Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation

MEHRDAD RESHADI
University of California Irvine
PRABHAT MISHRA
University of Florida
and
NIKIL DUTT
University of California Irvine

20

Instruction-set simulators are critical tools for the exploration and validation of new processor architectures. Due to the increasing complexity of architectures and time-to-market pressure, performance is the most important feature of an instruction-set simulator. Interpretive simulators are flexible but slow, whereas compiled simulators deliver speed at the cost of flexibility and compilation overhead. This article presents a hybrid instruction-set-compiled simulation (HISCS) technique for generation of fast instruction-set simulators that combines the benefit of both compiled and interpretive simulation. This article makes two important contributions: (i) it improves the interpretive simulation performance by applying compiled simulation at the instruction level using a novel template-customization technique to generate optimized decoded instructions during compile time; and (ii) it reduces the compile-time overhead by combining the benefits of both static and dynamic-compiled simulation. Our experimental results using two contemporary processors (ARM7 and SPARC) demonstrate an order-of-magnitude reduction in compilation time as well as a 70% performance improvement, on average, over the best-known published result in instruction-set simulation.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems

General Terms: Design, Performance

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712.
Authors' addresses: M. Reshadi, Center for Embedded Computer Systems, University of California Irvine, CA 92697; email: reshadi@cecs.uci.edu; P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, FL 32611; email: prabhat@cise.ufl.edu; N. Dutt, Center for Embedded Computer Systems, University of California Irvine; email: dutt@cecs.uci.edu.

DOI 10.1145/1509288.1509292 http://doi.acm.org/10.1145/1509288.1509292

Additional Key Words and Phrases: Compiled simulation, interpretive simulation, instruction set architecture, partial evaluation

ACM Reference Format:

Reshadi, M., Mishra, R., and Dutt, N. 2009. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. ACM Trans. Embedd. Comput. Syst. 8, 3, Article 20 (April 2009), 27 pages. DOI = 10.1145/1509288.1509292 http://doi.acm.org/10.1145/1509288.1509292

1. INTRODUCTION

An instruction-set simulator (ISS) is a tool that runs on a host machine to mimic the behavior of an application program running on a target machine. ISSs are indispensable tools in the development of new processor architectures. They are used to validate an architecture design, a compiler design, as well as to evaluate architectural design decisions during design space exploration. These simulators should be fast to handle the increasing complexity of processors; flexible to handle features of applications and processors such as runtime self-modifying codes and multi-mode processors; and retargetable to support a wide spectrum of architectures.

Traditional interpretive simulation is flexible but slow. In this technique, an instruction is fetched, decoded, and executed at runtime, as shown in Figure 1. Instruction decoding is a time-consuming process in a software simulation. It also affects the performance of the execute stage in the simulation loop.

Compiled simulation performs compile time decoding of application program to improve the simulation performance, as described in Section 3. However, all compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is more runtime static. Due to this assumption, many application domains are excluded from the utilization of compiled simulators. Similarly, compiled simulators are not applicable in embedded systems that use processors having multiple instruction sets. These processors can switch to a different instruction-set mode at runtime. For instance, the ARM processor [ARM7] uses the *Thumb* (reduced bit-width) instruction-set to reduce power and memory consumption. This dynamic switching of instruction-set modes cannot be considered by a simulation compiler, since the selection depends on runtime values and is not predictable. Similarly, applications with runtime dynamic program code, as provided by operating systems, cannot be addressed by compiled, simulators. Furthermore, compiled simulators rely on a compiler to compile and optimize the decoded program. As a result, it introduces other limiting factors such as compilation time overhead and the size of the input application program that the compiler can handle.

Due to the restrictiveness of the compiled technique, interpretive simulators are typically used in embedded systems design flow. This article presents a novel technique for generation of fast ISSs that combines the performance of traditional-compiled simulation with the flexibility of interpretive simulation. Our instruction-set compiled simulation (ISCS) technique achieves high performance by optimizing the *Decode* and *Execute* stages of interpretive simulation (Figure 1). The time-consuming decode stage is moved to compile

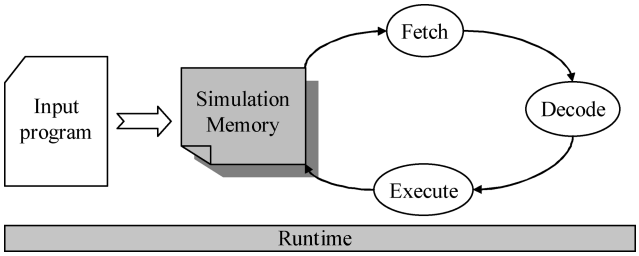

Fig. 1. Traditional interpretive simulation flow.

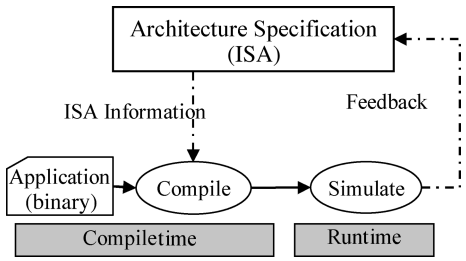

Fig. 2. Design space exploration.

time while maintaining the flexibility of interpretive simulation by applying decode and optimization at the instruction-level granularity. We use a *template-customization* technique to generate aggressively optimized decoded instructions to improve the performance of the execute stage. Our experimental results, using two contemporary processor models (ARM7 and SPARC), demonstrate 70% performance improvement on average over the best-known published result in this category.

Although the ISCS technique improves simulation performance by moving the decode step to compile time and performing various compile time optimizations, it introduces compilation challenges in terms of both compilation time and memory usage. These compilation challenges are also present in traditional-compiled simulation. Since the whole target program is converted into a source code that must be compiled and optimized by a compiler, compiled simulation 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. In general, compilation is done once and the simulation is performed multiple times. As a result, the longer compilation time is amortized over multiple faster simulation runs. However, a long compilation time is not acceptable, especially in the context of early design space exploration.

During design space exploration an architect, tries to figure out the best-possible instruction-set architecture (ISA) for the given set of application programs under various design constraints such as code size and performance. As shown in Figure 2, an application program is compiled using the current instruction set, and the simulation result (feedback) is used to modify the

ISA. Both compilation and simulation time are equally important, since they contribute to the overall evaluation time. Furthermore, since all of the instructions in the entire input program are decoded irrespective of whether all of them will be executed or not, the decoded information may consume a lot of memory at runtime.

To address the compile time overhead in ISCS, we propose a hybrid compilation technique that includes a static analysis of the input program during compile time followed by a dynamic analysis at runtime. In the static part, the input program is analyzed to produce the source code of an optimized decoder for that particular program. In the dynamic part, the decoder analyzes the input program at runtime 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 runtime. Using two contemporary processor models (ARM7 and SPARC), we demonstrate that our technique can drastically reduce the compilation time—from thousands of seconds to tens of seconds.

This article makes two primary contributions in design of fast and flexible instruction-set simulation. First, we developed an ISCS technique that combines the benefit of both interpretive and compiled simulation. Second, we present a hybrid compilation technique to drastically reduce the compilation overhead. The rest of the article is organized as follows. Section 2 presents related work addressing ISA simulation techniques. Section 3 compares the static and dynamic-compiled simulation approaches. The ISCS technique is presented in Section 4. Section 5 presents our hybrid compilation technique. Section 6 presents simulation results using two contemporary processor models: ARM7 and SPARC. Finally, Section 7 concludes the article.

2. RELATED WORK

An extensive body of recent work has addressed ISA simulation. The wide spectrum of today's instruction-set simulation techniques includes the most flexible but slow interpretive simulation and faster compiled simulation. Recent research addresses retargetability of ISSs using a machine description language.

Simplescalar [Simplescalar] is a widely used interpretive simulator that does not have any performance optimizations for functional simulation. Shade [Cmelik 1994], Embra [Witchel 1996], and FastSim [Schnarr 1998] simulators use dynamic binary translation and result caching to improve simulation performance. A fast and retargetable simulation technique is presented by Zhu and Gajski [1999] that improves traditional static-compiled simulation by 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 that use C as a code-generation interface.

