DynamoSim: A Trace-based Dynamically Compiled Instruction Set Simulator

Wai Sum Mong Jianwen Zhu

Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada
{mong, jzhu}@eecg.toronto.edu

ABSTRACT

Instruction set simulators are indispensable tools for the architectural exploration and verification of embedded systems. Different techniques have recently been proposed to speed up the simulation over the classical interpretation-based simulators, while maintaining their flexibility. In this paper, we introduce a suite of techniques inspired by recent advances in dynamic compilers to construct a hybrid simulation framework. Compared with compiled simulators reported earlier, our framework is more flexible, since any instruction can be interpreted; and faster, since only frequently executed instructions are translated on-the-fly into native code for direct execution, and the scope of our translation is extended from basic blocks to traces, and sophisticated register allocation is performed. Comprehensive results on SPEC2000 benchmarks are reported for the standard SimpleScalar processor to demonstrate the efficiency of proposed techniques.

1. INTRODUCTION

The increasing market demands of innovative electronic products result in a growing demand of processors embedded in systems-on-chip. In order to meet the required performance and power consumption constraints, the architecture of such an embedded processor is often application-specific and need to work with the rest of intellectual-property components that are often custom designed hardware. To verify such systems and perform the so-called architectural exploration, instruction set simulators (ISS) are widely used to validate and evaluate new processor architectures, and study their interactions with the rest of the on-chip components [1], before the actual hardware is built.

Instruction set simulation is a software technique that mimics the behavior of executing binary instructions on the target processor. The processor on which the binary instructions should run is called the *target processor*, while the processor on which the ISS runs is called the *host processor*.

The simplest ISS is usually *interpretation based*. An example of an interpreted simulator is the widely used *SimpleScalar toolset* [2]. Such simulators go through a loop of *fetch-decode-execute* cycle for each instruction to be simulated. Despite the flexibility it offers, an interpretation based ISS suffers from a performance problem. For example, the *gzip* benchmark in the standard SPEC2000 suite takes one week to simulate in SimpleScalar.

Compiled simulators translate the simulated target instructions into host instructions for direct execution, thereby eliminating the redundant computation involved in the decoding stage of interpreted simulation when an instruction is executed multiple times. Such binary translation can be carried out at compile time, called *static compiled simulation* [3, 4], or on-the-fly, called *dynamic compiled simulation* [5, 6, 7]. While compiled simulators can be made almost as fast as native execution [4], such performance is achieved without simulating all target machine states, for example, the conditional codes; or simulating the detailed memory architecture, such as caches. In addition, self-modifying codes which dynamically update the program instruction, cannot be simulated with compiled simulators, especially static compiled simulators.

To address the flexibility problem of compiled simulators, recent efforts explore the option called *cache-compiled simulation* [8, 9]. Instead of emitting native code for instruction to be simulated, this technique instead caches the decoded information of an instruction. The cost of instruction decoding can therefore be amortized over loops. On the other hand, the simulation framework is still interpretation-based and therefore flexibility is retained.

In this paper, we extend the line of work on compiled simulators based on dynamic instruction translation to further push the achievable simulation performance envelop, while retaining the flexibility to simulate arbitrary binary. More specifically, we make the following contributions in the paper: First, we propose an infrastructure that allows **hybrid** simulation: an instruction is by default interpreted, and is translated for native execution only when it is profitable. The flexibility offered by interpreted simulators are thus preserved. Second, inspired by recent advances in Just-in-Time compilers, we propose to extend the scope of dynamic translation, which is traditionally limited to basic blocks, to frequently executed **traces** spanning multiple basic blocks. A larger instruction-level parallelism can therefore be exploited by the host processor. Third, we perform a **trace-wide register allocation** to map the target machine state directly to host machine registers, thereby reducing the cost of frequent memory accesses.

The remainder of this paper is organized as follows. Section 2 gives a more detailed review of the related work reported in the literature. In Section 3, we describe our simulator infrastructure. In Section 4, we describe the trace-based compiled simulation technique. In Section 5, we describe the register allocation technique. We report our experimental results in Section 6 before we draw conclusions. Our simulator is built and evaluated for the SimpleScalar processor with the PISA instruction set. Since it is widely used in the academic community, throughout the paper we assume the target program is in PISA. However, our proposed technique is by no means limited to PISA.

