A RETARGETABLE, HIGH-PERFORMANCE ISA SIMULATOR IN JAVA

Marco Kaufmann, Thomas B. Preußer, Rainer G. Spallek

Institut für Technische Informatik
Technische Universität Dresden
01062 Dresden, Germany

{marco.kaufmann, thomas.preusser, rainer.spallek}@tu-dresden.de()

Abstract

This paper presents Jahris — a retargetable, high-performance ISA simulator entirely written in Java. It seamlessly integrates a high simulation speed and instruction-accurate observability, both of which are conflicting design goals, by a transparently adapting simulation execution. Jahris implements advanced just-in-time compilation techniques with a platform-independent hotspot engine targeting Java bytecode. It uses large translation units formed by branch-free instruction sequences to widely eliminate the simulation dispatch overhead. Jahris includes its own ISA description language, which aims at both a high simulation speed and the rapid modelling of target architectures. Its flexibility has been proven by the modelling and successful testing of a diverse set of architectures including DLX, i8086, ARMv4 and 32-bit PowerPC. The simulation performance has been determined to achieve up to 78 percent of that of QEMU with an extremely reduced modelling effort.
This paper makes two major contributions. Firstly, it proves the Java Runtime Environment (JRE) to be a virtual but feasible simulation platform. This enables the platform-independent implementation of high-performance JIT-compiling simulation engines, which take advantage of the widely-available and sophisticated JVM implementations. As these handle the further translation to the native code of the actual simulation host, Jahris further benefits from their internal and possibly platform-specific code optimization. Secondly, a novel ISA description language is introduced, which provides a strict separation between the decoder and the instruction behavior of ISA models. This allows the extraction of the behavior from code sequences to enable sophisticated simulation techniques such as behavior caching and block compilation.

Keywords: Retargetable, Dynamic compilation, ISA simulation, Hot Spot, Java

Presenting Author's Biography

Marco Kaufmann is a researcher at the Institute of Computer Engeneering at the Faculty of Computer Science of the Technische Universität Dresden. He received his Diplom in Computer Science from the Technische Universität Dresden in 2009 and is now working towards his Ph.D. His research interests include the modelling and simulation of computer systems and compiler and language design.

1 Introduction

Simulators have become an essential tool in the design process of instruction set architectures (ISAs). Serving as debuggers and profiling tools, they enable the continuous evaluation of design choices in the context of the targeted application domain prior to the actual implementation of the ISA. They further provide a virtual implementation of the architecture that not only allows an early start of the software development but also provides a valuable debugging environment.

ISA simulators are expected to meet several, often conflicting, demands. Most prominently, they should offer a high simulation speed while also providing the fine-grained observability of the target architecture. As to reflect the programmer's view on the target architecture correctly, the accurate observability is generally required for the instruction level whereas a cycle-accurate pipeline model is typically of no concern. However, already the computation of the consistent architectural states at all instruction boundaries for observability clearly impacts the simulation performance as it prevents the optimization of the execution of the instruction behavior across these boundaries.

Traditionally secondary demands for simulators are their portability, or even platform independence, and their retargetability. As these goals enable the preservation and reuse of achieved design and simulation know-how across host and target architectures, their importance grows with increasing numbers of special-purpose architectures with short-lived revisions. Unfortunately, the implementation of these demands typically also requires a compromise with respect to the simulation performance.

This paper presents Jahris, an ISA simulator offering a valuable tradeoff between the simulator design goals by demoting their mutual conflicts. A key and unique feature in the implementation of Jahris is the use of the Java platform as virtual simulation host. This platform offers a target for sophisticated dynamic compilation techniques, which enable the fast execution of the binary target applications. Nonetheless, Jahris itself remains platform-independent while even benefitting from platform-specific optimizations implemented in the dynamic code generators of the host JVMs. While the use of large translation units further increases the simulation performance, their fully transparent implementation allows an instant retreat to instruction-accurate observability. Finally, retargetability is achieved by an ISA description language, which enables sophisticated simulation techniques by a clean separation of concerns between instruction decoding and execution, and yet allows the rapid modelling of target architectures for the simulator.

