Fast and Accurate Simulation using the LLVM Compiler Framework [⋆]

Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler

Christian Doppler Laboratory: Compilation Techniques for Embedded Processors
Institute of Computer Languages, Vienna University of Technology
{brandner,andi}@complang.tuwien.ac.at

Abstract. Development of future generation computer architectures requires fast and accurate simulation tools that allow to test, verify, and analyze the behavior of the given architecture along with the intended workload. We present a simulation framework based on a structural architecture description language that uses the open source compiler infrastructure LLVM to dynamically translate instruction sequences of the simulated architecture into machine instructions of the host machine. We show that the optimizations in the simulator and the LLVM compiler lead to an outstanding runtime performance: A 5-stage MIPS core is simulated at a peak performance of up to 800 MHz.

1 Introduction

Computer architects designing future generation processor architectures heavily depend on simulation tools to pinpoint performance bottlenecks and gain further insight into the behavior of the given architecture under realistic workloads. In order to actually achieve improvements, the simulation is required to deliver accurate data as fast as possible. Combining these two opposing requirements is hard, and often leads to unsatisfactory solutions.

Instruction set simulation (ISS) eliminates much of the overhead by focusing on the architecture state that is visible to programmers according to the instruction set architecture. Other architectural features are only modeled if this is necessary, for example to guarantee correctness. The majority of ISS tools rely on the following techniques: (1) interpretation, (2) static compilation, and (3) dynamic compilation. Interpretation offers the lowest simulation speed, but is relatively easy to adopt for a new architecture. The two compilation techniques translate instruction sequences of the simulated architecture into machine instruction of the host machine. In the case of static systems this is done offline, i.e., before the simulation is actually run. Dynamic compilation systems perform this translation during the simulation run. Both compilation techniques are often combined with an interpreter, in the case of dynamic compilation to lower

[⋆] This work is supported in part by ON DEMAND Microelectronics and the Christian Doppler Forschungsgesellschaft.

the translation overhead, for static compilers to allow the execution of code that is not statically known.

We present a retargetable dynamic-compiling simulation framework based on the open source compiler infrastructure LLVM [LA04]. The LLVM *just-in-time* compiler generates high-quality code, such that the achieved simulation speed reaches up to several hundred MHz. Retargeting the simulator requires only minimal programming effort, because all architectural features are derived from an architecture model specified using a structural *architecture description language* (ADL). Our ADL also allows to derive other software tools, such as a C compiler [BEK07], from the same architecture model.

2 Related Work

Techniques for computer architecture simulation have been heavily researched, [YL06] gives a broad overview on frameworks, benchmarks and methodology. SimOS supports several independent simulators including Embra [WR96], a high-performance simulator based on dynamic compilation. Shade [CK94] is a dynamic binary translator offering a rich and highly optimized profiling and tracing interface. Bala et.al. present Dynamo [BDB00] a dynamic optimizer that improves the runtime of programs by profiling it's execution and applying optimizations accordingly. Nohl et.al [NBS^{+}02] present a retargetable dynamic-compiling simulator based on the LISA architecture description language. Other ADLs allow to derive retargetable static-compiling simulators [PHM00,RBMD03,FKH07].

The LLVM [LA04] compiler infrastructure provides facilities to build highly optimizing static and dynamic compilers. Besides the static compilers for various programming languages (C/C++, Scheme, etc.) and architectures, LLVM is also used in various projects relying on dynamic code generation. SVA [CLDA07] extends LLVM to define a *secure virtual architecture*. The Linux Kernel has been ported to SVA, and allows to execute a complete operating system within a safe virtual machine. Geoffray et.al [GTCF08] use LLVM to develop a Java virtual machine that is competitive compared with open source and commercial Java implementations.

3 Simulator Implementation

Our simulation framework is based on an architecture description language, i.e., all architecture dependent simulation functions are derived from a concise architecture model. This includes all data registers, pipeline registers and memories required to capture the architecture state, and interpreter and compiler specifications for all instructions that faithfully model the execution of instructions within the pipeline.

The generated simulator executes input programs using a mixed-approach by interpretation and direct execution of host machine instructions. Compilation is done dynamically during the simulation in two phases: basic block and *region* compilation. Basic blocks are compiled to linear simulation functions that do not contain any control flow, which allows fast translation. Hot basic blocks are in turn compiled into a more general structure called region.