2. RELATED WORK

Most instruction set simulators reported in embedded systems research are interpreted [10, 11, 12, 13, 14, 15]. Their major research goal is to equip retargetability to the simulators. In this paper we limit the discussion on simulation performance and flexibility only. Our retargeting strategy is reported in [16].

Zivojnvic et al. reported a static compiled simulator for DSP processors. The performance is 200-640 faster than the corresponding interpretive-based simulators. However, the simulation speed still ranges from 0.8 MIPS to 2.5 MIPS, partly due to the fact that bit true simulation of DSP instructions is more complex than RISC instructions. Zhu and Gajski reported a very fast static compiled simulator that can simulate up to 100 MIPS (within 1.6 times native time) by aggressively utilizing host machine resources. Dynamic compilation-based approach is pioneered by the Shade [5], Embra [6] and FastSim [7], which typically simulate RISC processors within 3-10 times native time. Compared to the proposed method, these simulators are not flexible, and do not simulate target machine state such as conditional codes, which can significantly slow down simulation.

The cache compiled simulator is first proposed by Nohl et al. [8] and later improved by Reshadi et al. [9]. Compared to the interpreted SimpleScalar [2], they report a speedup of around 4 for ARM7 processor on the JPEG and ADPCM benchmarks. Our work shares the same goal as cache compiled simulators for both speed and flexibility, however we achieve the goal by extending the work of dynamic compiled simulation using more aggressive techniques.

Two other fields are closely related to the compiled instruction simulators. Binary translation [17, 18, 19, 20, 21] promises to emulate the software of one platform, for example, a Microsoft Windows application, on another platform, for example, a Sun workstation. It is obvious that our technique can be used for the purpose of binary translation. However, the reverse is not true. The reason is that binary translation only needs the translated executable to produce the same result as the original. On the other hand, the simulation also needs to correctly maintain the target machine state at every simulated machine cycle. Given such freedom, binary translation can potentially achieve better performance by performing the so-called dynamic compilation.

Dynamic compilation is commonly used to optimize programs based on information which is not available until runtime. Widely used dynamic compilers include Just-in-Time Java compilers [22] which emit optimized native code for Java byte code. Since delaying all compilation tasks until runtime is expensive and unnecessary, systems like the DyC system [23] adopt the selective dynamic compilation approach. The same strategy is adopted in our simulator. As another example, the Dynamo [24] compiler interprets the input application and identifies the frequently taken paths (traces) at the same time. A trace is compiled and optimized into a fragment and it is cached for reuse. The overhead of Dynamo's operations will be amortized over the repeated executions of the fragments found in cache. The dynamic trace concept in our simulator is exactly like Dynamo. Our contribution here is to adapt it in a simulation environment and show it is beneficial.

3. HYBRID SIMULATOR INFRASTRUCTURE

Past simulators are either interpreted or compiled. In this section, we describe the proposed hybrid simulator infrastructure which features both a interpretation engine and dynamic compilation engine.

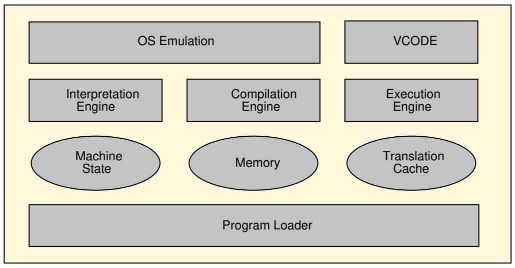

Figure 1: DynamoSim infrastructure.

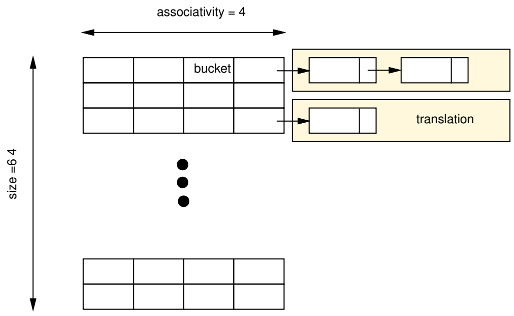

Figure 2: Translation cache.