Architecture description languages (ADLs) have been successfully used for specifying processor architectures and generating efficient software toolkit

including compiler, simulator, assembler, and debugger. The focus of early ADLs [Ramsey 1997; Onder 1998] was to specify the instruction-sets efficiently and enable assembler/simulator generation. Retargetable fast simulators based on an ADL have been proposed within the framework of FACILE [Schnarr 2001], Sim-nML [Hartoog 1997], ISDL [Hadjiyiannis 1997], MIMOLA [Leupers 1999], ANSI C [Engel 1999], LISA [Pees 2000; Braun 2001], and EXPRESSION [Halambi 1999]. The simulator generated from a FACILE description utilizes the *Fast Forwarding* technique to achieve better performance. All of these simulation approaches assume that the program code is runtime static. In summary, none of the previous approaches achieve the flexibility and high-simulation performance at the same time.

Various simulation platforms use efficient interpreters such as threaded-code interpreters [Bell 1973] and variations of runtime code generation for performance improvement. For example, SimICS [Magnusson 2002] uses the SimGen specification language to encode various aspects of the instruction-set and apply partial evaluations to improve simulation performance. The binary decoder is a performance bottleneck in ISSs. Qin and Malik [2003] generates efficient decoder from an instruction pattern specification using a decision tree and cost models. The SyntSim simulator generator [BurtScher 2004] improves simulation performance by a judicious combination of compiled and interpreted simulation modes as well as several optimizations, including unwanted code elimination; NOP removal; label minimization (preservation of basic blocks); inlining; PC update minimization; and the hard coding of constants, register specifiers, and displacement/immediate values. SyntSim selects the instructions to be compiled based on either profile data (if available) or built-in heuristics to guess the possible instructions to both improve simulation performance and reduce compilation overhead. However, these methods of instruction selection for compiled mode are not always applicable. Furthermore, due to compiled simulation, the full flexibility of interpretive simulation is lost. Our technique does not require such selections and still achieves the execution speed of compiled simulation and full flexibility of interpretive simulation.

Braun et al. [2001] have proposed a static scheduling technique based on the LISA machine description language. A just-in-time cache-compiled simulation (JIT-CCS) technique is presented by Nohl et al. [2002]. The objective of the JIT-CCS technique is similar to the one presented in this article—combining the full flexibility of interpretive simulation with the speed of the compiled technique. The JIT-CCS technique integrates the simulation compiler into their simulator. The compilation of an instruction takes place at runtime, *just-in-time* before the instruction is going to be executed. Subsequently, the extracted information is stored in a simulation cache for direct reuse in a repeated execution of the program address. The simulator recognizes if the program code of a previously executed address has changed and initiates a recompilation. The JIT-CCS technique improves performance of interpretive simulation by reducing the decoding overhead in ISS via a software cache that stores the decoded information. It translates the input instructions to general structures without further optimizations. In addition to caching the decoded information, our technique optimizes the execution of instructions in the

simulator and improves the simulation speed by 70%, on average, over the JIT-CCS technique.

The previous efforts in compiled simulation either ignored the compilation overhead, or avoided it by generating nonoptimized decoded information at runtime. Amicel and Bodin [2002] have 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. The SyntSim simulator generator [BurtScher 2004] addresses the compilation overhead issue by selecting 15% to 37% of the static instructions to be compiled such that 99.9% of the dynamic instructions run in compiled mode to amortize the synthesis and compilation time. There is no direct effort to reduce the compilation overhead. Our technique can reduce the compilation time by an order of magnitude, as demonstrated in Section 6.2.

3. COMPILED SIMULATION ISSUES

Compared to interpretive simulation, compiled simulation has two major disadvantages: (i) it is more complicated especially in retargetable simulators, and (ii) it has an extra time and memory overhead depending on the level of optimizations. In static-compiled simulation, the optimizations are applied to the whole input program allowing for the generation of highly optimized code; however, this is at the expense of potentially large compilation time. In dynamic-compiled simulation, the optimizations *may* be applied to a single instruction (or a basic block) at runtime. Therefore, the compilation time overhead is omitted, but the generated code may not be as optimized as possible. In general, a compiled simulation technique is preferred over an interpretive technique if the following equation holds true:

$$\mathrm{D_c} + \mathrm{C_c} + \mathrm{E_c} \leq \mathrm{D_i} + \mathrm{E_i} \tag{1}$$

Here, D_c, E_c, D_i, and E_i are decode and execution times in compiled and interpretive simulations, respectively. C_c is the compilation time overhead in static-compiled simulation and software cache overhead in dynamic-compiled simulation. The goal in compiled simulation is to make E_c significantly less than E_i. This leads to a larger D_c or C_c. In contrast, our technique focuses on reducing all of these values (D_c, C_c, and E_c) simultaneously. In the remainder of this section, we analyze the static and dynamic-compiled simulation techniques in detail.

3.1 Static-Compiled Simulation

In static-compiled simulation, the whole target program is decoded into a source file that is functionally equivalent with the input program. This input is then compiled to generate an executable binary on the host machine. When executed on the host, this program initializes the simulated environment and executes the input program in that environment (Figure 3).

This technique works particularly well when a program is simulated many times. In Equation (1), the decode time (D_c) depends on the number of

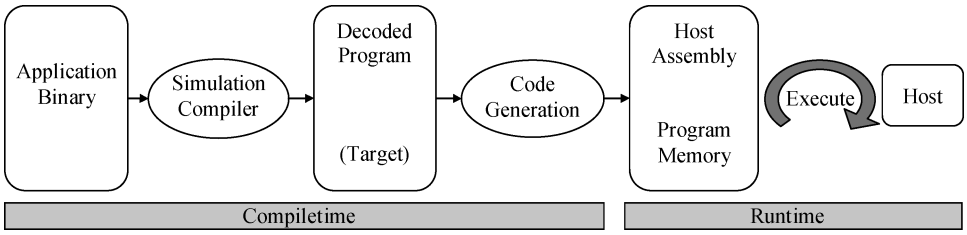

Fig. 3. Traditional compiled simulation flow.

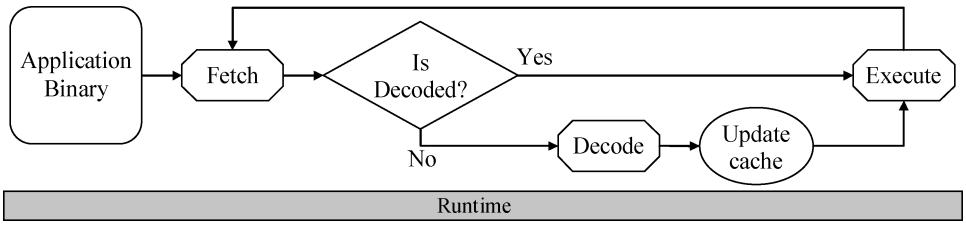

Fig. 4. Dynamic compiled simulation flow.

instructions in the input program (and not the number of executed instructions). Compile time (C_c) depends on the following aspects:

—Input program: Its size and the architectural features that it uses.

—Generated source code: The size, structure, and the complexity of the generated source code play a major role. Examples include number of files and functions, the number of defined variables, and the access mechanism, use of inlining and level of optimizations that are applied to the generated code.

—The target language and the used features. For example, use of C language can lead to a faster compilation compared to use of C++ since templates or macros in C++, can increase the amount of work at compile time.

—Level of detail in simulation: Detailed information collection during simulation requires more code to be instrumented between simulated instructions, leading to an increase in the size of the source code.

Similarly, the amount of memory consumption depends on the size of input program. In general, static-compiled simulation is the best choice when a target program must be simulated many times so that only one compilation is necessary.

