A Retargetable, Ultra-fast Instruction Set Simulator

Jianwen Zhu, Daniel D. Gajski
CECS, Information and Computer Science
University of California
Irvine, CA 92717-3425, USA
jzhu, gajski@ics.uci.edu

Abstract

In this paper, we present new techniques which further improve the static compiled instruction set architecture (ISA) simulation by the aggressive utilization of the host machine resources. Such utilization is achieved by defining a low level code generation interface specialized for ISA simulation, rather than the traditional approaches which use C as a code generation interface. We are able to perform the simulation at a speed up to 10^2 *millions of simulated instructions per second (MIPS). This result is only* $1.1 - 2.5$ *times slower than the native execution on the host machine, the fastest to the best of our knowledge. Furthermore, the code generation interface is organized to implement a RISC like virtual machine, which makes our tool easily retargetable to many host platforms.*

1 Introduction

An *instruction set simulator* is a tool that runs on a *host* machine, typically a workstation, to mimic the behavior of, or *simulate* a program running on a *target machine*, which either does not yet exist, or not available. Typically, instruction set simulation allows the user to examine the internal state of the target machine, such as the value of processor registers, during the execution of each instruction.

Instruction set simulators are indispensable tools in the development of conventional computer systems. They help to *validate* the processor design, the compiler design, as well as *evaluate* architectural design decisions such as cache sizes. Instruction set simulators play an even more important role in the development of modern *embedded systems*, which typically integrate one or more processors, acceleration hardwares, and sometimes analog frontends, on one chip to implement one specific application, such as cellular phone and personal communication systems. *Hardware/software cosimulation* [5], of which instruction set simulation is one of the most important parts, must be performed in order to validate and evaluate not only architectural decisions, but also implementation decisions such as how the functionality of the application is partitioned into hardware and software before any such systems are built. Such capability of *virtual prototyping* is essential to the success of a product.

It is obvious that the most important quality metric of an ISA simulator is its *simulation speed*, which is especially relevant for the development of high performance systems, where being able to perform simulation in *real time* is desired. Hardware *emulation*, despite its cost, has to be used when real time simulation is impossible. Other quality metrics include *compilation speed*, which has to do with how fast simulator can bring an application into a simulatable state; *tracability*, which has to do with how flexible the simulator can collect useful statistics, such as instruction profiling; *retargetability*, which has to do with how easy the tool can be extended to handle new target machines and new host platforms; *interoperatability*, which has to do with its capability to integrate with other tools, such as debugger, hardware simulator, etc.

Due to its importance, numerous ISA simulators have been developed, which can be categorized into three types (Section 2), namely, *interpretation based*, *static compilation based* and *dynamic compilation based*.

The tool presented in this paper is a static compilation based simulator. In addition to the advantages inherited, our tool makes several contributions, which lead to its superior performance. First, we propose to use a RISC like *virtual machine*, which has a predefined instruction set and an unlimited number of virtual registers, to serve as the intermediate to which the target instructions get translated, and from which the host instructions are generated. This is in contrast to the dynamic approaches which usually directly emit host instructions, where retargetability has to be sacrificed; and the traditional static approaches which emit C, where the direct manipulation of host machine resources is impossible.

Second, we use an aggressive, yet extremely simple *register allocator*, which is tailored for the purpose of ISA simulation. Effectively, this allows the direct mapping of target machine registers to host machine registers, while retaining retargetability. Such effect is hard, if not impossible to achieve in the traditional C emitting approach, even when sophisticated optimizations are used.

In addition, the low level interface proposed allows us to bypass the host machine calling conventions, which effectively expose more registers for the register allocator to manipulate on host machine architectures with register windows, such as SPARC. In combination, we have been able to simulate the benchmarks only 1.1-2.5 times slower than the execution of their counterparts directly compiled on the host machine, when tracing is off. This result is on average 2 times faster than the state of the art [4] [3]

[6].

The remainder of this paper is organized as follows. Section 2 gives more detailed description on the various approaches and compare their trade-offs. Section 3 presents the detail of our simulator. Section 4 discusses the extensions and limitations. Section 5 gives the results on the benchmarks chosen.

