{t}_{ex,insn} + \bar{t}_{comp,insn} * \epsilon_{miss}} \quad (6)$$

$$\epsilon_{miss} = f(\text{cachesize}) \quad (7)$$

4.4 Pipelined Processor Models

Previous explanations of the JIT-CCS have assumed that each cache address can be overwritten directly in a following simulator control step. While this is perfectly true for instruction accurate simulators where a complete instruction is processed within each control step, this assumption is not maintainable in case of cycle bases simulation of pipelined architectures. The lifetime of an instruction within the pipeline is no longer predictable. Therefore cache entries of instructions currently processed in the pipeline have to be protected against being overwritten. This is achieved by maintaining a look-up table that keeps the instruction addresses which are currently in use. Since the number of (VLIW) instructions present in the pipeline at the same time cannot exceed the number of pipeline stages, the size of the look-up table is defined by the number of stages. Concerning the cache access, three cases have to be considered :

1. cache hit,
2. cache miss, cache address not in use, and
3. cache miss, cache address in use.

In case of a cache hit an instruction can be taken directly from the compiled simulation cache. In the second case the instruction is not present in the cache, but the corresponding cache entry may be overwritten with the recently compiled data. Furthermore, the instruction address is registered in the look-up table. Table entries are written in a circular manner to displace instructions which have already left the pipeline. In the third case, a cache miss occurs and it is determined that the cached instruction which is to be replaced still remains in the pipeline. One solution would be a set-associative cache, however, due to the very unlikely occurrence of this case the simulator switches to interpretive mode for the particular instruction.

Figure 6 illustrates this procedure. The instruction `MOV Y, 1` at address `8000` is going to be decoded. Therefore the cache address is generated (1) and the corresponding entry is looked up in the cache (2). A cache collision is detected because the cache entry keeps the instruction `JMP 8000` from address `0`. Before the entry can be overwritten the pipeline-protection table has to be checked (3) whether the `JMP` instruction is still in use or not. Since this is the case, the instruction `MOV Y, 1` is not cache compiled but interpreted instead, without updating the cache.

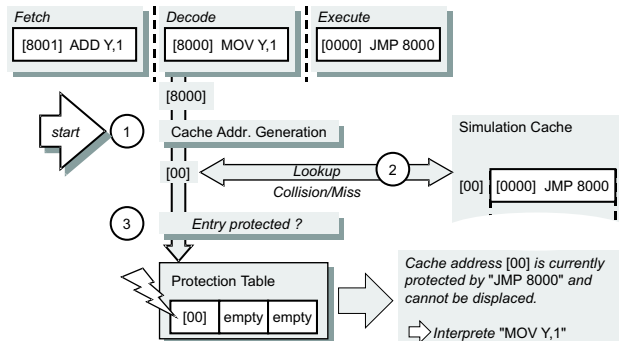


Figure 6: Cache Collision Scenario

5. RESULTS

The applicability and the efficiency of the retargetable just-in-time cache compiled simulation technique has been evaluated on the basis of various LISA processor models. The simulation results of the Advanced Risc Machines ARM7 and ST Microelectronics ST200 LISA models are presented in this section. The processor models have been developed within previous work and the generated tools were verified accurately against the vendor tools. Within this section the instruction throughput of the JIT-CCS will be compared to the performance of the interpretive and traditional compiled technique (both generated from LISA 2.0 descriptions). Therefore, the applications that have been selected for the benchmark fulfill the requirements of the traditional compiled simulation technique: constant program code. Furthermore, it is investigated how the performance of the JIT simulator is influenced by the cache size.

Performance results of the different generated simulators were obtained using an 1200 MHz Athlon PC, 768 MB RAM running the Microsoft Windows 2000 operating system. The generated simulator code has been compiled using the Microsoft Visual C++ 6.0 compiler with optimizations turned on (/O2).

Figure 7 shows the benchmark results of instruction accurate simulators for the ARM7 running a jpeg2000 codec. The dark-grey bars visualize the simulation performance for different simulators in MIPS (million instructions per second). The two leftmost bars show the performance of the compiled and interpretive simulation techniques. The remaining dark-grey bars correspond to the JIT-CCS performance for different cache sizes. The light-grey bars visualize the relative cache-miss rate for the examined cache configurations.

It is clearly recognizable that the JIT-CCS simulation performance is improving with a growing cache size. This effect becomes clear when looking at the continuously decreasing cache-miss rate. Examining the results it can be seen that more than 95% of the compiled simulation performance is achieved with a 4096 entries size cache. This cache allocates less than 2MB of host memory. Compared to the compiled simulator which requires approx. 23MB for all 47354 instructions of the application³ the JIT-CCS is very memory efficient. Considering big applications such as operating systems it is obvious that also the memory consumption of compiled simulators can become a serious problem. In contrast to compiled simulators the JIT-CCS is characterized by a user defined performance vs. memory trade-off. Due to the execution locality of programs the JIT-CCS perfor-

³The memory consumption of the compiled simulation is proportional to the application size.

