Precise Simulation of Interrupts using a Rollback Mechanism [*]

Florian Brandner
Vienna University of Technology
brandner@complang.tuwien.ac.at

Abstract

Instruction set simulation based on dynamic compilation is a popular approach that focuses on fast simulation of user-visible features according to the instruction-set-architecture abstraction of a given processor. Simulation of interrupts, even though they are rare events, is very expensive for these simulators, because interrupts may occur anytime at any phase of the programs execution. Many optimizations in compiling simulators can not be applied or become less beneficial in the presence of interrupts.

We propose a rollback mechanism in order to enable effective optimizations to be combined with cycle accurate handling of interrupts. Our simulator speculatively executes instructions of the emulated processor assuming that no interrupts will occur. At restore-points this assumption is verified and the processor state reverted to an earlier restore-point if an interrupt did actually occur. All architecture dependent simulation functions are derived using an architecture description language that is capable to automatically generate optimized simulators using our new approach.

We are able to eliminate most of the overhead usually induced by interrupts. The simulation speed is improved up to a factor of 2.95 and compilation time is reduced by nearly 30% even for lower compilation thresholds.

1. Introduction

The development of future generation micro-processor architectures heavily depends on fast and accurate simulation tools that allow the runtime characteristics of these processors to be measured under real workloads. This is especially true for embedded systems, where application specific instruction extensions are often used to improve the performance for a specialized domain, e.g., video or speech processing.

The highest level of accuracy is achieved using simulation of hardware components at the register transfer level (RTL). Event-based simulation of logical elements and wires is complex, and thus prohibitively slow for architecture evaluation and design space exploration. Architecture simulation frameworks, such as the Liberty Simulation Environment [30] and Unisim [1], operate at a higher level of abstraction, which results in improved simulation speed. Nevertheless, event-based modeling is too slow for the simulation of complex benchmarks executing billions of cycles.

An alternative technique based on statistical sampling avoids the complex simulation of architectural features and relies on estimates derived from a limited set of characteristics gathered during the execution of an instrumented version of the benchmark program. This approach leads to very fast execution at the expense of accuracy.

Instruction set simulation is a popular alternative that balances the need for fast simulation and the need for accurate data. The simulation is limited to user-visible architectural features of the instruction-set-architecture abstraction of the given processor. Other details are usually omitted and only modeled to guarantee correctness. The simplest form of instruction set simulation is based on *interpretation*, where each instruction of the simulated architecture is assigned one or more simulation functions that model the execution of the complete instruction or phases thereof. These functions are invoked from within a simulation loop, which controls the decoding of new instructions and the execution of the program. Interpreters usually have a low implementation complexity and can easily be adapted to new architectures.

Compiling simulators translate instructions to native code of the host machine to improve simulation speed. Overhead, such as the repetitive decoding of the same instructions and the dispatch of the simulation functions, is eliminated and compiled into the native code. In the case of *static-compiling* simulators the translation is performed offline before the simulation run, while for the *dynamic-compiling* approach the translation is done at simulation time. Both approaches are often combined with interpretation to allow the execution of code not statically known in the first and to reduce the compilation overhead in the latter case.

[*] © 2009 by EDAA.

Interrupts cause considerable overhead for compiling simulators and may drastically reduce optimization opportunities. The problem arises from the asynchronous nature of interrupts that can redirect execution at any phase of the simulation and may cause arbitrary instructions to be aborted. This impedes aggressive optimizations across instruction boundaries, because all intermediate states need to be retained. In addition, extra code is required for the interrupt dispatch and pipeline control. As a result, cycle-accurate simulation of interrupts is often avoided by postponing the actual dispatch to specific points, e.g., basic block boundaries. However, this simplification may result in false assumptions on the distribution of interrupts and may lead to unexpected behavior of the same program running on a real processor.

In this work we present an effective simulation technique that is able to combine fast dynamic-compiled simulation with cycle-accurate handling of interrupts using a rollback mechanism. The code generation phase of the simulator assumes that interrupts do not occur within the translated code sequence and generates aggressively optimized code. Before storing the architectural state this assumption is verified. In almost all cases this assumption is correct and simulation proceeds normally. However, if an interrupt is found to be pending, the simulation needs to revert to a previous state and restart the simulation using an interpreter that faithfully models interrupt handling. We have implemented the rollback mechanism using a simulation framework based on the open source compiler infrastructure LLVM 2.3 [21] and an architecture description language (ADL) [6] that generates all architecture dependent components. Programs of the simulated architecture are executed using a mixed approach based on interpretation and dynamic compilation with two optimization levels.

