Constructing Portable Compiled Instruction-set Simulators — An ADL-driven Approach

Joseph D'Errico Wei Qin
Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215, USA
{jderrico, wqin}@bu.edu

Abstract

Instruction set simulators are common tools used for the development of new architectures and embedded software among countless other functions. This paper presents a framework that quickly generates fast and flexible instruction-set simulators from a specification based on a C-like architecture-description language. The framework provides a consistent platform for constructing and evaluating different classes of simulators, including interpreters, static-compiled simulators, and dynamic-compiled simulators. The framework also features a new construction method for dynamic-compiled simulator that involves no low-level programming. It profiles and translates frequently executed regions of simulated binary to C++ code and invokes GCC to compile such code into dynamically loaded libraries, which are then loaded into the simulator at run time to accelerate simulation. Our experimental results based on the MIPS architecture and the SPEC CPU2000 benchmarks show that our dynamic-compiled simulator is capable of achieving up to 11 times speedup compared to our fast interpreter. Compared to other dynamic-compiled simulators requiring significant system programming expertise to construct, the proposed approach is simpler to implement and more portable.

1. Introduction

Instruction-set simulators (ISS)'s mimic target architectures on a host machine. They are used by computer architects to validate new architecture designs and by software developers to verify the functional correctness of compilers or application programs. They have also been used in microarchitecture simulators to update architectural states or to generate traces.

Most ISS's can be grouped into one of three major classes. *Interpretive simulation* is the most traditional method. Under this method, instructions are fetched, decoded, and executed one by one, as is shown in Figure 1. Although it is the slowest approach, interpretive simulation is considered the most flexible. It is capable of pausing and altering program flow at arbitrary positions during run-time, interacting with debuggers or co-simulators, and simulating self-modifying code such as boot-loaders. *Static-compiled simulation* decodes and translates the entire target binary prior to run-time. The elimination of fetch/decode overhead causes static-compiled simulators to run at significantly faster rates than interpretive simulators. However, the approach is less flexible since it does not support self-modifying code, and its simulation flow is hard to control. The third method, *dynamic-compiled simulation*, combines concepts from the first two classes. A dynamic-compiled simulator uses run-time code generation techniques to translate chunks of target binary code to host binary during execution. Translation can be performed selectively to only frequently executed regions of the target binary or to all executed instructions. This method is slightly slower than static-compiled simulation due to the translation overhead during run-time, but it is more flexible and can simulate self-modifying code. Although generally considered the superior simulation technology, dynamic-compiled simulation has limited use due to the extensive system-level programming skills that it demands from simulator developers.

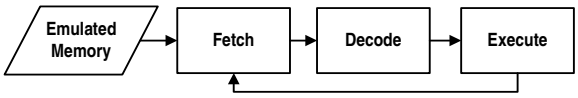


Figure 1. Interpretive Simulator

This paper presents a retargetable framework that can synthesize all three classes of simulators and their variations from a common target architecture description in a C-like architecture description language (ADL). Depending on their requirements of flexibility and speed, users can

3-9810801-0-6/DATE06 © 2006 EDAA

choose to generate the desired type of simulator without modifying the description. Since the framework uses a universal infrastructure of key building blocks throughout the different simulators, it also provides an ideal platform to evaluate and compare the simulators.

The paper, furthermore, describes a new approach for building retargetable dynamic-compiled simulators which are fast, flexible, and portable. It is based on the dynamically loaded library (DLL) feature of modern operating systems, the interface of which is simple and well-documented. The approach involves no low-level assembly programming and can be robustly implemented in a short period of time.

The remainder of the paper has the following organization. Section 2 describes related work in the field of ISS design. Section 3 explains the structure of the framework and our methods to construct simulators. Section 4 presents the experiment results and Section 5 discusses the features and properties of the generated simulators. At the end, Section 6 summarizes our findings and concludes the paper.