3.2 Dynamic-Compiled Simulation

Dynamic-compiled simulation, such as Shade [Cmelik 1994], eliminates the compilation time overhead by performing a more complex decode at runtime. As Figure 4 shows, in dynamic-compiled simulation, instead of decoding all the input instructions, only those that get executed are decoded and the results are stored (cached) for later use. Therefore, in Equation (1), decode time (D_c) depends on the number of executed instructions. In this equation, C_c is the

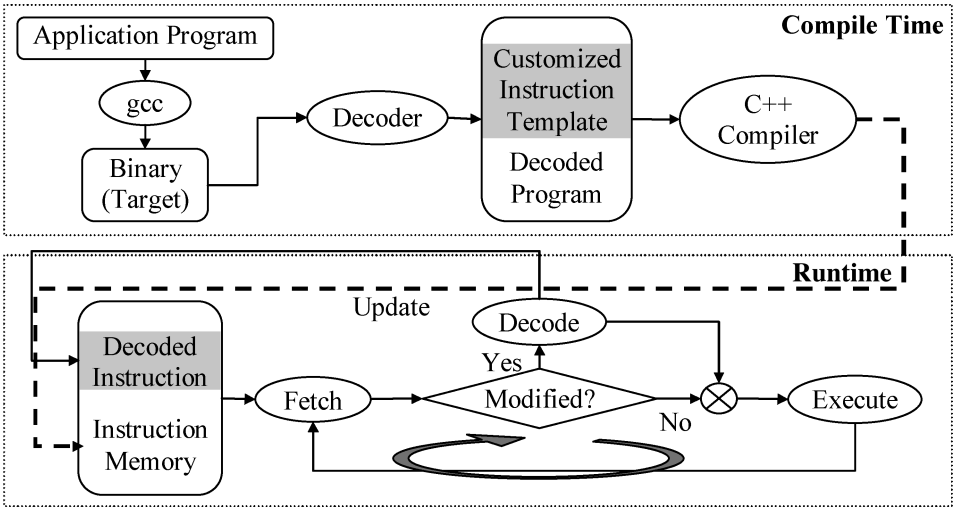

Fig. 5. Instruction-set-compiled simulation flow.

overhead of accessing and updating the storage (cache) of decoded information. In Figure 4, the Fetch, Decode, and Execute phases are usually applied to a block of consecutive instructions (basic blocks).

Since in dynamic-compiled simulation the whole program is not compiled in advance, it can handle much larger input programs than static-compiled simulation. Compared to interpretive simulation, the decode phase of dynamic-compiled simulation is more complex, and this technique works best when instruction are re-executed a lot.

4. INSTRUCTION-SET-COMPILED SIMULATION

We have developed an ISCS technique with the intention of combining the full flexibility of interpretive simulation with the speed of the compiled simulation. The basic idea is to move the time-consuming instruction decoding to compile time and perform aggressive customizations, as shown in Figure 5. The application program, written in C/C++, is compiled using a compiler, such as *gcc*, configured to generate binary for the target machine. The instruction decoder decodes one binary instruction at a time to generate the decoded program for the input application. The decoded program is then compiled by a C++ compiler and linked with the simulation library to generate the simulator. The simulator recognizes if the previously decoded instruction has changed and initiates redecoding of the modified instruction. If any instruction is modified during execution and subsequently redecoded, the corresponding location in instruction memory is updated with the re-decoded instruction.

In traditional interpretive simulation (e.g., Simplescalar [Simplescalar]) the decoding and execution of binary instructions are done using a single monolithic function. This function has many if-then-else and switch-case statements that perform certain activities based on bit patterns of opcode, operands, addressing modes, and so on. In advanced interpretive simulation (e.g., LISA [Nohl 2002]),

the binary instruction is decoded and the decoded instruction contains pointers to specific functions. This information is stored and reused every time the instruction is executed. There are many variations of these two methods based on efficiency of decode, complexity of implementation, and performance of execution. However, none of these techniques exploit the fact that a certain class of instructions may have a constant value for a particular field of the instruction that can be exploited for further optimizations. For example, a majority of ARM instructions execute unconditionally, and hence, it is a waste of time to check the *condition* field for such instructions every time they are executed. In a simplified situation, the condition field can be replaced by the pointer of one of two possible functions: One function always returns true and the other checks the actual condition depending on the value of condition filed in the instruction binary. The execution can be further optimized by using one function for every possible value of the condition (e.g., 16 functions for a 4-bit condition field). However, Section 4.1 demonstrates that there are too many possible scenarios and it is not always practical to write all possible functions and perform compile time function selection.

Our ISCS technique enables generation of customized functions during compile time for each instruction binary by using C++ templates. The basic idea is to define C++ templates for each instruction class in the ISA and use a C++ compiler to customize the template for each specific instruction binary during compile time. The following procedure shows the major steps for developing a simulator based on our ISCS technique.

Procedure to develop ISCS – Instruction-Set Compiled Simulation

Inputs: 1. Application Program Binary (*Application*)
2. Instruction-Set Description (*ISA*)
Output: Optimized ISCS Simulator.
Begin
1- Identify the instruction classes in *ISA*
2- Develop C++ templates for instruction classes.
3- Decode *Application* by generating a customized template for each instruction
4- Compile customized templates
5- Perform interpretive simulation using optimized and decoded instructions
End

The first step in ISCS is to identify all the instruction classes in the instruction set of the architecture. Typically, this information is readily available from the ISA manual. For example, the ARM processor has six instruction classes: data processing, branch, loadStore, multiply, multiple load-store, software interrupt, and swap. The second step is to develop C++ templates for each instruction class. The first two steps in the algorithm need to be performed manually by the simulator developer once for an ISA. The manual development effort for the first step is negligible (in the order of hours) in case the designer is knowledgeable about the ISA. In case the designer is not aware of the instruction set, then the first step may require several days to identify the instruction classes

from the architecture manual. The second step will require a week, since the number of instruction classes is typically small (5–10). Therefore, the manual development effort will be in the order of days. The amount of manual effort will vary during design space exploration depending on the required modification such as how many new instruction classes are created. On average, each iteration during exploration may add a new class and, therefore, the manual effort will be several hours (1 day) per iteration. It is important to note that the simulation efficiency is not affected by changing the number of classes, since each optimized function will have smallest number of computations irrespective of the originating template size. However, it may effect the compilation time, as discussed in Section 5.

In the third step, each instruction binary in the application program is decoded to generate its corresponding template parameter values, that is, the customized template is generated. In the next step, the customized templates are compiled and optimized to generate optimized decoded instructions. Steps 1 and 2 are performed only once for the ISA, but steps 3, 4, and 5 must be repeated once for every application. Steps 3 and 4 correspond to the "Compile Time" section of Figure 5 while Step 5 corresponds to the "Runtime" section of that figure. Steps 4 and 5 determine the overall performance of the simulation, and our goal is to speed up both of them. The compile time optimizations in Step 4 improve the speed of the simulation in Step 5. Section 4.1 describes the template customization procedure using an illustrative example followed by the compile time decode and customization algorithms in Section 4.2. Finally, the decoded instructions are used for fast and flexible simulation. Section 4.3 describes the simulation engine that offers the full flexibility of interpretive simulation.

4.1 An Illustrative Example

Consider the data-processing instructions of the ARM processor described in Example 1. Figure 6 shows an implementation for instructions described in Example 1 using function pointers. The actual values of pointers to functions are determined in a big switch-case statement in the decoder and stored in the *DPInst* structure. For example, if an instruction is executed unconditionally, the *condition* variable will point to *Always()* function. To execute an instruction, its corresponding data is passed to the *execute_dp()* function, which implements the behavior of the data-processing instructions of ARM.

Example 1: Data-Processing instructions in ARM processor.

An ARM data-processing instruction is a 32-bit instruction in the following format:
opcode {*condition*}{*S*} *Rd*, *Rn*, *ShifterOperand*
Where:
opcode: There are 16 operations such as *Add, Sub, ...*
condition: There are 16 conditions such as CarryClear, Always, Never, …
S: Indicates whether the flag bits (Carry/Zero/Negative/Overflow) should be updated (true) or not (false).
Rd and *Rn*: Destination and Source registers, respectively.
ShifterOperand: Second source operand and may have any of the followings formats:
#*immed*: An integer constant.
Rm shift #<*immed*>: A register shifted by a constant value.
Rm shift *Rs*: A register shifted by the value of another register.
There are four shift operations: LogicalShiftLeft, ArithmeticShiftLeft, ShiftRight, and RotateRight.
The combinations of above parameters create 9 different possibilities. LogicShiftLeft and RotateRight with constant 0 are considered special cases. Therefore, there are 11 different possible ShifterOperand formats.

```
struct DPInst {
  bool (*condition) (void);          //pointer to a function such as Always() that returns a Boolean.
  int   (*operation)(int, int);      //pointer to a function that gets 2 integer inputs and returns an integer.
  int   dest, src1;                  //index of destination and source registers
  int   (*shifterOperand)(int);      //pointer to a function that gets the opcode and returns an integer.
  int   opcode;  //opcode of the corresponding instruction
  bool updateFlags                   //if true, the flag bits must be updated
};

void execute_dp(DPInst inst)         //A general function that simulates the behavior of DataProcessing instructions
{
  if ( inst.condition() )
  {
    REGS[inst.dest] = inst.operation( REGS[inst.src1] , inst.shifterOperand( inst.opcode ) );
    if ( inst.updateFlags )
      Update_the_flag_bits(…);
  }
}

//possible condition functions
bool  Always()     { return true; }
bool  Never()      { return false; }
…
```

Fig. 6. A sample implementation for DataProcessing instructions in ARM.