After section 2 has given an overview on the related work and state-of-the-art simulation techniques, section 3 will describe Jahris' simulation engine. In section 4, the basic principles of Jahris' ISA description language, *HPADL*, are explained. Benchmark results and a comparison with QEMU are, finally, given in section 5 before section 6 concludes this paper.

2 Related Work

The simplest approach to ISA simulation is its implementation as an interpreter. Such simulators fetch and decode every target instruction each time it is to be executed and apply their behavior to update the state of the simulated architecture. Besides its simplicity, this approach is very flexible providing natural support for self-modifying or dynamically-reloaded code and even allowing computationally adapted instruction sets [1]. Its simulation of the target architecture is also faithful and allows the inspection of a consistent architectural state at any time. Unfortunately, these benefits come at the cost of a low simulation performance not acceptable for the simulation of larger target applications.

On the other extreme, there is the statically-compiled simulation as described by Mills et al. [2]. It essentially moves the continuous decoding overhead of the interpretive execution to a one-time compilation effort. The compiler fetches and decodes the instructions of the target application, performs a static code analysis and compiles the application into native code of the host machine ahead of time. This leads to a significant speedup of the simulation. However, this comes at the expense of observability and flexibility. A consistent architectural state can only be provided at pre-determined checkpoints. Self-modifying or dynamically-generated code is not supported. The long startup time caused by the compilation step can only be amortized by long simulation runs. The runtime profile of the target application cannot be considered so that even code blocks, which are actually never invoked during the simulation, are compiled.

A compromise between both of these extremes is established by the dynamically-compiling ISA simulation, which aims at combining both performance and flexibility. This approach does not compile the target application ahead of time. Instead, the code blocks of the target application are fetched and decoded during the simulation just in time before their initial execution. The compiled results are cached for possible later reuse. This compilation on demand enables a short startup time while still eliminating more and more of the decoding overhead as the simulation proceeds. The compilation is limited to the code blocks actually encountered in the execution of the simulated application. Code modified or added dynamically is easily handled by this approach and merely requires a careful management of cached compilations. Moreover, execution profiling can be employed to intensify the optimization of the compilation of application hot spots.

An implementation of a dynamically-compiling ISA simulator with a pipeline-accurate modelling of the target architecture and a cycle-accurate ISA simulation is described by G. Braun et al. [3]. It is based on a LISA specification [4] of the target architecture's micro operations. A hybrid approach combining static and dynamic compilation has been proposed by Reshadi et al. [5]. They practice an as-far-as-possible static compilation of the target application ahead of time. Only code blocks not discovered during the static analysis or modified during the simulation are compiled dynamically.

Jones and Topham suggest the use of large translation units for the dynamic compilation of target applications into native host code [6]. This improves the simulation performance in two aspects. First of all, the dispatch of the individual code blocks is further reduced so that a larger share of the simulation time can be designated to the behavior execution. Secondly, larger translation units provide the dynamic compiler with a greater scope for optimization. Their ISA simulator *EHS* supports a fast *Dynamic-Binary-Translation* simulation mode using large translation units as well as a slow interpreter mode for instruction-accurate simulation.

Dynamically-compiling simulation techniques are also employed by the full-system-emulator QEMU [7]. It uses micro operations, which are compiled into the platform-independent representation as *TCG*[1] *ops* ahead of time. The dynamic compilation then assembles these micro operations according to the instructions contained in a code block to feed a TCG backend for native host code generation [8]. A simulation cache is utilized for translation units similar to basic blocks. A further increase of performance is achieved by *Direct-Block-Chaining*, a technique that tries to bypass the cache to jump to translated blocks directly. QEMU achieves very high simulation speeds. However, its field of application is emulation rather than simulation. Thus, observability or instruction-accurate modelling are of no concern. The retargation of QEMU requires the manual modelling of an architecture in C using the provided framework. There is no designated ISA description language.