2 Techniques for ISA Simulation

2.1 Interpretation Based Simulation

Interpretation based simulation builds in memory a data structure representing the state of the target processor. It then enters a loop, the body of which executes the sequence of actions : *fetch*, which reads an instruction word from memory; *decode*, which analyzes the instruction and extracts the opcode field of the instruction; *dispatch*, which use a switch statement to jump to the appropriate code to handle a particular instruction; *execute*, which update the processor state according to the semantics of the instruction.

A representative, widely used interpretative simulator for MIPS processor is described in [2]. All most all commercially available simulators are interpretative. Despite ease of implementation and flexibility, interpretive simulators suffer performance problems, mainly due to the tremendous overhead spent on instruction fetching, decoding and dispatching, which, from simulation point of few, is *unproductive*. The simulator [2] reports a 25 times slow down of the native execution. [6] reported that it takes DSP simulators provided by vendors 6.4 hours to simulate G.726 speech transcoder for 13 seconds of speech signals, in contrasts to the 7 seconds of native execution time.

2.2 Compilation Based Simulation

Compilation based approaches reduce the runtime overhead by translating each target machine instruction directly to a series of host machine instructions which manipulate the simulated machine state. For example, the MIPS code in Figure 1 get translated to the SPARC code in Figure 2 for simulation. Here, `sp__sim` is the memory location which hosts the value of the simulated `sp` register.

```
addu $sp,$sp,-80
```

Figure 1. Target code

```
sethi %hi(sp__sim), %l0
ld [%lo(sp__sim)+%l0], %l1
add %l1, -80, %l2
sethi %hi(sp__sim), %l3
st [%lo(sp__sim)+%l3], %l2
```

Figure 2. Simulation code

Such translation can be done either at compile time, as in the case of static compiled simulation, where the overhead is completely eliminated; or at load time, as in the case of dynamic compiled simulation, where the overhead is amortized over the loops which repeatedly execute the same code.

2.3 Related Works

Static compiled simulation usually translates the target program into C code, and then use an optimizing C compiler (e.g., gcc with option -O3) to translate the C code into host machine instruction. In [6], Such simulators are developed for DSP processors. The authors reported 200-640 times speed up than the corresponding interpretative simulator. 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.

Dynamic compiled simulation translates the target program into host machine code on the fly. This approach is pineered by the shade simulator [3], where the SPARC V8, V9 and MIPS instruction set can be simulated at 3-10 times native time. Inspired by [3], the Embra simulator [4] performs complete machine simulation with similar performance.

The techniques discussed in this paper are not limited to embedded system design. It is also closely related to binary translation, which promises to emulate software of one platform, for example, a windows application, on another platform, for example, a SUN workstation.

3 A New Approach for Static Compiled Simulation

As shown in Figure 3, our simulator looks like, and in fact is integrated into, a retargetable compiler. The backend (e.g., *MIPS target* in Figure 3) which emits simulation code for a particular architecture, however, is slightly different from the corresponding cross compilation backend in that for every target instruction to be emitted, it emits a series of virtual machine instructions (Section 3.2) through the simulation code generation interface (Section 3.1) instead. The code generation interface is in turn implemented by a *host*, which translates each virtual machine instruction into a form which can be compiled into host machine instructions. The hosts might manage the host machine registers by the help of a *register allocator* (Section 3.4), which is designed to be machine independent.

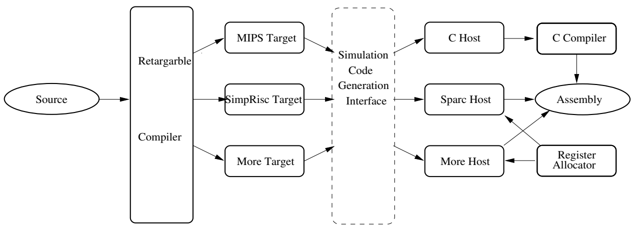

Figure 3. Simulator organization

```
public enum SegKind {
  SEG_CODE = 1, SEG_BSS, SEG_DATA, SEG_LIT
  }

public interface Host {
  void begin();
  void end();
  void exportSymbol( String symbol );
  void importSymbol( String name, int size );
  void segment( SegKind seg );
  void beginFunction( String name );
  void endFunction( String name );
  void emitConstantValue( Type type,
    Object value );
  void emitAddressValue( String name );
  void emitStringValue( int n, String name );
  void emitSpace( int size );
  void emitSymbol(
    String name, int size,
    int align, int isstatic
    );
  void emitInstrn(
    Opcode opcode, Type type,
    TargetExpr dest,
    TargetExpr op1, TargetExpr op2
    );
  int declGlobal( String name );
  int declLocal();
  void undeclAllLocals();
  }
```