The rationale behind such a hybrid structure is as follows. First, our simulator offers flexibility. With an interpretation engine always acting as a backup, any "irregular" code that cannot be handled well by compiled simulators, such as dynamic program code, can be adequately dealt with. Second, with the strategy that only selected code is dynamically compiled, our simulator does not suffer the problem of enormous memory usage experienced by dynamic compiled simulators. Our own experience indicate that each target instruction may be translated up to 20 host machine instructions. Therefore translating all instructions is not an option for large applications. Third, by selecting only code that is profitable for compilation, we can achieve the true speedup. Typical candidate code for compilation are those contained in loops. Code outside the loop is unprofitable to compile since the overhead offsets the gain.

Figure 1 gives the block diagram of the proposed simulator infrastructure. Like interpreted simulators, the infrastructure maintains the simulated target machine states, including general purpose registers and conditional codes, as well as the simulated memory. Unlike interpreted simulators, the infrastructure also maintains a data structure, called *translation cache*, which maps the *selected code regions* in the target program to the corresponding compiled native instructions. As shown in Figure 2, the translation cache is organized to mimic a configurable set associative cache. Each *tag* of the cache identifies a unique target code region characterized by its starting address. The content of the cache line is a *bucket* which points to a *translation*, which itself is a linked list of fixed sized blocks used to hold dynamically emitted native instructions. The link list data structure is necessary since we do not know in advance how large the selected code region and the translated host machine code will be. Therefore, a translation should be allowed to grow its size on demand.

The core of the infrastructure includes three engines, called the interpretation engine, compilation engine and execution engine. In the simulation main loop, the simulator is always executing one of the three engines. Three peripheral modules manage the interaction with the outside world: the program loader is responsible for the reading of binary executable; the OS emulation is responsible for handling operating system related system calls; and the VCODE [25] package is responsible for on-the-fly host code generation.

In the text that follows, we explain in detail each of the module contained in Figure 1 starting by the peripherals. Since the goal of our infrastructure is to establish a common framework so that different implementation schemes can be evaluated and compared, the description in this section will be general. The discussion will be substantiated in Section 4 when our proposed techniques are described.

3.1 Program Loader

The software program loader is responsible for loading the input executable stream into the simulated memory and initializing the execution environment. The loader requires the understanding of the binary file format in order to load the instructions and appropriate data to the simulated memory. We leverage the API supplied by GNU's BFD library [26] to identify the loadable sections in the executable. For each such section, memory as large as the section size is allocated, and APIs are invoked to copy the data from the executable on the disk to the allocated memory. Then, the loaded section data is copied to the simulated memory at the specified virtual memory address.

3.2 Operating System Emulation

Since in this work we do not simulate operating systems, an emulation layer has to be constructed to handle system calls, which are invoked by a trap instruction. According to the trap number carried in a simulated target register (e.g. `GPR(2)` in PISA), the required system call parameters are first obtained from the simulated registers, and the system call is then routed to the equivalent system call at the host system. The returned value is then copied back to the simulated registers. Implementing a small subset of POSIX system calls is usually enough to execute the common benchmarks.

3.3 The VCODE System

The VCODE (very fast dynamic code generation) is a dynamic code generation system developed by Engler [25]. We use the VCODE system to add the dynamic compilation ability to the *sim-safe* simulator. The VCODE system is a set of C macros and support functions that allow users to generate machine code at runtime. The VCODE interface provides the client with a view of a simple load-store RISC architecture, which is independent to the ISA of the generated instructions. Through the interface, an application can dynamically create, compile and run a *function*, which is the smallest unit of code generated by VCODE at a time.

3.4 The Interpretation Engine

The interpretation engine executes a loop where each iteration goes through the usual cycle of instruction interpretation. At the *fetch* stage, the instruction addressed by the PC register is copied from the simulated memory to the simulated instruction register (IR). The fetched instruction is then decoded at the *decode* stage, which extracts the necessary information needed for execution from the instruction word. Decoding complex instruction set typically involves walking down a decision tree implemented by nested switch statements. By using the decoded value, the appropriate instruction semantics definition is selected and executed.