The main contributions of this work are as follows:

- An efficient simulation technique for interrupt handling in an aggressively optimizing dynamic-compiling simulator.
- An ADL-based simulation framework that automatically optimizes the generated simulator using our new approach.
- A detailed evaluation of our new approach with respect to compilation time and simulation time.

The following section gives an overview of related work. Section 3 introduces the fundamentals of our ADL and presents how interrupts can be modeled. A basic overview of the simulation framework is given in Section 4. In the following section, we show how interrupts can be handled in a compiling simulator and present our new rollback mechanism. Section 6 presents detailed data of our experimental evaluation. Finally, we conclude in Section 7.

2. Related Work

Architecture simulation is of utmost importance for designing future processor generations, it is thus heavily researched – a broad overview is presented in [34].

Simulation techniques aiming at fast calculation of estimates often use sampling and statistical models [31, 33, 13]. Based on a calibration run, either offline prior to the simulation or during the simulation, a statistical model is derived that is able to predict cycle numbers and power consumption within an error margin of a few percent. Gao et.al. [16] integrate the native execution of C programs on the host machine with simulation. The system is able to switch between the native program and simulation at function boundaries and thus allows to skip large parts of the initialization phase of the program. The detailed simulation is limited to relevant parts only.

There are several simulation frameworks available using interpretation and event-based simulation: Simics [22], SimpleScalar [2], Liberty [30], Unisim [1], and many others. Unisim is tightly coupled with SystemC[1] and can interface with other SystemC components, e.g., external devices, buses, and other simulators.

SimOS [28] is a full-system simulator that offers various simulators including Embra [32] a fast dynamic translator. Embra focuses on efficient simulation of the memory hierarchy, including memory address translation, memory protection and caches. Shade [8] is another dynamic-compiling simulator that aims at fast execution tracing. It offers a rich interface to trace and process events during simulation. Jones et.al. use large translation units (LTUs) to speed up the simulation of the ARC architecture. Several basic blocks are translated at once within an LTU, leading to reduced compilation overhead and improved simulation speed. This technique is similar to regions that are used within our simulation environment. Cycle-accurate simulation of interrupts is usually not a major objective of the mentioned simulation frameworks and is thus not optimized further. Usually interrupts are processed at the boundaries of translated code, i.e., basic blocks.

Ebcioğlu et.al. present DAISY [12, 11] a VLIW architecture designed specifically for fast binary translation. Gschwind and Altman [18] use the hardware rollback mechanism of DAISY for aggressive optimization of the translated code. The technique is similar to our approach, however, DAISY does not perform cycle-accurate simulation. External interrupts are processed in hardware, but require support from the compiler to determine program points that are safe to actually deliver the interrupt, e.g., to prevent interrupts to be serviced while an instruction is simulated. Similar approaches are found in BOA [29] and Transmeta's Crusoe [10].

[1]`http://www.systemc.org`

Bala et.al. use Dynamo [3] to gather profile information during the execution of a program and apply optimizations accordingly. Dynamo is not intended for architecture simulation and is limited to native programs of the host machine. Nevertheless, the applied techniques are the same and are adapted by many architecture simulators.

Deriving simulators and other software development tools from a concise architecture specification can be done using several ADLs. Deriving a static-compiling simulator and accompanying compilers is possible using DSPXplore [14, 15], Expression [27, 20] and LISA [7, 26]. The ADL used in this work allows to derive a compiler [6, 4] and a dynamic-compiling simulator [5]. Nohl et.al. present a dynamic-compiling simulator based on the LISA language [24]. For most languages it is not clear how interrupts are described and finally simulated. The interrupt dispatch is usually embedded into the behavioral description of every instruction, which impedes automatic optimization.

LLVM is a open source compiler infrastructure that offers both static and dynamic code generation facilities. The dynamic compiler has successfully been used to implement a Java Virtual Machine [17] that is competitive compared to other open source and commercial Java implementations. Criswell et.al extend LLVM to define a safe virtual architecture SVA [9]. The Linux operating system has been ported to SVA and can be executed in a controlled and secure environment.

3. Architecture Description

The ADL used in this work targets in-order-pipelined architectures and VLIW processors. The language itself is based on XML and models the architecture using a structural approach, i.e., the modeling focuses on the hardware structure instead of the instruction set. An architecture specification consists of three major parts: A configuration section, declarations, and instantiations. The configuration section is used to define architecture parameters, e.g., the data width, the issue width or the number of registers. It also covers conventions of the application binary interface (ABI), such as register-usage and calling conventions.

The declaration section defines reusable templates for register files, memories, caches and functional units. In addition, templates for meta-information like binary encoding and assembler syntax are specified. All these templates are reusable across different architectures, processor variants and generations.

