A Retargetable Framework for Instruction-Set Architecture Simulation

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

Instruction-set architecture (ISA) simulators are an integral part of today's processor and software design process. While increasing complexity of the architectures demands high-performance simulation, the increasing variety of available architectures makes retargetability a critical feature of an instruction-set simulator. Retargetability requires generic models while high-performance demands target specific customizations. To address these contradictory requirements, we have developed a generic instruction model and a generic decode algorithm that facilitates easy and efficient retargetability of the ISA-simulator for a wide range of processor architectures, such as RISC, CISC, VLIW, and variable length instruction-set processors. The instruction model is used to generate compact and easy to debug instruction descriptions that are very similar to that of architecture manual. These descriptions are used to generate high-performance simulators. Our retargetable framework combines the flexibility of interpretive simulation with the speed of compiled simulation. The generation of the simulator is completely separate from the simulation engine. Hence, we can incorporate any fast simulation technique in our retargetable framework without introducing any performance penalty. To demonstrate this, we have incorporated fast IS-CS simulation engine in our retargetable framework which has generated 70% performance improvement over the best known simulators in this category. We illustrate the retargetability of our approach using two popular, yet different, realistic architectures: the SPARC and the ARM.

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

General Terms: Design, Language, Performance

Additional Key Words and Phrases: Retargetable instruction-set simulation, generic instruction model, instruction binary encoding, decode algorithm, architecture description language

This research was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to acknowledge Nikhil Bansal for his contribution in generating the SPARC simulator.
Authors' addresses: Mehrdad Reshadi and Nikil Dutt, Center for Embedded Computer Systems, University of California Irvine, CA 92697; email: {reshadi,dutt}@cecs.uci.edu; Prabhat Mishra, Department Computer and Information Science and Engineering, University of Florida, FL 32611; email: prabhat@cise.ufl.edu.

1. INTRODUCTION

Instruction-set architecture (ISA) simulators are indispensable tools in the development of new architectures. They are used to validate an architecture design, a compiler design, as well as to evaluate architectural design decisions during design space exploration. Running on a *host* machine, these tools mimic the behavior of an application program on a *target* machine. 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 multimode processors; and retargetable to support a wide spectrum of architectures. Although in the past years, performance has been the most important quality measure for the ISA simulators, retargetability is now an important concern, particularly in the area of the embedded systems and system-on-chip (SoC) design.

A retargetable ISA simulator requires a generic model, supported by a language, to describe the architecture and its instruction set. The simulator uses the architecture description to decode instructions of the input program and execute them. The challenge is to have a model that is efficient in terms of both quality of the description and performance of the simulator. To have a high-quality description, the model must easily capture the architectural information in a natural, compact and manageable form for a wide range of architectures. On the other hand, to generate a high-performance simulator and to reduce the operations that the simulator must do dynamically at runtime, the model should provide as much static information as possible about the architecture and its instruction-set.

In general, the instruction model, used in a retargetable ISA simulation, must capture the complexities of addressing modes, binary encoding, and execution semantics of the instructions in an architecture. Designing an efficient model that captures a wide range of architectures is a hard problem, because such architectures have different instruction-set format complexities. There is a tradeoff between speed and retargetability in ISA simulators. Some of the retargetable simulators use a very general processor model and support a wide range of architectural features, but are slow, while others use some architectural or domain specific performance improvements, but support only a limited range of processors. Furthermore, existing languages require lengthy descriptions of all possible formats of instructions to derive a fast simulator.

An efficient instruction model should support extensive reuse of descriptions. To facilitate reuse, all architecture-description languages (ADLs) use the notion of symbol or identifier to represent an entity. Therefore, instead of repeating the description of an entity, its corresponding symbol is used to refer to it. This is similar to the use of variable names and function calls in programming languages. The existing architecture-description languages use symbols that correspond to only one entity. Besides, these symbols only represent *operands* of instructions. As a result, they are limited by the *number*, as well as the *type*, of entities that they can represent. In contrast, in our proposed instruction model, a symbol represents more than one entity. It can also represent different types of entities such as *operands* and *opcodes* of instructions. This leads to a compact architecture-description and also facilitates exploitation of all possible symbol values to generate highly optimized instruction-set simulators.

In this paper, we present a retargetable simulation framework that supports many variations of architectures with complex instruction sets, while generating high-performance ISA simulators. To achieve maximum retargetability, we have developed a generic instruction model coupled with a decoding technique that flexibly supports variations of instruction formats for widely different contemporary processors. A combination of different addressing modes with various operations results in different instruction formats. Our model can also be used to exploit all possible instruction formats to generate optimized code for simulating them. We use this generic model to capture the behavior and binary encoding of the instructions. The EXPRESSION ADL [Halambi et al. 1999] is used to capture the structure of the architecture. The instruction descriptions, based on our generic model, are very compact and easy to debug and verify. In our framework, we have used the instruction-set compiled simulation (IS-CS) technique [Reshadi et al. 2003] that has the flexibility of interpretive simulation and the speed of compiled simulation. The IS-CS technique uses instruction templates to aggressively generate optimized code for executing instructions. We automatically generate the templates from the instruction descriptions. However, our retargetable framework is generic enough to incorporate any other simulation optimization techniques.

The rest of the paper is organized as follows. Section 2 presents related work addressing ISA simulator-generation techniques and distinguishes our approach. Section 3 outlines our retargetable simulation framework. The three key components of the framework, i.e., a generic instruction model, a decoding algorithm, and the simulation code generation, are described in Section 4, 5, and 6, respectively. Section 7 compares the efficiency of our instruction model with other ADLs, and presents simulation performance results using two contemporary processor architectures: ARM7 and SPARC. Finally, Section 8 concludes the paper.

2. RELATED WORK

An extensive body of recent work has addressed retargetable instruction-set architecture simulation. A fast and retargetable simulation technique is presented by Zhu and Gajaski [1999]. It improves traditional static compiled simulation by mapping the target machine registers to the host machine registers through a low-level code generation interface at compile time. The code-generation interface explicitly defines the mappings between the target architecture instructions and the host assembly. FACILE [Schnarr et al. 2001] is a language that retargets FastSim simulator [Schnarr and Larus 1998], but compromises its performance significantly. The nML language [Freericks et al. 1991] is a hierarchical formalism for instruction-set modeling and is used by several code generation and instruction-set simulation tools. Sim-nML [Hartoog et al. 1997] is based on nML. It targets DSP processors and generates relatively slow simulators. ISDL [Hadjiyiannis et al. 1997] is mainly suitable for assembly/binary code generation and follows a grammar similar to that of *lex* and *yacc* tools used for describing programming languages. The JACOB system [Leupers et al. 1999] generates both interpretive and compiled simulators from the MIMOLA [Bashford et al. 1994], which describes the architecture in RTL level. FLEXWARE simulator [Paulin et al. 1997] uses a VHDL model of a generic parameterizable model. SimC [Engel et al. 1999] is based on a machine description in ANSI C. It uses compiled simulation and has limited retargetability. Babel [Mong and Zha 2003] was originally designed for retargeting the binary tools and has been recently used for retargeting the SimpleScalar simulator [Simplescalar].

