A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation

Wei Qin
Boston University
Boston, MA, U.S.A.
wqin@bu.edu

Joseph D'Errico
Cavium Networks, Inc.
Marlborough, MA, U.S.A.
joseph.derrico@caviumnetworks.com

Xinping Zhu
Northeastern University
Boston, MA, U.S.A.
xzhu@ece.neu.edu

ABSTRACT

Traditionally, instruction-set simulators (ISS's) are sequential programs running on individual processors. Besides the advances of simulation techniques, ISS's have been mainly driven by the continuously improving performance of single processors. However, since the focus of processor manufacturers is shifting from frequency scaling to multiprocessing, ISS developers need to seize this opportunity for further performance growth. This paper proposes a multiprocessing approach to accelerate one class of dynamic-compiled ISS's. At the heart of the approach is a simulation engine capable of mixed interpretative and compiled simulation. The engine selects frequently executed target code blocks and translates them into dynamically loaded libraries (DLLs), which are then linked to the engine at run time. While the engine performs simulation on one processor, the translation tasks are distributed among several assistant processors. Our experiment results using SPEC CINT2000 benchmarks show that this approach achieves on average 197 million instructions per second (MIPS) for the MIPS32 ISA and 133 MIPS for the ARM V4 ISA. Compared with the uniprocessing configuration under the same general approach, multiprocessing offers higher performance and improved speed consistency. To our best knowledge, this is the first reported approach that uses multiprocessing to accelerate functional simulation.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Performance, Verification

Keywords

instruction set simulator, compiled simulation, retargetable

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

1. INTRODUCTION

An instruction-set simulator (ISS) functionally mimics the behavior of a target processor on a host workstation. It is an important tool for the development of both embedded and general-purpose computer systems. The classic ISS style is *interpretation*. It features a simulation loop which repeatedly fetches, decodes and executes instructions one after another. Over the past two decades, a few advanced simulation techniques have been developed to improve the performance of ISS's. These techniques fall into either the *static-compiled* or the *dynamic-compiled* categories. The former performs binary translation prior to simulation and the latter does so at run-time. In general, compiled simulation is free of the instruction fetching and decoding overhead and therefore offers much faster speed than interpretation.

Traditionally, all ISS's are developed as sequential software running on a single processor. As the performance of high-end processors continuously improves, ISS's have been enjoying a free ride. However, recent trends indicate that microprocessor manufacturers are shifting their focus from frequency scaling toward multiprocessing. For example, the clock rates of Intel processors improved by a modest 2.5x over the past five years, in contrast to a 7.5x from 1995 to 2000 and a 6x from 1990 to 1995 [9]. At the same time, several multiprocessor implementations have been introduced, including the STI Cell processor [10], the SUN Niagara processor [11], and dual-cores from AMD, Intel, and IBM [6]. Multiprocessing improves the overall throughput of a computer, but it is not directly useful for inherently sequential ISS's. Consequently, ISS developers face a new challenge — to seize the multiprocessing opportunity so as to keep pace with growing needs of users. To address this challenge, this paper proposes a multiprocessing approach to accelerate dynamic-compiled simulation. The approach is highly portable and retargetable, and is capable of handling self-modifying code. To our best knowledge, it is the first reported approach that uses multiprocessing to accelerate functional simulation.

The remainder of this paper has the following organization. Section 2 describes related work in the ISS field. We then explain our method to construct dynamic-compiled simulators in Section 3. Section 4 presents the experiment results. In Section 5 we compare different techniques and discuss performance issues. At the end, Section 6 summarizes our findings and concludes the paper.

2. RELATED WORK

A static-compiled ISS translates a target program to a host program with identical functional behavior. Zhu et al. [20] created a

static-compiled ISS framework. It translates target instructions to a virtual representation, and then to either host assembly or to C code. Finally, the output code is assembled or compiled into host binary. SyntSim [3] directly uses C as its intermediate representation. It translates a target program into a huge C function which may sometimes overflow the C compiler. To alleviate the problem, it uses a profile of the target program to select only the most frequently executed instructions to translate. The other instructions are interpreted. Different from SyntSim, FSCS [1] is capable of translating a complete target program into C++. It does so by dividing the target code into blocks, and translating each block into a C++ function. Since a function is small, it will not overflow the C++ compiler. Hence, this approach is scalable and has been successfully applied to large benchmarks.