Unlike traditional interpreted simulators, our interpretation engine handles the additional task of instruction *monitoring*: depending on the type of the instruction being interpreted, the interpretation engine may perform a translation cache lookup. If it is a hit, it will exit the interpretation loop and hand off the control to the execution engine. It may also keep track of a *counter* so that when a threshold value is reached, it will exit the interpretation loop and hand off the control to the compilation engine.

3.5 Compilation Engine

The compilation engine is triggered when some candidate target code regions are identified for dynamic compilation. It is responsible for translating target machine instructions into host machine instructions stored in the corresponding translation.

Our compilation engine translates one instruction at a time using the VCODE interface. Since it may bring some performance improvement, we actually interpret each instruction as we translate it. The emitted native code consists of three parts: the *prologue*, consisting of code to load simulated machine state from memory to host machine registers, the *epilogue*, consisting of code to commit values contained in host machine registers to the simulated machine state, as well as the *body*, consisting of codes implementing the semantics of the instruction. It is important to note that to simulate all the machine state, simulating an *add* instruction on the target involves much more than finding an *add* instruction on the host. In fact, in order to find the correct conditional code after executing the target *add* instruction, a list of 10 extra instruction has to be emitted. We found for the PISA instruction set, an average number of host instructions per target instruction is 20.

The scope of translation is referred to as regions, and each such region is traditionally a basic block, in which case a branch taken instruction signals the end of the region. In addition, an instruction which is the starting address of an already compiled region also signals the end of current region.

When the compilation engine reaches the end of the region, it hands off control to the execution engine, or the interpretation engine, depending on whether there is a hit in the translation cache lookup.

3.6 Execution Engine

The execution engine is responsible for the execution of translations, or dynamically generated native code for the corresponding regions. VCODE dictates that dynamically generated codes are encapsulated in procedures. Therefore the execution of translations amounts to an indirect procedure calls to the translation. When returned, a lookup to the translation cache will be performed with the current PC. If it is a hit, the found translation will be invoked. Otherwise, a counter maybe updated and depending on whether its value exceeds certain threshold, control may be handed off either to the interpretation engine or the compilation engine.

4. TRACE-BASED DYNAMIC COMPILED SIMULATION

It is obvious that the most profitable regions for dynamic compilation are those frequently executed codes residing in loops. However, the scope of dynamic compilation of the traditional techniques is limited to basic blocks. Operating with such a fine granularity brings many disadvantages. First of all, the overhead associated with executing a translation cannot be amortized over large number of instructions. One example of such overhead is the cost of indirect function call for translation invocation. Another example is the cost of executing prologue and epilogue. Second, with a small code size, the instruction level parallelism inherent in the host processor cannot be readily exploited. Therefore it is desirable to extend the compilation region beyond the scope of basic blocks.

ALGORITHM 1. *Trace-based Dynamic Compiled Simulation.*

```
interp:
  while( true ) {
    interpret instruction at PC;
    if( it is a taken branch ) {
      translation = cacheLookup( NPC );
      if( translation != null ) {
        update PC, NPC; goto execute;
        }
      if( it is a backward branch ) {
        counter( NPC ) ++;
        if( counter( NPC ) > threshold ) {
          update PC, NPC; goto compile;
          } } }
    update PC, NPC;
    }
compile:
  start a new trace translation;
  while( true ) {
    interpret instruction at PC;
    emit simulation code in translation;
    if( it is a taken branch ) {
      translation = cacheLookup( NPC );
      if( translation != null ) {
        emit code updating PC, NPC;
        update PC, NPC; goto execute;
        }
      if( it is a backward branch ) {
        counter( NPC ) ++;
        if( counter( NPC ) > threshold ) {
          emit code updating PC, NPC;
          update PC, NPC; goto compile;
          }
        else {
          emit code updating PC, NPC;
          update PC, NPC; goto interpret;
          } } }
    update PC, NPC;
    }
execute:
  while( translation ) {
    invoke native code in translation;
    translation = cacheLookup( PC );
    }
  counter( PC ) ++;
  if( counter( PC ) > threshold )
    goto compile;
  else
    goto interp;
```

In recent studies of dynamic compilation [24], the concept of *hot trace* has been demonstrated for improving the performance of dynamically compiled programs dramatically. Trace is defined to be a sequence of frequently executed instructions. A trace could include branches, therefore span beyond the scope of basic blocks, however, does not span across loops. While the concept of trace has been proposed much earlier, the authors of [24] proposed a way to detect trace dynamically.

