An Efficient Retargetable Framework for Instruction-Set Simulation

Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, Nikil Dutt
Center for Embedded Computer Systems, University of California, Irvine.
{reshadi, nbansal, pmishra, dutt}@cecs.uci.edu

ABSTRACT

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. 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 loosing performance. 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

Keywords: Retargetable Instruction-Set Simulation, Generic Instruction Model, Instruction Binary Encoding, Decode Algorithm, Architecture Description Language.

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 all features of applications and processors, e.g. runtime self modifying codes, multi mode 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 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 run time, the model should provide as much static information as possible about the architecture and its instruction set.

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 architectures but are slow, while others use some architectural or domain specific performance improvements but support only a limited range of processors. Also in some description languages, deriving a fast simulator requires lengthy descriptions of all possible formats of instructions.

In this paper, we present a retargetable simulation framework that supports many variations of architectures with any instruction-set complexity 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 differing contemporary processors. This model can also be used to exploit all possible instruction formats to generate optimized code for them. We use this generic model to capture the behavior and binary encoding of the instructions. The EXPRESSION Architecture Description Language (ADL) [1] 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 [2] to generate fast and flexible simulators by automatically generating the instruction templates from the descriptions. Our retargetable framework is generic enough to incorporate any other simulation optimization technique as well.

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 the retargetable simulation framework. It describes three key components of the framework: a generic instruction model, a decoding algorithm, and the simulation code generation. Section 0 compares the efficiency of the instruction model with other architecture description languages and presents simulation performance results using two contemporary processor architectures: ARM7 and SPARC. Section 5 concludes the paper.

2. RELATED WORK

An extensive body of recent work has addressed instruction-set architecture simulation. A fast and retargetable simulation technique is presented in [3]. It improves traditional static compiled simulation by mapping the target machine registers to the host machine registers through a low level code generation interface. Retargetable simulators based on an ADL have been proposed within the framework of FACILE [4], Sim-nML [5], ISDL [6], and MIMOLA [7]. The proposed ADL in ISDL is mainly suitable for assembly/binary code generation. FACILE is optimized for out-of-order processor simulation; Sim-nML only supports DSP processors. FLEXWARE Simulator [8] uses a VHDL model of a generic parameterizable model. SimC [9] is based on a machine description in ANSI C. It uses compiled simulation and has limited retargetability. Babel [10] was originally designed for retargeting the binary tools and has been recently used for retargeting the SimpleScalar simulator [11].

The just-in-time cache compiled simulation (JIT-CCS) [12] technique, the closest to our approach, combines some 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 and efficient support of complex instruction formats requires extensive coding in this language.

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

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 general but slow or very fast but restricted to some architecture domain. However, the instruction model significantly affects the complexity of decode and the quality of execution. We have developed a generic instruction model coupled with a simple decoding algorithm that lead to an efficient and flexible execution of decoded instructions.

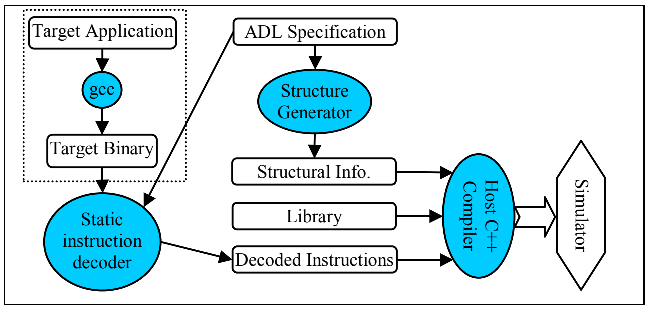

Figure 1- Generating the simulator from ADL

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 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 3.1. Using the instruction specifications from ADL, the *Static Instruction Decoder* decodes the target program one instruction at a time, as described in Section 3.2. It then generates the optimized source code of the decoded instructions (Section 3.3), that is loaded in the instruction memory.

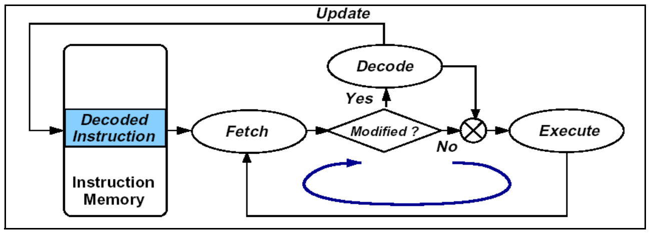

Figure 2- Simulation Engine Flow

The S*tructure 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 run time.