2. Related Work

Considerable studies have explored varied instruction set simulation techniques and have consequently led to the availability of many different simulators. Among the recent improvements to interpretive simulation is JIT-CCS [8], which caches decoding results of instructions for reuse. IS-CS [10] further optimizes decoding by moving it to simulator building time. Both JIT-CCS and IS-CS are flexible and retargetable. But their performance is constrained by their fine-grained kernel loops which fetch and execute one instruction per iteration.

Studies have also sought to accelerate static-compiled simulation. Zhu et al. [14] aggressively manipulates host resources, especially registers, to boost simulation performance. The FSCS [2] uses a simulator synthesis framework based on the ArchC ADL. The framework translates target program code to C functions, which are then compiled into simulator instances. During translation, it performs control-flow related optimizations to improve simulation performance. SyntSim [4] similarly uses C as the intermediate representation. However, its generated C code are in one huge function and may occasionally cause the C compiler to run out of memory. Thus it requires a profile of the program to select the most frequently executed instructions to translate. The rest instructions are interpreted.

Early work on dynamic-compiled simulators focuses on performance. Shade [5] is likely the first simulator in this class. Embra [13] used Shade's dynamic binary translation technique for full system simulation. However, none of these simulators is retargetable or portable.

More recently, research in dynamic-compiled simulation has focused on the addition of retargetability. The Strata [11] infrastructure was designed to build retargetable dynamic-compiled simulators. Target-specific functions, provided as extensions to Strata, can be added for new targets. However, Strata is limited only to x86 hosts and porting Strata would require extensive knowledge of the host architecture. In contrast, QEMU [3] has been ported to a variety of hosts. Porting QEMU requires some assembly code and intimate knowledge of the inner workings of the host architecture and OS.

The above approaches have all made significant improvements to their respective simulator classes by using novel optimizations and infrastructures. However, other factors, such as the coding style, and the features included in the simulator, can play a significant role in determining its speed. Common simulator features affecting speed include the use of statistics counters, the capability of detecting memory faults of the target program, and interoperability with other simulators. Since such differences exist, it is hard to compare the performance of the simulators solely based on the merits of their approaches.

3. Our Simulator Synthesis Framework

To improve productivity, we created a light-weight ADL from which all three types of ISS's can be synthesized. Since the ADL only contains necessary information for ISS generation, it is much simpler than more comprehensive ones such as nML [6] or LISA [9]. It can serve as the interface between a full-scale ADL and an ISS generation backend based on our framework. The ADL uses a C-like syntax, shown in Figure 2, similar to that in GenISSLib [1]. For each instruction, it describes the instruction encoding and the instruction semantics. It borrows features from FSCS [2] by separating the description of control flow instructions from others, which is helpful for constructing dynamic-compiled simulators. It also distinguishes between control flow instructions with constant targets, which are statically hard-coded in complied simulation, and those with variable targets, which must be determined at run time.

```
op add(000000:rs:rt:rd:-----100000) {
execute = "
  WRITE_GPR(READ_GPR($rs$) + READ_GPR($rt$), $rd$);
  "
}

op beq(000100:rs:rt:imm) {
condition="READ_GPR($rs$) == READ_GPR($rt$)"
ctarget="$pc$+4+((hword_t)$imm$ << 2)"
}

op jr(000000:rs:---------------001000) {
vtarget="READ_GPR($rs$)"
}
```

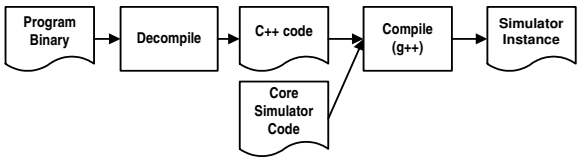
Figure 2. Sample ADL code

For each architecture description input, our framework will analyze the instruction encoding information and generate an efficient instruction decoder that is shared by all synthesized simulators. It also shares highly optimized key elements such as the target memory emulators, the system call emulator and the program loader across all simulators. Maximal sharing of infrastructure code among simulators not only saves our development effort, but also enables fair comparison of different simulator classes and provides consistent performance data for trading off speed and flexibility. Below we will first describe the generation flow of three types of simulators.