We adapted the techniques in [24] so that it works for the purpose of simulation. The key to identify trace is actually very simple: First, all *branch taken targets* are detected by looking at the target address of branch instructions. Second, the detection of the so-called *backward branches* can be achieved by comparing the values of the branching target and the PC. Therefore branching to a target with an address value smaller than the current PC is identified as a backward branch. Third, a counter is maintained for every potential start of trace, which can either be a branch taken target, or the NPC of the backward branch. This way, any executed loop header will be considered as potential start of trace, and any branching to another loop in a trace will be forced as one exit of a trace. Whenever such a trace candidate is encountered in interpretation, the corresponding counter is incremented. When the counter value reaches certain threshold, dynamic compiler will be invoked to compile the trace into native code. Interested readers are referred to [24] for detail.

With these simple techniques, we can substantiate the generic algorithm described in Section 3 to perform selected dynamic compiled simulation at the trace level. The algorithm is organized as a state machine, where *interp*, *compile* and *execute* are labeled states. State transition follows the *goto* statements in the algorithm.

It is important to note that not all techniques reported in [24] can be used for the purpose of simulation. Our dynamic compiler has to strictly adhere to the requirement of simulation, which means no compiler optimization such as out-of-order scheduling should be performed. Instead, the target machine states need to be faithfully maintained. As another example, the design of our translation cache and its replacement policy are also significantly different from that of [24]. The memory requirement of cache in [24] is unacceptably high for the purpose of simulation. The target machine instructions can be translated into equivalent host machine instructions of almost equal size in the case of [24], however in our case, a ten fold increase of host machine instructions is not uncommon. We choose to use a 4-way set associative cache for this purpose and use a least recently used replacement policy. In contrast, [24]'s cache replacement policy occurs at a much coarse grained level, meaning recompilation occur at a large scale. Since the cost of our translation is much higher, we cannot afford to adopt this policy.

5. REGISTER ALLOCATION

To simulate a typical target instruction, the value of its two operands need to be retrieved from the simulated machine states maintained in the simulator memory, and after certain computation specific to the opcode of the instruction, the computed result needs to be committed to its destination, meaning the simulator needs to write the result into the simulated machine states. Thus, at least three separated memory accesses are needed to simulate an instruction. This is an expensive process and the purpose of register allocation is to reduce this cost by mapping the simulated target machine registers directly to host machine registers.

Note that our register allocation is under different constraints from the register allocation in compilers, and therefore is a completely different problem. For hybrid simulators, a specific requirement is to maintain the consistency between the interpretation engine, which keeps target machine states in memory, and the execution engine, which keeps target machine states in host machine registers. In addition, it is very likely that the host machine does not have enough registers to hold all target machine states to be simulated. The register allocator therefore has to build an abstraction between the compilation engine and the VCODE module so that the described complications can be hidden.

A natural scope of register allocation is the trace described in Section 4. As the execution engine enters the trace, the prologue of a target instruction will be executed to copy the value stored in simulated machine state to a host machine register determined by the register allocator at compile time (when the compilation engine is executed). Subsequent references to the corresponding target machine register can then be directly routed to the allocated host machine register. As the execution engine leaves the trace, the epilogue will be executed so that all machine states maintained in host machine registers will be committed back to the simulated machine states. The coherence between the interpretation engine and execution engine can therefore be maintained.

We have just described an ideal case when every target machine registers used in the trace are allocated with a host machine register and such mapping holds for the entire life time of the trace.