Figure 4. Simulation code generation interface

3.1 Simulation Code Generation Interface

Figure 4 defines the interface that every host has to implement. `begin` and `end` gives the host an opportunity to initialize and finalize its internal data structure. As their name implies, `exportSymbol` and `importSymbol` exports and imports symbols. `segment` switches the current segment to either text segment (`SEG_CODE`), or uninitialized data segment (`SEG_BSS`), or data segment (`SEG_DATA`), or constant data segment (`SEG_LIT`). `beginFunction` and `endFunction` signals the beginning and the end of a function. `emitConstantValue`, `emitAddressValue`, and `emitString` emits compile time values. `emitSpace` emits uninitialized data. `emitSymbol` emits either a data symbol or a label.

The interface also abstracts the host machine resources by a virtual machine, as defined in Section 3.2. The interface functions `emitInstrn` and `declGlobal`, `declLocal`, `undeclAllLocals` manage the virtual instructions and the virtual registers of the virtual machine respectively.

The retargetability of our simulator attributes to the fact that the hosts are completely decoupled from the targets thanks to the code generation interface. The host can emit C code (e.g., *C Host* in Figure 3), an approach equivalent to [6]; or directly emit host machine assembly (e.g., *Sparc Host* in Figure 3).

3.2 Virtual Machine

The virtual machine that we define has an instruction set that resembles [8], which in turn is derived from the intermediate representation of [9]. Each instruction is represented as a value tuple of opcode, type, destination and operands. The opcodes include arithmetic/logical operations, load/store operations and control transfer operations. The types further constrains the operations to work on a byte (signed or unsigned), halfword, word, long, single and double precision floating point, pointer value. They are defined in Figure 5.

```
public enum Opcode {
  OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_MOD,
  OP_AND, OP_OR, OP_XOR, OP_SHL, OP_SHR,
  OP_COMP, OP_NOT, OP_NEG, OP_MOV, OP_SET,
  OP_CNVI, OP_CNVU, ...,
  OP_LD, OP_ST,
  OP_RET, OP_J, OP_JAL,
  OP_BLT, OP_BLE, OP_BGT, OP_BGE,
  OP_BEQ, OP_BNE,
  OP_NOP
  }

public enum Type {
  TYPE_C, TYPE_UC, TYPE_S, TYPE_US,
  TYPE_I, TYPE_U, TYPE_L, TYPE_UL,
  TYPE_F, TYPE_D, TYPE_P, TYPE_V,
  }
```

Figure 5. Virtual machine definition

The operands can be either a constant, a symbol, an expression which manipulate constants and symbols, or a virtual register. The destination is always a virtual register.

Our virtual machine has an unlimited number of virtual registers. The virtual registers are categorized into *global* registers, which is alive during the entire program execution; and *local* registers, whose value only last a short time, typically one simulated instruction.

3.3 Target Implementation

A target uses the code generation interface to emit simulation code. It first allocates a set of global virtual registers, which usually correspond to the target machine registers. It then emits a set of virtual instructions for every target instruction, while making sure that they have the same semantics. Note that usually one virtual instruction is enough for a target instruction. In case not, local virtual registers have to be allocated for temporary storage. For example, the MIPS instruction in Figure 1 is mapped to the virtual instruction `add_i vsp, -80, vsp`, where `vsp` is a virtual register.

The target calls other interface functions to emit data and other assembly directives.

3.4 Machine Independent Register Allocator

Most virtual instructions apply certain operations on some source virtual registers and write the result to the destination virtual registers. Each virtual register has a memory location in the simulation code to hold its value. For efficiency, the virtual registers should be cached in the host machine registers, called the *hard* registers. The policy towards how the virtual registers are cached comprises the job of the register allocator.