All ARM instructions are predicated, that is, they are executed only if some condition is true. A majority of the ARM instructions execute unconditionally (*condition* field has value *Always*) and hence it is a waste of time to check the condition for such instructions every time they are executed. Generally, when certain input values are known for a class of instructions, the partial evaluation technique [Futamura 1971] can be applied to optimize the execution. The partial evaluation technique specializes a program with part of its input to get a faster version of the same program. For example, for unconditional ARM instructions, a function that does not check the conditions will run faster. Furthermore, it is possible to have separate functions for different operations and different *ShifterOperand* addressing modes. Similarly, there can be two implementations for instructions that update the flag bits and those that do not update the flag bits. Examples of such optimizations are shown in Figure 7. The decoder at runtime assigns an instruction to the fastest function that simulates the instruction's behavior.

To take advantage of such customizations, we need to have separate functions for each and every possible format of instructions so that the functions can be optimized by the compiler at compile time and produce the best performance at run time. Unfortunately, this is not an easy task. For example, the ARM data-processing instructions have $16 \times 16 \times 2 \times 11 = 5{,}632$ possible formats.[1] Generating all of the customized functions for the whole instruction-set requires complex algorithms and may also impose a huge load on the compiler.

[1]They have 16 conditions, 16 operations, an update flag (true/false), and one of the source operands is called ShifterOperand with 11 types.

```
//A customized function for unconditional ARM data processing instructions
void execute_dp_unconditional(DPInst inst)
{
    REGS[inst.dest] = inst.operation( REGS[inst.src1] , inst.shifterOperand( inst.opcode ) );
    if ( inst.updateFlags )
      Update_the_flag_bits(…);
}

//A customized function for unconditional ARM data processing instructions that do not update the flags
void execute_dp_unconditional_noupdate(DPInst i)
{
    REGS[inst.dest] = inst.operation( REGS[inst.src1] , inst.shifterOperand( inst.opcode ) );
}

//A customized function for unconditional ARM data processing instructions that update the flags
void execute_dp_unconditional_update(DPInst i)
{
    REGS[inst.dest] = inst.operation( REGS[inst.src1] , inst.shifterOperand( inst.opcode ) );
    Update_the_flag_bits(…);
}

//A customized function for unconditional ADD instructions in ARM that do not update the flags
void execute_dp_unconditional_noupdate_add(DPInst i)
{
    REGS[inst.dest] = REGS[inst.src1] + inst.shifterOperand( inst.opcode );
}
```

Fig. 7. Customized implementations for DataProcessing instructions in ARM.

To solve this problem, we define instruction classes, where each class contains instructions with similar formats. Most of the time, this information is readily available from the ISA manual. For example, we defined only one instruction class for all data-processing instructions of ARM. Totally, we defined six instruction classes for the whole ARM instruction set such as *DataProcessing*, *Branch*, *LoadStore*, *Multiply*, *Multiple Load-Store*, *Software Interrupt*, and *Swap*. Each instruction class is expanded by the compiler and customized for instances of the instructions in the input program. To decode and generate code for instructions of a program, we define a set of masks for each instruction class. The mask consists of "0," "1" and "x" symbols. A "0" ("1") symbol in the mask matches with a "0" ("1") in the binary pattern of an instruction at the same bit position. An "x" symbol matches with both "0" and "1." Example 2 shows the masks for the data-processing instructions in ARM.

Example 2. Bit masks defined for the DataProcessing instruction class in ARM.

```
"xxxx-001x xxxx-xxxx xxxx-xxxx xxxx-xxxx"
"xxxx-000x xxxx-xxxx xxxx-xxxx xxx0-xxxx"
"xxxx-000x xxxx-xxxx xxxx-xxxx 0xx1-xxxx"
```

We use C++ *templates* to implement the functionality of each class of instructions. These templates are in fact parameterizable data structures. The values of the template parameters are specified at compile-time, and hence the compiler can use them for optimizations. If any of these parameters has a

```
template <class Cond, class Op, bool UpdateFlag, class SftOper>
class DataProcessing
{
  SftOper _sftOperand;
  Reg _dest, _src1;
public:
  virtual void execute()
  {
    if (Cond::execute())
    {
      _dest = Op::execute(_src1, _sftOperand.getValue());
      if ( UpdateFlag )
      {
        Update_the_flag_bits(…);
      }
    }
  }
};
```

Fig. 8. A C++ template for DataProcessing instructions in ARM.

static value that does not change at run time, the compiler can safely propagate the corresponding value and remove unnecessary operations. Another benefit is that since we use functions of templates rather than function pointers, the compiler can inline the body of smaller functions that are called for simulating one instruction (see Figure 6) and then optimize the whole code all together. In other words, we not only avoid the over head function calls through pointer, but also create more optimization opportunity for the compiler. Figure 8 shows the C++ template for the ARM data-processing instructions. The template has four parameters such as, *Condition*, *operation*, *updateFlag*, and *shifterOperand*. The shifter operand itself is a template having three parameters such as, *operand type*, *shift options* and *shift value*. In this example, if the *condition* is *Always*, the compiler knows that the condition is always true and removes the *if* statement from the *execute()* function. A more detailed example is shown in Section 4.2. We also use a Mask Table for the mapping between mask patterns and templates. It also maintains a mapping between the mask patterns and the values of the corresponding template parameters. This template customization technique is used to generate aggressively optimized decoded instructions, as described in Section 4.2.

4.2 Instruction Decoder

Algorithm 1 shows how instructions are decoded to generate the final decoded program. In this algorithm, each binary instruction is decoded one at a time. To decode each instruction, first the proper template is determined and then the selected template is customized for the corresponding instruction.

Algorithm 1- Instruction decoding

```
Inputs: Application Program Application (Binary), MaskTable maskTable.
Output: Decoded Program DecodedProgram.
Begin
  TempProgram = {}
  foreach binary instruction inst with address addr in Application
    template = DetermineTemplate(inst, maskTable)
    template_inst = CustomizeTemplate(template, inst)
    newStr = "InstMemory[addr] = new template_inst"
    TempProgram = AppendInst(TempProgram, newStr)
  endfor
  DecodedProgram = Compile(TempProgram)
End
```

Finally, the proper code for instantiating the customized template is added to the program source code. The program source code, *TempProgram*, is fed to a C++ compiler that performs necessary optimizations to take advantage of the partial evaluation technique to produce the *DecodedProgram*. The *DecodedProgram* is loaded into instruction memory, which is a separate data structure than main memory. While the main memory holds the original program data and instruction binaries, each cell of instruction memory holds a pointer to the optimized functionality as well as the instruction binary. The instruction binary is used to check the validity of the decoded instruction during runtime.

Algorithm 2 briefly describes how the proper template is determined for each instruction binary. In *DetermineTemplate* function, the instruction binary is compared with all binary masks in the *maskTable*, in which each binary mask is associated with a template. When an instruction matches with a mask, the corresponding template is selected and returned.

Algorithm 3 describes the template customization process. The algorithm's basic idea is to extract the values from specific fields of the binary instruction (e.g., opcode, operand) and assign those values to the corresponding template.

We illustrate the power of our technique to generate an optimized decoded instruction using a single data-processing instruction. Example 3 shows the binary as well as the assembly of an *ADD* instruction.

Algorithm 2- DetermineTemplate

```
Inputs: Instruction inst (Binary), and Mask Table maskTable.
Output: Template.
Begin
  foreach entry < mask; template > in maskTable
    if mask matches inst
      return template
  endfor
End
```

Algorithm 3- CustomizeTemplate

```
Inputs: Template template, Instruction inst (Binary).
Output: Customized Template with Parameter Values.
Begin
  switch instClassOf(inst)
    case Data Processing:
      switch (inst[31:28])
        case 1110: condition = Always endcase
        ...
      endswitch
      switch (inst[24:21])
        case 0100: opcode = ADD endcase
        ...
      endswitch
      ......
      return template < condition, opcode, …>
    endcase /* Data Processing */
    case Branch: ... endcase
    ...
  endswitch
End
```

Example 3- The binary and assembly of a data processing instruction in ARM.