This does not happen in reality since host machine registers are not always available. To solve this problem, we reserve a few *scratch registers* in the host machine, such that whenever allocation fails, the scratch registers are used. The restriction of scratch registers is that their mapping to target machine registers are temporary. Therefore, immediate synchronization with the simulated target machine states is needed.

Algorithm 2 and Algorithm 3 show two APIs supplied by the register allocator to the compilation engine to facilitate the dynamic code generation when operands and destinations of the target instruction need to be accessed. As can be seen from these two similar algorithm, the register allocator first tries to find if the requested target register has already been allocated by consulting the mapping *ireg_mapping*. If it has not been allocated, the VCODE register allocator is called to allocate a host machine register and the mapping *ireg_mapping* is updated accordingly. If VCODE allocator fails, the specified scratch register is accessed.

ALGORITHM 2. *Target register read access.*

```
ra_getreg( int reg_index, int scratch_index ) {
 if( ireg_mapping[reg_index] != -1 )
   return ireg_mapping[reg_index];
 vreg = invoke vcode register allocator;
 if( successful ) {
  ireg_mapping[reg_index] = vreg;
  emit code to load simulated register to vreg;
  }
 else {
  emit code to load simulated register to scratch[scratch_index];
  return scratch[scratch_index];
  }
}
```

ALGORITHM 3. *Target register write access.*

```
ra_putreg( int vreg, int reg_index ) {
 if( ireg_mapping[reg_index] != -1 )
  emit code to copy vreg to ireg_mapping[reg_index];
 ireg_mapping[reg_index] = invoke vcode register allocator;
 if( successful )
  emit code to copy vreg to ireg_mapping[reg_index];
 else
  emit code to copy vreg to simulated register;
}
```

With the abstraction built by the register allocator, the compiler engine is not too much from the one without the register allocator. On the other hand, performance can be improved. This is especially true when the granularity of the trace is large. The advantage will be less obvious if the target region is much smaller, for example, at the basic block level.

6. EXPERIMENTAL RESULT

To evaluate the proposed techniques, we report results of four different simulation configurations under a common simulator framework. The first configuration is a pure interpreted simulator. In fact, we directly used *sim-safe* from the SimpleScalar tool suite. The second configuration is a pure dynamic compiled simulator with the same strategy as [27, 6]. The scope of dynamic compilation is basic blocks. The third configuration is the use of hybrid simulation, where the scope of dynamic compilation is trace (Section 4). The fourth configuration is the use of hybrid simulation in addition to register allocation (Section 5).

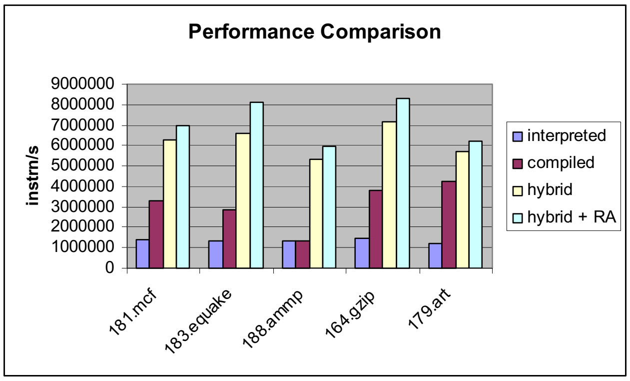

Figure 3: Simulation speed comparison.

Benchmark	interpreted (sim-safe)	compiled (basic block)		hybrid (trace)		hybrid+RA (trace)	
	MIPS	MIPS	speedup	MIPS	speedup	MIPS	speedup
181.mcf	1.37	3.32	2.42	6.29	4.59	6.95	5.07
183.equake	1.35	2.86	2.12	6.56	4.86	8.13	6.02
188.ammp	1.31	1.32	1.01	5.33	4.07	5.97	4.56
164.gzip	1.45	3.80	2.62	7.18	4.95	8.29	5.72
179.art	1.23	4.26	3.46	5.73	4.66	6.21	5.05
average			2.33		4.63		5.28

Table 1: Simulation speed.

We report results on benchmarks from SPCE2000, including two integer benchmarks (181.mcf and 164.gzip) and three floating point benchmarks (183.equake, 188.ammp and 179.art). Our results are collected on a 750MHz SunBlade 1000 workstation.