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 get the final ISA simulator. Figure 2 shows the flow of the simulation engine. This engine fetches the decoded instructions from the instruction memory and executes them. If the simulator detects that the program code of a previously executed address has changed it initiates a re-decoding and then updates the instruction memory. The simulation engine is specified by the *Library* component. Therefore, by modifying this component, we can integrate other simulation techniques and optimizations in our retargetable framework.

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

3.1 Generic Instruction Model

A major challenge in retargetable simulation is the ability to capture a wide variety of instructions. We propose an instruction model that is generic enough to capture 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 model the integer arithmetic instructions of the Sparc V7 processor. The completer description is shown in Figure 3.

Example 1: Sparc V7 [15] is a single-issue processor with 32-bit instruction. The integer-arithmetic instructions, *IntegerOps* (as shown below), perform certain arithmetic operation on two source operands and write the result to the destination operand. 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.

In this model, an *instruction* of a processor is composed of a series of *slots*, $I=<sl_0, sl_1,\ldots>$, and each slot contains only one *operation* from a subset of operations. All the operations in an instruction execute in parallel. Each operation is distinguished by a mask pattern. Therefore, each slot (sl_i) contains a set of operation-mask pairs (op_i, m_i) and is defined in the following format. The length of an operation is equal to the length of mask pattern.

$$sl_i=<(op_i^0, m_i^0) \mid (op_i^1, m_i^1) \mid \ldots>$$

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 *clops*: $sl_i=<(clOps_i, m_i)>$. For example, integer arithmetic instructions in Sparc V7 can be grouped in a class (IntegerOps) as shown below:

I_{SPARC} = <(*IntegerOps*, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx)| … >

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 will be: $dest = f_{opcode}(src1, src2)$. 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 run time, whereas the value of an immediate integer symbol is static and is known at compile time. Each symbol in an operation has a possible set of types. A general operation class is then defined as:

$$clOps=<(s_0, T_0), (s_1, T_1), \ldots \mid exp(s_0, s_1, \ldots)>$$

where *(s_i, T_i)* are *(symbol, type)* pairs and *$exp(s_0, s_1,\ldots)$* 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*) or an immediate constant (∈ *Constants*) or can be based on certain *micro-operations* (∈ *Operations*). For example, a data processing instruction in ARM (e.g., add) uses shift (micro-operation) 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:

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

For example, the opcode symbol in Example 1 can be any of valid integer arithmetic operations and 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 this provides more freedom for describing the operations because here 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 also use a specific register with a fixed index, as in a branch instruction that update the program counter. A register is defined by:

$$r= [regClass, i, j] \mid [regClass, index]$$

where *i* and *j* define the boundary of index bit slice in the instruction. For example, the *dest* symbol (in Example 1) is from 25th to 29th bits in the instruction, and is an integer register. Its type can be described as:

DestType = [IntegerRegClass, 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 $c = \#type, i, j\# \mid \#type, value\#$ where *i* and *j* show the bit positions of the constant and *type* is a scalar type such as integer, Boolean, float, etc.

```
SPARCInst = $
   (InegerOps, 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, 29, 25];
Src1Type = [IntegerRegClass, 18, 14];
Src2Type = {
   ([IntegerRegClass,4,0], xxxx-xxxx xxxx-xxxx xx0x-xxxx xxxx-xxxx),
   (#int,12,0#, xxxx-xxxx xxxx-xxxx xx1x-xxxx xxxx-xxxx)
};
```

Figure 3- Integer arithmetic instcutions in SPARC

Figure 3 shows the complete description of integer-arithmetic instructions in SPARC processor (Example 1). Figure 4 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, has three fields: shift operand (register/immediate), shift operation (5 types) and shift value (register/immediate). 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 << 10)*” expression. If indicated in the instruction opcode, the flag bits (Z, N, C, and V) are updated. Therefore, 16x16x(2x5x2)x2=10240 formats of instructions binaries are possible in this class of instructions. All these formats are covered by the description of Figure 4. In the next 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 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.