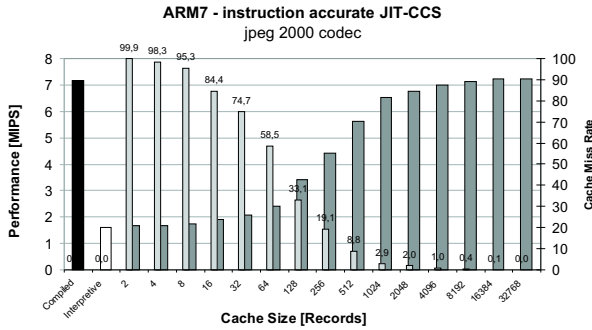


Figure 7: Simulator Performance ARM7 – jpeg2000
 performance saturation is achieved rapidly, resulting in manageable small simulation caches. Unlike the voluminous jpeg

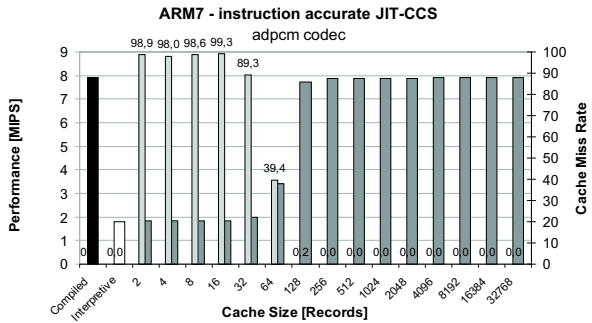


Figure 8: Simulator Performance ARM7 – adpcm
 2000 codec with large (>1024 instructions) loop kernels, typical DSP applications like the adpcm codec require much smaller caches. The biggest loop of the adpcm codec compiled for the ARM7 spans 122 instructions. The benchmark results for the adpcm codec presented in figure 8 reflect this characteristic. Here a cache size of 128 entries is sufficient to achieve 95% of the compiled simulation performance. The investigation of further applications has shown, that a reasonable big cache size (approx. 4096-16384) is absolutely sufficient for a >95% approximation of the compiled performance. Almost the same applies to the instruction accurate JIT-CCS of the ST200 architecture. Figure 9 shows the simulation results of an adpcm codec on this machine.

Compared to the ARM7 benchmark results, the extremely poor performance of the interpretive simulator is conspicuous. This is due to the complex instruction decoder which consequentially also influences the performance of the JIT-CCS for small caches. The ST200 processor is a VLIW architecture with many possible parallel instruction combinations. The 128bit instruction word (VLIW) allows multiple combinations of parallel instructions with and without extensions for immediate values. Together with distributed

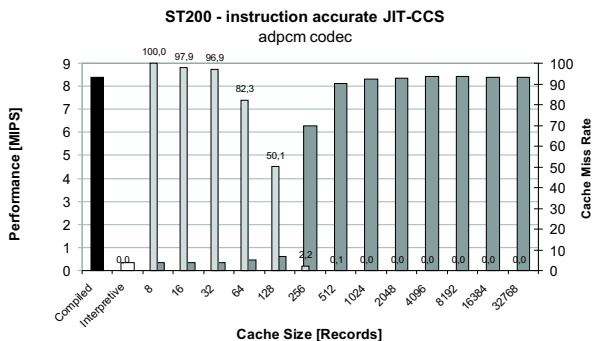


Figure 9: Simulator Performance ST200 – adpcm

op-codes the instruction decoding process of the software simulator dominates the actual behavioral execution. Here, a cache size of at least four entries is required to store the parallel instructions of the VLIW instruction word.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel instruction-set simulation technique, addressing retargetability, flexibility, and finally high performance at the same time. Results for various real world architectures have proven the applicability and the high simulation speed of this technique. Future work will concentrate on modelling further real world architectures and verifying the presented simulation methodology. Beyond simulation, current research aims at the retargeting of HLL compilers and the generation of synthesizable HDL descriptions from a LISA 2.0 processor model.

7. REFERENCES

- [1] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe*, 1999.
- [2] A. Hoffmann and T. Kogel and A. Nohl and G. Braun and O. Schliebusch and A. Wiefenink 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, 2001.
- [3] E. Schnarr and J.R. Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.
- [4] E. Schnarr and M.D. Hill and J.R. Larus. Facile: A Language and Compiler For High-Performance Processor Simulators. In *Proc. of the Int. Conf. on Programming Language Design and Implementation*, 1998.
- [5] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proc. of the Conf. on Measurement and Modeling of Computer Systems*, 1996.
- [6] F. Engel and J. Nuhrenberg and G.P. Fettweis. A Generic Tool Set for Application Specific Processor Architectures. In *Proc. of the Int. Workshop on HW/SW Codesign*, 1999.
- [7] G. Braun and A. Hoffmann and A. Nohl and H. Meyr. Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description. In *Proc. of the Int. Symposium on System Synthesis*, 2001.
- [8] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference*, 1997.
- [9] M. Hartoog J.A. Rowson and P.D. Reddy and S. Desai and D.D. Dunlop and E.A. Harcourt and N. Khullar. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *Proc. of the Design Automation Conference*, 1997.
- [10] C. Mills, S. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software - Practice and Experience*, 21(8):877–889, 1991.
- [11] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proc. of the Asia South Pacific Design Automation Conference*, 1999.