Reducing Compilation Time Overhead in Compiled Simulators

Mehrdad Reshadi, Nikil Dutt
Center for Embedded Computer Systems, University of California, Irvine
{reshadi, dutt}@cecs.uci.edu

Abstract

Compiled simulation is a well known technique for improving the performance of instruction set simulators at the cost of compilation time. However the compilation time overhead makes such usage of compiler optimizations impractical especially for large applications. In this paper, we propose a hybrid compiled simulation approach that is simple, generates an optimized decoder and has almost no compilation overhead comparing to static compiled simulation. Using two contemporary processor models--ARM7 and Sparc-- we demonstrated that our technique can reduce the compilation time by 99% on the average, from several thousands of seconds to only tens of seconds.

1. Introduction

Instruction-set simulators are indispensable tools in the development of new architectures. An important quality measure for these tools is simulation performance and it depends on the overhead of simulating target instructions vs. executing them natively. *Interpretive simulation* is the simplest way of doing this process but has poor performance. In interpretive simulation, each instruction is fetched, decoded and executed at run time. *Compiled simulation* reduces the overhead of simulation and improves the performance by removing the *decode* phase of each instruction from the simulation execution loop and doing it once for all instructions. It may also generate optimized code for instructions and hence further improves the simulation speed. The main core of this technique is the translation of the input instructions to an executable binary that can run on the host machine. In *dynamic compiled simulation*, such as [4], instruction translation is repeatedly applied to portions (usually basic blocks) of the input program that are executed and the results are stored for later reuse (Figure 1).

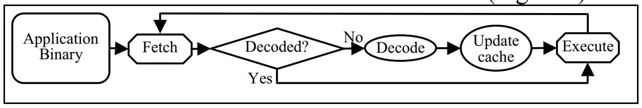

Figure 1- Dynamic compiled simulation flow

In *static compiled simulation*, such as [1],[2] the target program binary is analyzed and compiled into a source code that is functionally equivalent with the input program. This source code is then optimized and compiled into the host binary and executed on the host machine (Figure 2). Since the whole target program is converted into a source code that must be compiled and optimized by a compiler, this technique is only applicable if the compiler can handle the size of the generated source code and can finish the compilation in an acceptable amount of time. On the other hand since the entire input program instructions are decoded irrespective of being executed, the decoded information may consume a lot of memory at run time. The compilation time and memory usage depends on: size of input program; size, structure and complexity of generated source code; the target language and the used features; and level of details in simulation. Since in dynamic compiled simulation the whole program is not compiled in advance, it can handle much larger input programs than static compiled simulation. However, it is very difficult to generate optimized code in dynamic compiled simulation.

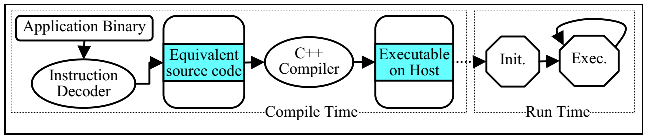

Figure 2- Static compiled simulation flow

The previous efforts in compiled simulation either ignored the compilation overhead and did not address it, or avoided it by generating non-optimized decoded information at run time. Among the works that have used compiled simulation technique for ISA simulation, only [5] has explicitly investigated means of reducing compilation time. In their approach, the output source file is partitioned into smaller functions and the effect of the number of functions on the compilation time is demonstrated. They use assembly code of the input program rather than the executable binary.

In this paper we propose a hybrid compiled simulation that includes a static analysis of the input program followed by a dynamic analysis at run time. In its static part, the input program is analyzed to produce the source code of an optimized decoder for that particular program. In the dynamic part, this decoder analyzes the input program at run time and generates optimized code for the instructions as if they were statically compiled and optimized. This technique significantly reduces the compilation time and memory usage while utilizing compiler optimizations for generating optimized decoded instructions at run time. Using two contemporary processor models--ARM7 and Sparc-- we demonstrated that our technique can reduce the compilation time by 99% on the average, from several thousands of seconds to only tens of seconds. This hybrid approach is a general method that can be applied to any simulation technique.