3 The Simulation Engine

3.1 Simulation Architecture

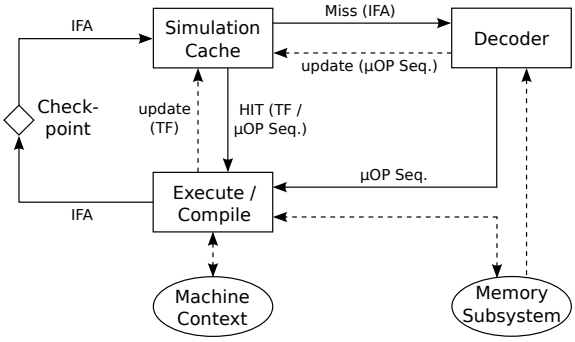
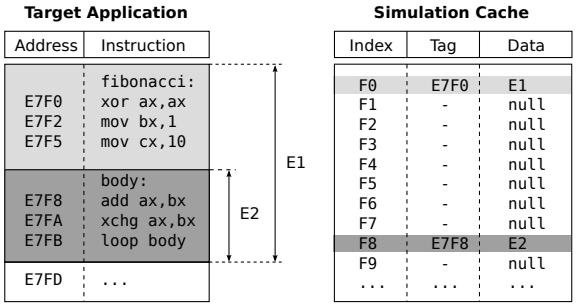


Fig. 1 - Simulation control flow

The core control flow of the simulation loop is depicted in Fig. 1. A single pass always starts at a *checkpoint* representing a consistent state of the simulated architecture. This state is transformed into a new one by the execution of an *execution unit*. The appropriate execution unit is identified by the current *instruction fetch address* (IFA) and may be a *micro-operation sequence* (μOP sequence) to be interpreted or a compiled *translated function* (TF). It is first looked up in the *simulation cache*.

[1] Tiny Code Generator [8]



For the purpose of clearness, the target application is displayed in assembler mnemonics, though actually fetched and decoded in binary. Also, the default cache size is 1M rather than 256 entries as suggested by the cache index in the figure.

Fig. 2 - Overlapping execution units (E1 and E2)

Only if it cannot be located there, the decoder will be involved to construct a micro-operation sequence from the binary application image in the memory of the target architecture. The decoded μOP sequence is buffered in the simulation cache. It will itself trigger its replacement by an equivalent translated function when its execution counts exceeds a configurable *compilation threshold*, which currently defaults to 10 runs. The final output of each execution unit is the next IFA, which interpreted μOP sequences derive explicitly from the architectural state whereas translated function include an optimized compiled computation.

Jahris uses a simulation cache to store execution units that have already been fetched and decoded. The used direct-mapped cache is of configurable size. It is indexed by the lower part of the IFA, which in whole is used as the tag associated with a stored execution unit. The current cache implementations allows an IFA of up to 64 bits.

The decoder is responsible for the initial fetching and decoding of the application code into functionally-equivalent μOP sequences. These are composed directly from the original instruction stream of an execution unit. The input to the decoder is the initial IFA of the execution unit and the memory bus, from which to fetch the instruction words. Note that the decoder may not further process the current architectural state so as to produce a true cachable behavioral model of the execution unit that is applicable to this and all its future invocation without regard to the particular architectural state.

3.2 Execution Units

The smallest execution unit possible for a non-cycle-accurate ISA simulator is a single target instruction. As suggested previously (Sec. 2), larger execution units lead to a higher simulation performance. Similar to QEMU, Jahris uses block execution units, which reach from an entry point (label) up to and including the next control-transferring target instruction. It may span an arbitrary number of instructions. Because such a block may include further entry points, they are not identical to what is known as basic blocks. Execution units may,

in fact, even overlap with one being the suffix of the other with a distinct entry point. An illustrating example for such a situation is shown in Fig. 2.