Static-compiled ISS's do not support simulating self-modifying code. They are also inconvenient to use since for every target program to simulate, an instance of the simulator needs to be built. Therefore, more research has explored the construction of dynamic-compiled ISS's. Early work on dynamic-compiled ISS's focuses on performance. Shade [4] is likely the first simulator in this class. Embra [19] used Shade's dynamic translation technique for full system simulation. However, none of these simulators is retargetable or portable.

More recently, a few retargetable dynamic-compiled simulation approaches have been proposed. The Strata [17] infrastructure defines an extension interface for adding new targets. However, its current implementation is tied to the IA32 platform. Porting it to another host requires extensive effort. In contrast, the JIT-CCS [12] and the IS-CS [16] are completely based on C/C++ and hence independent of the host platform. They utilize high-level target architecture descriptions for better retargetability. The execution engines of both JIT-CCS and IS-CS have resemblance to interpretative ISS's in their fine-grained structure — each calling of a compiled routine simulates only one instruction. This imposes significant per-instruction overhead. QEMU [2] is an open-source dynamic-compiled ISS featuring very high performance. Similar to the work of Zhu et al.[20], it uses an intermediate representation to simplify the translation of most instructions. However, like many other just-in-time engines, porting or retargeting QEMU still requires mixed C/assembly programming and extensive debugging effort.

In the field of embedded systems, many specialized processors need to be simulated. Therefore it is highly valuable to have good retargetability through some abstraction means such as an architecture description language (ADL). It is also desirable to have portability since host architectures and operating systems never stop evolving. FSCS, SyntSim, JIT-CCS and IS-CS fully use C/C++ in their implementation and are therefore highly portable. They are also retargetable through their respective architecture abstraction means. Consequently, we view these approaches more suitable to use in design automation tools for embedded systems.

3. PROPOSED APPROACH

Previously in [5], we described an approach to construct retargetable and portable dynamic-compiled ISS's. The approach is very similar to those of FSCS and SyntSim in use of C++ code as the intermediate representation of translation. However, it is more flexible than those approaches since it performs translation during simulation and therefore supports simulating self-modifying code. The approach also features significantly higher performance than JIT-CCS and IS-CS due to its more sophisticated engine. The engine simulates tens of instructions in each calling of a compiled routine. In contrast, JIT-CCS or IS-CS simulates one instruction in each calling. The following section briefly reviews the approach in general.

```
0x8000: 8d280000     lw $8, 0($9)
0x8004: 1500fffe     bnez $8, 8000
0x8008: 21290004     addi $9, $9, 4
0x800c: 01201021     move $2, $9
```

(a) A 4-instruction page

```
addr_t page_8000(iss_t *iss, addr_t pc)
{
  assert(pc >= 0x8000 && pc < 0x8010);

  switch (pc) {
    case 0x8000:
      L8000: iss->set_reg(8,
        iss->mem_read_word(iss->get_reg(9)));
      // load MEM[R9] into R8
      // fall through to next instruction
    case 0x8004:
      if (iss->get_reg(8)!=0) {
        // run delay slot before jump
        iss->set_reg(9, iss->get_reg(9) + 4);
        goto L8000;
      }
      // if (R8!=0)
      //   goto 0x8000 after delay slot
      // fall through to next instruction
    case 0x8008:
      iss->set_reg(9, iss->get_reg(9) + 4);
      // increment R9 by 4
      // fall through to next instruction
    case 0x800c:
      iss->set_reg(2, iss->get_reg(9));
      // copy R9 to R2
      // end of page
   }
   return 0x8010;
}
```