3.4.1 Greedy Allocation

The straightforward solution is to fetch the source virtual register values from the memory to some scratch registers, compute it, and then stores the result immediately to the memory. An example of such strategy is shown in Figure 2.

3.4.2 Lazy Allocation

A better policy is to perform lazy fetching, that is, virtual register values need not to be loaded from the memory if it is not recently written after it is last read from the same basic block; and lazy flushing, that is, virtual registers need not to be written to the memory until the end of a basic block. Here, the basic block refers to a piece of code which contains a single entry and does not contain control transfer instructions except the last one. On the other hand, in case no hard register is available, *spilling* has to be performed. Essentially, spilling select a virtual register to give up its occupancy of the corresponding hard register, by first flushing its value if it is "dirty", or, its value is inconsistent with that stored in the memory.

3.4.3 Fixed Allocation

Lazy allocation inserts fetching code for the first use of virtual registers in the basic block, the spilling code which flushes virtual register, and an epilogue for every basic block which flushes all the "dirty" virtual registers, for every basic block. These overheads are needed because the mapping between virtual registers and hard registers are different across different basic blocks. An observation is that if the mapping is consistent across the entire program, then these overhead can be eliminated. This is of course not always feasible since there might not be enough hard registers to hold all the virtual registers. But still, some virtual registers, are so frequently used, such as those which correspond to the stack pointer, program counter, and target scratch registers, that they deserve to have one fixed hard register allocated whenever possible.

3.4.4 Hybrid Approach

This leads to a hybrid approach in which the hard registers are partitioned into two sets: one is the *fixed* register set, the member of which is assigned to a global virtual register throughout the entire program execution; the other is the *temporary* register set.

This strategy is adopted by our simulator, where a global virtual register is assigned a fixed hard register on a first-come-first-get basis. Those globals that fail to obtain a fixed hard register are mapped to the temporary registers together with the locals according the the lazy allocation mechanism.

Note that our algorithm is of linear complexity. This is in contrast to standard approaches based on liveness analysis and graph coloring, which is (1) an overkill for allocation of locals since their lifetime only last one simulated instruction; (2) unable to handle globals like ours without expensive interprocedural analysis and execution profiling. Also worthy of mention is that although compilers such as gcc provide ways to allow user to map global variables to a machine register (e.g. by declaring `register int sp_sim asm( ''%g4'' )`, these methods are unflexible and unportable.

3.5 Host Implementation

A host implements the interface defined by Section 3.1. The majority of the work is usually devoted to the implementation of every virtual instruction using host machine instructions, while the management of virtual registers can be delegated to the machine independent register allocator discussed in Section 3.4. To use the register allocator, the hard registers as well as how they are partitioned has to be provided.

Worthy of mention is that how the virtual instruction is implemented sometimes has an influence on the number of hard registers that can be made fixed. For example, on the SPARC architecture, if the standard calling convention is followed, the register window will be shifted, which make most registers renamed to physically different registers upon every function call, and hence make them illegible to be partitioned into the fixed set. In our implementation of the SPARC host, the shifting of register window is suppressed thanks to the low level interface defined. Otherwise if a C emitting approach is followed, only g4 through g7 is available on SPARC.

4 Limitations

There are limitations for the static compiled approach in general. Simulators that fall into this category cannot handle self-modifying code, code which load dynamic libraries. Our tool is not immune to these problems. Fortunately, these cases are rare in embedded systems.

There are also limitations specific to our tool. First, our tool works best on high performance host machines with large register sets. When the host has a limited number of registers, the performance will degrade, however, not to the level worse than those without register allocation. Second, the difference on endianess between the target machine and the host machine is ignored. Third, currently the code generation from target machine to virtual machine is directly built on a retargetable compiler, rather than a separate one which accepts assembly or binary as input. While the replacement of additional parsing with direct function call can certainly speed up the compilation, it also ties our tool with a specific compiler. Fortunately, one can build a "binary translation" version of our tool fairly easily.

