Generation of Executable Representation for Processor Simulation with Dynamic Translation

JiaJia Song[1], HongWei Hao[1]
[1]School of Information Engineering,
University of Science and Technology Beijing
Beijing 10083, China
jjsong@liama.ia.ac.cn, hhw@ies.ustb.edu.cn

Claude Helmstetter[2], Vania Joloboff[2]
[2]INRIA
LIAMA, Beijing 100080, China
claude@liama.ia.ac.cn,
vania@liama.ia.ac.cn

Abstract

Instruction-Set Simulators (ISS) are indispensable tools for studying new architectures. There are several alternatives to achieve instruction set simulation, such as interpretive simulation, static translation and dynamic translation. This paper presents a simulator where we have developed and integrated three techniques: an interpretive simulator and two variants of dynamic translation. In the third variant, the simulator caches an intermediate representation that consists of pseudo instructions. These pseudo instructions use semantic functions that can be specialized using partial evaluation technique and a code generator. These three methods have been used to run the same simulated programs and compare their performance. The experiments show that the partial evaluation technique increases performance and flexibility, but also shows that it may have adverse effects.

1. Introduction

An instruction-set simulator (ISS) is used to mimic the behavior of a target computer processor on a simulation host machine. The main task of an ISS is to carry out the computations that correspond to each instruction of the simulated program. There are several alternatives to achieve such simulation. In *interpretive simulation*, each instruction of the target program is fetched from memory, decoded, and executed, as shown in Figure 1. This method is flexible and easy to implement, but the simulation speed is slow as it wastes a lot of time in decoding. Interpretive simulation is used in Simplescalar [11].

[0]The work is supported by the National Natural Science Foundation of China #60675006 and the National Key Technology R&D Program during the 11th Five-Year Plan Period #2006BAI03A09

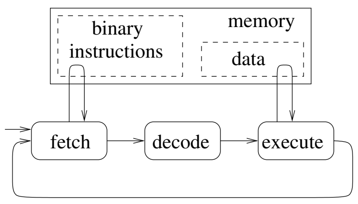

Figure 1. Interpretive simulation

A second technique is compiled simulation (see Figure 2), also called *static translation*. The application program is decoded at compile time and translated into a new program for the simulation host. The simulation speed is vastly improved, but it is not as flexible as interpretive simulation. The application program must be known at compile time, before simulation starts. This method is hence not suitable for application programs which will dynamically modify the code, or dynamically load new code at run-time.

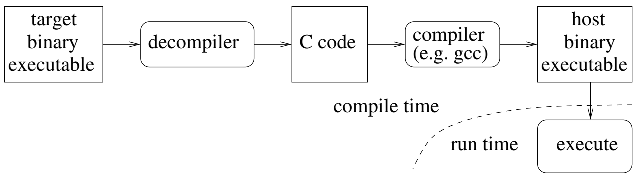

Figure 2. Compiled simulation

A third technique to implement ISS is *dynamic translation* [2, 12, 9]. With dynamic translation (see Figure 3), the target instructions are fetched from memory at runtime, like in interpretive simulation. They are decoded on the first execution and the simulator translates these instructions into another representation which is stored into a cache. On further execution of the same instructions, the translated cached version is used. If the code is modified during runtime, the simulator invalidates the cached representation. Dynamic translation combines the advantage of interpretive simulation and compiled simulation as it supports the simulation of programs that have either dynamic loading or self-modifying code,

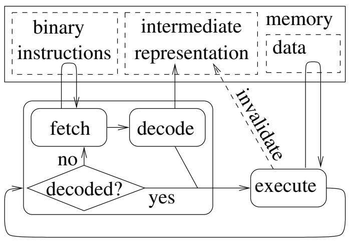

Figure 3. Dynamic translation

This paper presents a method to define an intermediate representation (IR) for a dynamic translation simulator using partial evaluation (specialization). To evaluate its performance, we have also implemented an interpretive simulator and a simpler dynamic translation schema. Benchmarks programs can then be run on the same simulation host with the three techniques and compared. As we have written a code generator to generate specialized functions, we can also experiment with different degrees of specialization.

The remainder of this paper has the following organization: section 2 describes related work in the domain of dynamic translation; our dynamic translation method is presented in section 3. Section 4 describes the experiments result and section 5 concludes the paper.

2. Related work

There is an extensive literature about Instruction Set Simulation. In IS-CS [8] they use an instruction abstraction technique to generate aggressively optimized decoded instruction that further improve simulation performance. The basic idea of JIT-CCS [6] is to store the extracted information in a simulation cache, and this technique is integrated in the retargetable LISA processor design platform.

Different simulators use different kinds of intermediate representation (IR) to improve the performance or to increase modularity and flexibility. GXemul [5] is a framework for full-system computer architecture emulation. The IR in GXemul consists of arrays of pointers to functions, and a few arguments which are passed along to those functions. The functions are mostly implemented in manually hand-coded C. The performance of using this kind of executable IR is obviously lower than what can be achieved by emulators using native code generation, but can be significantly higher than using a naive fetch-decode-execute interpretation loop.