Binary:	1110	000	0100	0	0010	0001	01010	00	0	0011
	cond	000	op	S	Rn	Rd	immed	shift	0	Rm

```
Assembly: ADD                 r1   r2   r3  LSL    #10
            op   {cond} {S} Rd, Rn, Rm Shift #immed
```

Since the instruction binary of Example 3 matches with the second mask in Example 2, the *DetermineTemplate* function (Algorithm 2) returns the *DataProcessing* template (shown in Figure 8). The *CustomizeTemplate* function (Algorithm 3) generates the customized template for the execute function by matching and replacing the template parameters with the field values at corresponding locations in the instruction. For example, the first 4 bits of the *ADD* instruction in Example 3 is used to determine the corresponding condition for executing the instruction. The value "1110" in this portion of instruction corresponds to the condition *Always*. Similarly, other portions of instruction are decoded to find the template parameter values. Figure 9 shows the customized *execute()* function for the *DataProcessing* template (Figure 8) using the parameter values.

After compilation using a C++ compiler, several optimizations occur on the *execute()* function. The *Always::execute()* function call is evaluated to *true*. Hence, the check is removed. Similarly, the *UpdateFlag* is evaluated to *false*. As a result, the branch and the statements inside it are removed by the compiler.

```
void DataProcessing<Always, Add, false, SftOper<Reg, ShiftLeft, Imm>>::execute()
{
  if ( Always::execute() )
  {
    _dest = Add::execute( _src1, _sftOperand.getValue() );
    if ( false )
    {
      Update_the_flag_bits(…);
    }
  }
}
```

Fig. 9. Customized *execute()* function for *ADD* instruction in Example 3.

```
void DataProcessing<Always, Add, false, SftOper<Reg, ShiftLeft, Imm>>::execute()
{
  _dest = _src1 + _sftOperand._operand << 10;
}
```

Fig. 10. Optimized *execute()* function for instruction of Example 3.

Finally, the two function calls *Add::execute()* and *_sftOperand.getValue()* get inlined as well. Consequently, the *execute()* function gets optimized into one single statement, as shown in Figure 10.

Furthermore, in many ARM instructions, the shifter operand is a simple register or immediate. Therefore, the shift operation is actually a no shift operation (according to the ARM manual, a shift left zero) and is removed by the compiler. In this way, an instruction similar to the previous example would have only one operation in its *execute()* method.

4.3 Simulation Engine

Due to compile time decoding and our template customization technique, the simulation engine is fast and simple. In this section, we briefly describe the three basic steps in the simulation kernel such as, fetch, decode (if necessary) and execute (shown in the "Runtime" section of Figure 5). The simulation engine fetches one decoded instruction at a time. As mentioned earlier, each instruction entry contains two fields such as, binary and the pointer to the optimized functionality for the instruction. Before executing the fetched instruction, it is necessary to verify that the current instruction is valid (i.e., this instruction is not modified during runtime). The simulation engine compares the binary part of the current instruction having address *addr* with the binary instruction of the application program stored in memory at address *addr*. If they are equal, the decoded instruction is valid and the engine executes the optimized functionality referenced by the instruction.

However, if the instruction is modified, the modified binary is redecoded. This decoding is similar to the one performed during compile time decoding of instructions except that it uses a pointer to an appropriate function. While

we develop the templates for each class of instructions, we also develop one function for each class. The mask table mentioned in Section 4.2 maintains the mapping between a mask for every class of instruction and the function for that class. The decoding step during runtime consults the mask table and determines the function pointer. It also updates the *instruction memory* with the decoded instruction (i.e., it writes the new function pointer in that address). The execution process is very simple. It simply invokes the function using the pointer specified in the decoded instruction.

Since the number of instructions modified during runtime is usually negligible, using a general unoptimized function for simulating them does not degrade the performance. It is important to note that since the engine is still very simple, we can easily use traditional interpretive techniques for executing modified instructions while the instruction-set compiled technique can be used for the rest (majority) of the instructions. Thus, our ISCS technique combines the full flexibility of interpretive simulation with the speed of the compiled simulation.

5. COMPILE TIME REDUCTION

In static-compiled simulation, the generated simulator is optimized by the C++ compiler. Therefore, it delivers good simulation performance, but the compilation time can be extremely large since the complete input program is decoded into source code. In dynamic-compiled simulation, the compilation time overhead is removed and the instructions are optimized at runtime during simulation. However, only simple optimizations that do not impose huge performance penalty are possible in these approaches. As a result, the simulation performance is sacrificed. We propose a hybrid technique that combines the advantages of both static- and dynamic-compiled simulation. In our hybrid ISCS, first the program is decoded and analyzed statically. Instead of generating source code for every instruction in the program, we generate the source code of a decoder that is customized for the input program. This decoder along with the rest of the simulation engine is compiled by a standard C++ compiler to generate the customized hybrid ISCS simulator for the program, as shown in Figure 11. Since the source code of the customized decoder is much smaller than the source code of whole decoded program, generating the hybrid ISCS takes significantly less compilation time compared to that of standard static-compiled simulation techniques. Furthermore, the hybrid ISCS can implement much more sophisticated optimizations than dynamic-compiled, simulations, since the decoder is optimized at compile time. Moreover, the hybrid ISCS is as flexible as an interpretive simulator, since it executes one instruction at a time. It is important to note that our improved simulation flow in Figure 11 (Hybrid-ISCS) is similar to the flow in Figure 5 (ISCS) except that the Figure 11 supports hybrid dynamic compilation, whereas Figure 5 relys on static compilation. On a related note, the compile time reduction techniques (presented in this article) have no effect on the simulation efficiency, since each optimized function will have the smallest number of computations irrespective of the compilation technique.

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

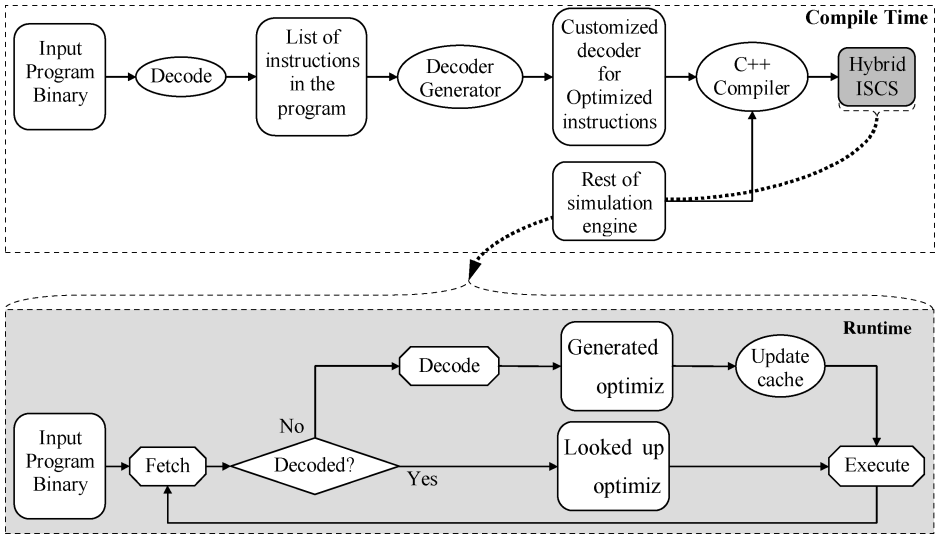

Fig. 11. Hybrid instruction-set-compiled simulation flow.

number of *instruction types*[2] 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, many instruction instances, such as *Add R1, R2, R3*; *Add R4, R5, R6*; and *Add R1, R2, #10*, correspond to only two instruction types from the instruction set, that is, *Add* R_x, R_y, R_z and *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 classes 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 runtime, and for each instruction instance, 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 runtime.

One may like to explore various ways of defining instruction classes and study their impact on compilation efficiency. It is important to note that the code size will be comparable irrespective of how many templates (instruction classes) are used in dynamic compilation. However, it may affect the compilation time. If the number of templates is less than instruction classes in the architecture manual, the corresponding templates will be very complex, since it is trying to cluster different types of instructions. As a result, the compilation time may increase. On the other hand, if too many instruction classes are used, it may not decrease compilation time, since the number of possible alternatives that the compiler is trying to explore will remain the same. As mentioned earlier,

[2]An instruction class (described earlier) consists of a set of similar instruction types.