Overlapping execution units in the simulation cache imply that their overlapping parts have been fetched and decoded multiple times. While this certainly incurs an additional decoding overhead, it also maximizes the size of the code sequences that can be executed without an intermediate dispatch. A thorough enforcement of mutually exclusive execution units would also require either a static control flow analysis to identify all possible entry points or the discarding of already compiled units when a previously unknown entry point is discovered during the simulation. These approaches do, however, not promise any benefit that would justify their costs.

3.3 Hot-Spot Simulation Compiler

Decoded execution units in the simulation cache are stored either in their interpretive form as a micro-operation sequence (μOP sequence), or in their compiled form as a *Translated Function* (TF). The cached μOP sequences keep track of their invocation count. If this invocation count exceeds the configurable compilation threshold, the particular μOP sequence is compiled into an equivalent TF. This step assumes that execution units that were invoked frequently in the past, i.e. the execution *hot spots*, will also be invoked frequently in the future so that the time spent for the compilation will actually amortize by the accelerated future runs of the execution unit. The configurable compilation threshold defaults to 10 executions. The special threshold values of 0 and -1 request an instant compilation and a permanent interpretation, respectively.

The translation of execution units into native code of the host typically involves two compilation steps, only the first of which is implemented by Jahris itself. In fact, Jahris assembles appropriate Java bytecode into a specialized class representation of a TF and holds it prepared in a custom class loader. From there, the class is loaded into the host Java Virtual Machine (JVM) and instantiated for the simulation cache entry to be used in the future. In modern JVM implementations, the execution of the TF will again be profiled as to trigger another hot-spot compilation step eventually. Thereafter, the TF will even be executed as native code implementation on the host machine.

The two-stage compilation approach implemented for Jahris may appear somewhat expensive. Indeed, it involves the generation of Java bytecode as intermediate representation which is embedded into a flattened class-file structure, which then needs to be parsed and verified by the host JVM. Nonetheless, the use of the JVM as quasi-host for Jahris also bears many benefits. First of all, the Java platform offers a compilation target that itself is platform-independent. It is this feature that allows Jahris to implement a hot-spot simulation compiler that is available an all host machines providing a JVM. Classic dynamic compilers, on the other hand, would have to rely on a platform-dependent compiler backend. Furthermore, Jahris benefits from the highly-sophisticated compilers integrated in modern JVM implementations. All their compilation know-how even including platform-dependent code optimizations can be exploited by Jahris. Its own bytecode compiler can, thus, avoid any larger own effort in optimization. Note that even most Java compilers are non-optimizing leaving this effort to the JVM, which can even make improved optimization decisions based on the actual host machine architecture and on application profiling information.

3.4 Breakpoints, Stepping and Asynchronous Halt

At the end of each simulation cycle, the checkpoint of the simulation loop is passed (cf. Fig. 1) and the simulation break flag is checked. If it is set, the simulation is stopped. Thus, the break flag enables the asynchronous simulation halt with a precision to the execution unit. As execution units do never contain branches and, thus, no loops, their execution must terminate and the halt of the simulation can be guaranteed. Also, the limited precision of the halt to the boundaries of execution units is not a deficiency due to the asynchronous nature of the halt, which makes no assumption about the simulation time within execution units.

Despite the use of large execution units possibly comprising a long sequence of target instructions, Jahris also facilitates the synchronous, instruction-precise simulation halt via instruction stepping and breakpoints. The implementation of both these features is non-invasive and does neither rely on any particular support by the simulated target architecture nor on a modification of the application binary. Whenever the simulation halts, the target architecture is, of course, in a consistent state, which can be inspected and modified by the user.