MDES [Gyllenhaal et al. 1996] was designed for compilation and has been used for simulator generation in Trimaran framework [Trimaran]. However, it allows retargetability only for the class of architectures in the HPL-PD family. It does not appear to have language support for specifying instruction syntax and semantics and the attendant task of instruction binary decoder generation. The Liberty Simulation Environment (LSE) [Vachharajani et al. 2002] models processors by connecting the hardware modules through their interfaces. It generates cycle-accurate simulators from such descriptions. This environment does not provide any facility for capturing the instruction behavior, binary encoding, and generating the decoder. The instruction decoder must be explicitly described in C language

The just-in-time cache compiled simulation (JIT-CCS) [Nohl et al. 2002] technique is the closest to our approach. It combines retargetability, flexibility, and high simulation performance. It uses the LISA machine description and its performance improvement is gained by caching the decoded instruction information. LISA supports simple RISC-like instruction formats. Efficient support of complex instruction formats requires extensive coding in this language.

In contrast, in our simulation framework, the proposed instruction model efficiently supports a wide variety of instruction formats supported by contemporary processor architectures, as well as architectures with complex hybrid instruction-sets. While our generic instruction model results in a compact and easy to debug description, the proposed decode algorithm can extract the required information for any simulator generation technique from the specification.

3. RETARGETABLE SIMULATION FRAMEWORK

In a retargetable ISA-simulation framework, the range of architectures that can be captured and the performance of the generated simulators depend on three issues: first, the *model* based on which the instructions are described; second, the *decoding algorithm* that uses the instruction model to decode the input binary program; and third, the *execution method* of decoded instructions. These issues are equally important and ignoring any of them results in a simulator that is either very slow, but general, or very fast, but restricted to some architecture domains. However, the instruction model significantly affects the complexity of decode and the quality of execution. A simple instruction model will require a simple decode algorithm, but may not provide enough information for aggressive execution optimizations. On the other hand, a very detailed

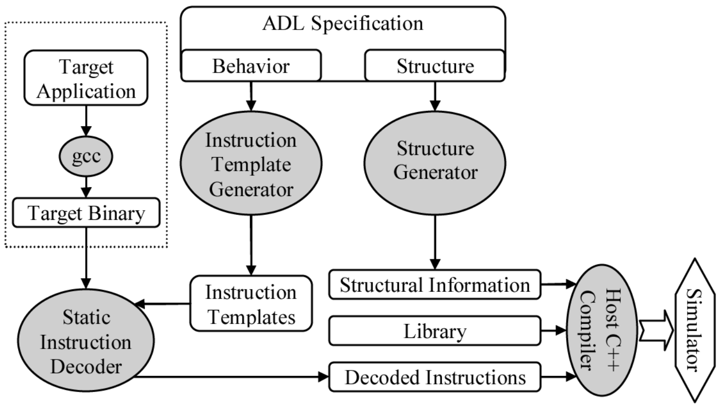

Fig. 1. Generating the simulator from ADL.

and complex instruction model may include all the information needed for optimizations, but may require a very complex decode algorithm or may result in complex descriptions. A complex description is less readable and maintainable. Similarly, a more human friendly instruction description may have negative effects on the complexity of decode and applicability of optimizations. We have developed a generic instruction model coupled with a simple decoding algorithm that lead to an efficient and flexible execution of decoded instructions.

Figure 1 shows our retargetable simulation framework that uses the ADL specification of the architecture and the application program binary (compiled by *gcc*) to generate the simulator. The ADL captures the behavior and structure of the target architecture. We describe the binary encoding and behavior of instructions, based on our generic instruction model, as described in Section 4. Using the instruction specifications from ADL description, the *Static Instruction Decoder* decodes the target program, one instruction at a time, as described in Section 5. We use the IS-CS technique [Reshadi et al. 2003] for simulating the instructions. It achieves high-performance by executing aggressively optimized instructions. The optimizations are done through instruction templates, which are automatically generated from the behavioral description of the architecture. After decoding the instructions, the *Static Instruction Decoder* uses the extracted instruction templates to generate the optimized source code of the decoded instructions (Section 6.1), which is loaded in the instruction memory. In other words, the instruction model is used for both decoding the instructions and generating the corresponding source codes for simulation.

The *Structure Generator* compiles the structural information of the ADL into components and objects that keep track of the state of the simulated processor. It generates proper source code for instantiating these components at runtime. The target independent components are described in the *Library*. This library is finally combined with the *Structural Information* and the *Decoded Instructions* and is compiled on the host machine to obtain the final ISA simulator. Figure 2 shows the flow of the simulation engine. This engine fetches each instruction from instruction memory and looks it up in a software cache, which provides a mapping between <address, opcode> of the instruction and its corresponding

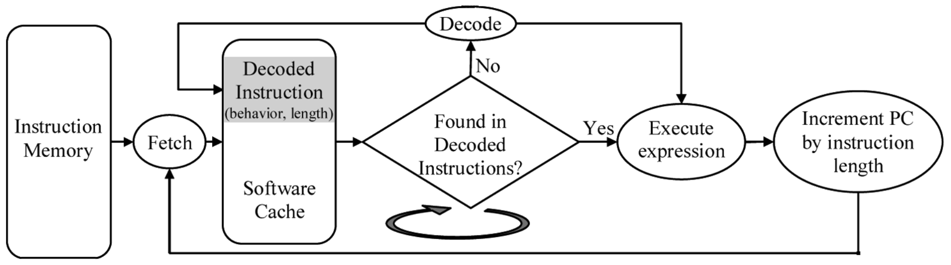
Instruction Memory
Fetch
Decoded Instruction (behavior, length)
Software Cache
Decode
No
Found in Decoded Instructions?
Yes
Execute expression
Increment PC by instruction length

Fig. 2. Simulation engine flow.

decoded information. The decoded information contains the behavior and the length of the instruction. If the decoded information of the instruction is found in the software cache, it is executed. Otherwise, the instruction is decoded prior to its execution, and the decoded instruction is stored back in the software cache. Finally, PC is incremented by the length of the instruction.

In order to support self-modifying codes, both the address and the opcode of instructions are used for look up in the software cache. In our framework, the original instructions of a program are statically decoded and the optimized behaviors are stored in the software cache. At runtime, an instruction is decoded if the opcode of the instruction differs from the one stored in the cache; otherwise the decoded information is reused. Details of generating executable behaviors of instructions are described in Section 6.

The simulation engine is specified by the *Library* component. Therefore, we can integrate other simulation techniques and optimizations in our retargetable framework by modifying the library component.

In the remainder of this section, we describe the generic instruction model for capturing the binary encoding and behavior of the instruction set. Next, we explain how the decoding algorithm decodes the program binary using the description of instructions in the ADL. Finally, we show how optimized code is generated for fast simulation.