2. Hybrid Compiled Simulation

We propose a hybrid technique that combines both static and dynamic compile simulation. As Figure 3 shows, in this technique instead of generating a source code that is equivalent to the input program, we generate the source code of a decoder that is customized for that input program. In traditional static compiled simulation, each instruction in the input program has a corresponding code in the generated source code. However, a more careful investigation of the instructions of a typical program shows that the number of instruction *types* is significantly less than the number of instances of instructions. An instruction *type* is any variation of the instruction set of the target architecture. For example in a program, there may be many *Add* instructions in the form of *Add* R_x, R_y, R_z and many others in the form of *Add* R_x, R_y, *#immed*. Therefore instead of repeatedly generating code for instruction *instances*, we can generate customized code for each instruction *type* that exists in the program. Since number of instruction *types* is much less than that of instruction *instances*, the generated source code is smaller and requires considerably less time to compile. This code is then compiled and optimized to generate a decoder that decodes the input program again at run time, and for each instruction *instance*, simply instantiates the corresponding optimized code (instruction *type*). In this way, we use the static compiled simulation approach to utilize the compiler optimizations at compile time and then use the dynamic compiled simulation approach to dynamically decode instructions to their corresponding optimized codes at run time. In the next subsections, we analyze different possible scenarios that this hybrid technique can be used and then will compare them in the result section.

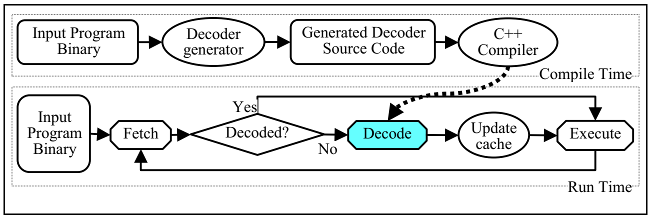

Figure 3- Hybrid compiled simulation flow

2.1 Static decode of one program

This approach is the same as static compiled simulation. As shown in Figure 4, the whole program is decoded at compile time and for each instruction *instance* in that program a customized code is added to the source code. The generated source code is a set of functions that create instruction objects at run time and load them in the instruction memory.

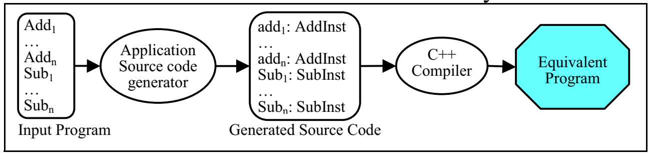

Figure 4- Static decode of one program

For example, if the program contains 1000 similar *Add* instructions, there would be 1000 corresponding codes in the generated source code and 1000 instantiations at run time.

2.2 Dynamic decode of one program.

As shown in Figure 5, in this approach the instructions of the input program are analyzed and the individual instruction *types* are detected. The generated source code is in fact a decoder that contains a customized code for each instruction *type* that exists in the input program. It analyzes the instructions of the program at run time and decodes them by instantiating the optimized code of the corresponding instruction *type*. The size of the generated source code in this case is significantly smaller than the static decode and hence the compilation time is considerably less. For example, if the program contains 1000 similar *Add* instructions, only one customized code is added to the decoder for that *Add* instruction. At run time, each time the decoder detects such an *Add* instruction, this code is instantiated. Therefore there would be one customized code in the generated source code and 1000 instantiated at run time.

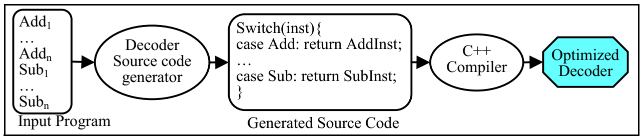

Figure 5- Dynamic decode of one program

2.3 Dynamic decode of multi-programs

It is also possible to analyze a group of input programs and detect their instruction types and then generate one decoder for all of them as shown in Figure 6. Our experiments show that a large number of instruction *types* are common among different programs. Therefore the size of the decoder is only slightly bigger than that of a single program.