Breakpoints are managed by the simulator and may be set at any location of the target application, even inside execution units. When a breakpoint is set, all execution units that contain its code location are removed from the simulation cache. Hence, the invocation of the affected code must necessarily cause a cache miss and the decoder will be requested to assemble a new execution unit. Whenever it encounters a breakpoint during this process, it will discontinue the assembly at this point. The resulting incomplete execution unit will be executed and, as desired, end right at the breakpoint. To avoid spoiling the cache with fragments, the so obtained execution units are not cached. Note that the simulation and its performance up to the code section containing a breakpoint is not affected at all. Thus, breakpoints do not downgrade simulation performance. Jahris will also restore expelled execution units to the simulation cache as soon as the last breakpoint in their covered code section is cleared.

Instruction stepping makes the simulation proceed by a precisely-defined number of instructions unless a breakpoint is encountered before. Its implementation relies on a counter, which maintains the count of instructions yet to go. Before each invocation of an exe-

cution unit, it is verified that it contains fewer instructions than the current counter value. If so, the counter is decremented accordingly and the execution unit is invoked. Otherwise, the decoder is queried for an auxiliary incomplete execution unit comprising exactly the remaining number of instructions. As with the incomplete execution units produced for breakpoints, these are not cached. Note again that all but the last execution unit are executed normally on a whole. Merely, the management of the counter adds a marginal overhead as compared to the regular simulation.

4 The Architecture Description

The retargetability of Jahris is achieved by its own ISA description language *High Performance Architecture Description Language* (HPADL). It is used to describe target architectures entirely and aims at both the rapid ISA modelling and the compatibility with advanced simulation techniques such as simulation caches and large translation units. Its key feature is the strict separation of concerns between the instruction decoding and the instruction behavior.

An ISA description in HPADL is subdivided into three sections: the structural section, the decoder description and the behavioral description. This section covers the basic principles of each of them. Amongst others, a complete language specification including example code and hints for the creation of efficient architecture descriptions is given by Marco Kaufmann in [9].

4.1 Structural Section

The structural section models the memory subsystem and the machine context of the target architecture. The memory subsystem consists of an arbitrary set of busses, each of which has its own autonomous address space. A bus specification contains the name by which the bus is identified in the scope of the architecture description, the specification of its access interfaces and a device map. The latter instantiates representatives for bus devices and maps them into the address space. Access interfaces are specified by their endianess, the size of an addressable unit in bits, and the allowed access widths in addressable units. Uncommon values such as an addressable unit of 7 bits or accesses of 3 addressable units are supported. For bus accesses in the behavioral description, the requested access interface is selected by its identifier.

The predefined classes of bus devices comprise RAM and ROM components. Additional user-defined classes can be included into an architecture description using the `include` clause. User-defined bus device classes must be provided as Java classes. They may realize aribitrary device functions, and even provide an interface to the host plattform. For example, they can be used to implement IO devices such as timers, keyboards, UARTs or VGA interfaces. Thus, Jahris is also applicable as a full-system simulator. An example for a memory subsystem including user-defined bus device classes is provided by listing 1.

```
1  import io.Timer;
2
3  bus mem {
4      unit = 8;
5      endian = BIG;
6      access {
7          byte = 1;
8          word = 2;
9          dword = 4;
10     }
11     devices {
12         export ram mem {
13             base = 0;
14             size = 0x40000000;
15         }
16         export io.Timer timer {
17             base = 0x40000000;
18             size = 8;
19         }
20     }
21 }
```

Listing 1 - Example memory subsystem description

```
1  context {
2    // standard registers
3    export int pc;     // program counter
4    export int iar;    // interrupt address
5    export int[32] r; // general purpose
6
7    // floating-point registers
8    export boolean fpsr; // status
9    export int[32] f;    // scratch
10 }
```

Listing 2 - Example machine context declaration (DLX)

The machine context is the set of variables that - aside from the memory subsystem - define the state of the target architecture. This, at least, comprises variables to represent the register set and may further include auxiliary variables of the machine context that are required in the scope of the behavioral description. An example for a machine context declaration is given in listing 2.

4.2 Decoder Description

As to enable the caching and re-use of decoded and possibly dynamically-compiled execution units, the fetch and decode process must necessarily be indipendend from the instruction execution process and thus indipendend also from the current machine context. In HPADL, this is achieved by a strict separation of concerns between instruction decoding and instruction behavior whereas other description languages typically mingle them.