4. GENERIC INSTRUCTION MODEL

In general, several instructions of an instruction set have very similar behavior, i.e., they have similar components with identical relations among them. Such instructions can be described using a single template. This template describes the components of these instructions and the relation between the components. For example, a set of three operand instructions can be described as `<opcode dest src1 src2>`, and the relation can be described as `dest` = f_{opcode}(`src1, src2`). Such instructions have four components, viz., `opcode, dest, src1`, and `src2`; execution of the instruction results in applying the functionality of `opcode` on the values of `src1` and `src2` and storing the result in the `dest`. We call such a template an *OperationClass*. While components of the instruction may have different values, their relation is always preserved as described by the template. For example, `opcode` can be any functionality, such as addition or subtraction and `src2` can be any kind of operand, such as constant or register; however, their relation always remains the same. This style of description is also used in many architecture manuals. The similarity in the behavior of a group of instructions stems from the fact that these instructions are usually executed by the same hardware unit in the architecture. Previously proposed ADLs have partially exploited such commonalities in instruction behaviors, by using symbols or parameters for operands; however, none of them have exploited the possibility of using parameters for both operands and operations in the instructions. Besides, the parameters in these ADLs can represent only one value. In our proposed instruction model, symbols represent both functionalities and values. Furthermore, a symbol can represent multiple entities depending on the actual instruction instance. For instance, in different instruction instances, symbol `src2` in the above example may represent a constant (value) or an addressing mode (functionality).

A major challenge in retargetable simulation is the ability to capture a wide variety of instructions. Our proposed instruction model is generic enough to capture the variations of instruction formats of contemporary processors. The focus of this model is on the complexities of different instruction binary formats in different architectures. As an illustrative example, we use the integer arithmetic instructions of the SPARC V7 processor, shown in Example 1, to describe the model. The complete description is shown in Figure 5 (see later).

Example 1 (*SPARC integer arithmetic instructions*). SPARC V7 [SPARC] is a single-issue processor with 32-bit instruction width. The integer-arithmetic instructions, `IntegerOps` (as shown below), perform certain arithmetic operation on two source operands and write the result to the destination operand. The destination and first source operand are always register operands. The second source operand can be either a register operand or a constant integer. This subset of instructions is distinguished from the others by the following bit mask:

`Bitmask:`	`10xxxxx0`	`xxxxxxxx`	`xxxxxxxx`	`xxxxxxxx`
IntergerOps: `<opcode dest src1 src2>`				

A bit mask is a string of “1”, “0” and “x” symbols and it matches the bit pattern of a binary instruction if, and only if, for each “1” or “0” in the mask, the binary instruction has a 1 or a 0 value in the corresponding position respectively. The “x” symbol matches with both 1 and 0 values.

Figure 3 shows an overview of our instruction model. In this model, each *VLIW instruction* consists of some *slots*. Each slot contains one of the possible *operation classes*. An operation class consists of a set of *symbols* and a *behavior*, described based on the symbols. A symbol can be a *constant*, a *register* or a *microoperation*. A *microoperation* is itself an operation class that represents a subfunctionality within other operation classes. A non-VLIW instruction is considered as a VLIW instruction with only one slot. In Figure 3, a node is selected if the binary of the instruction matches with the mask. In the remainder of this section, we describe the instruction model in detail. Figure 4 shows the complete description of the instruction model.

In this model, an *instruction* of a processor is composed of a series of *slots* and each slot contains only one *operation* from a subset of operations. All the

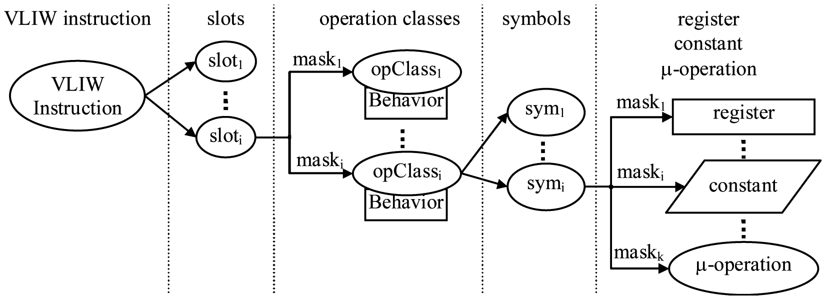

Fig. 3. Overview of proposed instruction model.

```
instruction::= slot-list
slot-list::= slot [ '|' slot-list ]
slot::= '$' opClass-mask-list '$'
opClass-mask-list::= '(' opClass ',' mask ')' ['|' opClass-mask-list]
mask::= (0 | 1 | x)*
opClass::= '<' symbol-type-list '|' expression '>'
symbol-type-list::= '(' symbol ',' '{' types '}' ')'['|'symbol-type-list]
symbol::= identifier
types::= '(' type ',' mask ')' [ | types ]
type::= register | constant | opClass
register::= '[' registerClass ',' index-function ']'
constant::= '#' constant-extraction-function '#'
```

Fig. 4. Grammar of instruction description.

operations in an instruction execute in parallel. Each operation is distinguished by a mask pattern. Therefore, each slot ($\mathtt{sl}_\mathtt{i}$) contains a set of operation-mask pairs ($\mathtt{op}_\mathtt{i}$, $\mathtt{mask}_\mathtt{i}$) and is defined by the following format. The length of an operation is equal to the length of its corresponding mask pattern.

$$\mathtt{slot}_\mathtt{i} = \mathtt{<(op}_\mathtt{i}^0\mathtt{,\ mask}_\mathtt{i}^0\mathtt{)\ |\ (op}_\mathtt{i}^1\mathtt{,\ mask}_\mathtt{i}^1\mathtt{)\ |\ \ldots>}$$

An *operation class* refers to a set of similar operations in the instruction set that can appear in the same instruction slot and have similar format. The previous slot description can be rewritten using an operation class: $\mathtt{slot}_\mathtt{i}$ = <($\mathtt{opClass}_\mathtt{i}$, $\mathtt{m}_\mathtt{i}$)>. For example, integer arithmetic instructions in SPARC V7 can be grouped in a class (`IntegerOps`), as shown below:

$$\mathtt{I}_{\mathtt{SPARC}} = \mathtt{<(IntegerOps,\ 10xx\text{-}xxx0\ xxxx\text{-}xxxx\ xxxx\text{-}xxxx\ xxxx\text{-}xxxx)\ |\ \ldots>}$$

An operation class is composed of a set of *symbols* and an *expression* that describes the behavior of the operation class in terms of the values of its symbols. For example, the operation class in Example 1 has four symbols: `opcode`, `dest`, `src1`, and `src2`. The *expression* for this example is: $\mathtt{dest = f}_{\mathtt{opcode}}\mathtt{(src1,\ src2)}$. Each symbol may have a different *type* depending on the bit pattern of the operation instance in the program. For example, the possible types for `src2` symbol in Example 1 are register and immediate integer. The value of a symbol depends on its type and can be static or dynamic. For example, the value of a register symbol is dynamic and is known only at runtime, whereas the value of an immediate integer symbol is static and is known at compile time. Each symbol in an operation class has a possible set of types T. A general operation class is defined as:

$$\texttt{opClass = <(sym}_0\texttt{, T}_0\texttt{), (sym}_1\texttt{, T}_1\texttt{), ...| exp(sym}_0\texttt{, sym}_1\texttt{, ...)>}$$

where $(\texttt{sym}_i, \texttt{T}_i)$ are *(symbol, types)* pairs and $\texttt{exp(sym}_0\texttt{, sym}_1\texttt{,...)}$ is the behavior of the operations based on the values of the symbols.