QEMU [1] is a machine emulator. The compile time tool called `dyngen` uses the object file containing the micro operations as input to generate a dynamic code generator. This dynamic code generator is invoked at runtime to generate a complete host function which concatenates several micro operations.

SimIt [7] uses a special simulation technique that translates the target binary code to C code, and the uses GCC to do run-time host binary translation. To translate one page of C code, GCC requires almost one second. SimIt uses multi-processors to improve performance: translation runs on one processor while execution runs in parallel on another processor. Compared to the direct translation approach in QEMU, the GCC-based approach is easier to implement, but at the cost of translation speed.

3. Instruction Set Simulation

In order to compare different techniques, we have implemented three kinds of instruction simulation corresponding to three modes that the simulator can run in, for the ARM architecture.

The first mode, named D0, is interpretive simulation. This is the basis from which we can compare performance. The second mode (D1) is dynamic translation with no specialization. This mode shows the performance improvement obtained with dynamic translation compared to interpretive simulation. The third mode (D2) is dynamic translation with specialized pseudo instructions as described below. This mode shows the performance improvement obtained with specialization over standard dynamic translation as in D1 mode.

Partial evaluation is a compiling optimization technique, also known as specialization [4]. The basic concept of specialization is to transform a generic program P, when operating on some data d into a faster specialized program Pd that executes specifically for this data. Specialization can be advantageously used in processor simulation, because data can often be computed at decoding time, and a specialized version of the generic instruction can be used to execute it. The simulation code then uses fewer tests, fewer memory accesses and more immediate instructions. This technique has been used to some extent in the IC-CS simulator [6] and we have experimented it further.

3.1. ARM architecture overview

There are two instruction sets in ARM [10], ARM32 instruction set and Thumb instruction set. The Thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb instructions are half the size of ARM32 instructions (16 vs 32 bits), resulting in greater code density.

Almost all ARM32 instructions can be conditionally executed, which means that they are only executed if a condition specified in the instruction is satisfied. ARM32 has 15 such condition codes. These conditions are based on the value of the four N, Z, C and V flags in the CPSR. If the flags do not satisfy the condition, the instruction acts as a NOP: that is, execution advances to the next instruction.

3.2. Intermediate representation

In both D1 and D2 mode, the dynamic translation process constructs an intermediate representation (IR) for the original instructions that is suitable for execution on the simulation machine.

The function `fetch_decoded()` is used to fetch and decode the target code. If the instruction has not been decoded, it translates it and stores the decoded form into the intermediate representation cache for further usage.

For the D1 mode IR, we have defined a generic C++ class `ARM_Instruction`. Each type of ARM instruction stores the relevant data obtained from decoding and has an `execute()` method that performs the instruction. The simulator then runs the execute function for each instruction.

In the specialized mode D2, the IR consists of an array of pseudo-instructions. The translator generates one *pseudo instruction* corresponding to each target machine instruction. Each pseudo instruction includes two things. The first is a pointer to a function, called *semantic function*. The second is a structure which contains data used by the semantic function. This data is filled at decoding time.

3.3. Semantic function

The purpose of a semantic function is to perform each individual operation for the target processor. Consider for example the `ADD` instruction, noted in assembly language as: `ADD{<cond>}{S} <Rd> ,<Rn>, <operand>`. Its semantic is described in ARM Architecture Reference Manual [10] as follows:

```
if ConditionPassed (cond) then
 Rd = Rn + operand
  if S == 1 and Rd == R15 then
    CPSR = SPSR
  else if S == 1 then
    N Flag = Rd[31]
    Z Flag = if Rd == 0 then 1 else 0
    C Flag = CarryFrom(Rn + operand)
    V Flag = OverflowFrom(Rn + operand)
```

The `S` bit indicates whether the instruction updates the CPSR or not. There are 11 modes used to calculate the operand in an ARM data-processing instruction, such as immediate, using a register, shifted or non shifted, and so on. In order to maximize performance, we can use many specialized functions that each carry a very specific task, when the task can be discovered at decoding time. In order to test multiple specialization possibilities, we have written a generator which can generate the semantic functions automatically based on input specifications. For example, we can specialize semantic functions on the condition code, the `S` bit, the operand mode and the registers of data-processing instructions. As an example, the generated semantic function below is the `ADD` instruction when the condition code is `EQ`, the `S` bit is 0, and the operand is not shifted.

```
PseudoStatus add_Ceq_S0_Mreg
(Processor &proc, PseudoInstruction &pi)
{ if (!proc.cpsr.z) return OK;
  uint32_t tmp = proc.regs[pi.dpi.Rn];
  uint32_t opv = proc.regs[pi.dpi.Rm];
  uint32_t r = tmp + opv;
  proc.regs[pi.dpi.Rd] = r;
  return OK; }
```

In D1 mode, one function is used to implement the ADD operation. In contrast, in D2 mode, 330 specialized functions are used to implement ADD corresponding to: 15 (condition code) * 2 (`S` bit) * 11 (operand mode).