Finally, the previously defined templates are instantiated to define the architecture using a network of components. The interconnections between the components can be annotated with information for hazard detection, hazard resolution, and pipelining. The use of abstractions and templates simplifies the development of architecture models, and leads to very compact and intuitive specifications. In contrast to other languages, e.g., Expression and LISA, which require several interdependent models for the instruction set, the hardware structure, and the instruction semantics, our approach relies on a single model. The instruction set is not defined explicitly but is automatically derived using software tools available with the language. These tools provide a consistent view of the instruction set, the pipelines, the hardware resources, the timing behavior, and instruction semantics. So far several generator backends have been developed for the ADL. For example, a compiler generator [6] that automatically customizes a C compiler, including the instruction selector, register allocator and scheduler. The completeness of the instruction selector can be proved automatically [4]. In addition, a dynamic-compiling simulator can be derived, details on the simulator are given in Section 4.

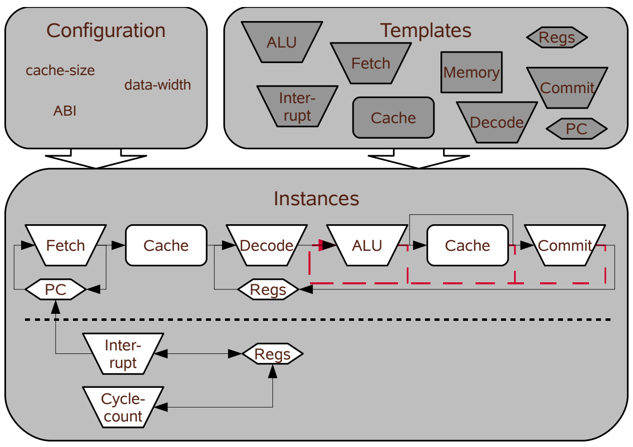


Figure 1. Overview of an example architecture model.

Figure 1 depicts an example model for a simple architecture. Several templates are available, describing functional units for fetch, decode, arithmetic computations, and register writeback. Each functional unit contains a set of *operations* specifying its behavior. In addition, storage elements, such as caches, memories, and registers, are declared. The instances derived from the templates are connected to build the data path. Templates can be reused, for example the `Cache` template is instantiated twice – the first instance models the instruction cache, the second a data cache.

The instruction set is extracted from *paths* through the network of components using a breath first search. Ignoring the interrupt dispatch below the dotted line, three paths can be found for the example architecture: `Fetch-Cache-Decode-ALU-Commit`, `Fetch--Cache-Decode-ALU-Cache-Commit`, and `Fetch--Cache-Decode-ALU-Cache`. Branches and regular arithmetic instructions are defined along the first path by combining the operations attached to the functional units.

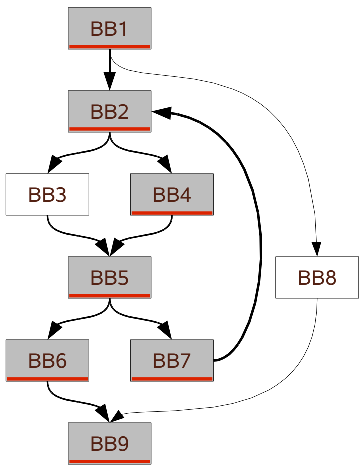


Figure 2. Basic blocks in gray are executed frequently and are thus compiled into a single region. The region contains complex control flow and loops, but can only be entered through `BB1`.

Load and store instructions are extracted from the second and third path respectively. The red dotted lines represent connections for data forwarding and data hazard resolution.

3.1. Modeling Interrupts

The ADL allows to specify several parallel pipelines within an architecture model. Not all of these pipelines need to be filled with instructions fetched from a cache, memory or ROM. Instructions that are not fetched from memory are considered to be executed every cycle in parallel with the ordinary instructions of the other pipelines.

The interrupt dispatch is simply modeled using such a *parallel* instruction. A dedicated functional unit, that is connected to the program counter (PC) and some status registers, repeatedly checks for pending interrupts and conditionally triggers a jump to the interrupt routine. Using this approach, interrupts are modeled apart from the other instructions and are explicitly visible to the ADL-tools. It is thus possible to apply specialized optimizations to the interrupt dispatch mechanism, for example within the simulation framework.

In the case of the example architecture from Figure 1, the parallel instructions are used to implement a cycle counter and the interrupt dispatch.

4. Simulator Generation

The architecture specifications provide enough information to automatically customize and generate software development tools, such as a C compiler [6] and a simulator.

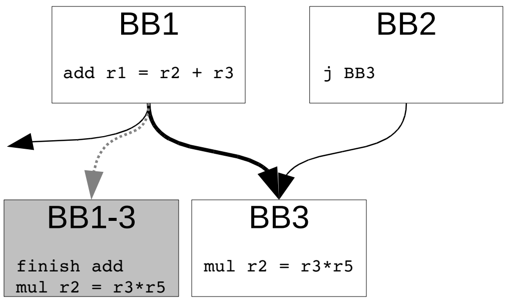


Figure 3. Compiled basic blocks are specialized for hot predecessors.