The type of a symbol can be defined as a register (∈*Registers*), an immediate constant (∈*Constants*) or a *microoperation* (∈*Operations*). A microoperation is itself an operation class that performs a subfunctionality in another operation class. For example, a data-processing instruction in ARM (e.g., add) uses shift (microoperation) to compute the second-source operand, known as `ShifterOperand`. Each possible type of a symbol is coupled with a mask pattern that determines what bits in that operation must be checked to find out the actual type of the corresponding symbol. Possible types of a symbol are defined as:

$$\texttt{T} = \{(\texttt{t, m}) \mid \texttt{t} \in \texttt{Operations} \cup \texttt{Registers} \cup \texttt{Constants, m} \in (1|0|\texttt{x})^*\}$$

Each mask represents the value of one or more bytes. The right-most byte in the mask represents the LSB and, therefore, bytes from right to left in the mask are compared to bytes from lower to higher addresses in the program binary stream. The little/big endianness of the processors is supported by either ordering the bytes correctly in the masks or updating the decode algorithm to compare bytes in correct order. Pairs of <operation, mask> in a slot or <type, mask> in a set of types are processed from left to right during decode. Therefore, the right mask must not be a subset of the left mask for any two pairs. For example, the opcode symbol in Example 1 (valid integer arithmetic operations) can be described as:

```
OpTypes = {
  (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
  (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx), ...
}
```

Note that, use of a binary mask instead of an integer provides more freedom for describing the operations, because the symbols are not directly mapped to some contiguous bits in the instruction and a symbol can correspond to multiple bit positions in the instruction binary.

The actual register in a processor is defined by its class and its index. The index of a register in an instruction is defined by extracting a slice of the instruction bit pattern and interpreting it as an unsigned integer. An instruction can use a specific register with a fixed index, as in a branch instruction that update the program counter. A register is defined as:

```
r = [regClass, index-function(...)]
```

where index-function() is a function that extracts the actual register index from the instruction binary or returns a constant value. For example, the `dest` symbol (in Example 1) is indicated by 25th to 29th bits in the instruction and is an

```
//Functions
int extract-slice(int hiBit, int loBit) {…}
//
SPARCInst = $(IntegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) | … $;
IntegerOp = < (opcode, OpTypes), (dest, DestType), (src1, Src1Type),
              (src2, Src2Type) | { dest = opcode(src1, src2); } >;
OpTypes = {
  (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
  (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx),
  (Or, xxxx-xxxx 0010-xxxx xxxx-xxxx xxxx-xxxx),
  (And, xxxx-xxxx 0001-xxxx xxxx-xxxx xxxx-xxxx),
  (Xor, xxxx-xxxx 0011-xxxx xxxx-xxxx xxxx-xxxx), …
};
DestType = [IntegerRegClass, extract-slice(29, 25)];
Src1Type = [IntegerRegClass, extract-slice(18, 14)];
Src2Type = {
   ([IntegerRegClass, extract-slice(4,0)], xxxx-xxxx xxxx-xxxx xx0x-xxxx
xxxx-xxxx),
   (#extract-slice(12,0)#, xxxx-xxxx xxxx-xxxx xx1x-xxxx xxxx-xxxx)
};
```

Fig. 5. Integer arithmetic instcutions in SPARC.

integer register. Its type can be described as:

```
DestType = [IntegerRegClass, extract-slice(29, 25)]
```

Similarly a portion of an instruction may be considered as a constant. For example, one bit in an instruction can be equivalent to a Boolean type or a set of bits can make an integer immediate. It is also possible to have constants with fixed values in the instructions. A constant type is defined by:

```
c = #constant-extraction-function(...)#
```

The functions that extract the constants or register indexes are defined with a C-like language. The instruction binary is an implicit parameter of these functions and does not need to appear in the parameter list. The actual body of the function depends on the implementation details of how the binary stream is represented. Figure 5 shows the complete description of the integer-arithmetic instructions in SPARC processor (Example 1).

Figure 6 describes how to capture data-processing instructions of the ARM processor using our instruction model. ARM has complex 32-bit instruction formats that are all conditional. In data-processing operations (`DPOperation`), if the condition (16 possibilities) is true, some arithmetic operation (16 possibilities) is performed on the two source operands and the result is written in the destination operand. The destination and the first source operand are always registers. The second source operand, called `ShifterOperand` (11 possibilities), has three fields: shift operand, shift operation, and shift value. The shift value shows the number of shifts that must be performed on the shift operand by the specified shift operation. For example, the "`ADD r1, r2, r3 SL #10`" is equivalent to "`r1 = r2 + (r3` $\ll$ `10)`" expression. If indicated in the instruction opcode, the flag bits (Z, N, C, and V) are updated. Therefore, $16 \times 16 \times 11 \times 2 = 5632$ formats of instruction binaries are possible for the data-processing instructions. All these formats are covered by the description in Figure 6. In

```
ARMInst = $
   (DPOperation, xxxx-001x xxxx-xxxx xxxx-xxxx xxxx-xxxx) |
   (DPOperation, xxxx-000x xxxx-xxxx xxxx-xxxx xxx0-xxxx) |
   (DPOperation, xxxx-000x xxxx-xxxx xxxx-xxxx 0xx1-xxxx) | …
$;
DPOperation = <
  (cond, Conditions), (opcode, Operations),
  (dest, [intReg, extract-slice(15,12)]),
  (src1, [intReg, extract-slice(19,16)]), (src2, ShifterOperand),
  (updateFlag, {(true, mask(32, 20, "1"), (false, mask(32, 20, "0")})
  | {
   if (cond()) {
    dest = opcode( src1, src2);
    if (updateFlags) {/*Update flags*/}
   }
  }
>;
Conditions = {
   (Equal, mask(32, 31, "0000"), (NotEqual, mask(32, 31, "0001"),
   (CarrySet, mask(32, 31, "0010"), (CarryClear, mask(32, 31, "0011"),
   …,
   (Always, mask(32, 31, "1110"), (Never, mask(32, 31, "1111")
};
Operations = {
   (And, mask(32, 24, "0000"), (XOr, mask(32, 24, "0001"),
   (Sub, mask(32, 24, "0010"), (Add, mask(32, 24, "0100"), …
};
ShifterOperand = <
   (op, {([intReg, extract-slice(11,8)], mask(32,4,"0")),
         (#int, extract-slice(11,7)#, mask(32,7,"0xx1"))}),
   (sh, {(ShiftLeft, mask(32,6,"00)), (ShiftRight, mask(32,6,"01)), …}),
   (val, {([intReg, extract-slice(3,0)], mask(32,25,"0")),
          (#int, extract-slice(7,0)#, mask(32,25,"1"))})
   | { sh(op, val) }
>;
```

Fig. 6. Data processing instructions in ARM.

the remaining sections, we show how all these possibilities are explored for generating an optimized code for each type of instruction. We defined a set of macros that can be used for compact description. For example mask(8,2,"10") macro generates an 8-bit mask that has a "10" at position 2, i.e., xxxx-x10x. In other words, it copies "1" at position 2, and "0" at position 1, and fills the rest with "x". The position number starts from 0 for the right-most bit of the mask and increases from right to left.