The simulation speeds for different benchmarks under different configurations are reported in Table 1 in terms of millions of simulated instruction per second (MIPS) as well as speed up against sim-safe. It is interesting to observe that pure dynamic compilation achieves an average speedup of 2.33, and under one occasion (188.ammp) it is almost at the same speed interpretation. This rather modest gain is mostly due to the unprofitable translations of infrequently executed instructions. In contrast, the hybrid simulation using trace-based compilation achieves an average speedup of 4.63, and about 2 times faster than purely compiled simulation. When adding register allocation, the speedup of 5.28 is achieved in hybrid simulation. This result is visualized in Figure 3.

In order to precisely study the effects of proposed techniques, we made an effort to ensure the comparison is made fairly. It is important to note that compiled instruction set is routinely reported with hundred time speedup [5, 6, 4]. These results are reported based on binary translation technology which does not keep track of special machine states such as conditional code. In addition, memory simulation is not performed. Therefore, it is typical to simulate one target instruction with only a few host machine instructions. The decoding overhead eliminated by compilation therefore occupies a significant portion of total simulation time. On the other hand, the majority of our emitted instructions are actually code that keep tracks of such registers. Therefore, on average 20 VCODE instructions need to be emitted for each target instruction. The speedup is therefore significantly smaller but we believe this is a more realistic figure in embedded systems verification, since many machine features are of interest.

While it is clear that our proposed method is faster than purely dynamic compiled simulation, it is still not clear how it exactly

Statistics	compiled	hybrid	hybrid+RA
#translation	190850	194	195
avg. region size	4.7	53.0	51.0
avg. translation size	122.7	994.8	423.9
#cache accesses	60518844	18204469	18415286
cache miss rate	0.32%	0.02%	0.02%
#instrn interpreted	0	40514	41842
#instrn compiled/executed	419091513	419050999	419049671

Table 2: Various statistics - 181.mcf.

compares with cached compiled simulation in terms of speed. While desirable, such conclusion is difficult to draw since a comprehensive results are not reported in [8, 9], and report for a different processor (ARM7). On the other hand, our speedup seems to fall in the same order of magnitude as [8, 9].

It is instructive to compare various statistics we collect for benchmark *181.mcf* in Table 2, which can shed more insight in the proposed techniques. It can be observed that the average target region size for trace based method is about 50 target instructions, whereas for basic block based method is about 5 target instructions, a factor of about 10. As a result, the average size of the translation of trace based method is also about 10 times more host machine instructions. It clearly showed the much coarser granularity of trace-based approach. The translation cache miss rate if trace-based approach is also significantly lower for the same cache configuration. It is interesting to note for our hybrid approach, many instructions never get compiled and are always interpreted.

7. CONCLUSION

In this paper, we argue the importance of instruction set simulators in general in the context of embedded systems-on-chip, as well as the importantance of simultaneously achieving fast simulation speed and flexibility.

We introduce a new path of achieving the goal different from recently proposed cached compiled simulators. Inspired by recent advances in dynamic compilers, we introduce the strategy of hybrid simulation that combines the flexibility of instruction set simulation and the speed of dynamic compilation. With our study, we conclude that this strategy does achieve the goal. In addition, the proposed technique of trace-based dynamic compilation can lead to a similar performance gain in simulation to that experienced in compilers. In addition, our trace-wide register allocation scheme is effective.

8. REFERENCES

[1] J. Rowson, “Hardware/software co-simulation,” in *Proceeding of the Design Automation Conference (DAC)*, 1994.

[2] *SimpleSclar LLC*, `http://www.simplescalar.com`.

[3] V. Zivojnvic, S. Tjiang, and H. Meyr, “Compiled simulation of programmable DSP architectures,” in *Proceedings of the 1995 IEEE Workshop on VLSI Signal Processing*, Sakai, Japan, 1995.

[4] J. Zhu and D.D. Gajski, “A retargetable, ultra-fast instruction set simulator,” in *Proceedings of the Design Automation and Test Conference in Europe (DATE)*, Munich, Germany, March 1999.