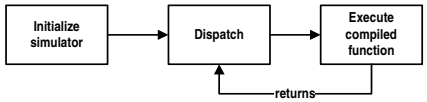
3.1. Interpretive Simulation

The interpretive simulator follows the traditional model shown in Figure 1. The decoder employs multi-level table-lookup to dispatch the execution flow to one of the interpretation routines corresponding to the opcode of the fetched instruction. Both the decoder and the interpretation routines are synthesized from the ADL description. The framework emits two interpreters by using two different memory emulators. One does memory fault checking and will report on all invalid memory accesses such as misaligned addresses or violation of protection. The other is optimized for speed and does not check safety.

3.2. Static-compiled Simulation



(a) Compiled Time



(b) Run Time

Figure 3. Static-Compiled Simulator

The two-step process of static-compiled simulation is shown in Figure 3. The first step involves a decompiler which is synthesized from the ADL description. The decompiler divides the program binary into aligned pages and translates each page into a C++ function. All C++ functions are then compiled by GCC and linked with the core library components to create a *simulator instance*. By default, each page contains 1K instructions, the starting address of which is aligned at a 1K-word boundary. The page size can also be specified by the user.

The structure of the generated functions is similar to that of FSCS [2]. Each function contains a *switch-case* statement. Each case block holds one translated instruction. A function receives the program counter as the argument, and upon entry steers execution to the corresponding case. If a case holds a non-branch instruction, after its evaluation the execution flow will fall through to the next case. For a branch instruction, if its target address can be statically determined and is within the same page, the a *goto* statement is used. Otherwise, the function returns.

In the second step (Figure 3(b)), simulation is performed. Depending on the value of the program counter, the dispatcher will call the corresponding compiled function. Free of fetching and decoding overhead, the loop in Figure 3(b) executes much faster than the one in Figure 1. Moreover, each function call evaluates around 50 instructions on average. Thus significantly less loop iterations are needed by Figure 3(b) than Figure 1 to simulate one program, further expanding the speed gap between the two techniques.

3.3. Dynamic-Compiled Simulation

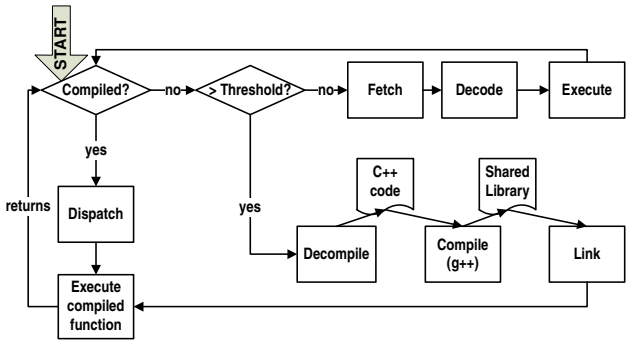


Figure 4. Dynamic-Compiled Simulator

The dynamic-compiled simulator utilizes building blocks of both the static-compiled and interpretive simulators, as can be observed in Figure 4. The interpretation loop from the interpretive simulator exists although it is augmented by conditional statements which check if a page of target code has already been compiled or is ready for compilation. If either condition is true, the simulator leaves the interpretation loop and performs compiled simulation. Different from the static-compiled case, the translation of aligned code pages is done at run time and the results, in the form of DLLs, are immediately linked to the simulator.

We aim to run frequently executed code pages in compiled mode and the rest in interpretation mode. To determine

frequently executed pages, we use a constant threshold. If the dynamic instruction count of a page exceeds the threshold, the page will be translated into C++ and then compiled into a DLL. The threshold can be specified by the user. Section 5 will discuss the trade-off faced when choosing the threshold as well as the page size.

Unlike sophisticated simulators in the same class which perform binary translation internally, we rely on GCC and the DLL interface of the host operating system. This choice greatly simplifies the implementation and increases portability since no assembly programming is involved. However, it does suffer from the slow translation speed. To mitigate the problem, the simulator caches the DLLs in the hard drive so that subsequent runs of the same target program can directly reuse them.