In this model, instructions that have similar format are grouped together into one class. Most of the time, this information is readily available from the instruction set architecture manual. For example, we defined six instruction classes for the ARM processor viz., Data Processing, Branch, LoadStore, Multiply, Multiple LoadStore, Software Interrupt, and Swap.

4.1 Detecting the Length of Instructions

In some processors, the instruction set is divided into distinct subsets and usually an internal processor mode determines which subset is used for decoding the instructions. For example, the ARM instruction set consists of a set of 32-bit

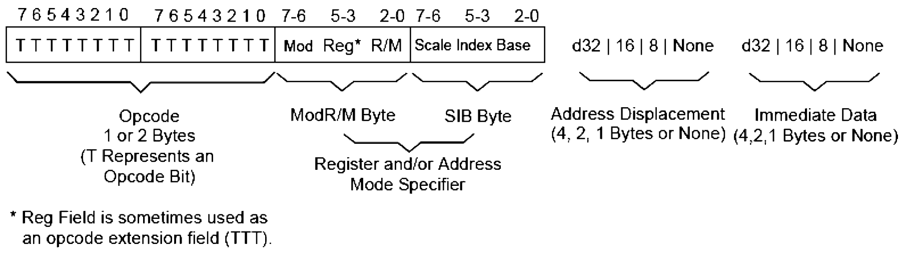
7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
7-6 5-3 2-0
7-6 5-3 2-0
T T T T T T T T
T T T T T T T T
Mod Reg* R/M
Scale Index Base
d32 | 16 | 8 | None
d32 | 16 | 8 | None
Opcode
1 or 2 Bytes
(T Represents an Opcode Bit)
ModR/M Byte
SIB Byte
Register and/or Address Mode Specifier
Address Displacement (4, 2, 1 Bytes or None)
Immediate Data (4,2,1 Bytes or None)
* Reg Field is sometimes used as an opcode extension field (TTT).

Fig. 7. Pentium general instruction format.

Table I. Field Values of ModR/M and Their Effect on Instruction Length

mod	reg	r/m	Description	#Bytes
00	xxx	110	Memory access using 2-byte address displacement	3
00	xxx	xxx	Memory access no address displacement	1
01	xxx	xxx	Memory access using 1-byte address displacement	2
10	xxx	xxx	Memory access using 2-byte address displacement	3
11	xxx	xxx	Register Access	1

instructions and a set of 16-bit instructions, known as Thumb instructions. The Intel IA32 also has a 16-bit and a 32-bit mode, which also determines the size of the constant field of instructions. In such cases, each subset of instruction set can be easily modeled by our instruction model and then during decode, the proper set of instruction specifications is selected and passed to the decode algorithm (described in Section 5).

Our instruction model is also applicable to architectures with complex ISA. The ISA description of such architectures must first detect the actual length of instructions and then capture their field structure and behavior. The previous examples of this section showed how different fields of an instruction are described and then used in the behavior description of the instruction. In the remainder of this section, we show how to detect the length of a group of instructions in the Pentium architecture [IA-32], in order to demonstrate how variable length instructions can be captured in our model.

Figure 7 shows the general instruction format of Pentium [Figure B-1 in IA-32]. The instructions of Pentium may have one or two bytes of opcode, followed by a ModR/M byte. Depending on the opcode and ModR/M field, the instructions may have an SIB byte and one or two constant fields (address displacement and immediate data). The length of each constant field depends on the configuration of the opcode and ModR/M, as well as the processor mode. In the 16-bit mode, instructions do not have an SIB byte and their constant fields are, at most two bytes.

The ModR/M field provides 8 register and 24 possible addressing modes. Table I shows the values of subfields of ModR/M and the number of bytes that is added to the length of instruction in each case. The details of ModR/M field can be found in Table 2-1 in [IA-32].

For each operation, the Pentium ISA has multiple opcodes and different instruction structures to perform that operation on different types of operands. For example, in the 16-bit mode, operations such as *Add*, *Sub*, *Or*, and *Xor* have

Table II. Field Values of ModR/M and Their Effect on Instruction Length

opcode	ModR/M byte	Immediate data bytes	#Bytes
0xxx-00xx	1	0	2+address displacement (if any)
1xxx-00x0	1	1	3+address displacement (if any)
1xxx-00x1	1	2	4+address displacement (if any)
xxxx-0100	0	1	2
xxxx-0101	0	2	3

```
Pentium16bit = $
   …
  (IS1, mask(32, 15, "00xx-x110 0xxx-00xx")) |
  (IS1, mask(16, 15, "00xx-xxxx 0xxx-00xx")) |
  (IS1, mask(24, 15, "01xx-xxxx 0xxx-00xx")) |
  (IS1, mask(32, 15, "10xx-xxxx 0xxx-00xx")) |
  (IS1, mask(16, 15, "11xx-xxxx 0xxx-00xx")) |
  (IS1, mask(40, 15, "00xx-x110 1xxx-00x0")) |
  (IS1, mask(24, 15, "00xx-xxxx 1xxx-00x0")) |
  (IS1, mask(32, 15, "01xx-xxxx 1xxx-00x0")) |
  (IS1, mask(40, 15, "10xx-xxxx 1xxx-00x0")) |
  (IS1, mask(24, 15, "11xx-xxxx 1xxx-00x0")) |
  (IS1, mask(48, 15, "00xx-x110 1xxx-00x1")) |
  (IS1, mask(32, 15, "00xx-xxxx 1xxx-00x1")) |
  (IS1, mask(40, 15, "01xx-xxxx 1xxx-00x1")) |
  (IS1, mask(48, 15, "10xx-xxxx 1xxx-00x1")) |
  (IS1, mask(32, 15, "11xx-xxxx 1xxx-00x1")) |
  (IS1, mask(16, 7, "xxxx-0100")) |
  (IS1, mask(24, 7, "xxxx-0100")) | …
$;
IS1 = <(opcode, {…}), (op1, {…}), (op2, {…}) | { op2 = opcode(op1, op2);
} >;
```

Fig. 8. Detecting the length of a subset of instructions in Pentium.

five different opcode configurations. Table II shows possible opcodes for such operations and their effect on the length of an instruction. This information is extracted from Table B-10 in [IA-32]. We can group these operations and describe them by one OperationClass in our model. We call this OperationClass IS1.

In order to detect the length of instructions described by OperationClass IS1, we need to combine the information of the first three rows of Table I (which have a ModR/M byte) with the rows of Table II. Therefore, there will be 17 masks ($3 \times 5 + 2$) that detect this set of instructions. Once the length of the instructions are detected, the IS1 OperationClass can be described based on its fields. The description of fields may need to consider different combinations of ModR/M byte and its effect on the length. Other than that, the effort for describing the IS1 OperationClass will be similar to that of previous examples. Figure 8 shows the structure of the IS1 OperationClass as well as the masks that detect the length.

In this section, we have demonstrated two key features of our instruction model: first, it is generic enough to capture architectures with complex instruction sets; second, it captures the instructions efficiently by allowing instruction grouping.