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.

```
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,15,12]),
   (src1, [intReg,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,11,8], mask(32,4,"0")), (#int,11,7#, mask(32,7,"0xx1"))}),
   (sh, {(ShiftLeft, mask(32,6,"00)), (ShiftRight, mask(32,6,"01)), …}),
   (val, {([intReg,3,0], mask(32,25,"0")), (#int,7,0#, mask(32,25,"1"))})
   | {  sh(op, val)  }
>;
```

Figure 4- Data processing instructions in ARM

3.2 Generic Instruction Decoder

A key requirement in a retargetable simulation framework is the ability to automatically decode application binaries of different processors 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.

```
Algorithm 1: StaticInstructionDecoder
Input: Target Program Binary Appl, Instruction Specifications InstSpec;
Output: Decoded Program DecodedOperations;
Begin
      Addr = Address of first instruction in App;    DecodedOperations={};
   While (Appl not processed completely)
      BinStream = Binary stream in Appl starting at Addr;
      (Exp, AddrIncrement) = DecodeOperation (BinStream, InstSpec);
       DecodedOperations  = DecodedOperations  U <Exp, Addr>;
      Addr = Addr + AddrIncrement;
   EndWhile;
   return  DecodedOperations ;
End;
```

Algorithm 1 describes how *Static Instruction Decoder* of Figure 1 works. 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 gets a binary stream and a set of specifications containing operation or micro-operation 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 types of symbols are determined by comparing their 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 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;
```

Consider the following SPARC Add operation example and its binary pattern:

Add g1, #10, g2	31 1000-0100	23 0000-0000	15 0110-0000	7 0000-1010

Using the specifications of Figure 3, in the first line of Algorithm 2, the (InegerOps, 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.*

```
Algorithm 3: ValueOf
Input: Type t, Operation Binary OpBinary;
Output: Extracted Value extValue;
Begin
   Switch (t)
      case #type, value#: extValue = (type) value; endcase
      case #type, i, j#: extValue = (type) OpBinary[i:j]; endcase
      case [regClass, index]: extValue = REGS[regClass][index]; endcase
      case [regClass, i, j]: extValue = REGS[regClass][ OpBinary[i:j]]; endcase
      case Operation Spec: (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 micro-operation specification, the decode algorithm (Algorithm 2) is called again and the result is returned. If the type is not a fixed constant (register), the value is calculated by interpreting the proper portion of the operation binary (*OpBinary[i:j]*) as a constant (register index).

In the previous example, the four symbols (*opcode, dest, src1, src2*) are decoded using Algorithm 3. Symbol *opcode*'s type is *OpTypes* in which the mask pattern of *Add* matches the operation pattern. So 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.

The complexity of the decoding algorithm is $O(n{*}m{*}log_2m)$, where *n* is the number of operations in the input binary program and *m* is the number of operations in the instruction set.

3.3 Generating Fast Code for Simulators

Typically in simulators, for each instruction of the simulated architecture instruction set, there is a general piece of code (in form of a function or switch-case statements) that 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 [14], 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 run time. 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 of the instruction formats of an instruction set may not be feasible in practice. For example, as discussed in section 3.1, there are 10240 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 micro-operation 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 [2]. Figure 5 shows the extracted template and its parameters for data-processing instructions in ARM, described in Figure 4.

```
/* 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);  …}
      }
   }
};
```

Figure 5- Code generation for an ARM instruction

Note that the decode algorithm, described in Section 3.2, relies only on the descriptions of instructions to extracts the values of symbols corresponding to an instruction instance. These values can be used either dynamically in some conditional statements or statically to generated the source code of proper functions. Therefore the generic instruction model and the proposed decode algorithm can be used in any simulation technique and does not depend on IS-CS technique.

4. EXPERIMENTS

In order to evaluate the applicability of our framework, we modeled two contemporary, yet very different, processors: ARM7[16] and SPARC [15] 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 [11] 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* [13] simulator. We have used benchmarks from SPEC 95 and DSP domains. In this section we show the results using three application programs: *adpcm*, *099.go* and *129.compress*.