The simulation framework is based on the open source compiler infrastructure LLVM 2.3 [21] that provides a portable and highly optimizing just-in-time code generator. To avoid the compilation overhead for code that is not executed repetitively an interpreter is used until a specified threshold has been reached.

Frequently executed code is compiled using two optimization levels. First, sequential code, i.e., basic blocks, is compiled using a fast, moderately optimizing code generator configuration. Only some simple scalar optimizations are applied that cause low compilation overhead, but usually lead to a considerable runtime improvement. The transition from the interpreter or compiled predecessor blocks into the compiled code of a given basic block is critical, because the initial pipeline state depends on the instructions that were issued before entering the block. In order to minimize the transition overhead a specialized version of the basic block is compiled for every predecessor, but only if this particular transition reaches the compilation threshold, i.e., is executed frequently. This specialization may cause additional compilation overhead when multiple versions of a basic block are compiled. Fortunately, most basic blocks only have few predecessors and even fewer of them actually trigger the compilation, consequently, excessive duplication of compiled code does not occur.

Figure 3 illustrates the specialization of `BB3`. The white blocks represent the original control flow. When `BB3` is executed the state of the pipeline depends on the predecessors, and may thus require to either simulate a jump or an addition. However, only the transition between `BB1` and `BB3` is frequently executed and a specialized block `BB1-3` is compiled solely for this particular transition.

Hot basic blocks, i.e., blocks that are still executed frequently, are then combined to *regions* and recompiled using more aggressive optimizations. LLVM is intended for compilation of programming languages and imposes some restrictions on the form of the compiled code. For example, only *functions* can be translated that have a single entry. The generated code also follows the calling convention of the

```
Cycle  Simulation-Action
    N  (a) increment time
       (b) simulate instructions
       (c) if (interrupt) jump exit
  N+1  (a) increment time
       (b) simulate instructions
       (c) if (interrupt) jump exit
    ⋮
       exit: store state
```

Figure 4. A simple approach: Conditional exits (c) increase the memory consumption and reduce to benefits of compiler optimizations.

```
Cycle  Simulation-Action
    N  simulate instructions
  N+1  simulate instructions
    ⋮
  N+n  simulate instructions
       increment time by n
       if (int-missed) jump rollback
       exit: store state
       rollback: revert state
```

Figure 5. Assuming that interrupts do not occur allows to eliminate useless code and increases optimization opportunities.

host machine and thus imposes some overhead to save and restore state on function entry and exit. Regions are limited to have a single entry but are not constrained otherwise. In particular, regions are allowed to contain loops, non-linear control flow and several exits.

Building a region is similar to *trace* formation found in other simulators. Starting from a seed block, frequently executed successor blocks are added iteratively to the region. The processing stops, when only infrequently executed successors are left or a predefined threshold is reached. The code generation for regions is heavily simplified by relying on the LLVM function inliner. For each block of the region a simple call is added to the LLVM function representing the region that invokes the corresponding LLVM function of the basic block. During the code generation and optimization phase the LLVM inliner decides whether the call is inlined, i.e., replaced with a copy of the called simulation function of the basic block.

Consider the control flow depicted in Figure 2. The region starts with the seed block `BB1` and is enlarged iteratively by adding other frequently executed blocks marked gray in the picture. The region contains complex control flow and even a loop, but can only be entered through the seed block. For example, the region can not be entered using the transition between `BB8` and `BB9`, or `BB3` and `BB5`.

5. Interrupts and Rollback

In its simplest form all interrupt instructions are executed every cycle along with the ordinary instructions that are fetched from memory. This leads to a tremendous performance overhead, because:

- For each instruction an additional interrupt instruction needs to be simulated. Due to the low number of interrupts most of the interrupt instructions never actually trigger an interrupt dispatch.
- In the rare event of an interrupt the compiled code needs to be exited using a conditional jump. This requires extra code to save and possibly restore the architecture state after each cycle.
- Increased compile time and code size for the translated instructions in basic blocks and regions. A large portion of the generated code is actually useless as it is never executed.
- Compiler optimizations, such as constant propagation, dead code elimination and strength reduction, are less beneficial, because intermediate results need to be retained for the case of an interrupt.

As a consequence, compilation takes longer, the resulting code requires more memory and executes much slower. Most of the issues disappear, when interrupts are not directly handled within compiled code. We propose a rollback mechanism that assumes that interrupts do not occur during the execution of a compiled code sequence. This assumption is verified at so called *restore-points*. When an interrupt is actually found to be pending the processor state is reverted to an earlier restore-point. The execution is then restarted using the slower interpreter that faithfully models interrupts.