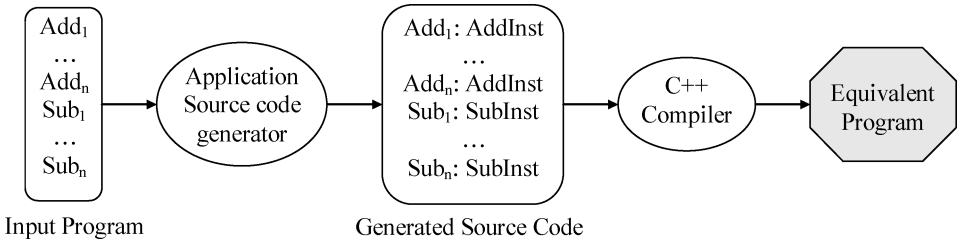

Fig. 12. Static decode of one program.

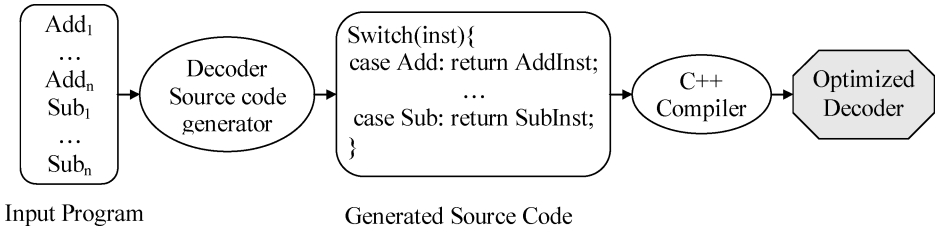

Fig. 13. Dynamic decode of one program.

the basic idea of using templates is to reduce the manual effort for a simulator developer. Therefore, it is optimal to use the same number of instruction classes as in the architectural manual to improve readability, maintainability, and as well as initial template development time. In the remainder of this section, we analyze different possible scenarios where our hybrid technique can be used. We compare these scenarios in Section 6.2.

5.1 Static Decode of One Program

This approach is same as static-compiled simulation. As shown in Figure 12, 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 runtime and load them in the instruction memory. For example, if the program contains 1,000 similar *Add* instructions of which only 500 execute at runtime, there will be 1,000 corresponding codes in the generated source code and 1,000 instantiations at runtime.

5.2 Dynamic Decode of One Program

As shown in Figure 13, 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 runtime and decodes them by instantiating the optimized code of the corresponding instruction type. In this case the size of the generated source code is significantly smaller than the static decode, and hence, the compilation time is considerably less. For example, if the program contains 1,000 similar *Add* instructions of which 500 execute at runtime, only one customized

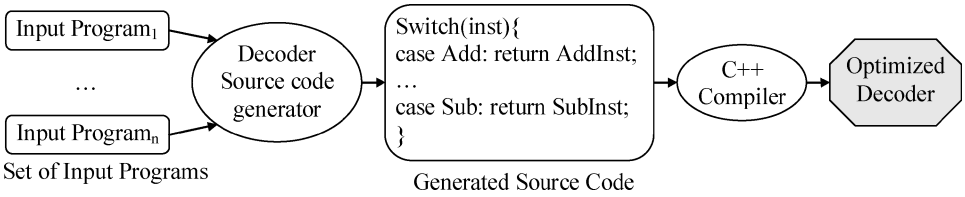

Fig. 14. Dynamic decode of multiple program.

code is added to the decoder for that *Add* instruction. At runtime, 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 500 instantiated at runtime.

5.3 Dynamic Decode of Multiple 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 14. 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. 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.

5.4 Dynamic Decode for All ISA

The instructions of a program are a subset of all possible variations of the instructions in the 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.

For example, the SPARC processor has a simple instruction set and the number of variations of the instructions is less than 1,000. 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 ($\sim$500k). Thus, using this approach for ARM processor not only has a long compilation time, but also consumes a lot of memory for the decoder and hence may not be practical. For example, in the case of SPARC processor, the number of instruction types required for dynamic decode of multiple programs is 126 (as shown in Figure 20), whereas the number of variations in SPAC processor is approximately 1,000. Therefore, there will be a 10 $\times$ increase in code size (and hence the compilation time) if dynamic decode for all ISA is used. Similarly, there will be a 1,000 $\times$ code size increase in case of ARM processor. Clearly, it is not a good idea to use dynamic decode for all ISA during exploration. However, once exploration phase is over using dynamic decode for one program (Section 5.2) or multiple programs (Section 5.3), the final optimized decoder can be built using dynamic decode for all ISA approaches.

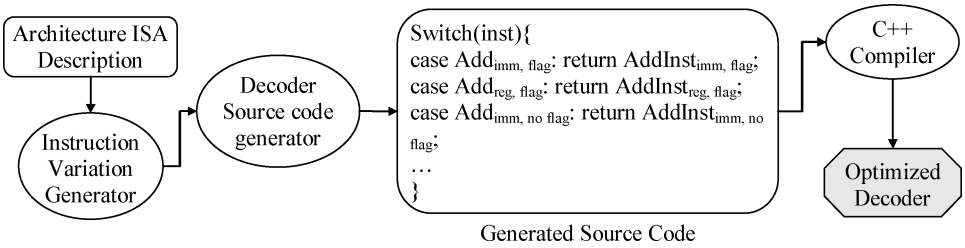

Fig. 15. Dynamic decode of all ISA.

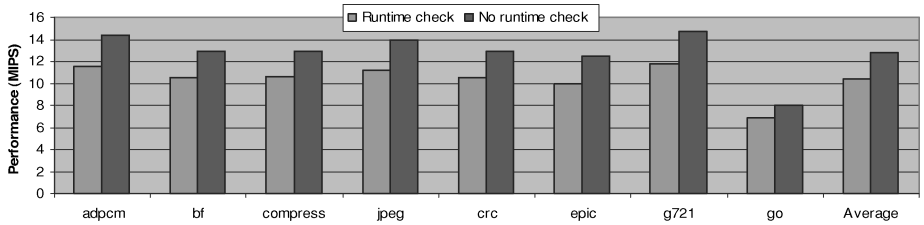

Fig. 16. Hybrid instruction-set-compiled simulation performance for ARM7 processor model.

6. EXPERIMENTS

In this section, we present simulation results using two contemporary processors, ARM7 [ARM7] and SPARC [SPARC], to demonstrate the usefulness of our approach. The ARM7 processor is a RISC machine with fairly complex instruction set. We used *arm-linux-gcc* for generating target binaries for ARM7. Performance results of the different generated simulators were obtained using a Pentium 3 (1GHz) with 512MB RAM running Windows 2000. The generated simulator code is compiled using the Microsoft Visual Studio .NET compiler with all optimizations enabled. The same C++ compiler is used for compiling the decoded program as well. The SPARC V7 is a high-performance RISC processor with 32-bit instructions. We used *gcc3.1* to generate the target binaries for SPARC and validated the generated simulator by comparing traces with *Shade* [Cmelik 1994] simulator. We show the results using eight application programs: *adpcm*, *jpeg*, *blowfish* (*bf*), *compress*, *crc*, epic, *g*721, and *go*. The *adpcm* and *jpeg* benchmarks are used to compare our simulator performance with the previously published results [Nohl 2002].

We present our results in two categories. First, we present the experimental results to demonstrate the simulation performance improvement using our ISCS technique. Next, we present the results on compile time reduction using our hybrid compilation technique.

6.1 Simulation Performance

Figure 16 and Figure 17 show the simulation performance using our technique for the ARM7 and SPARC processor models, respectively. The first bar shows the simulation performance of our technique with runtime program modification check enabled. The simulation performance can be improved if it is known prior to execution that the program is not self-modifying. The second bar represents

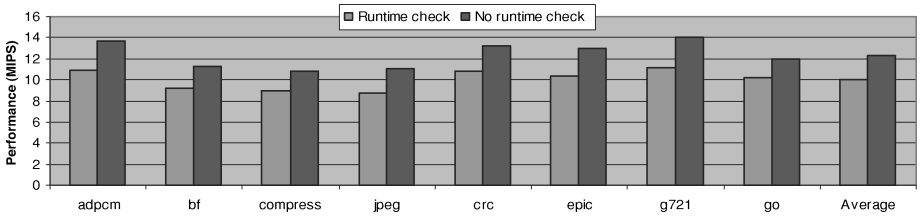

Fig. 17. Hybrid instruction-set-compiled simulation performance for SPARC processor model.