5. GENERIC INSTRUCTION DECODER

A key requirement in a retargetable simulation framework is the ability to automatically decode application binaries of different processor architectures. This necessitates a generic decoding technique that can decode the application binaries based on instruction specifications. In this section, we propose a generic instruction-decoding technique that is customizable, depending on the instruction specifications captured through our generic instruction model. Based on our proposed instruction model, the main task of decoding algorithm is to find the actual values of symbols for each individual instance of an instruction binary. This is done by comparing the instruction binary with different mask patterns used in the description.

Algorithm 1: *StaticInstructionDecoder*

```
Input:   Target Program Binary Application,
         Instruction Specifications InstSpec;
Output: Decoded Program DecodedOperations;
Begin
   Addr = Address of first instruction in Application;
   DecodedOperations = {};
   While (Application not processed completely)
     BinStream = Binary stream in Application starting at Addr;
     (Exp, Length) = DecodeOperation (BinStream, InstSpec);
     DecodedOperations = DecodedOperations ∪ <Exp, Addr>;
     Addr = Addr + Length;
   EndWhile;
   return DecodedOperations;
End;
```

Algorithm 1 describes the operation of the *Static Instruction Decoder* shown in Figure 1. This algorithm accepts the target program binary and the instruction specification as inputs and generates a source file containing decoded instructions as output. Iterating on the input binary stream, it finds an operation, decodes it using Algorithm 2, and adds the decoded operation to the output source file. Algorithm 2 also returns the length of the current operation that is used to determine the beginning of the next operation.

Algorithm 2: *DecodeOperation*

```
Input: Binary Stream BinStream, Specifications Spec;
Output: Decoded Expression Exp, Integer DecodedStreamSize;
Begin
   (OpDesc, OpMask) = findMatchingPair(Spec, BinStream);
   OpBinary = initial part of BinStream whose length is equal to OpMask;
   Exp = the expression part of OpDesc;
   ForEach pair of (s, T) in the OpDesc
     Find t in T whose mask matches the OpBinary;
     v = ValueOf (t, OpBinary);
     Replace s with v in Exp;
   EndFor
   return (Exp , size(OpBinary));
End;
```

Algorithm 2 gets a binary stream and a set of specifications containing operation or microoperation classes. The binary stream is compared with the elements of the specification to find the specification-mask pair that matches with the beginning of the stream. The length of the matched mask defines the length of the operation that must be decoded. The type of a symbol is determined by comparing its masks with the binary stream. Finally, using the symbol types, all symbols are replaced with their values in the expression part of the corresponding specification. The resulting expression is the behavior of the operation. This behavior and the length of the decoded operation are produced as outputs.

Algorithm 3: *ValueOf*

```
Input:  Type t, Operation Binary OpBinary;
Output: Extracted Value extValue;
Begin
  Switch (t)
    case Constant:
      extValue = constant-extraction-function(OpBinary);
    endcase
    case Register:
      extValue = REGS[registerClass][index-function(OpBinary)];
    endcase
    case microoperation:
      (extValue, tmp) = DecodeOperation(OpBinary, t);
    endcase
  EndSwitch;
  return extValue;
End;
```

Algorithm 3 gets a symbol type and an operation binary (`OpBinary`) and returns the actual value of the corresponding symbol. If the type itself is a microoperation specification, the *DecodeOperation* function (Algorithm 2) is called again and the result is returned. If the type is a constant, the appropriate function is called to extract the intended portion of the instruction binary and convert it to a constant. If the type is a register, the corresponding function is called to extract the index of the register and then corresponding register is selected.

Example 2 shows the decoding process of an Add instruction in SPARC processor. The Add instruction belongs to the class of integer-arithmetic instructions described in Example 1. In the worst case, each instruction of the input program should be compared with all the mask patterns in all the operation classes of the description. Therefore, the complexity of the decoding algorithm is $O(n \times m)$, where n is the number of operations in the input binary program and m is the number of binary masks in the description.

Example 2 (*Decoding an Add instruction of the SPARC processor*). Consider the following SPARC Add operation example and its binary pattern:

`Add g1, #10, g2`	1000-0100	0000-0000	0110-0000	0000-1010

Using the specifications of Figure 5, in the first line of Algorithm 2, the (IntegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) pair matches with the instruction binary. This means that the `IntegerOps` operation class matches this operation. It calls Algorithm 3 to decode the symbols of `IntegerOps viz. opcode, dest, src1, src2`. Symbol `opcode's` type is `OpTypes` in which the mask pattern of `Add` matches the operation pattern. Thus the value of `opcode` is `Add` function. Symbol `dest's` type is `DestType`, which is a register type. It is an integer register whose index is bits 25th to 29th (00010), i.e., 2. Similarly, the values for the symbols `src1` and `src2` can be computed. By replacing these values in the expression part of the `IntegerOps`, the final behavior of the operation would be: `g2 = Add(g1,10)`, which means `g2 = g1 + 10`.

6. GENERATING EXECUTABLE INSTRUCTION BEHAVIORS

The *DecodeOperation* (Algorithm 2) and *ValueOf* (Algorithm 3) algorithms can be called both statically at compile time and dynamically at runtime. The only difference between the static and dynamic versions of these algorithms is the structure of the expressions that they generate after decoding the instructions. In the dynamic version, the expressions are a combination of pointers to constants, registers, and functions. This is a traditional implementation that other simulators such as JIT-CCS [Nohl et al. 2002] have used before. However, using our instruction model, the instructions can also be statically decoded at compile time. The static version of decode algorithms will generate C++ source code that are optimized at compile time to speed up the execution of instructions during simulation. The basis of the static code generation is described in the next section.

6.1 Generating Optimized Code for Fast Simulation

Typically in simulators, for each instruction, a general piece of code (in the form of a function or switch-case statements) simulates the behavior of the instances of that instruction. However, these instances may have a constant value for a particular field that can be used for further optimizations. For example, 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. By considering these constant (static) values and applying the partial evaluation technique [Futamura 1971], it is possible to generate a customized code for different formats of instructions. To take advantage of such situations, we need separate functions (or *case* statements) for each and every possible format of instructions so that the function can be optimized by the compiler at compile time and produce the best performance at runtime. In our instruction model, all of these formats and their corresponding functions can be constructed by generating all of the permutations of the symbol values in an operation class. The number of generated formats (functions) can be controlled by excluding some of the symbols or iterating only on a subset of symbol values. Controlling the level of optimizations and number of generated formats using the same small description is one of the unique features of our model.

However, generating all the instruction formats of an instruction set may not be feasible in practice. For example, as discussed in Section 4, there are

```
/* extracted template for data processing operations of ARM*/
template<class Conditions, class Operations, class ShifterOperand, bool
updateFlag>
class DPOperations {
 intReg dest, src1;
 ShifterOperand src2;
public:
 …
 virtual void execute() {
  if (Conditions::f()){
   dest = Operations::f(src1, src2);
   if (updateFlag){ Z = (dest == 0); N= (dest < 0); …}
  }
 }
};

/*Customization of execute() method for ADD r1, r2, r3 sl #10 */
DPOperations<Always, Add, ShifterOperand<Reg, ShiftLeft, Const>,
fasle>::execute()
{
 if (Always::f()){
  dest = Add::f(src1, src2);
  if (false){ Z = (dest == 0); N= (dest < 0); …}
 }
}
/*After optimization*/
DPOperations<Always, Add, ShifterOperand<Reg, ShiftLeft, Const>,
fasle>::execute()
{
  dest = src1 + (src2.op << 10);
}
```

Fig. 9. Code generation for an ARM instruction.