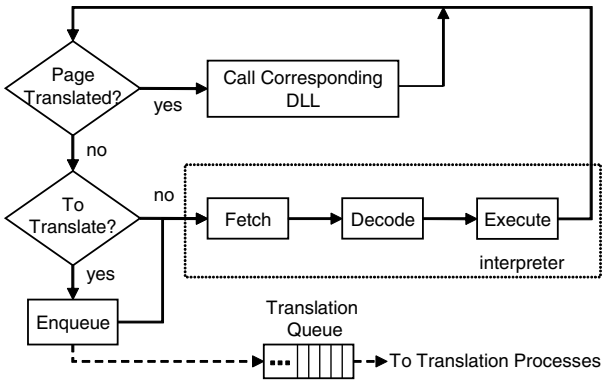
(b) Equivalent C++ code

Figure 1: A translation example

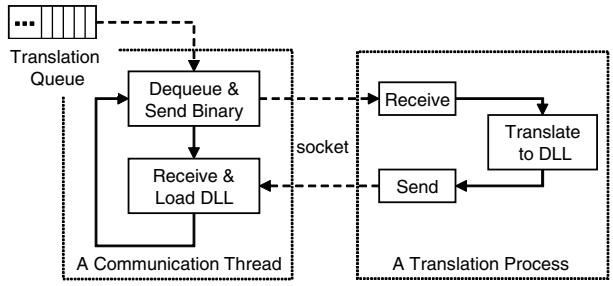
3.1 General Approach

During the simulation of a target program, the proposed dynamic-compiled simulator decodes and translates portions of the program into C++ code, which is then compiled by GCC[7] into dynamically loaded libraries (DLLs). The libraries are immediately linked to the simulation engine at runtime and are called when the simulated program counter enters the compiled portions of the target program.

The basic translation unit is a *page*, that is, a contiguous block of target instructions. We always translate a page into a C++ function. Figure 1 illustrates an example of four MIPS32 instructions before and after translation using this approach. The resultant function receives the program counter as an argument. Upon entry it switches execution to the corresponding case block and runs until the execution flow goes out of the range of the page. In real implementations, pages are much larger so that a single calling of the function is capable of simulating many instructions. By default, we



(a) Engine organization



(b) Inter-process communication

Figure 2: Structure of the simulator

use 512-word pages. To quickly compute page index, all pages are aligned at page-size boundaries.

The above translation approach is very similar to the one in FSCS except that we dynamically translate pages into DLLs during run time. The advantage of dynamic run-time translation is the capability to simulate self-modifying code, such as in the case of a boot-loader or a just-in-time engine. To simulate self-modifying code, we implemented an exception handling mechanism using the *setjmp* and *longjmp* functions of ANSI C. We monitor all memory instructions in the target program. When an instruction writes to a page that has been complied, an exception is triggered to interrupt compiled simulation. The simulator will unload the DLL corresponding to the modified page, and resume simulation. It may translate the modified page again when the execution flow reaches the same page.

3.2 Simulator Organization

Because of the latency of compiling translated pages in C++, the page translation process has some notable delay. In our experiment environment, it takes about 0.9 second to compile a translated 512-instruction page using GCC. If a page is rarely executed after the translation, there will be little return for the resource spent to translate it. Therefore, we selectively translate the most-frequently executed pages run-time profiling.

Figure 2(a) shows the general organization of our simulation engine. It is capable of hybrid interpretative and dynamic-compiled simulation. The interpreter simulates infrequently executed instructions and generates profiling statistics for selecting frequently executed pages to compile. We use a simple history-based heuristic for determining frequently executed pages. If, during interpretation, the dynamic execution count of a page exceeds a predefined threshold, the engine considers the page a candidate for binary translation and sends it to a translation queue. The engine continues interpreting the instructions in the page while the page awaits translation. After the page has been translated, the resulting code is linked to the simulator in the form of a DLL. The engine will invoke the DLL for future simulation of the instructions in the page.

In our previous single-processor implementation of the engine [5], the translation queue has zero capacity. In other words, once a candidate page is identified, the engine process immediately suspends simulation, translates the page, and then resumes simulation. This special case has been demonstrated beneficial for large benchmarks in our experiments.

In this paper, we propose a more general approach. First, we use a common queue of infinite capacity to buffer the indices of candidate pages. Second, we distribute translation tasks to other processes. The result of this extension is that we can utilize multiple processors in an asymmetric way in our simulator. A main processor is always fully utilized in running the the engine process. While the rest assistant processors actively perform the translation tasks on demand: whenever the queue becomes non-empty, an idle assistant processor, if any, pulls the first page from the queue and translates it. It is worth noting that the processors involved do not have to be homogeneous. As long as a processor is capable of generating DLLs compatible with the main processor, it can be utilized for translation.

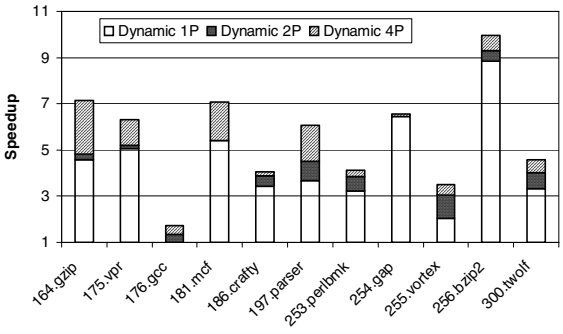
In our current implementation of the engine, we used several light threads to communicate with the translation processes through sockets. Figure 2(b) shows the interaction between such a communication thread and a translation process. This implementation choice involves some communication overhead for each translated page. However, it is highly flexible and allows us to perform experiments in a cluster of low cost workstations. The implementation can be easily adapted to run on a shared-memory multiprocessor server with reduced communication overhead.