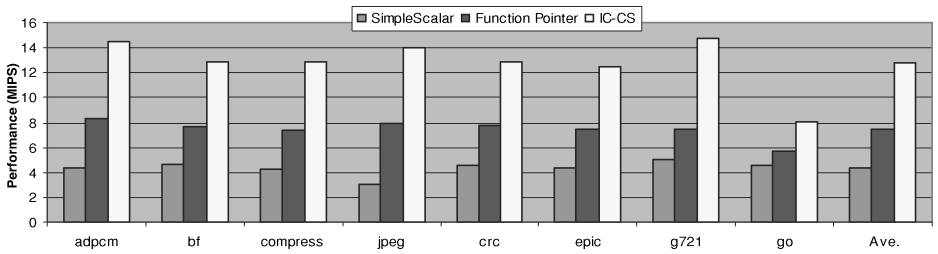

Fig. 18. Effect of different optimizations on ARM7 simulator.

the simulation performance of running the same benchmark by disabling the runtime check. We could achieve up to 25% performance improvement (23%, on average) by disabling the instruction modification detection and updation mechanism.

When runtime program modification check is enabled, the ARM7 simulation speed is up to 12MIPS using the P3 (1.0GHz) host machine. To the best of our knowledge, the best performance of a simulator having the flexibility of interpretive simulation has been JIT-CCS [Nohl 2002]. The JIT-CCS technique could achieve a performance upto 8MIPS on an Athlon at 1.2GHz with 768MB RAM. Since we did not have access to a similar machine, our comparisons are based on results running on a slower machine (Pentium 3 at 1GHz with 512MB RAM). On the *jpeg* benchmark, our ISCS technique performs 40% better than JIT-CCS technique. The same trend (30% improvement) is observed in case of *adpcm* benchmark as well. Clearly, these are conservative numbers, since our experiments were run on a slower machine.

There are two reasons for the superior performance of our technique: moving the time-consuming decoding out of the execution loop, and generating aggressively optimized code for each instruction. The effects of using these techniques are demonstrated in Figure 18. The first bar in the chart is the simulation performance of running the benchmarks on an ARM7 model of Simplescalar [Simplescalar] that does not use any of these techniques. The second bar shows the effect of decoding the instructions at runtime using function pointers and caching the decoded information. The use of function pointers is similar to JIT-CCS. The last bar shows the performance of our simulation approach that uses compile-time decode and C++ templates for optimized code generation. As Figure 18 shows, the ISCS technique improves the simulation performance by 70%, on average, compared to using function pointers, similar to JIT-CCS.

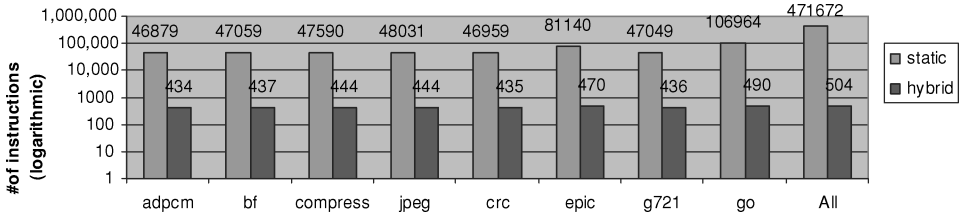

Fig. 19. Generated source file size in different techniques in ARM7 simulator.

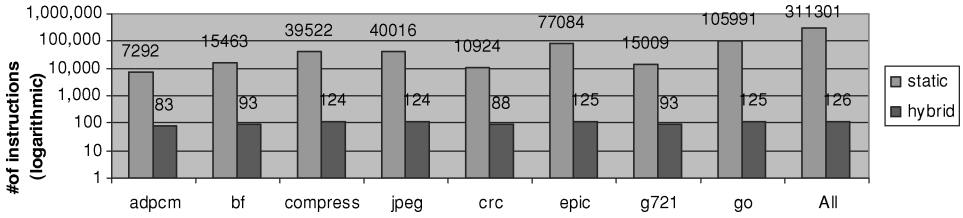

Fig. 20. Generated source file size in different techniques in SPARC simulator.

We have demonstrated that ISCS coupled with our template customization technique delivers the performance of compiled simulation while maintaining the flexibility of interpretive simulation. Our simulation technique delivers better performance than other simulators in this category, as demonstrated in this section.

6.2 Compile Time Reduction

We used the ISCS technique to implement the optimized decoder in our simulator. To exclude the effect of file structures from the compilation time comparisons, it is important that the generated files in both static-compiled simulation and hybrid-compiled simulation have similar structures. In all of our experiments, each source file contained up to 100 functions and each function contained up to 100 instruction decoding.

Figures 19 and 20 show the number of instructions that exist in the generated source files 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 benchmarks together (Section 5.3). Interestingly, compared to the number of instruction instances, the number of instruction types change slightly between benchmarks and have a lot of commonality. It is important to note that our hybrid compilation technique is flexible to incorporate any one of the dynamic compilation techniques (discussed in Sections 5.2, 5.3, and 5.4). In the experiments shown in Figures 19 through 24, the first bar in the figures represents the corresponding metric (code size, binary size, or compilation time) using static compilation (described in Section 5.1), and the second bar represents those of hybrid compilation (described in Section 5.2)

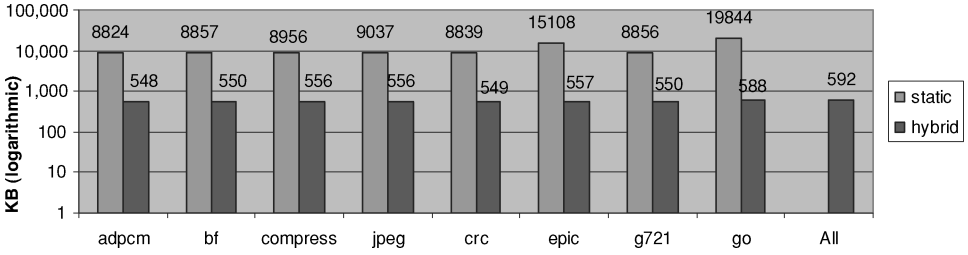

Fig. 21. Executable file size in different techniques in ARM7 simulator.

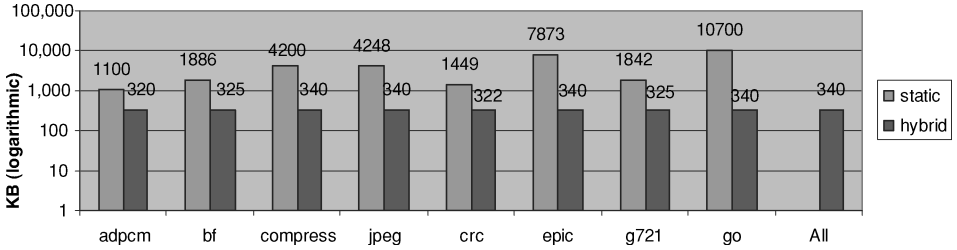

Fig. 22. Executable file size in different techniques in SPARC simulator.

for individual programs. The second bar in the last column (marked "All") represents the metrics for dynamic compilation for all programs (described in Section 5.3). As described in Section 5.4, it is not profitable to use dynamic decode for all ISA in the exploration phase. Instead it can be used to build the final optimized decoder once exploration is over.

Similarly, Figures 21 and 22 show 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 experiments, we got very similar performance results from both static- and hybrid-compiled simulation. However, we believe that in the hybrid approach, the instructions must be decoded again at runtime, but the smaller executable size improves the cache behavior of the hybrid simulator compared to that of the static-compiled simulator and, therefore, compensates the extra runtime decoding overhead.

The last bars in both figures show the size of customized decoder for all benchmarks. Note that this technique is not applicable to traditional static compiled simulation.

Figure 23 and Figure 24 show the comparison of the compilation time of hybrid- and static-compiled simulation. In our experiments, the average compilation time was about 3,474s for static-compiled simulation and about 28s for our hybrid-compiled simulation. This shows, on average, 100× reduction compilation time, while still benefiting from all the advantages of static-compiled simulation. As mentioned earlier, the last bars in both figures show the compilation time of decoder for all benchmarks. This compilation time is shared among all seven benchmarks. Therefore, the overall compilation overhead per benchmark is further reduced.