Dynamic compilation with LLVM

LLVM contains a complete set of high-level compiler optimizations, ranging from simple scalar simplifications to complex loop transformations. Prior to the actual code generation, various optimization passes are invoked by our simulator to increase the simulation speed. Deciding which optimizations to apply is a delicate task, and is highly dependent on the currently simulated program, the simulated architecture and the host machine.

The basic unit of translation in LLVM is a *function*, which has a single entry. The dynamically compiled functions follow the regular ABI conventions of the host architecture and thus contain some overhead caused by saving and restoring state. Basic blocks are translated to simple linear functions that do not contain any control except function exits. For all instructions comprising a basic block dedicated code generation functions are invoked that emit LLVM intermediate code modeling the instruction's execution. Buffering and shadow registers are used to simulate parallel events. Because of pipelining intermediate code for an instruction may cross basic blocks boundaries. We use basic block duplication to specialize a block when it has multiple predecessors. In all of our benchmark programs and architectures the increase in the number of basic blocks is less than 5.4%.

Hot basic blocks are compiled into regions when a specified threshold has been reached. Starting from a seed block the region is incrementally enlarged by adding basic blocks that can be reached from within the region built so far. A region is compiled to a LLVM function and may thus contain only a single entry, but we do not impose other restrictions on regions, specifically regions may contain loops. The LLVM intermediate code of a region is built from function calls to the corresponding LLVM functions of the blocks. The LLVM optimization passes eliminate most of these calls by inlining, in addition other advanced optimizations are applied to regions.

The compile time of the LLVM compiler is quite high (up to 100 times compared to other Java just-in-time compilers [GTCF08]). For some benchmark programs the compile time reaches 90% of the overall execution time. Therefore, the simulator does some optimizations on its own to reduce the number of intermediate code instructions. The simulator has more knowledge about constants, e.g. a register is always zero, and the use of data-forwarding. Constant expression evaluation and dead code elimination are thus performed internally by the simulator. Global values are loaded in a local copy on function entry and saved at function exit.

4 Evaluation

The evaluation was done for two different architectures, the MIPS, with a pipeline similar to the MIPS R2000, and the CHILI, a symmetrical 4-way VLIW archi-

		Cycles				Cycles	
Benchmark	LOC	MIPS	CHILI	Benchmark	LOC	MIPS	CHILI
prime	38	20459k	435607k	bitcount	925	48227	41718
jpeg	26106	36151k	17815k	blowfish	1924	14877k	28305k
crc32	281	645303k	1389091k	stringsearch	3259	8426k	11413k
sha	269	162018k	278669k	adpcm	300	26623k	51285k
dijkstra	179	350441k	644530k	gsm	6033	109551k	136304k
rijndael	1778	32619k	61327k				

Model	LOC	Instructions	Registers	Pipeline-Stages
MIPS-r2000	1054	54	32	5
CHILI	1650	771	64	7

Table 1. Benchmarks and architecture models used in the evaluation.

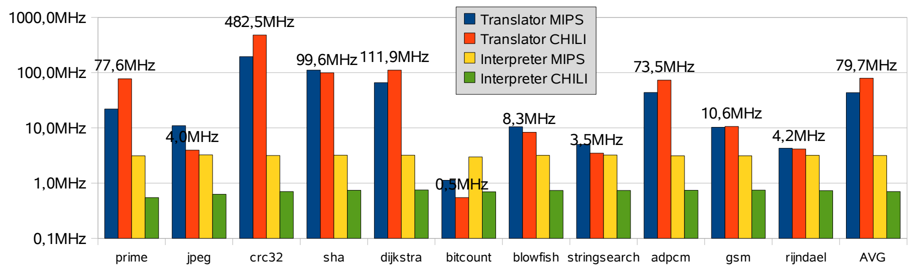

Fig. 1. Simulation speed in MHz for CHILI and MIPS with compilation enabled and disabled – note the logarithmic scale

tecture with delayed branches. A large subset of the *MiBench* suite was used to measure the simulation characteristics of embedded benchmarks. The benchmarks were compiled with optimization enabled (-O) using *gcc version 3.4.6* and *gcc version 4.2.0* for MIPS and CHILI respectively. All tests were performed on a single core *AMD Athlon(tm) 64 Processor 3500+* with 2200 MHz and 1 GB of RAM running a 32-Bit Linux operating system.