4. Experiment Results

To evaluate our framework we constructed ISS's of the popular MIPS architecture. All experiments were performed on an unloaded, 2.8GHz Pentium 4, Linux server with 2GB RAM. The three ISS's were compiled using GCC 3.3.3 with -O3 and *-fomit-frame-pointer* flags for optimization. When compiling the generated C++ code for both compiled simulators, the -O flag was used instead of -O3. Omitting the expensive optimizations of -O3 reduced compilation time without significantly affecting the simulation speed. For all experiments, we used 1K-sized pages for compilation and used a threshold of 64M to control dynamic compilation.

Our findings are based on the results of simulating all C-based SPEC CPU2000 [12] benchmarks. The benchmarks were compiled using a MIPS cross-GCC with the -O3 and *-static* flags. The only exception is for 176.gcc where we used the -O flag to work around a compiler bug. For all benchmarks, their first reference inputs provided by SPEC were used.

4.1. Speed Comparison

The interpretive simulators depend on the fine-grained fetch/decode/execute loop in Figure 1 and thus have the slowest speed. Our results show that the speed-optimized interpreter runs at an average of 37.3 million instructions per second (MIPS) with a small standard deviation of 2.5 MIPS. The interpreter that performs memory safety checking has a 18% slower speed of 30.7 MIPS due to the overhead associated with checking all memory accesses. We use the fast interpretive simulator as the baseline case to evaluate the speed of the compiled simulators.

Figure 5 displays the performance of the compiled ISS classes. The first two consecutive runs of the dynamic-compiled simulator were recorded because a difference in

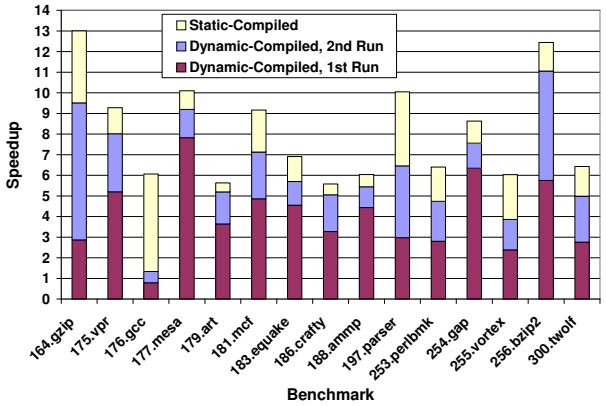


Figure 5. Speedup of Compiled Simulators

speed exists – the second run benefits from reusing the cached DLLs as a result of the first run. In other words, the second run does not contain the compilation overhead of the DLLs and therefore yields better performance.

Clearly, the static-compiled simulator dominates on pure performance, and it is, on average, 7.9 times faster than the interpretive simulation (computed by the total simulation time of all benchmarks). However, the performance of the static-compiled simulator is not nearly as predictable as that of the interpretive simulator. Its speedup ranges from 13 times to 5.6 times interpreter speed. Most likely, this is a result of the varying instruction mixes across different benchmarks. In the static-compiled simulator, it takes only one host instruction to interpret an arithmetic instruction of the target but more than 10 host instructions to interpret an memory instruction (need to translate the target memory address to a valid host address). A high percentage of hard-to-interpret instructions will result in a reduced speedup.

With some instructions interpreted, the dynamic-compiled simulator has slower but still formidable speed. It averages 4.8 times interpretation speed on the first run and 6.6 times on the second. In the best case (256.bzip2), the simulator achieves a speed of 450 MIPS, or 11 times the speed of the interpreter. Like the static-compiled case, the performance varies based on benchmark. For most benchmarks, such variations are akin to the variations in static-compiled speed-up. Only one benchmark, 176.gcc, fails to meet this paradigm. In fact, it has a slowdown for the first run. The next section, Section 4.2, discusses this anomaly further.

