# Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation

Mehrdad Reshadi Prabhat Mishra Nikil Dutt

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA
(reshadi, pmishra, dutt)@cecs.uci.edu
http://www.cecs.uci.edu/~aces

## ABSTRACT

Instruction set simulators are critical tools for the exploration and validation of new programmable architectures. Due to increasing complexity of the architectures and time-to-market pressure, performance is the most important feature of an instruction-set simulator. Interpretive simulators are flexible but slow, whereas compiled simulators deliver speed at the cost of flexibility. This paper presents a novel technique for generation of fast instruction-set simulators that combines the benefit of both compiled and interpretive simulation. We achieve fast instruction accurate simulation through two mechanisms. First, we move the time-consuming decoding process from run-time to compile time while maintaining the flexibility of the interpretive simulation. Second, we use a novel *instruction abstraction* technique to generate aggressively optimized decoded instructions that further improves simulation performance. Our *instruction set compiled simulation* (IS-CS) technique delivers upto 40% performance improvement over the best known published result that has the flexibility of interpretive simulation. We illustrate the applicability of the IS-CS technique using the ARM7 embedded processor.

## Categories and Subject Descriptors

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

## General Terms

Design, Performance

## Keywords

Compiled Simulation, Interpretive Simulation, Instruction Set Architectures, Instruction Abstraction


*DAC 2003,* June 2–6, 2003, Anaheim, California, USA.


## 1. INTRODUCTION

An *instruction-set simulator* is a tool that runs on a *host* machine to mimic the behavior of running an application program on a *target* machine. Instruction-set simulators are indispensable tools in the development of new programmable architectures. They are used to validate an architecture design, a compiler design, as well as to evaluate architectural design decisions during design space exploration.

Traditional interpretive simulation is flexible but slow. In this technique, an instruction is fetched, decoded, and executed at run time as shown in Figure 1. Instruction decoding is a time consuming process in a software simulation.

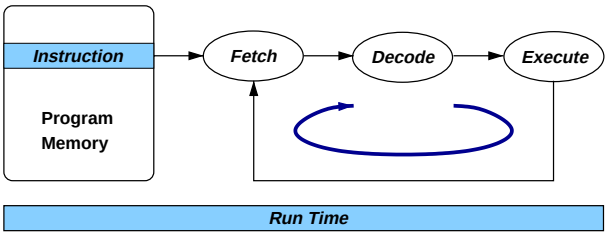


**Figure 1: Traditional Interpretive Simulation Flow**

Compiled simulation performs compile time decoding of application program to improve the simulation performance as shown in Figure 2. To improve the simulation speed further, static compilation based techniques move the instruction scheduling into the compilation phase [4]. However, all compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is further more run-time static. Due to this assumption many application domains are excluded from the utilization of compiled simulators. For example, embedded systems that use external program memories can not use compiled simulators since the program code is not predictable prior to runtime. Similarly, compiled simulators are not applicable in embedded systems that use processors having multiple instruction sets. These processors can switch to a different instruction set mode at run time. For instance, the ARM processor uses the *Thumb* (reduced bit-width) instruction set to reduce power and memory consumption. This dynamic switching of instruction set modes cannot be considered by a simulation compiler, since the selection depends on run-time values and is not predictable. Furthermore, applications with run-time dynamic program code, as pro-

vided by operating systems (OS), can not be addressed by compiled simulators.

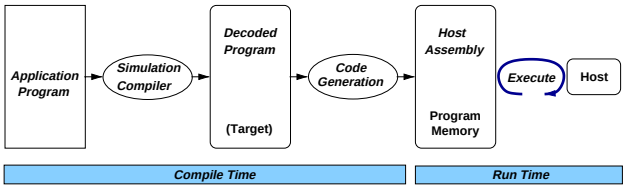


**Figure 2: Traditional Compiled Simulation Flow**

Due to the restrictiveness of the compiled technique, interpretive simulators are typically used in embedded systems design flow. This paper presents a novel technique for generation of fast instruction-set simulators that combines the performance of traditional compiled simulation with the flexibility of interpretive simulation. Our *instruction set compiled simulation* (IS-CS) technique achieves high performance due to two reasons. First, the time consuming instruction decoding process is moved to compile time while maintaining the flexibility of interpretive simulation. In case an instruction is modified at run-time, the instruction is re-decoded prior to execution. Second, we use an *instruction abstraction* technique to generate aggressively optimized decoded instructions that further improve simulation performance. The IS-CS technique delivers better performance than other published simulation techniques that have the flexibility of interpretive simulation. The simulation performance of the IS-CS technique is upto 40% better than the best known results [1] in this category.