Figure 4 illustrates the overhead caused by interrupts in compiling simulators. Besides the simulation of regular instructions, interrupts need to be handled every cycle (c). Optimization opportunities for the compiler are limited because intermediate results need to be retained. For example, the time counter needs to be incremented every cycle (b) and can not be eliminated. These limitations are eliminated in the case of rollbacks shown in Figure 5. The faster compilation, the reduced code size, and the improved runtime will out-weight the overhead induced by the rollback and the slower interpretation of the interrupt dispatch.

5.1. Restore Points

The existing simulator already buffers all computations in local variables and registers. Only the most recent version of each variable is committed to the global architecture state at the end of the compiled code. This improves performance even when interrupts are not to be simulated. Extending the existing infrastructure to support the rollback mechanism is simplified because of this buffering. In the case of a rollback the buffered values are simply discarded and not committed to the global architecture state. However, not all updates of the architecture state can be buffered. For example, memories and caches are not considered for buffering because of the memory overhead that would be associated with it.

Essentially there are three options to handle instructions that update memory. For the first option, writes directly update the memory. The address and the previous value are stored in a dedicated buffer that is consulted on a rollback to revert the memory to the original state. Option two is inverse to option one, the current values are stored in a buffer and loads consult that buffer to read the current value. This simplifies the rollback, but on the other hand introduces additional overhead for the simulation of load *and* store instructions. Finally, one can establish restore-points around stores such that execution reverts right before a store if an interrupt is pending. In addition, the current architecture state is committed directly following the memory update.

Currently, we decided for the last option, as it reduces the number of cycles that need to be reverted on a rollback and only requires moderate extensions to the existing simulator. As a side effect large blocks, that usually contain memory operations, are split into smaller chunks, such that a dedicated splitting of these blocks can be omitted.

5.2. Rollback in Regions

As mentioned in the previous sections, regions may contain complex control flow and most importantly loops that need to be considered for rollbacks. For example, if a busy-waiting loop, which is only exited after a particular interrupt has occurred, is compiled into a region, care has to be taken to prevent infinite iteration. Restore-points and interrupt checks need to be provided in order to detect the interrupt within the region.

Restore-points need to be inserted for every block of the region that potentially causes the execution to leave the region. This is required for correctness in order to prevent a false architecture state to be committed, e.g., before returning to the interpreter. However, this does not yet guarantee correct handling of loops, for example, if the region does not have any exits at all. The simplest solution is to attach restore-points at the end of all basic blocks within a region.

benchmark	avg	min	max	std. dev.
adpcm	5	1	11	2.74
bitcount	–	–	–	–
blowfish	396	1	422	100.41
crc32	8	1	12	3.7
dijkstra	7	1	16	4.17
gsm	275	1	477	228.64
jpeg	17	1	160	27.3
prime	4	2	5	1.33
rijndael	185	1	538	235.47
sha	19	1	32	7.21
stringsearch	4	1	9	1.84

Table 1. Average, minimal, and maximal number of simulated cycles reverted per rollback.

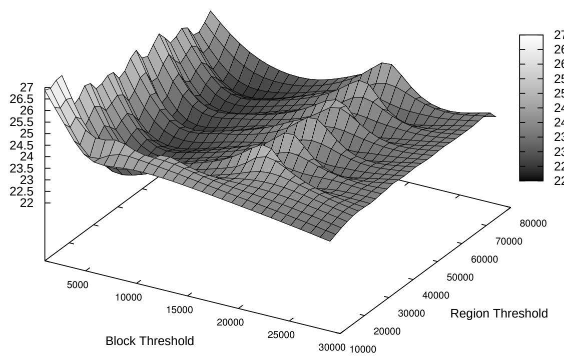
We have decided for this approach because of two reasons, even though some overhead during the simulation is introduced. Most importantly, this approach fits well with the existing infrastructure for regions, i.e., regions are compiled simply by inlining the LLVM functions of each basic block. All required restore-points are already compiled into these functions, eliminating them would require a complete duplication of these functions except for a small check. Consequently, the overall compilation overhead is reduced, in particular for large regions. Second, this approach reduces the number of cycles discarded by rollbacks and is thus expected to reduce the simulation overhead. Additionally, the bookkeeping for rollbacks itself is simplified, because the restore operation is kept local to a single basic block.

Consider the example presented in Figure 2 showing the red marked restore-points for a complex region with a loop. Several basic blocks (`BB1`, `BB2`, `BB9`) already require a restore-point for correctness. The other restore-points are inserted to minimize the overhead of restores.