4.2. Analysis of Simulation Performance

Because we use GCC to compile DLLs, it is expected that its long compilation latency may affect the performance. The speed gap between the first and the second runs

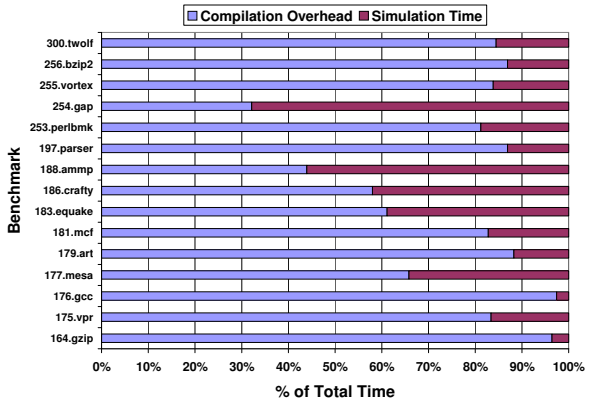


Figure 6. Compilation Overhead vs. Simulation Latency

of the dynamic-compiled simulators in Figure 5 confirms this, so does the long compilation delay of the first step of static-compiled simulation. On our experiment platform, it takes roughly 2 seconds for GCC to build an DLL containing 1K translated instructions.

For static-compiled simulation, Figure 6 illustrates the percentage of time that the entire simulation process (compiling and then running once) spends in compilation. In all but two cases, running the compilation takes longer than running the actual benchmark. With compilation overhead included, the average speed of static-compiled simulation drops to 2.5 times the speed of interpretive simulation. Certainly, if a simulator instance is reused several times, or if larger reference inputs are used so that simulation takes longer, the percentage of total time spent in simulation will increase and the speedup will improve.

For dynamic-compiled simulation, less time is spent in compilation because we selectively compile only the frequently executed pages of the program. Nevertheless, compiling a page may not be beneficial if the resulting DLL is not used frequently after it is compiled. This scenario unfortunately becomes true for 176.gcc, which has a relatively short execution trace and poor locality. During its first run, a total of 22 pages are compiled, but only 32% of the total 6G instructions benefit from compiled simulation. The remaining 68% are interpreted. The mere performance gain from the 32% compiled instructions is totally canceled by the compilation time for the 22 pages. Thus we observe a slowdown for its first run in Figure 5. In the second run, a total of 44% interpreted instructions still limits the performance gain significantly.[1]

To further characterize the performance of 176.gcc, we used all its seven reference inputs from SPEC, which are of different lengths. Depending on the length of the trace, the speed ranges from a slowdown on the smallest trace to a 5.5X speedup on the largest. This trend suggests that programs with longer running traces would perform better with our dynamic-compiled simulator. This is desirable since long programs need the speedup most. Accelerating the simulation of programs which run on the order of hours is more of a concern than accelerating those that run on the order of seconds or minutes.

5. Discussions

It is unfair to discuss the speed of the simulators without mentioning their capabilities. Among all simulators generated, the interpreter with memory checking has the most features but the slowest speed. Both interpreters support simulating self-modifying code and interfacing to a GDB remote debugging interface [7]. They are suitable to be used to debug application programs.

Both compiled simulators are optimized for speed and therefore do not include the memory checking capability. The static-compiled one can neither simulate self-modifying code nor interface to GDB. In contrast, the dynamic-compiled simulator is capable of both given a simple extension. To support self-modifying code, especially to allow a compiled page to modify itself, we implemented an exception handling mechanism by using *setjmp* and *longjmp* functions of ANSI C. We also monitor all memory write accesses. When an instruction writes to a page that has been complied, a *longjmp* is triggered so that compiled simulation is interrupted. After the DLL of the modified page is unloaded, simulation resumes from the interrupted location. The page will be interpreted until the threshold is met again. The mechanism, when activated, has a 20% impact on the performance of the dynamic-compiled simulator.

Table 1 summarizes the performance and capabilities of the different simulator configurations in our framework. The two speed numbers for the dynamic-compiled simulator are for the first and the second runs. The first number for the static-compiled simulator takes into account compilation time. We also performed experiment on the ARM architecture using all C-based SPEC CPU2000 integer benchmarks. The results show that the average speed of the dynamic-compiled simulator is 121 MIPS on first runs, 3.9 times of that of the interpreter. In comparison, the reported speed of JIT-CCS was around 7 MIPS on a 1.2GHz Athlon and that of IC-CS was 12 MIPS on a 1GHz P3, for the ARM ISA. Both are significantly slower than our dynamic-compiled simulators after the difference of the host machines is taken into account. This is mainly due to the high looping overhead of their fine-grained kernel loops.