The decoder description includes the term for the instruction fetch address (IFA), the decoder context specification and decoder operations. The IFA has an exceptional position, as it resides in the decoder section though it is actually not a part of the instruction decoder and even depends on the current machine context. The IFA expression is evaluated by the simulation engine in order to determine the address of the next execution unit. Only in the case of a cache miss, this is

actually passed back to the decoder as the initial fetch address. Note that the IFA computation may be more complex than evaluating a single PC register. So, the term `16 * CS + PC` is used for the i8086 architecture.

The decoder context contains all variables that must be globally visible to the decoder operations as well as those variables that are to be exported to the instruction execution context. The behavior of the instruction decoder is specified in terms of decoder operations. They fetch and decode an instruction word or parts of it and generate the appropriate μOP sequence. Arbitrary complex decoder algorithms can be specified. Incremental fetching and processing of instruction words is possible. Decoder operations may also invoke each other so that a modular decoder design is enabled. This makes decoder descriptions very compact and flexible so as to allow simple instruction decoders as well as decoders for complex variable-length instruction word formats.

Execution units are terminated by control transfer instructions, because they might break the sequential control flow of the target application (cf. Sec. 3.2). Such instructions may not be obvious to the instruction decoder as the change of the control flow may be an occasional side effect of rather innocuous instructions such as an arithmetic instruction having `R15`, the program counter, as destination on some ARM architectures. Therefore, HPADL requires that control transfer instructions be marked explicitly. For this purpose, the predefined μOP `exit` is provided, which has no behavioral meaning but merely tags control transfer instructions. Whenever this μOP is contained in the subsequence to be appended for another target instruction, the decoding processes completes after this instruction. Note that the exact position of this μOP within the subsequence is irrelevant.

Listing 3 provides a decoder example in HPADL using an excerpt from the DLX decoder description.

4.3 Behavioral Description

The behavioral section defines the instruction execution context and the μOPs available to the instruction decoder. The μOPs may freely implement any behavior. All memory busses of the target architecture are accessible, and both the machine and the instruction execution context are visible within the behavioral description. As an example, listing 4 shows an excerpt from the DLX behavioral description.

The instruction execution context contains variables that are globally visible to all micro operations. Unlike variables of the machine context, the lifespan of these variables is bound to one particular execution of one instruction. There is no memory of their value after this execution of the instruction completes. As these variables are implemented more efficiently than those in the machine context, all variables that need not keep their state across individual instruction executions should be put into the instruction execution context. A typical application for such variables is the transfer of intermediate results between μOPs.

```
1  decoder {
2      // instruction fetch address
3      fetch {
4          bus = mem;
5          pc;
6      }
7      // decoder context
8      context {
9          int iw, s1, s2, d;
10     }
11     // main decoder operation
12     default = dlx;
13     // j-format instruction operand
14     operation j {
15         s2 = (iw << 6) >> 6;
16     }
17     ...
18     // fetch and decode instruction
19     operation dlx {
20         iw = fetch.word;
21         IPC;
22         decode iw[26 to 31] {
23             0x00: alurr;
24             0x01: fpurr;
25             0x02: {j; J; exit;}
26             0x03: {j; JAL; exit;}
27             0x04: {i; BEQZ; exit;}
28             0x05: {i; BNEZ; exit;}
29             0x06: {i; BFPT; exit;}
30             0x07: {i; BFPF; exit;}
31             0x08: {i; ADDI;}
32             ...
33         }
34     }
35 }
```

Listing 3 - Example decoder description (DLX)

The instruction execution context may further import variables from the decoder context. These are read-only and keep a constant value for the whole execution of the target instruction. Such variables are even more efficient in simulation as the simulation compiler replaces them by constants. These read-only variables are suitable for all values the decoder can already compute such as immediates extracted from the instruction word.