[5] R. F. Cmelik and D. Keppel, “Shade: A fast instruction-set simulator for execution profiling,” in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[6] E. Witchel and M. Rosenblum, “Embra: Fast and flexible machine simulation,” in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[7] E. Schnarr and J. Larus, “Fast out-of-order processor simulation using memorization,” in *Proceeding of the International Conference on Architectural Support for Programming Lanuages and Operating Systems (ASPLOS)*, San Jose, California, October 1998.

[8] A. Nohl, G. Braun, O. Schilebusch, R. Leupers, H. Meyr, and A. Hoffmann, “A universal technique for fast and flexible instruction-set architecture simulation,” in *Proceeding of the Design Automation Conference (DAC)*, June 2002.

[9] M. Reshadi, P. Mishra, and Nikil Dutt, “Instruction set compiled simulation: A technique for fast and flexible instruction set simulation,” in *Proceeding of the Design Automation Conference (DAC)*, Anaheim, CA, June 2003.

[10] S. Sutarwala, P. Paulin, and Y. Kumar, “Insulin: An instruction set simulation environment,” in *Proceedings of CHDL-93*, Ottawa, Canada, 1993.

[11] W. Geurts, D. Lanneer, G. Goossens, and et al., “Design of dsp systems with CHESS/CHECKERS,” in *Handouts of the 2nd Int. Workshop on Code Generation for Embedded Processors*, Leuvan/Belgium, 1996.

[12] M. Hartoog, J. Rowson, P. Reddy, and et al., “Generation of software tools from processor descriptions for hardware/software codesign,” in *Proceeding of the Design Automation Conference (DAC)*, 1997.

[13] A. Fauth, “Beyond tool-specific machine descriptions,” in *Code Generation for Embedded Processors*. 1997, Kluwer Academic Publishers.

[14] G. Hadjiyiannis, S. Hanono, and S. Devadas, “Isdl: an instruction set description language for retargetability,” in *Proceeding of the Design Automation Conference (DAC)*, Anaheim, CA, June 1997.

[15] M. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khulla, “Generation of software tools from processor descriptions for hardware/software codesign,” in *Proceeding of the Design Automation Conference (DAC)*, Anaheim, CA, June 1997.

[16] W. S. Mong and J. Zhu, “A retargetable micro-architecture simulator,” in *Proceeding of the Design Automation Conference (DAC)*, Anaheim, June 2003.

[17] K. Andrews and D. Sand, “Migrating a CISC computer family onto RISC via object translation,” in *Proceeding of the International Conference on Architectural Support for Programming Lanuages and Operating Systems (ASPLOS)*, P. Marwedel, Ed., October 1992.

[18] R.L. Site et al., “Binary translation,” *Communication of the ACM*, Feberary 1993.

[19] C. Cifuentes, “Partial automation of integrated reverse engineering environment of binary code,” in *Proceedings Third Working Conference on Reverse Engineering*. November 1996, pp. 50–56, IEEE-CS Press.

[20] W. F. Kao and I. J. Huang, “Instruction retargeting based on the state pair notation,” in *Asia Pacific Conference on Hardware Description Languages*, 1997, pp. 114–120.

[21] A. Ghernoff et al., “A profile-directed binary translator,” *IEEE Micro*, pp. 56–64, March/April 1998.

[22] V.C. Screedhar, M. Burke, and J.D. Choi, “A framework for interprocedural optimization in the presence of dynamic class loading,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2000.

[23] B. Grant, M. Philipose, M. Mock, C. Chambers, and S.J. Eggers, “An evaluation of staged run-time optimizations in dyc,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[24] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A transparent dynamic optimization system,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2000.

[25] D. R. Engler, “VCODE: A portable, very fast dynamic code generation system,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[26] S. Chamberlain, *libbfd: the Binary File Descriptor Library*, Free Software Foundation, April 1991.

[27] Robert F. Cmelik and D. Keppel, “Shade: A fast instruction-set simulator for execution profiling,” Tech. Rep. SMLI 93-12, UWCSE 93-06-06, University of Washington, 1993.