4. Experiment results

We consider in this paper two benchmark programs that we have written to test the performance of our simulator, named `crypto.c` and `loop.c`. The `loop` program is a simple loop program. The `crypto` program is a more complex cryptographic program using functions from the XYSSL [3] library. The measurements reported have been made on an Intel Core2 processor at 2.13 GHz. We have tested these two programs with several compilation options of the C cross-compiler. The suffix `a0.x` means ARM executable (32 bits) compiled without optimization and `a3.x` means compiled with optimization. The suffix `t0.x` means THUMB executable (16 bits) compiled without optimization and `t3.x` means compiled with optimization.

For this experiment, we used specialization to a limited extent, with no specialization on the registers and no specialization on the condition codes.

The results (Table 1) show that D1 mode is about twice as fast as D0 mode, and D2 is about ten times as fast as D0 mode, showing the value of dynamic translation and specialization over interpretive simulation. Optimized code is faster since there are less instructions and registers are used more aggressively. A same program compiles to more thumb instructions than ARM32 instructions, since ARM32 instruction set is more expressive. Consequently, simulating a program in thumb mode takes more time than in ARM32 mode, whereas the simulation speed (in number of instructions per second) is almost the same.

Then, in other experiments we used further specialization to specialize semantic functions on the condition code.

Table 1. Comparison of interpretive and dynamic translation

	D0	D1	D2
crypto.a0.x arm32, no opt.	522 s 6.62 Mips	221 s 15.6 Mips	57.6 s 59.9 Mips
crypto.a3.x arm32, with opt.	139 s 6.84 Mips	62.2 s 15.3 Mips	11.6 s 82.3 Mips
crypto.t0.x thumb, no opt.	1996 s 5.01 Mips	576 s 17.3 Mips	153 s 65.4 Mips
crypto.t3.x thumb, with opt.	299 s 5.40 Mips	90.6 s 17.8 Mips	26.6 s 60.7 Mips
loop.a0.x arm32, no opt.	18.2 s 7.37 Mips	9.26 s 14.5 Mips	1.57 s 85.4 Mips
loop.t0.x thumb, no opt.	38.1 s 4.84 Mips	12.1 s 15.2 Mips	2.49 s 74.1 Mips

In many ARM application programs, the condition code of most non branching instructions is `AL` (always), so one would think specializing would gain performance.

Table 2. Comparing specialization of the condition code

	specialized cond	non specialized cond
crypto.a0.x arm32, no opt.	58.8 s 58.6 Mips	57.6 s 59.9 Mips
crypto.a3.x arm32, with opt.	12.2 s 78.3 Mips	11.6 s 82.3 Mips
loop.a0.x arm32, no opt.	2.04 s 65.8 Mips	1.57 s 85.4 Mips

However, we found that the less specialized code has better performance than the specialized one. The reason is that when the condition code is specialized, the number of semantic functions is roughly multiplied by 15 (as there are 15 conditions). The code size of the simulator grows significantly. Then there are more instructions cache misses on the host machine during execution of the simulation, and this degrades the performance, even though the code executes faster.

In a similar experiment, we also specialized instructions over use of registers. As there are 16 registers, it also multiplied the number of semantic functions. In that case, we got a performance degradation of about 5 percent.

5. Conclusion

In this paper we experimented methods of dynamic translation for Instruction Set Simulation (ISS), which provides speed and flexibility. Because we have built a parametrized code generator, we have been able to experiment different degrees of specialization for instruction set simulation. These experiments demonstrate two points. The first is that dynamic translation is clearly an order of magnitude faster than interpretive simulation. The second is that specialization of the semantic functions is valuable only to a certain extent. When the semantic functions become highly specialized, their number increases. The simulator code size then increases by an order of magnitude, and each individual function is used less frequently. It induces many more cache misses on the simulation machine, leading to worse simulation performance. Specialization is henceforth valuable to the extent that the specialized code can be held as much as possible in the host simulation cache.

Further work will concentrate on exploring other alternatives, in particular expanding our work to other architectures, recognizing patterns in the target code, so that several target instructions could be translated into a single pseudo-instructions, and generating native code.

References

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05*, Berkeley, CA, USA, 2005. USENIX Association.

[2] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS'94*, pages 128–137, New York, NY, USA, 1994. ACM.

[3] X. Community. Xyssl cryptographic open source code, 2007.

[4] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher Order Symbolic Computation*, 12(4):381–391, 1999.

[5] A. Garave. Gxemul, 2007. `http://www.gavare.se/gxemul/`.

[6] 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 *DAC'02*, pages 22–27, New York, NY, USA, 2002. ACM.

[7] W. Qin, J. D'Errico, and X. Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *CODES+ISSS'06*, pages 193–198, New York, NY, USA, 2006. ACM.

[8] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC'03*, pages 758–763, 2003.

[9] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. *SIGOPS Oper. Syst. Rev.*, 32(5):283–294, 1998.

[10] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] SimpleScalar LLC. SimpleScalar, 2004. `http://www.simplescalar.com/`.

[12] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24(1):68–79, 1996.