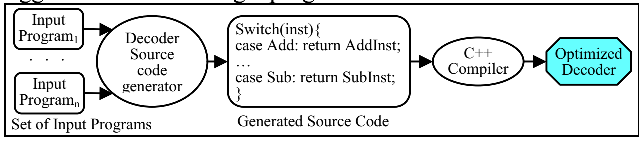

Figure 6- Dynamic decode of multiple program

The major benefit of this approach is that it requires one compilation for all of the programs while in the previous approaches, for each input program, the generated source code must be compiled.

2.4 Dynamic decode for all ISA

The instructions of a program are a subset of all the variations of the instructions in the instruction-set (ISA). Therefore, instead of analyzing an input program and generating the decoder for that particular program, it is better to generate all possible variations of instructions in the instruction set and have a decoder that can decode any input program on a specific architecture. However, this approach is only applicable if the number of these variations is not very large or if the simulator is used for a fixed architecture and not in a design exploration loop.

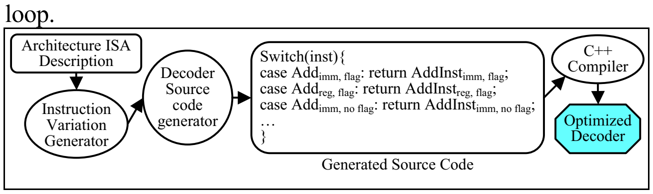
Architecture ISA Description
Instruction Variation Generator
Decoder Source code generator
Switch(inst){
case Add$_{imm, flag}$: return AddInst$_{imm, flag}$;
case Add$_{reg, flag}$: return AddInst$_{reg, flag}$;
case Add$_{imm, no flag}$: return AddInst$_{imm, no flag}$;
…
}
Generated Source Code
C++ Compiler
Optimized Decoder

Figure 7- Dynamic decode of all ISA

For example, the Sparc processor has a simple instruction-set and the number of variations of the instructions is less than 1000. On the other hand, the ARM processor has a very complex instruction set and the number of variations of instructions is in the range of several hundred thousand (~500k) instruction types. Thus, using this approach for ARM processor not only has a long compilation time, but also consumes a lot of memory for decoder and hence is not practical at all.

3. Memory usage

Despite of generating optimized code, the decoder in our hybrid approach is similar to the decoder in any other simulation technique and can replace it. Therefore, its memory usage and the decoded information can be handled similarly by well known

1063-6404/03 $ 17.00 © 2003 IEEE

techniques and data structures such as a software cache. On the other hand, while the decoder generated in our approach can generate optimized code, it does not need to implement any optimization algorithm and simply uses the pre-optimized codes that the compiler has generated. In this way, if the number of instruction types in the program is not very large, the size of our decoder is comparable to (or even less than) the size of a traditional dynamic compiled simulation decoder that performs some optimizations during decode at run time. Our experiments show that usually the number of instruction types is very low even when multiple programs are processed to generate a single decoder for all of them.

4. Results

We conducted our case studies with two contemporary processor models: ARM7 and Sparc. We used the Instruction-Set Compiled Simulation (IS-CS) technique [3] to implement the optimized decoder in our simulator. In this section we show the results using four application programs: adpcm, jpeg, 099.go and 129.compress. The results were obtained on a 2.4 GHz Pentium 4 with 512 MB of RAM. In all experiments, each source file contained up to 100 functions and each function contained up to 100 instruction decoding.