5632 possible formats for the data-processing instructions of the ARM processor and generating all these formats, imposes a huge overhead on the compiler. To solve this problem in our framework, we generate the customized code only for the instruction instances of the simulated program. Furthermore, instead of generating distinct functions, we use C++ templates and customize them during decode. We generate a template for each operation or microoperation class specification. For each symbol in the operation class, the corresponding template has a parameter in its parameter list. During the decode phase, these parameters are replaced with the values of the symbols. Finally, during the compilation on host machine, these customized templates are optimized by the compiler. After extracting the templates in this way, we can use IS-CS technique for generating a high-performance simulator. The details of using and optimizing these templates in IS-CS technique are described in Reshadi et al. [2003]. Figure 9 shows the extracted template and its parameters for data-processing instructions in ARM (described earlier in Figure 6).

Note that the decode algorithm, described in Section 5, relies only on the descriptions of instructions to extract the values of symbols corresponding to an instruction instance. These values can be used either dynamically in some conditional statements or statically to generate the source code for appropriate functions. Therefore, although we use the IS-CS technique in this paper, the generic instruction model and the proposed decode algorithm can be coupled with any other existing simulation techniques.

7. EXPERIMENTS

In order to evaluate the applicability of our framework, we modeled two contemporary, yet very different, 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 and validated the generated simulator by comparing traces with Simplescalar-arm [Simplescalar] simulator. 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 and Keppel 1994] simulator. The benchmarks are taken from MiBench [MiBench] (BlowFish, crc), MediaBench [Lee et al. 1997] (adpcm, jpeg, epic, g721), and SPEC95 [SPEC] (compress, go) suites.

To evaluate the instruction model we consider two aspects: the efficiency of the description and the performance of the generated simulator. We first compare the descriptions of our model with that of LISA and Babel in terms of compactness. Next, we present the simulation performance results of running the selected benchmarks on two processor models.

7.1 Efficiency of Description

It took us one man-month to study the manual and generate the corresponding simulator for each processor. This very short generation time was mainly due to three reasons. First, the description of the instructions in our model is very similar to their representation in the architecture manual, and therefore, we needed a simple mapping between manual and the language. Second, the description is very compact and efficient because of extensive reuse. For example, in Figure 6, to add a new instruction, we need to add a small code for an operation and its mask to the `Operations`. This way, we reuse the rest of the description of operations classes that use the `Operations`. Third, since all of the operations in an operation class share the same expression, it is very easy to debug and verify the descriptions. For example, in Figure 6, if the expression of `DPOperations` class works correctly for Add, it will also work well for Sub, and other operations that can be replaced by symbol `opcode`.

Figure 10 shows a sample code in LISA language taken from a recent publication [Nohl et al. 2002]. In this figure, an operation Add with register parameters is described. To describe an Add operation with immediate integer operands the “`OPERATION ADD`” section (lines 5–10) must be repeated again with minor modification. In other words, for every possible addressing mode in the architecture, a separate section for an operation is needed in LISA description. In LISA, a C function is generated in the simulator for the `BEHAVIOR` section of each operation. Therefore, to exploit different instruction formats and generate faster simulation, the formats must be explicitly included in the description. For example, consider the data-processing instructions in ARM (Section 4). In

```
01:  RESOURCE {
02:    PROGRAM_MEMORY byte8 prog_mem[0x0..0x1000];
03:    REGISTER word32 R[1..15];
04:  }
05:  OPERATION ADD {
06:    DECLARE { GROUP dst,src1,src2 = {Register}}
07:    CODING { 0b01011 0b0000 src1 src2 dst}
08:    SYNTAX { "ADD" dst "," src1 "," src2 }
09:    BEHAVIOR{ dst = src1 + src2}
10:  }
11:  OPERATION Register {
12:    DECLARE { LABEL index; }
13:    CODING { index=0bx[4]}
14:    SYNTAX { "R" index }
15:    EXPRESSION { R[index]}
16:  }
```

Fig. 10. LISA sample code for Add operation.

these operations since one of the sources can be a `ShifterOperand` addressing mode, each instruction needs at least 11 `OPERATION` sections, one for each type of `ShifterOperand`. Now consider the optimizations discussed in Section 6.1. Since all ARM instructions are predicated, but the majority of them execute unconditionally in a program, we can generate two formats for each instructions: the conditional version that checks the proper condition and the unconditional one that executes faster. This is only one of the many possible optimizations. Considering 11 formats for the `ShifterOperand` addressing mode and only 2 formats for the optimization, each instruction needs 22 `OPERATION` sections similar to the one presented in lines 5–10 (5 lines) in Figure 10. Finally for 16 instructions in this group, we need at least $16 \times 2 \times 11 \times 5 = 1760$ lines in the LISA description. As shown in Figure 6, in our model, this group of instructions is represented by only three operation classes (`DPOperation`, `Conditions`, and `Operations`) in less than 50 lines.

A similar approach to LISA is used in Babel [Mong and Zhu 2003]. In Babel, a separate section is needed to describe each individual format of an instruction. For example, SPARC description in Babel is more than 2300 lines long, while its description in our model contains less than 400 lines of code.

In addition, in these languages, an instruction can contain only contiguous fields. Therefore, dummy or multiple fields are needed to describe noncontiguous opcodes (as in SPARC) making the description less readable. These extra fields not only increase the size of the description, but also make the decoder inefficient. On the other hand, our model is more natural and does not require such tricks because of the use of bit masks.

7.2 Performance of the Simulator

The JIT-CCS technique relies on LISA description of instructions for generating the instruction decoder and utilizes a software cache for reusing the decoded instructions and improving the simulation speed. To the best of our knowledge, they have reported the best interpretive simulation performance among

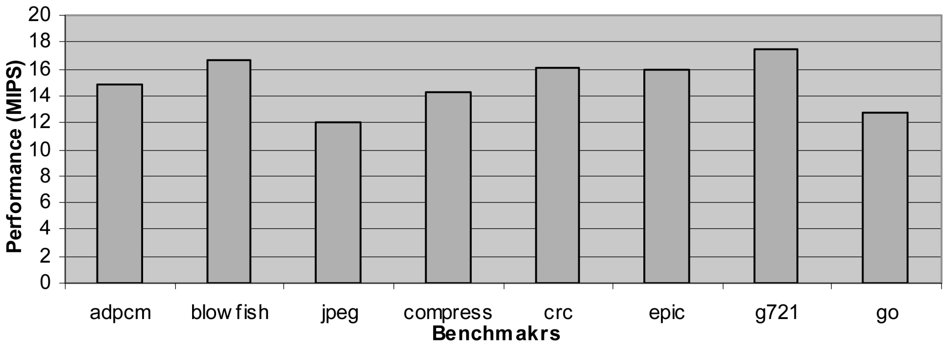

Fig. 11. Simulation results of ARM7 processor.