1 On a second run, all cached pages are loaded on simulation start and do not go through page profiling. Therefore less instructions are interpreted.

Simulator	Speed(MIPS)	Mem-Check	Self-Mod
Interp. 1	30.7	Yes	Yes
Interp. 2	37.3	No	Yes
Dynamic 1	150/195	No	Yes
Dynamic 2	181/245	No	No
Static	91.8/294	No	No

Table 1. Summary of Simulators

Specifying the threshold and the page size can alter results for the dynamic-compiled simulator. If the size of pages is too large, compilation time will increase quickly due to some quadratic optimizers in GCC. However, small pages will cause more iterations of the leftmost loop in Figure 4, which will slow down the simulation speed due to the dispatcher overhead. A higher threshold will reduce the number of pages to translate, but will increase the percentage of interpreted instructions. The default parameters for our framework, 1K instructions per page and a 64M threshold, achieve a good balance for large benchmarks such as the SPEC CPU2000 based on our observations.

The idea of invoking GCC to dynamically generate DLLs is perhaps not brand new. However, we are not aware of any previous report on its application to accelerating instruction-set simulation. Thus we view it a contribution of this paper to introduce the idea. The most notable feature of the approach is its *simplicity* and *portability*. In fact, our framework is purely based on C++. Its development is not dependent on the particular host architecture. This is contrary to the necessary expert-level understanding of the host platform needed to port QEMU [3] or Embra [13]. As long as a C++ compiler and a DLL interface are available for the host architecture, our framework can be quickly ported. Both requirements are readily satisfiable for modern workstations or personal computers. The drawback of our approach, the occasional slowing down of simulation due to long compilation latency, can potentially be overcome by distributing the compilation tasks to other processors or workstations. This task remains a topic of our ongoing research.

6. Conclusions

In this paper, we described a framework that generates three major classes of instruction set simulators from an ADL description. Since the framework utilizes a universal infrastructure to construct all simulators, it provides a consistent platform for evaluating different classes of simulation techniques. The framework also features a new technique to construct dynamic-compiled simulators. Aside from good performance and portability, the technique is simple to implement. It is also flexible enough to support self-modifying code and a GDB interface. We believe that it is a practical technique to use and deploy in place of interpreters.

7. Acknowledgments

This research is partially supported by a UROP Faculty Matching Grant from Boston University. We thank Prof. Patrick Schaumont and the anonymous reviewers for their invaluable comments to improve the paper.

References

[1] `http://www.microlib.org`, 2005.

[2] M. Bartholomeu, R. Azebedo, S. Rigo, and G. Araujo. Optimizations for compiled simulation using instruction type information. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 74–81, 2004.

[3] F. Bellard. `http://www.qemu.org`, Sep 2005.

[4] M. Burtscher and I. Ganusov. Automatic synthesis of high-speed processor simulators. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, 2004.

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

[6] A. Fauth, J. V. Praet, and M. Freericks. Describing instructions set processors using nML. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 503–507, Paris, France, 1995.

[7] Free Software Foundation, Inc. `http://www.gnu.org/software/gdb/gdb.html`, June 2005.

[8] 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 *Proceedings of Design Automation Conference*, pages 22–27, 2002.

[9] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of Design Automation Conference*, pages 933–938, 1999.

[10] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *Proceedings of Design Automation Conference*, 2003.

[11] K. Scott and J. Davidson. Strata: A software dynamic translation infastructure. In *Proceedings of the IEEE 2001 Workshop on Binary Translation*, 2001.

[12] Standard Performance Evaluation Corporation. `http://www.spec.org`, Aug 2005.

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

[14] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of Conference on Design Automation and Test in Europe*, 1999.