4. EXPERIMENT RESULTS

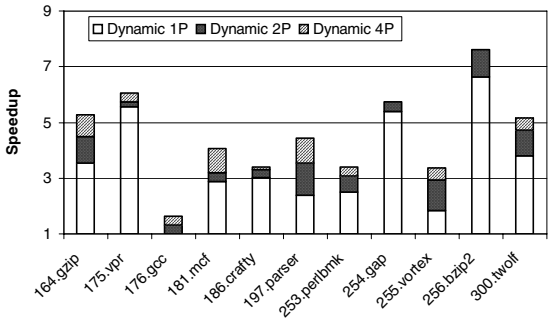
4.1 Setup

We implemented the proposed simulation approach in C++. For retargetability, we used a simple architecture description scheme to specify the binary encoding and the semantics of target instructions. We described the user-level subsets of two architectures, the MIPS32 ISA and the ARM V4 ISA. We used the descriptions to synthesize the proposed dynamic-compiled ISS's. In addition, we also synthesized two interpretive ISS's as reference simulators.

We performed our experiments on six 2.8GHz Pentium 4 workstations, each equipped with 1024KB L2 cache and 1GB RAM. The workstations run the Linux operating system with the 2.6 kernel. We used GCC 3.3.3 with the -O3 and *-fomit-frame-pointer* flags to compile the engine of the ISS's. The translated pages for the ISS's are compiled by the same GCC with the -O flag, which reduces compilation time without significantly affecting the quality of translated code. For all experiments, we use a page size of 512 words. Our results are based on simulating C-based SPEC CINT2000 [18] benchmarks. The benchmarks are compiled using cross-GCCs with the -O3 flag. For all benchmarks, their first reference inputs provided by SPEC are used.

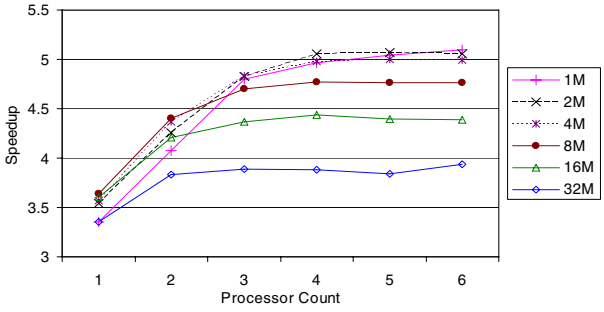


(a) MIPS32

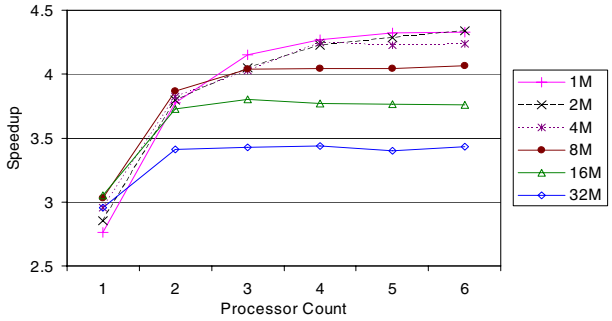


(b) ARM V4

Figure 3: Speedup for benchmarks



(a) MIPS32



(b) ARM V4

Figure 4: Variation of Speedup

4.2 Results

For each architecture, we first ran the interpreter for all benchmarks. We then ran the dynamic-compiled simulator using 1, 2, and 4 processors. We used a 2-million-instruction threshold for page selection. Figure 3 shows the speedup factors of our compiled simulation configurations with respect to interpretation. From the results, it is clear that multiprocessing offers consistently superior performance to interpretation or uniprocessing.

Table 1 summarizes the results from Figure 3 using geometric means. The results are based on the total CPU time of the simulation engine, which is a meaningful indication of the delay that the user will experience when waiting for a target program to finish simulation.

Several factors affect the speed of the dynamic compiled simulators, including the number of processors and the threshold value. To understand their effect, we vary the value of the threshold from 1M to 32M, and the number of processors from 1 to 6. Figure 4 shows the resulting speedup factors. The results indicate that the improvement due to multiprocessing saturates when more than 4 processors are used. In addition, it shows that the threshold value has mixed effects on performance. Using a smaller threshold means that more pages will be translated, and that more instructions will benefit from compiled simulation. However, when only one processor is used, the additional translation overhead may dominate the benefit of the increased percentage of compiled-simulation, resulting in lower performance. In contrast, the use of multiple processors significantly reduces such additional translation overhead and realizes the benefit of the smaller threshold. In general, 2 or 4-million seems a reasonable threshold value to use for all tested configurations.