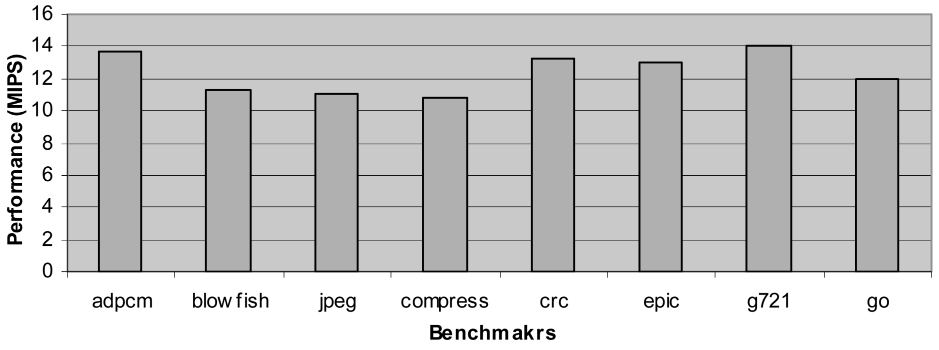

Fig. 12. Simulation results of SPARC processor.

retargetable simulators. For an ARM processor, they have reported a simulation performance of 8 and 7.5 MIPS for *adpcm* and *jpeg* benchmarks, respectively [Nohl et al. 2002]. On a similar machine, the ARM simulator using IS-CS technique [Reshadi et al. 2003] runs at 15 and 12 MIPS for *adpcm* and *jpeg* benchmarks, respectively.

The IS-CS performance results presented in Reshadi et al. [2003] are based on generating the instruction templates manually. We extracted similar instruction templates automatically from our instruction model and generated the simulator. Typically retargetable approaches slow down the simulation. In our framework, since the generation of the simulator and extraction of the templates is completely separate from the simulation engine itself, there are no negative effects imposed on the performance. Figure 11 shows the simulation performance of our technique on ARM7 processor model. The performance results of the retargetable framework are exactly similar to that of manually generated instruction templates in [Reshadi et al. 2003].

Figure 12 shows the simulation performance on SPARC processor model. Note that, the overall performance of ARM simulator is slightly better than that of SPARC. ARM instructions are more complex and, in most cases, are equivalent to more than one SPARC instruction. Therefore, optimizing one ARM instruction is equivalent to optimizing multiple instructions in SPARC. Also simulating SPARC model on a Pentium host machine requires a data encoding translation (Big-Endian to Little-Endian) that causes performance degradation.

The proposed instruction model results in compact, readable, easy to debug and easy to maintain descriptions. In addition, the decode algorithm has no negative effect on the execution method of instructions. As a result, the performance results of the retargetable framework using IS-CS technique are exactly the same as that of former nonretargetable simulators.

8. CONCLUSION

In this paper, we presented a retargetable framework for generating fast and flexible instruction set simulators. We proposed a generic instruction model as well as a generic decode algorithm that can specify and decode many variations of instruction binary formats with various complexities. We demonstrated the applicability of the approach on two radically different architectures, viz., ARM and SPARC processors. Use of symbols in the generic instruction model enables maximum reuse of descriptions among operations, and results in very compact descriptions. It also simplifies the debug and verification of the description. Furthermore, describing instructions in our generic model is very simple because of its similarity with the architecture manual.

The instruction set described using our generic instruction model is an order of magnitude smaller in size than other languages, such as LISA and Babel. To achieve high-performance simulation, we have integrated the IS-CS simulation technique in our retargetable framework by automatically extracting the required templates for simulating the instructions. Since the generation of the simulator is completely separate from the simulation engine, we can incorporate any other fast simulation techniques without any performance penalty. Future work will concentrate on using this framework for cycle accurate simulation of complex architectures including reconfigurable platforms.

REFERENCES

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

Bashford, S., Bieker, U., Harking, B., Neumann, A. and Voggenauer, D. 1994. The MIMOLA Language V4.1. *Technical Report, University of Dortmund, Dept. of Computer Science*.

Cmelik, B., and Keppel, D. 1994. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review 22*, 1, 128–137.

Engel, F., Nührenberg, J., and Fettweis, G. P. 1999. A generic tool set for application specific processor architectures. *The Eighth International Workshop on Hardware/Software Codesign*. 126–130.

Freericks, M. 1991. “The nML Machine Description Formalism.” *Tech. Rep. 1991/15, TU Berlin, Fachbereich Informatik*.

Futamura, Y. 1971. Partial evaluation of computation process—An approach to a compiler–compiler. *Systems, Computers, Controls 2*, 5, 45–50.

Gyllenhaal, J. C., Hwu, W. W., and Rau, B. R. 1996. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3. University of Illinois at Urbana-Champaign, Urbana-Champaign, IL.

Hadjiyiannis, G., Hanono, S., and Devadas, S. 1997. ISDL: An instruction set description language for retargetability. *Design Automation Conference* (*DAC*).

Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *Design Automation and Test in Europe* (*DATE*).

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. *Design Automation Conference* (*DAC*).

IA-32 Intel® *Architecture Software Developer's Manual*, Vol. 2: Instruction Set Reference, http://www.intel.com.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (Micro 30)*.

LEUPERS, R., ELSTE, J., AND LANDWEHR, B. 1999. Generation of interpretive and compiled instruction set simulators. In *Proceeding of Asian-Pacific Design Automation Conference*, Hong Kong.

MiBench benchmarks available at http://www.eecs.umich.edu/mibench

MONG, W. S. AND ZHU, J. 2003. A retargetable micro-architecture simulator. *Design Automation Conference* (*DAC*). 752–757.

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. *Design Automation Conference* (*DAC*). 22–27.

PAULIN, P. G., LIEM, C., MAY, T. C., AND SUTARWALA, S. 1997. Flexware: A flexible firmware development environment for embedded systems. *Code Generation for Embedded Processors*. 67–84.

RESHADI, M., MISHRA, P., AND DUTT, N. 2003. Instruction-set compiled simulation: A technique for fast and flexible instruction-set simulation. *Design Automation Conference* (*DAC*). 758–763.

SCHNARR, E. AND LARUS, J. R. 1998. Fast out-of-order processor simulation using memoization. In *Proceedings of ASPLOS98*, San Jose CA. 283–294.

SCHNARR, E., HILL, M. D., AND LARUS, J. R. 2001. Facile: A language and compiler for high-performance processor simulators. *Programming Language Design and Implementation* (*PLDI*). 321–351.

Simplescalar Home page: http://www.simplescalar.com

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

SPEC benchmarks available at http://www.specbench.org

Trimaran Home page: http://www.trimaran.org

VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. 2002. Microarchitectural exploration with Liberty. *International Symposium on Microarchitecture*.

ZHU, J. AND GAJSKI, D. D. 1999. A retargetable, ultra-fast instruction set simulator. *Design Automation and Test in Europe* (*DATE*). 298–302.

Received September 2003; revised January 2005; accepted July 2005