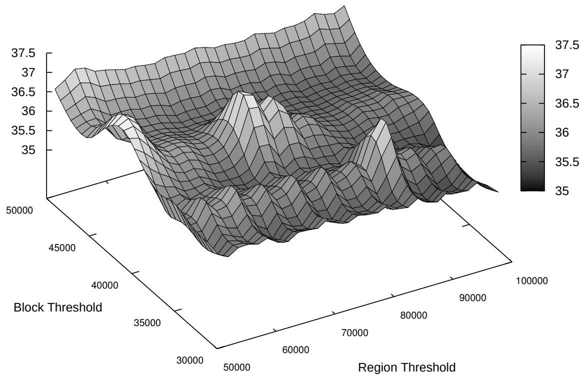
6. Evaluation

We have evaluated our approach using a MIPS R2000 [25] model describing the integer instruction set and additionally, the basic configuration registers and instructions required for timer interrupts as specified in the MIPS32 architecture reference manual [23]. Including comments the model consists of 1,128 lines of ADL-code. 148 of which specify interrupt related components, such as the cycle counter, the interrupt dispatch unit, and status registers.

Memory loads require a delay cycle before the data is guaranteed to be available. Branches are executed in the third stage of the five-stage pipeline, resulting in a branch penalty of one cycle due to the delay slot. The `eret` instruction, which resumes the normal execution after the in-



(a) Rollback



(b) Simple

Figure 6. Total execution time for all benchmarks using different thresholds for basic block compilation and region compilation. The rollback mechanism enables the use of lower thresholds.

terrupt handler has completed, does not define a branch delay slot, leading to a penalty of two cycles. On an interrupt dispatch the first three pipeline stages are flushed. Usually the instruction at stage three is later restarted upon a return from interrupt. If that instruction is executing in a branch delay slot the branch itself is restarted. In our setup the interrupt handler consists of a single `eret` instruction, the penalty of an interrupt thus varies between 6 and 7 cycles.

Two simulators were generated using the extended MIPS model to compare the simple approach and our new rollback mechanism described in Section 5. A MIPS model without interrupt facilities serves as an additional reference. The three generated simulators were tested using a large subset of the MiBench benchmark suite [19] on a 3 GHz Intel Xeon server machine with 24 GB RAM and Linux 2.6.18. LLVM and the simulators were compiled using the GCC compiler version 3.4 with standard optimizations enabled. The benchmarks where compiled using GCC 4.2 with standard optimizations for the *mips-elf* target. For the performance comparison each benchmark was run 10 times in order to get reasonable results.

The first timer interrupt is triggered in cycle 10, all following interrupts are signaled periodically every 20,000 cycles. For a processor running at 200 MHz this results in a timer resolution of 100 μ-seconds, which is a reasonable value for the domain of embedded systems. On average over all benchmarks, 0.85 rollbacks are performed per interrupt. In total 2,627,699 cycles are reverted by only 92,335 rollbacks during the simulation of more than 1.5 billion cycles. Table 5.2 presents the average, the minimal, and the maximal number of cycles reverted per rollback, the last column shows the standard deviation. The relative number of cycles reverted for the benchmarks is usually far bellow one percent. The largest fraction can be observed for blowfish and gsm where 1.67% and 1.22% of the simulated cycles are reverted. As can be seen in Figure 9, these comparatively large values do not have an impact on simulation speed – gsm is among the benchmarks with the largest speedup. From this data we can conclude that the overhead induced by rollbacks is very low. It is thus likely that the initial design decisions to minimize the number of cycles per rollback was over-conservative. Many restore-points around stores and within regions can probably be eliminated without negative effects on the simulation performance.

6.1. Compilation Overhead

We found that the simple interrupt handling scheme leads to considerable compilation overhead. Our first experiment, shown in Figure 6, compares the total simulation time over all benchmarks for different execution thresholds for basic block and region compilation in the range of 1,000 to 100,000. The rollback mechanism allows for lower thresholds for basic blocks. The threshold for regions usually has a lower impact for both techniques.

The simple approach is very unpredictable, good solutions are spread all over the search space and no continuous development can be observed. The best result is achieved using a threshold of 33,000 and 73,000 for basic blocks and regions respectively. The rollback mechanism is much more predictable and achieves the best results with a basic block threshold of 5,000 and a region threshold of 72,000.