5 Benchmark and Comparison with QEMU

The target application employed for the benchmark was a sieve test for the ARMv4 architecture. Within one iteration, it computes a prime number sieve in the memory of the target arcitecture covering the numbers 3 to 16381. For comparison, this application was executed on both Jahris and QEMU. The performance of both simulators can be expected to increase with longer simulation runs as the impact of the dynamic compilation costs decreases. For the verification of this hypothesis, performance measurements were obtained for different numbers of sieve iterations ranging from 1 000 to 10 000 executions within one benchmark. The employed performance metric is the average number of target instructions executed per second throughout the

```
1  behavior {
2      // instruction execution context
3      context {
4          decoder s1,s2,d;
5          float fs1,fs2,fd;
6          double ds1,ds2,dd;
7      }
8      // micro operations
9      operation IPC  {pc += 4;}
10     operation J    {pc += s2;}
11     operation JAL  {r[31] = pc; pc += s2;}
12     operation BEQZ {if (r[s1] == 0) pc += s2;}
13     operation BNEZ {if (r[s1] != 0) pc += s2;}
14     operation ADD  {r[d] = r[s1] + r[s2];}
15     operation ADDI {r[d] = r[s1] + s2;}
16     operation LW  {
17         r[d] = mem.word[r[s1]+s2];
18     }
19     operation LBU {
20         r[d] = mem.byte[r[s1]+s2];
21     }
22     operation LHU {
23         r[d] = mem.half[r[s1]+s2];
24     }
25     ...
26 }
```

Listing 4 - Example behavioral description (DLX)

Tab. 1 - Host configuration

CPU	2 × Dual-Core AMD Opteron(TM) @2.4GHz
Cache	L1: 2×128 kb, L2: 2× 1 MB
Main Memory	4GB
OS	Ubuntu 9.04, Kernel 2.6.28-16-generic
JRE	Java(TM) SE Runtime Environment (build 1.6.0_16-b01)
JVM	Java HotSpot(TM) 64-Bit Server VM (build 14.2-b01, mixed mode)
QEMU Version	0.10.0
QEMU Target	ARM Versatile/PB (ARM926EJ-S)
QEMU Guest OS	Linux 2.6.18 for Versatile (Debian 2.6.18.dfsg.1-23+versatile)

run of the benchmark specified in Million Instructions Per Second (MIPS). The same ARMv4 target application binary in ELF format was used for the Jahris as well as for the QEMU benchmark test both executed on the same host machine. The configuration of this host is summarized in Tab. 1.

The obtained results are summarized in Tab. 2 and Fig. 3. As expected for both simulators, the simulation performance increases with the length of the run. The performance of Jahris consistently amounts to 63 to 66 percent of that of QEMU regardless of the benchmark size. Note that the performance of Jahris can be increased significantly by running the JVM with more aggressive non-default options:

- `-Xverify:none` deactivates the bytecode verifier, thus, reducing the costs of class loading also for the simulation compiler.
- `-XX:+AgressiveOpts` enables aggressive code optimization, which raises the execution speed at the cost of additional compilation effort for the bytecode to native code translation. This option is particularly valuable for longer simulation runs.

Tab. 2 - Jahris and QEMU simulation performance

Sieve-Benchmark		Simulation Performance		
Run length		QEMU	Jahris	
Iter.	Instructions	MIPS	MIPS	%[1)]
1 000	494832396	174.24	114.70	65.83
2 000	989664396	181.59	119.57	65.85
5 000	2474160396	200.45	126.01	62.86
1 0000	4948320396	206.18	129.26	62.69
			161.38[2)]	78.27[2)]

1) compared to QEMU

2) with JVM switches (see text)