4.1 Efficiency of description

It took us one man-month for each processor to study the manual and generate the corresponding simulator. This very short generation time was mainly because of the three following reasons. First, the description of 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 descriptions are very compact and efficient. For example by adding a very small code for an operation and its mask to the *Operations* class in the Figure 4, we can add a new instruction and reuse the rest of the description. 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 4 if the expression of *DPOperations* class works correctly for *Add*, it will also work well for *Sub* and other operations that can be replaced with symbol *opcode.*

```
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]}}
```

Figure 6- LISA sample code for Add operation

Figure 6 shows a sample code in LISA language taken from a recent publication [12]. 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 slight 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 3.1). In these operations since one of the sources can be a ShifterOperand addressing mode, each instruction needs at least five OPERATION sections. Now consider the optimizations discussed in Section 3.3. Since all ARM instructions are predicated but 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 possible optimizations. Considering five formats for the ShifterOperand addressing mode and only two formats for the optimization, each instruction needs ten OPERATION sections similar to the one presented in lines 5-10 in Figure 6. Finally for sixteen instructions in this group we need at least 16x10x5=800 lines in the LISA description. As shown in Figure 4, in our mode, 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 [10]. 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 with our model contains less than 400 lines of code.

Additionally, in these languages, the instruction can contain only contiguous fields. Therefore dummy or multiple fields are needed to describe non-contiguous opcodes (as in Sparc). These extra fields not only increase the size of description but also make the decoder inefficient. On the other hand, since in our model we use bit masks, the descriptions are more natural and does not require such tricks in similar cases.

4.2 Performance of the simulator

The IS-CS performance results presented in [2] 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.

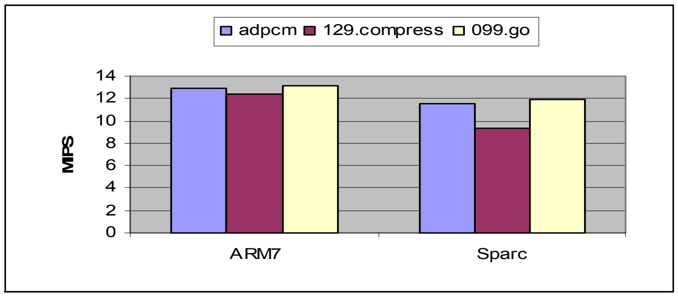

Figure 7- Simulation Results of ARM7 & Sparc processors

Figure 7 shows the simulation performance of our technique on both ARM7 and Sparc processor models. The performance results of the retargetable framework are exactly similar to that of IS-CS. We achieved 30%-50% performance improvement compared to best published results in this category of simulators. 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).

5. SUMMARY

In this paper, we presented a retargetable framework for generating fast and flexible ISA simulator. 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 any complexity. 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 whole description. Furthermore, describing instructions in our generic model is very simple due to its similarity with the architecture manual. The ISA 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 technique without loosing performance. Future work will concentrate on using this framework for cycle accurate simulation of complex architectures including reconfigurable platforms.

6. ACKNOWLEDGMENTS

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

7. REFERENCE

[1] A.Halambi et al. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. DATE, 1999.

[2] M.Reshadi et al, Instruction-Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation, DAC, 2003.

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

[4] E.Schnarr et al. Facile: A language and compiler for high-performance processor simulators. PLDI, Jun. 2001.

[5] M.Hartoog et al. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. DAC, 1997.

[6] G.Hadjiyiannis et al. ISDL: An instruction set description language for retargetability. In Proc. DAC, 1997.

[7] R.Leupers et al. Generation of Interpretive and Compiled Instruction Set Simulators. DAC, 1999.

[8] P.Paulin et al. FlexWare: A flexible firmware development environment for embedded systems. In Proc. Dagstuhl Code GenerationWorkshop, 1994.

[9] F.Engel et al. A generic tool set for application specific processor architectures. (CODES), May 2000.

[10] W.S.Mong et al, A Retargetable Micro-architecure Simulator. DAC, 2003.

[11] Simplescalar Home page: http://www.simplescalar.com

[12] A.Nohl et al. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. DAC, 2002.

[13] R.F.Cmelik et al. Shade: A fast instruction set simulator for execution profiling. ACM SIGMETTRICS Conference on Measurment and Modeling of computer systems, Philadelphia, 1996.

[14] Y. Futamura. Partial Evaluation of Computation Process: an Approach to a Compiler-Compiler. Systems, Computers, Controls, 1971.

[15] Sparc Version 7 Instruction set manual: http://www.sun.com

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