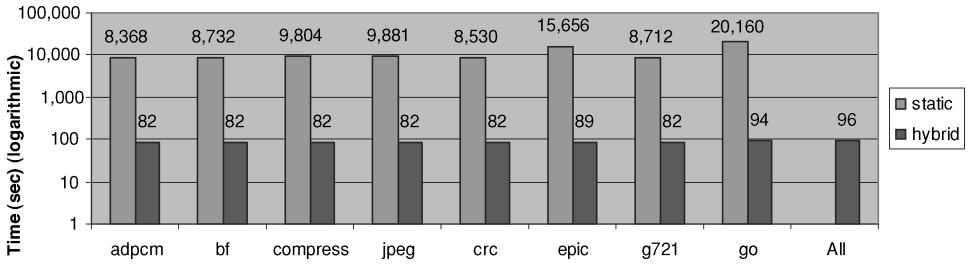

Fig. 23. Compilation time in different techniques in ARM7 simulator.

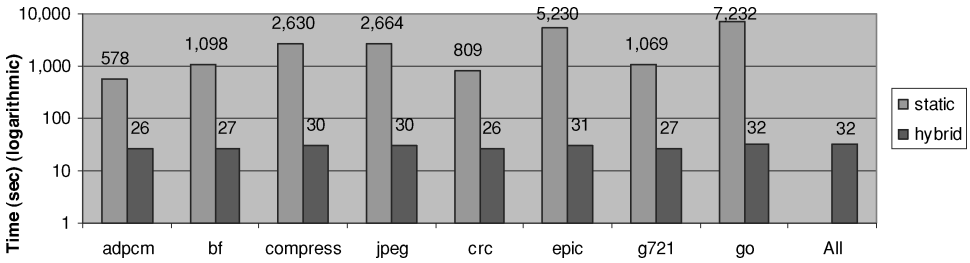

Fig. 24. Compilation time in different techniques in SPARC simulator.

7. CONCLUSIONS

Design of programmable processors and embedded applications require ISSs for early exploration and validation of candidate architectures. Due to the increasing complexity of programmable architectures and time-to-market pressure, performance is the most important feature of an ISS. This article made two important contributions in improving simulation performance: ISCS for improving runtime performance, and hybrid-compiled simulation for reducing compilation time in compiled simulators. Due to the simple interpretive simulation engine and optimized pre-decoded instructions, our ISCS technique achieves the performance of compiled simulation, while maintaining the flexibility of interpretive simulation. The performance can be further improved by disabling the runtime change detection that is suitable for many applications that are not self-modifying. The ISCS technique achieves its superior performance for two reasons: using templates to produce aggressively optimized code for each instance of instructions, and moving time-consuming decode optimizations to compile time. We demonstrated performance improvement of 70%, on average, over the best-published results on an ARM7 model.

A major challenge in ISCS technique is the compilation time overhead that makes usage of compiler optimizations impractical, especially for large applications. The problem is also present in traditional-compiled simulators. We developed a hybrid-compiled simulation technique that utilizes the advantages of both static- and dynamic-compiled simulation and reduces the compilation time. In this approach, the input program is first analyzed and an optimized decoder is generated for that program using a conventional (C or C++) compiler. The results showed two orders of magnitude reduction in compilation time

without any performance penalty. Our future work will concentrate on using this technique for modeling other real-world architectures.

REFERENCES

AMICEL, R., AND BODIN, F. 2002. Mastering startup costs in assembler-based compiled instruction-set simulation. In *Proceedings of Workshop on Interaction between Compilers and Computer Architectures (INTERACT'02)* IEEE, Los Alamitos, CA.

ARM7, The ARM7 User Manual. http://www.arm.com

BELL, J.R. 1973. Threaded code. *Comm. ACM 16*, 370–372.

BRAUN, G., HOFFMANN, A., NOHL, A., AND MEYR, H. 2001. Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description. In *Proceedings of the International Symposium on Systems Synthesis (ISS'01S)*. ACM, New York, 57–62.

BURTSCHER, M., GANUSOV, I. 2004. Automatic synthesis of high-speed processor simulators. In *Proceedings of the International Symposium on Microarchitecture (MICRO'4)*. ACM, New York, 55–66.

CMELIK, B., AND KEPPEL, D. 1994. Shade: a fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Perform. Eval. Rev. 22*, 128–137.

ENGEL, F., NÜHRENBERG, J., AND FETTWEIS, G.P. 1999. A generic tool set for application specific processor architectures. In *Proceedings of the 8th International Workshop on Hardware / Software Codesign*. IEEE, Los Alamitos, CA, 126–130.

FUTAMURA, Y. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Syst. Comput. Controls 2*, 45–50.

HADJIYIANNIS, G., HANONO, S., AND DEVADAS, S. 1997. ISDL: an instruction set description language for retargetability. In *Proceedings of the Design Automation Conference (DAC'97)* IEEE, Los Alamitos, CA.

HALAMBI, A., GRUN, P., GANESH, V., AND KHARE, A. 1999. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the Design Automation and Test in Europe (DATE'99)*. Springer, Berlin, Germany.

HARTOOG, M.R., ROWSON, J.A., REDDY, P.D., DESAI, S., DUNLOP, D.D., HARCOURT, E.A., AND KHULLAR, N. 1997. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of the Design Automation Conference (DAC'97)*. IEEE, Los Alamitos, CA.

LEUPERS, R., ELSTE, J., AND LANDWEHR, B. 1999. Generation of Interpretive and Compiled Instruction Set Simulators. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'99)*. IEEE, Los Alamitos, CA.

MAGNUSSON, P.S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., WERNER, B. 2002. Simics: a full system simulation platform. *IEEE Comput. 35*, 50–58.

NOHL, A., BRAUN, G., HOFFMANN, A., SCHLIEBUSCH, O., MEYR, H., AND LEUPERS, R. 2002. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the Design Automation Conference (DAC'02)*. IEEE, Los Alamitos, CA, 22–27.

ONDER, S., GUPTA, R. 1998. Automatic generation of microarchitecture simulators. In *Proceedings of the International Conference on Computer Languages*. IEEE, Los Alamitos, CA, 80–89.

PEES, S., HOFFMANN, A., AND MEYR, H. 2000. Retargeting of compiled simulators for digital signal processors using a machine description language. In *Proceedings of the Design Automation and Test in Europe (DATE'00)*. Springer, Berlin, Germany, 669–673.

QIN, W., MALIK, S. 2003. Automated synthesis of efficient binary decoders for retargetable software toolkits. In *Proceedings of the Design Automation Conference (DAC'03)*. IEEE, Los Alamitos, CA, 764–769.

RAMSEY, N., FERNANDEZ, M.F. 1997. Specifying representations of machine instructions, *ACM Tran. Program. Lang. Syst. 19*, 492–524.

RESHADI, M., MISHRA, P., AND DUTT, N. 2003a. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings of the Design Automation Conference (DAC'03)*. IEEE, Los Alamitos, 758–763.

RESHADI, M., AND DUTT, N. 2003b. Reducing compilation time overhead in compiled simulators, In *Proceedings of the International Conference on Computer Design (ICCD'03)*. IEEE, Los Alamitos, CA, 151–153.

RESHADI, M., MISHRA, P., DUTT, N. 2006. A retargetable framework for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst. 5*, 2, 431–452.

SCHNARR, E., AND LARUS, J. R. 1998. Fast out-of-order processor simulation using memorization. In *Proceedings of the Programming Language Design and Implementation (PLDI'98)*. ACM, New York, 283–294.

SCHNARR, E. C., HILL, M. D., AND LARUS, J. R. 2001. Facile: a language and compiler for high-performance processor simulators. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 321–351.

SIMPLESCALAR HOME. http://www.simplescalar.com.

SPARC. Version 7 Instruction set manual: http://www.sun.com

WITCHEL, E., AND ROSENBLUM, M. 1996. Embra: fast and flexible machine simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (MMCS'96)*. ACM, New York, 68–79.

ZHU, J., GAJSKI, D. 1999. A retargetable, ultra-fast instruction set simulator. In *Proceedings of the Design Automation and Test in Europe (DATE'99)*. Springer, Berlin, Germany, 298–302.

Received September 2006; revised May 2007; accepted August 2007