The rest of the paper is organized as follows. Section 2 presents related work addressing instruction-set simulation techniques. The instruction set compiled simulation (IS-CS) technique is presented in Section 3. Section 4 presents simulation results using the ARM7 architecture, a commonly used embedded processor. Section 5 concludes the paper.

## 2. RELATED WORK

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

Simplescalar [3] is a widely used interpretive simulator that does not have any performance optimizations for functional simulation.

Shade [5], Embra [10] and FastSim [8] simulators use dynamic binary translation and result caching to improve simulation performance. Embra provides the highest flexibility with maximum performance but is not retargetable: it is restricted to the simulation of the MIPS R3000/R4000 architecture.

A fast and retargetable simulation technique is presented in [6]. It improves traditional static compiled simulation by aggressive utilization of the host machine resources. Such utilization is achieved by defining a low level code generation interface specialized for ISA simulation, rather than the traditional approaches that use C as a code generation interface.

Retargetable fast simulators based on an Architecture Description Language (ADL) have been proposed within the framework of FACILE [9], Sim-nML [12], ISDL [14], MIMOLA [16], ANSI C [11], LISA ([1], [2], [4]), and EXPRESSION [15]. The simulator generated from a FACILE description utilizes the *Fast Forwarding* technique to achieve reasonably high performance. All of these simulation approaches assumes that the program code is run-time static.

In summary, none of the above approaches (except [1]) combines retargetability, flexibility, and high simulation performance at the same time. A *just-in-time cache compiled simulation* (JIT-CCS) technique is presented in [1]. The objective of the JIT-CCS technique is similar to the one presented in this paper - combining the full flexibility of interpretive simulators with the speed of the compiled principle. The JIT-CCS technique integrates the simulation compiler into the simulator. The compilation of an instruction takes place at simulator run-time, *just-in-time* before the instruction is going to be executed. Subsequently, the extracted information is stored in a simulation cache for direct reuse in a repeated execution of the program address. The simulator recognizes if the program code of a previously executed address has changed and initiates a re-compilation. This technique makes an assumption to get performance closer to complied simulation: the number of repeatedly executed instructions should be very large such that 90% of the execution time is spent in 10% of the code. This assumption may not hold true for all real world applications. For example, the *176.gcc* benchmark from SPEC CPU2000 violates this rule.

We propose an *instruction set compiled simulation* (IS-CS) technique where the program is compiled prior to run time and executed interpretively as shown in Figure 3. The simulator recognizes if the program code of a previously executed address has changed and initiates a re-decoding. We achieve both the performance of compiled simulation and flexibility of interpretive simulation. The simulation performance of the IS-CS technique is upto 40% better than the best known result [1] in this category. There are two reasons for its superior performance. First, the time consuming instruction decoding process is moved to compile time while maintaining the flexibility of interpretive simulation. Second, we use a novel *instruction abstraction* technique to generate aggressively optimized decoded instructions that further improve simulation performance.

## 3. INSTRUCTION SET COMPILED SIMULATION

We developed the *instruction set compiled simulation* (IS-CS) technique with the intention of combining the full flexibility of interpretive simulation with the speed of the compiled principle. The basic idea is to move the time-consuming instruction decoding to compile time as shown in Figure 3. The application program, written in C/C++, is compiled using the *gcc* compiler configured to generate binary for the target machine. The *instruction decoder* decodes one binary instruction at a time to generate the decoded program for the input application. The decoded program is compiled by C++ compiler and linked with the simulation library to generate the simulator. The simulator recognizes if the previously decoded instruction has changed and initiates re-decoding of the modified instruction. If any instruction

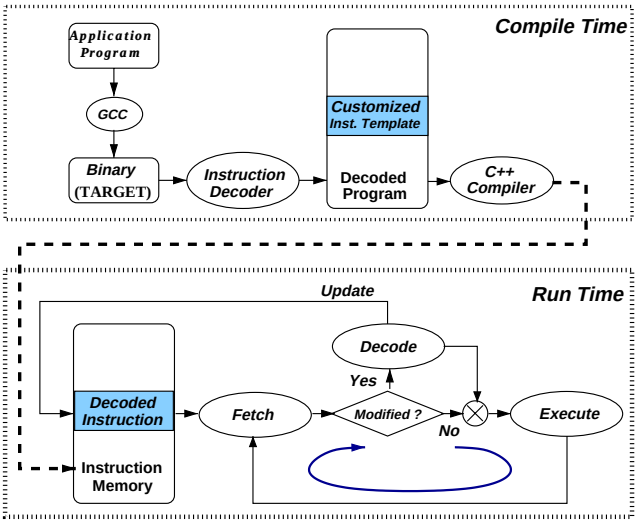


**Figure 3: Instruction Set Compiled Simulation Flow**

is modified during execution and subsequently re-decoded, the location in *instruction memory* is updated with the re-decoded instruction. To improve the simulation speed we use a novel *instruction abstraction* technique that generates optimized decoded instructions as described in Section 3.1. As a result the computation during run-time is minimized. This technique achieves the speed of compiled simulation due to compile-time decoding of application as described in Section 3.2. Section 3.3 describes the simulation engine that offers the full flexibility of interpretive simulation.