5 Experiment

We have selected a set of benchmarks to evaluate our simulator. *COUNTER* consists of a loop which simply increments a

Benchmark	hybrid	lazy	greedy	c w/o opt.	c w/t opt.	hybrid traced	c traced
COUNTER	1.0 272	9.1 30	9.1 30	6.0 45	3.6 75	1.27 214	4.7 58
IDCT	1.3 209	5.7 49	11.5 24	8.83 32	3.7 76	1.5 186	5.3 52
VITERBI	1.1 185	3.1 53	6.4 25	4.9 33	1.8 87	1.2 166	4.9 55
FIR	2.4 122	6.2 49	9.4 32	9.4 32	4.0 76	2.9 105	6.6 46
LEVISON_DURBIN	2.5 105	6.3 42	9.2 29	8.0 33	4.1 64	2.9 93	4.86 55

Figure 6. Comparison of simulation performance of various approaches

counter. *IDCT* is the inverse discrete cosine transform algorithm extracted from JPEG/MPEG. *VITERBI* is a popular channel coding algorithms. *FIR* and *LEVISON_DURBIN* are signal processing algorithms extracted from ITU speech coding standard g.723.

We studied the effects of different design decisions during the code generation and summarize the result in Figure 6, where each row corresponds to a benchmark, and each column corresponds one implementation of the code generation interface:

1. the first column corresponds to the hybrid approach discussed in Section 3.4.4;
2. the second column corresponds to the lazy allocation approach discussed in Section 3.4.2;
3. the third column corresponds to the greedy allocation approach discussed in Section 3.4.1;
4. the fourth column corresponds to the C emitting approach, where the executable is generated by gcc without optimization;
5. the fifth column corresponds to the C emitting approach, where the executable is generated by gcc with optimization (with option -O3 turned on);
6. the sixth column corresponds an implementation the same as the first one, except that the total instruction count is traced;
7. the seventh column corresponds an implementation the same as the fifth one, except that the total instruction count is traced.

The result is characterized by two numbers: one is the simulation time, normalized to the native execution time on the host machine; the other is the simulation speed in the unit of MIPS. Note that we choose to measure the simulation performance against native execution on the host machine, rather than the target machine. We believe it offers a better measurement on the performance of the simulator since the performance difference between the host machine and target machine is factored out.

We also measured the compilation speed, and observed an average of 10 times slow down for the approach that generates C than the other approaches. The additional time is mainly spent on gcc compilation and optimization.

6 Conclusion

In conclusion, we have described a technique which uses a virtual machine code generation interface for the static compiled ISA simulation. We argue that such a low level interface is more efficient, in terms of both the compilation speed and simulation performance, than the high level C interface. This interface is also retargetable, as our experience show that porting an ISA simulator to a new host takes only two days.

Our future work will extend this methodology to perform cycle accurate instruction set simulation, and hardware/software cosimulation, which present more challenges.

7 Acknowledgement

The authors would like to thank En-shou Chang for his careful review of the initial manuscript and the anonymous reviewers for providing constructive comments.

References

[1] Kristy Andrews and Duane Sand, *Migrating a CISC Computer Family onto RISC via Object Translation*. Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October, 1992.

[2] John Hennessy and David Patterson. Computer Organization and Design: The Hardware-Software Interface (Appendix A, by James R. Larus), Morgan Kaufman, 1993.

[3] Robert F. Cmelik, and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems May 1994, pp. 128-137.

[4] Emmett Witchel, Mendel Rosenblum. *Embra: Fast and Flexible Machine Simulation*. Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Philadelphia, 1996.

[5] J. Rowson. *Hardware/Software Co-simulation*. Proceedings of the 31st ACM/IEEE Design Automation Conference, 1994.

[6] Vojin Zivojnvic, Steven Tjiang, Heinrich Meyr. *Compiled Simulation of Programmable DSP Architectures*. Proceedings of the 1995 IEEE Workshop on VLSI Signal Processing, Sakai, Japan.

[7] S. Sutarwala, P. Paulin, and Y. Kumar. *Insulin: An Instruction Set Simulation Environment*. Proceedings of CHDL-93, Ottawa, Canada, 1993.

[8] Dawson R. Engler. *VCODE: A Portable, Very Fast Dynamic Code Generation System*. SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96), Philadelphia, PA, May 1996.

[9] C. W. Fraser, D. R. Hanson *A Code Generation Interface for ANSI C*. Software-Practice and Experience 21 (9), 963-988, Sep. 1991.