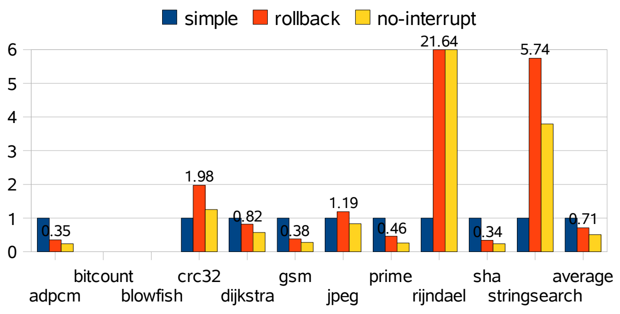


Figure 7. Compile time for all three simulators relative to the simple approach.

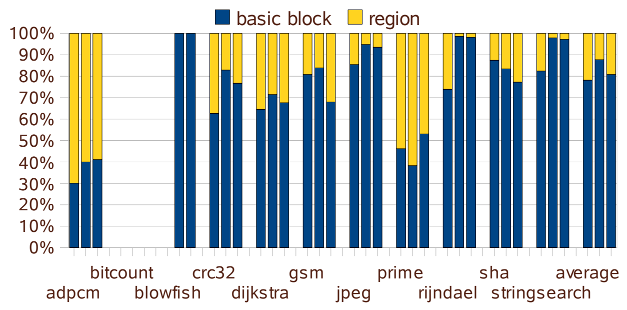


Figure 8. Relative compile time spent on translating regions and basic blocks, results for unoptimized interrupt handling are shown first, then interrupts with rollback and finally with interrupts disabled.

Increasing the basic block threshold consistently results in longer simulation time. Note, that the axis showing the basic block threshold for the simple approach is inverted to improve the 3D-view.

The best compilation thresholds obtained in the previous experiment were used to compare the compilation overhead for each benchmark individually. Figure 7 compares the time spent on compiling both regions and basic blocks. For almost all benchmarks the total compile time is reduced using the rollback mechanism even with lower thresholds, which generally lead to *more* compiling. Overall the compilation time is reduced by nearly 30%, which is close to the simulation without interrupts, which reduces the compilation time by about 50% compared to the simple implementation. Two benchmarks, rijndael and stringsearch, show a drastically increased compilation time, which is in part caused by the lower thresholds. It appears that some of the optimization passes of LLVM can not cope well with these two benchmarks. We where not yet able to determine the cause of these slowdowns. The simulation time for the bitcount benchmark is so short that compilation is never performed in any of the three simulators. Similarly, compilation time for the blowfish benchmark is below the resolution of the systems timer. Both benchmarks are thus not considered for this comparison.

Figure 8 shows the relative time spent on compiling regions and basic blocks for all three simulator setups. For each benchmark three bars are shown, the first bar represents the simple interrupt simulation approach, while the other bars show the results for the rollback mechanism and the simulation without interrupts. Although more optimizations are performed on regions the relative compilation time is comparatively small. Only about 20% of the compilation time is spent on regions, in the case of the optimized approach even less than 15%. Nevertheless, regions are more important to the overall simulation performance as will be shown in the next section.

6.2. Performance

We have already shown that the rollback mechanism is able to reduce the compilation time for our simulator. Figure 9 shows that simulation performance is similarly improved. Almost all benchmarks show significant improvements, on average the simulation using rollbacks is 68% faster compared to simulation using the simple approach. The best results are achieved for the sha, gsm and prime benchmarks which show improvements by a factor of up to 2.95. The rijndael and stringsearch benchmarks that already showed an inferior compile time also achieve the least simulation performance. As mentioned before, this appears to be in part a compiler problem.

Regions are an important optimization within our simulator framework. Regions not only allow for more aggressive optimizations but also increase the scope of these optimizations. In order for these optimizations to be beneficial it is important that a large portion of the cycles is simulated within regions. Figure 10 shows the relative number of cycles simulated using the interpreter and using compiled code of basic blocks or regions. The first bar again shows the results for the simple approach, while the other two represent the simulators with the rollback mechanism and with interrupts disabled. On average about 80% of the cycles are simulated using compiled code and more than 50% using regions. The impact of interrupts on this values is rather limited, because the overall number of interrupts is too low.

7. Conclusions and Future Work

The simulation of interrupts has great impact on the overall performance of a dynamically compiling simulator. We have shown that the slower compilation, increased code size, and reduced performance can effectively be eliminated

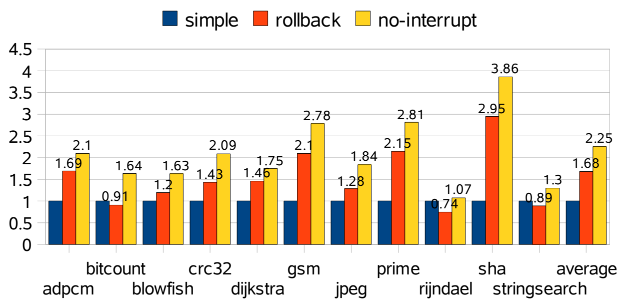


Figure 9. Relative simulation performance for each benchmark compared to the simple implementation.