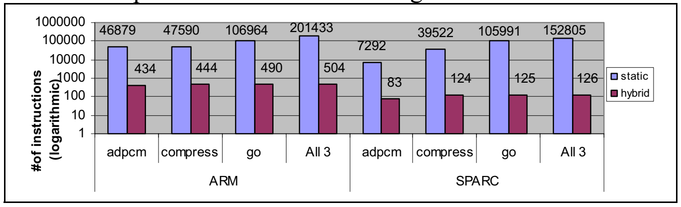

Figure 8- Source file size in different techniques

Figure 8 shows the number of instructions that are generated in the output source file in each technique for both processor models. For each benchmark, the first bar shows the total number of instruction instances in the input program binary (and hence the output of static compiled simulation) and the second bar shows the number of distinct instruction types that exists in that benchmark (and hence the output of hybrid compiled simulation). The last pair of bars shows these numbers for all 3 benchmarks together. Interestingly, compared to the number of instruction *instances*, the number of instruction *types* change slightly between benchmarks and have a lot of commonality.

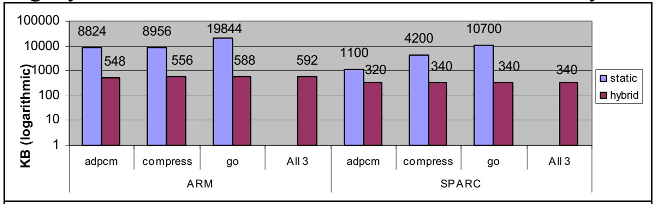

Figure 9- Executable file size in different techniques

Similarly, Figure 9 shows the size of the executable binary file after compilation. Note that in the static compiled simulation, all of the instructions are decoded even if they are not executed at all. In our experience, we got very similar performance results from both static and our hybrid compiled simulation. However, se believe that in the hybrid approach, the instructions must be decoded again at run time, but the smaller executable size improves the cache behaviour of the hybrid simulator compared to that of the static compiled simulator and therefore compensates the extra run time decoding overhead.

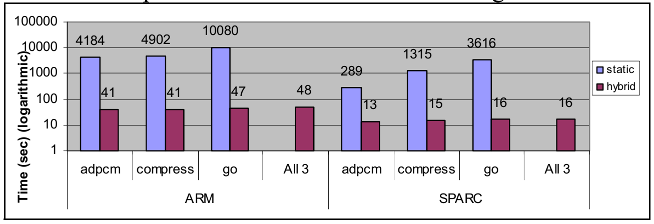

Figure 10- Compilation time in different techniques

Figure 10 shows the comparison of the compilation time of hybrid and compiled simulation. In our experiments, the average compilation time was about 4100 seconds for static compiled simulation and about 30 seconds for our hybrid compiled simulation. This shows an almost 99% reduction in average compilation time, while still benefiting from all the advantages of static compiled simulation. The hybrid complied simulationis described in more detail in [6].

5. Summary

In this paper we proposed a hybrid compiled simulation technique that utilizes the advantages of both static and dynamic compiled simulation and avoids their disadvantages. In this approach, the input program is first analyzed and an optimized decoder is generated for that program using a conventional (C, C++, etc.) compiler. The decoder is then used by the simulation engine to decode the simulated instructions to optimized decoded information at run time. While the technique is applicable to any execution model, we used our Instruction-Set Compiled Simulation (IS-CS) technique to show the advantages of the hybrid compiled simulation technique. The results showed a 99% reduction in compilation time without any performance loss.

6. Acknowledgments

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712.

7. References

[1] G. Braun et al. Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description. ISSS, 2001.

[2] J. Zhu et al. A Retargetable, Ultra-fast Instruction Set Simulator. DATE, 1999.

[3] M. Reshadi et al, Instruction-Set Compiled Simulation: A technique for fast and flexible instruction set simulation, DAC, 2003.

[4] R.F. Cmelik, et al. Shade: A fast instruction set simulator for execution profiling. Proceedings of 1994 ACM SIGMETTRICS Conference on Measurment and Modeling of computer systems, Philadelphia, 1994.

[5] R. Amicel et al. Mastering startup costs in assembler-based compiled instruction-set simulation. Proceedings of Workshop on interaction between Compilers and Computer Architectures (INTERACT'02), 2002.

[6] M. Reshadi, N. Dutt. Hybrid Compiled Simulation. ICS Technical Report#03-23, University of California, Irvine, 2003.