	Speed (MIPS)/Speedup	
	MIPS32	ARM V4
Interpreter	38.9/1	31.5/1
Dynamic 1P	138/3.55	90.0/2.86
Dynamic 2P	166/4.27	121/3.84
Dynamic 4P	197/5.06	133/4.22

Table 1: Comparison of speed

5. DISCUSSION

5.1 Multiprocessing vs. Uniprocessing

According to the experiment results in Figure 3, multiprocessing offers on average 40–50% performance improvement over uniprocessing for our dynamic-compiled approach. Such performance improvement is highly valuable to speed-demanding applications. For instance, for programmers relying on ISS's to verify the correctness of software, faster simulation speed means less waiting time and hence higher productivity. Another beneficial usage is in those microarchitecture simulators based on statistical sampling or distributed methods [13, 8]. Their simulation speed asymptotically approaches that of the functional ISS when an infinite number

workstations are used. Improving the speed of the functional ISS will proportionally increase the upper speed limit of those microarchitecture simulators.

In addition, multiprocessing is free of the speed consistency problem of uniprocessing. For benchmarks with short execution traces, a uniprocessing dynamic-compiled ISS is sometimes slower than an interpreter since the translation overhead cannot be absorbed by the overall short trace. For example, the 176.gcc benchmark from SPEC CINT2000 takes 156 seconds on the interpretive MIPS ISS but 277 seconds on the uniprocessing dynamic-compiled ISS, a slow-down of 1.78x. Such slow-down effect is prevalent for target programs with relatively short execution traces, limiting the applicability of the uniprocessing approach. The problem is overcome by multiprocessing since it performs translation in parallel with simulation. In the same 176.gcc case, the simulation took 116 seconds on two processors and 90.5 seconds on four processors, both faster than interpretation.

For multiprocessing, the average load of each translation processor decreases when more processors are employed. When six or more processors are used, the average load of each translation processor is below 10% in our experiments. Thus, the proposed approach does not necessarily require exclusive access to multiple workstations. In contrast, it can be inexpensively deployed to salvage unused computing power in an enterprise computing infrastructure. A translation processor can also be shared by multiple simulation engines.

5.2 Dynamic vs. Static

Different from interpreters and dynamic-compiled simulators, a static-compiled simulator is specific to a target program. For each target program to simulate, an instance of the static-compiled simulator needs to be constructed. The construction process involves translating the entire target program – all its code pages – into host binary. Because of its lengthy construction process, static-compiled simulation is less convenient to use. It is only beneficial when the long constructing time is dominated by the total usage time. To study the difference of the two compiled simulation styles, we constructed a set of static-compiled simulators for the benchmarks in Figure 3. We then ran the simulators with the same inputs, and tabulated their average speeds in Table 2. The first row in the table does not include the construction time; the second row assumes that one processor is used to construct the simulators; and the third row assumes that four processors perform parallel construction, with each translating a quarter of the code pages.

	Speed (MIPS)/Speedup	
	MIPS32	ARM V4
Static without overhead	282/7.25	179/5.68
Static with overhead, 1P	46.0/1.18	34.1/1.08
Static with overhead, 4P	123/3.17	86.7/2.75

Table 2: Speed of static-compiled simulation

Comparing Table 2 with Table 1, one can see that the speed of static-compiled simulation can be interpreted as either faster or slower than that of dynamic-compiled simulation, depending on whether the construction overhead of the former is taken into consideration. Clearly, if a target program needs to be simulated for only once, then dynamic-compiled simulation has the advantage. On the other hand, if the target program needs to be simulated repeatedly for so many times that the initial construction overhead becomes negligible, static-compiled simulation has a speed advantage of about 40%. To mitigate this speed gap, we implemented a DLL cache in the host file system for our dynamic-compiled simulators. In the first simulation run of a target program, the generated DLLs are stored into the cache. During subsequent simulation runs, the simulator first looks up the cache to determine whether the current program has been simulated before. In case of a hit, the cached DLLs from the previous run will be loaded directly. Our experiments show that this scheme improves the performance of subsequent runs of target programs by 20% in the 4-way multiprocessing dynamic-compiled case. It effectively reduces the speed gap by half and makes the dynamic-compiled simulation approach more appealing.