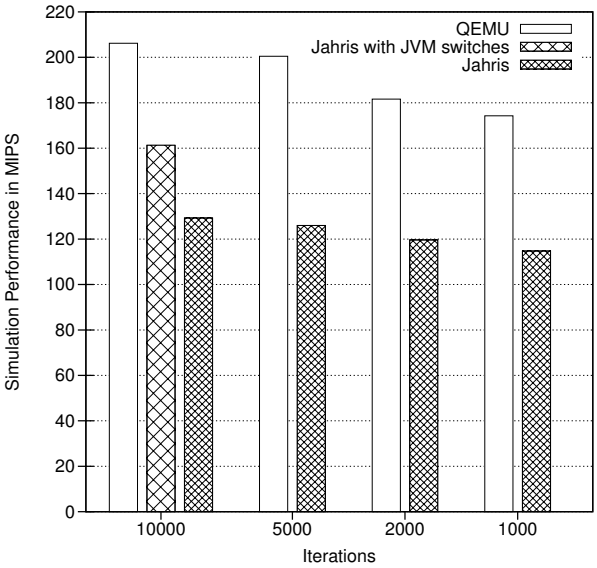


Fig. 3 - Jahris and QEMU simulation performance.

As shown for the benchmark of 10 000 sieve iterations, Jahris even achieves 78 percent of the QEMU performance with these switches.

While QEMU is, indeed, somewhat more performant than Jahris, its designated field of application is also quite different. Rather being a system emulator than simulator, observability or even instruction-accurate modelling are of no concern to it. Furthermore, the retargation of QEMU requires the manual modelling of the target architecture in C merely using the provided framework. Jahris, on the other hand, utilizes a compact designated ISA description language for the rapid modelling of target architectures. The greatest additional benefit of Jahris is its unmatched platform-independence achieved by the use of the Java virtual machine. This even allows Jahris to profit from any improvements in the JVM execution engines without requiring its own development effort as for a highly-sophisticated native compilation backend. Given these advantages, the achieved performance of up to 78 percent of that of QEMU is considerable.

6 Conclusion

In this paper, a retargetable, high-performance ISA simulator was presented that seamlessly integrates high simulation speeds and instruction-accurate observability. It is entirely implemented in Java and its hot-spot simulation compiler itself targets Java bytecode. This makes Jahris highly portable and even allows it to take advantage of the JVM's dynamic, highly-sophisticated code optimization. Jahris applies advanced simulation techniques such as the hot-spot compilation of the target application into Java bytecode and large translation units. It has been showed that high simulation speeds are achieved, which compare well with QEMU. The Java Virtual Machine has been proven to be a suitable platform for ISA simulation.

Jahris can be retargeted by HPADL, a new ISA description language that integrates rapid ISA modelling, flexibility and high simulation performance. This is achieved by a strict separation of concerns between the decoder and the instruction behavior within ISA models. Its flexibility has been proven by the modelling and successful testing of a diverse set of architectures including DLX, i8086, ARMv4 and 32-bit PowerPC.

Continuing work will focus on improved support for dynamically generated code and the further increase of the simulation performance.

7 References

[1] Thomas B. Preußer, Steffen Köhler, and Rainer G. Spallek. Modelling and simulating dynamic and reconfigurable architectures for embedded computing. In Yskandar Haman and Gamal Attiya, editors, *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*, 2004.

[2] C. Mills, S. C. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software, Practice and Experience*, 21(8):877–889, 1991.

[3] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 22–27, New York, NY, USA, 2002. ACM.

[4] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa machine description language and generic machine model for hw/sw co-design. In W. Burleson, K. Konstantinides, and T. Meng, editors, *VLSI Signal Processing, IX, 1996.*, October 1996.

[5] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 758–763, New York, NY, USA, 2003. ACM.

[6] Daniel Jones and Nigel Topham. High speed cpu simulation using ltu dynamic binary translation. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] Fabrice Bellard. Tiny code generator. `http://svn.savannah.gnu.org/svn/qemu/trunk/tcg/README`, 2009.

[9] Marco Kaufmann. Erschließung von Just-in-Time-Compilierungstechniken in der Realisierung eines retargierbaren Architektursimulators, December 2009.