We compared the simulation speeds of the interpreter and translator for both architectures (see Fig. 1). The MIPS simulator reaches about 3.2 MHz, the CHILI simulator about 0.7 MHz. The translator is up to 500 times faster for the longer running benchmarks and reaches up to 480 MHz. On average the MIPS simulator executes at a speed of 43 MHz, the CHILI simulator even reaches 79 MHz on average. Fig. 2 shows the peak simulation speed over time for three benchmarks for the MIPS architecture. With all optimizations enabled a peak simulation speed of 800 MHz can be reached for the `blowfish` benchmark. For the very short running `bitcount` benchmark the translator is slower since the compile time cannot be compensated.

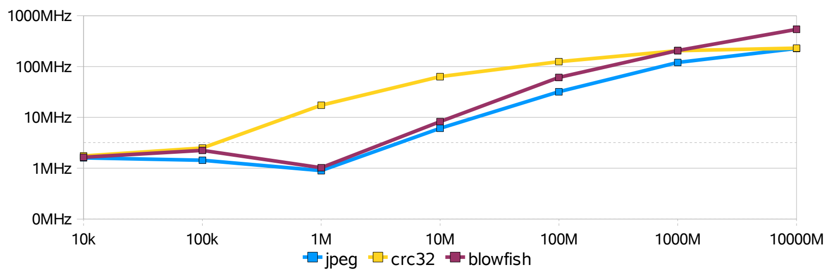

Fig. 2. Simulation speed over time for the MIPS architecture

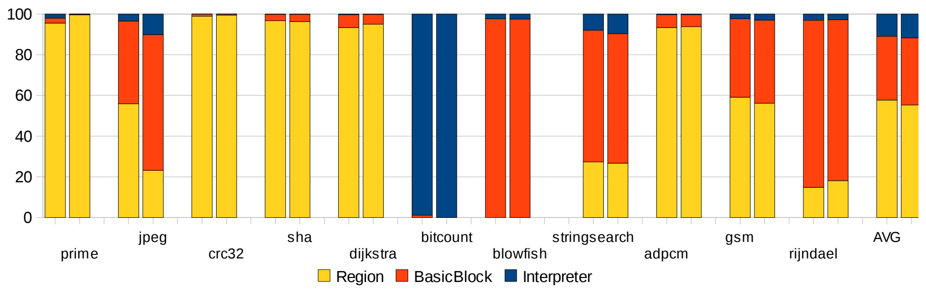

Fig. 3. Ratio of simulated cycles using the interpreter, JIT-compiled code in basic blocks, and compiled code in regions for the MIPS and CHILI architecture.

Fig. 3 shows the relative number of cycles simulated using interpretation or execution of JIT-compiled code in basic blocks and regions. Except for `bitcount` interpretation is only used to simulate a small fraction of the overall cycles, on average 11.7% for CHILI and 11% for MIPS. For `crc32` the complete main loop is compiled to a single region resulting in very high simulation speed.

5 Conclusion

We have presented a simulation framework for fast cycle-accurate emulation of computer architectures based on the LLVM compiler infrastructure. All architecture dependent simulation functions are derived from structural architecture specifications that can also be used to generate a VHDL processor model and a C compiler. The LLVM just-in-time compiler is used to compile basic blocks and non-linear regions of the simulated program to native code of the host machine. Optimizations of the simulator generator and the compiler framework enable a peak performance of the simulation speed of up to 800 MHz for the MIPS architecture. Future work on reducing the compile time is necessary to reduce the gap between the average simulation speed of 47 MHz for the MIPS (79 MHz for the VLIW CHILI) and the peak performance.

References

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev 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.

[BEK07] Florian Brandner, Dietmar Ebner, and Andreas 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.

[CK94] Bob Cmelik and David 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.

[CLDA07] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram 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.

[FKH07] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *EUROMICRO Journal of Systems Architecture*, 53(8):501–510, 2007.

[GTCF08] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil 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.

[LA04] Chris Lattner and Vikram 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.

[NBS$^+$02] 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 Conference on Design Automation*, pages 22–27. ACM, 2002.

[PHM00] Stefan Pees, Andreas Hoffmann, and Heinrich 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.

[RBMD03] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and Nikil 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.

[WR96] Emmett Witchel and Mendel 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.

[YL06] Joshua J. Yi and Fellow-David J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on Computers*, 55(3):268–280, 2006.