As mentioned earlier, dynamic-compiled simulators are suitable to run self-modifying code such as boot-loaders, just-in-time engines, or full operating systems. In addition, the dynamic-compiled simulators mentioned in Section 4 can be readily extended with a remote debugging interface to communicate with the popular software debugger GDB. The simulators will then become the target emulation back-ends of GDB, which interprets user commands, controls simulation progress and queries processor states. User commands such as adding or deleting a break point require the back-ends to modify the program code, an impossible task for static-compiled simulators. Therefore, our dynamic-compiled simulators are highly valuable to programmers who need to verify and debug software. The faster speed offered by the multiprocessing approach makes them even more attractive because of the reduced the *latency* between inputing a user command and receiving a response from the simulator. Smaller latency implies higher productivity of software development.

The dynamic-compiled simulators can also offer higher simulation *throughput* than static-compiled simulators for regression tests, which are typically performed by compiler developers over a set of programs on several workstations. As demonstrated by Table 2, static-compiled simulators pay significant compilation overhead for each program to simulate since the whole program needs to be translated. Consequently, their overall throughput is no higher than interpretation for SPEC CINT benchmarks if each program is simulated only once. In contrast, dynamic-compiled simulation has smaller compilation overhead and thus features higher throughput.

5.3 Other Performance Issues

Our dynamic-compiled simulators are significantly faster than JIT-CCS and IS-CS. As shown in Table 1, for the ARM architecture, the average simulation speed is 90 MIPS with uniprocessing and 133 MIPS with quad-processing. In comparison, the reported speed of JIT-CCS was around 7 MIPS on a 1.2GHz Athlon [12] and that of IS-CS was 12 MIPS on a 1GHz Pentium 3 [16], both for the ARM ISA. Both are significantly slower than our dynamic-compiled ISS's after the difference of the workstations is taken into account. Most likely this is due to the high overhead of their fine-grained simulation engine which executes one instruction per-iteration.

To further improve the simulation speed of our dynamic-compiled simulators, we are currently studying two optimization opportunities. Firstly, the history-based page selection heuristic may be replaced by a predictive heuristic which proactively compiles pages before they will be executed. Successful predictions can further reduce the number of interpreted instructions. Secondly, multiple optimization levels may be used when compiling the translated C++ pages. The current implementation translates code pages using the -O option of GCC so that DLLs can be quickly generated. However, it may be profitable to recompile the heavily executed pages at a higher optimization level. Similar multi-level optimization techniques have been successfully adopted in Java virtual machines.

The reported results in Section 4 are all based on simulating statically linked target programs. In high end target systems with full-featured operating systems, shared libraries are often utilized. For such systems, pre-translation of shared libraries is an additional viable means to reduce translation overhead and to improve simulation speed.

6. CONCLUSIONS

In this paper we presented the first approach to construct dynamic-compiled simulators that benefit from multiprocessing. The approach involves pure C++ programming and can be quickly ported to any platform with a C++ compiler and a DLL interface. The approach not only offers better average performance than our uniprocessing dynamic-compiled simulator, but also solves its problem of speed inconsistency. In conclusion, the approach is suitable to be used in many design tools for computer systems, such as software debuggers or system-level simulators. The proposed simulation approach has been incorporated into our open-source simulators SimIt-ARM [14] and SimIt-MIPS [15]. Both are available for free public access.

7. ACKNOWLEDGMENTS

This research is partially supported by a UROP Faculty Matching Grant from Boston University. Subhendra Basu contributed to an early implementation of the presented approach. We thank the anonymous reviewers for their invaluable comments to improve the paper.

8. REFERENCES

[1] 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.

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

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

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

[5] J. D'Errico and W. Qin. Constructing portable compiled instruction-set simulators – an ADL-driven approach. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 112–117, 2006.

[6] M. J. Flynn and P. Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, 2005.

[7] Free Software Foundation, Inc. `http://gcc.gnu.org/`.

[8] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 1–12, 2003.

[9] Intel Corporation. Intel microprocessor quick reference quide, `http://www.intel.com/pressroom/kits/quickrefyr.htm`, Dec 2005.

[10] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[12] 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.

[13] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, 2003.

[14] W. Qin. `http://simit-arm.sourceforge.net`.

[15] W. Qin. `http://simit-mips.sourceforge.net`.

[16] 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*, pages 758–763, 2003.

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

[18] Standard Performance Evaluation Corporation. `http://www.spec.org`.

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

[20] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 298–302, 1999.