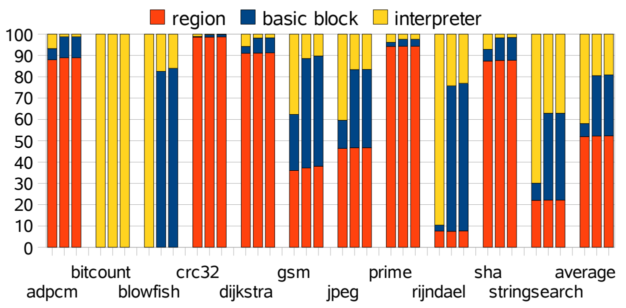


Figure 10. Percentage of simulated cycles per simulation technique, results for unoptimized interrupt handling are shown first, then interrupts with rollback and finally with interrupts disabled.

using a rollback mechanism. The frequency of interrupts is relatively low, which enables our new approach to improve overall performance even though the interrupt dispatch itself is executed using the slower interpreter. The execution performance is improved up to a factor of 2.95 compared to the simple approach, on average an improvement of 68% is achieved. The reduced compilation complexity enables the use of lower thresholds and at the same time reduce the overall compilation time by nearly 30%.

In the future we will implement and compare different strategies for restore-points in the presence of memory updates. In the current implementation scheme we tried to minimize the number of cycles reverted by rollbacks. The experiments indicate that the frequency of rollbacks and the associated overhead is not a problem and other strategies might prove profitable. In addition, restore-points within regions are not optimized at all, e.g., redundant restore-points could be eliminated.

We will also investigate how rollbacks might be used to improve other rare events during simulation. Possible candidates for research in that direction are branch buffers, branch predictors, and other predictors employed by modern micro-processors to improve performance. Usually these predictors achieve very good results, e.g., branch predictors correctly predict the direction in over 90% of the cases. It seems feasible to optimize the simulation of these predictors by predicting the result during compilation.

References

[1] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, 6(2):45–48, 2007.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2000.

[4] F. Brandner. Completeness of instruction selector specifications with dynamic checks. In *COCV '09: 8th International Workshop on Compiler Optimization Meets Compiler Verification*, 2009.

[5] F. Brandner. Fast and accurate simulation using the LLVM compiler framework. In *RAPIDO '09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2009.

[6] F. Brandner, D. Ebner, and A. Krall. Compiler generation from structural architecture descriptions. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 13–22. ACM, 2007.

[7] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. C compiler retargeting based on instruction semantics models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1150–1155. IEEE Computer Society, 2005.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137. ACM, 1994.

[9] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 351–366. ACM, 2007.

[10] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing^{TM} software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the International Symposium*

on Code Generation and Optimization, pages 15–24. IEEE Computer Society, 2003.

[11] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[12] K. Ebcioğlu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th annual International Symposium on Computer Architecture*, pages 26–37. ACM, 1997.

[13] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[14] S. Farfeleder, A. Krall, and N. Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *EUROMICRO Journal of Systems Architecture*, 53(8):501–510, 2007.

[15] S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. Effective compiler generation by architecture description. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, pages 145–152. ACM, 2006.

[16] L. Gao, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. A fast and generic hybrid simulation approach using C virtual machine. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 3–12. ACM, 2007.

[17] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach: Building a JVM with third party software. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 73–82. ACM, 2008.

[18] M. Gschwind and E. Altman. Optimization and precise exceptions in dynamic compilation. *ACM SIGARCH Computer Architecture News*, 29(1):66–74, 2001.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[20] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100. ACM, 1999.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75. IEEE Computer Society, 2004.

[22] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[23] MIPS Technologies. MIPS32® architecture for programmers volume III: The MIPS32® privileged resource architecture, July 2005. Version 2.50.

[24] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 22–27. ACM, 2002.

[25] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2007.

[26] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *DATE '00: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 669–673. ACM, 2000.

[27] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Co-Design and System Synthesis*, pages 13–18. ACM, 2003.

[28] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43, 1995.

[29] S. Sathaye, P. Ledak, J. Leblanc, S. Kosonocky, M. Gschwind, J. Fritts, A. Bright, E. Altman, and C. Agricola. BOA: Targeting multi-gigahertz with binary translation. In *In Proceedings of the 1999 Workshop on Binary Translation*, pages 2–11. IEEE Computer Society, 1999.

[30] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The Liberty Simulation Environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems*, 24(3):211–249, 2006.

[31] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[32] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79. ACM, 1996.

[33] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97. ACM, 2003.

[34] J. J. Yi and F.-D. J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on Computers*, 55(3):268–280, 2006.