## 3.1 Instruction Abstraction

In traditional interpretive simulation (e.g., Simplescalar [3]) the decoding and execution of binary instructions are done using a single monolithic function. This function has many if-then-else and switch/case statements that perform certain activities based on bit patterns of opcode, operands, addressing modes etc. In advanced interpretive simulation (e.g., LISA [1]) the binary instruction is decoded and the decoded instruction contains pointers to specific functions. There are many variations of these two methods based on efficiency of decode, complexity of implementation, and performance of execution. However, none of these techniques exploit the fact that a certain class of instructions may have a constant value for a particular field of the instruction. For example, a majority of the ARM instructions execute unconditionally (condition field has value *always*). It is a waste of time to check the condition for such instructions every time they are executed.

Clearly, when certain input values are known for a class of instructions, the *partial evaluation* [13] technique can be applied. The effect of partial evaluation is to specialize a program with part of its input to get a faster version of the same program. To take advantage of such situations we need to have separate functions for each and every possible format of instructions so that the function could be optimized by the compiler at compile time and produce the best performance at run time. Unfortunately, this is not feasible in practice. For example, consider the ARM data processing instructions. It can have 16 conditions, 16 operations, an update flag (true/false), and one destination followed by two source operands. The second source operand, called shifter operand, has three fields: operand type (reg/imm), shift options(5 types) and shift value (reg/imm). In total, the ARM data processing instructions have 16 x 16 x 2 x 2 x 5 x 2 = 10240 possible formats.

Our solution to this problem is to define instruction classes, where each class contains instructions with similar formats. 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. Next, we define a set of masks for each instruction class. The mask consists of '0', '1' and 'x' symbols. A '0' ('1') symbol in the mask matches with a '0' ('1') in binary pattern of the instruction at the same bit position. An 'x' symbol matches with both '0' and '1'. For example, the masks for the data processing instructions are shown below:

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

We use C++ templates to implement the functionality for each class of instructions. For example, the pseudo code for the data processing template is shown below. The template has four parameters viz., *condition*, *operation*, *update flag*, and *shifter operand*. The shifter operand is a template having three parameters viz., *operand type*, *shift options* and *shift value*.

```
Example 1: Template for Data Processing Instructions

template <class Cond, class Op, class Flag, class SftOper>
class DataProcessing :
{
    SftOper _sftOperand;
    Reg _dest, _src1;

    public:
      .......
      virtual void execute()
      {
        if (Cond::execute())
        {
          _dest = Op::execute(_src1, _sftOperand.getValue());
          if (Flag::execute())
          {
            // Update Flags
            .......
          }
        }
      }
};
```

We also use a *Mask Table* for the mapping between mask patterns and templates. It also maintains a mapping between mask patterns and functions corresponding to those templates.

This *instruction abstraction* technique is used to generate aggressively optimized decoded instructions as described in Section 3.2.

## 3.2 Instruction Decoder

Algorithm 1 decodes one binary instruction at a time to generate the decoded program for the input application. For each instruction in the application program it selects the appropriate template using Algorithm 2. It generates a customized template for the instruction using the appropriate parameter values. Algorithm 3 briefly describes the

customized template generation process. Finally, the customized template is instantiated and appended in the temporary program *TempProgram*. The *TempProgram* is fed to a C++ compiler that performs necessary optimizations to take advantage of the partial evaluation technique, described in Section 3.1, to produce the *DecodedProgram*. The *DecodedProgram* is loaded into *instruction memory* which is a separate data structure than main memory. While the main memory holds the original program data and instruction binaries, each cell of instruction memory holds a pointer to the optimized functionality as well as the instruction binary. The instruction binary is used to check the validity of the decoded instruction during run-time.

Algorithm 3 describes the template customization process. The algorithm's basic idea is to extract the values from specific fields of the binary instruction (e.g., opcode, operand etc.) and assign those values to the template. We maintain the information for each class of instructions, templates, field formats, and mask patterns. These information can be derived from the processor specification described using an Architecture Description Language such as LISA [1], EXPRESSION [7] and nML [12].

**Algorithm 1**: *Instruction Decoding*
**Inputs**: Application Program *Appl* (Binary), MaskTable *maskTable*.
**Output**: Decoded Program *DecodedProgram*.
**Begin**
TempProgram = {}
**foreach** binary instruction *inst* with address *addr* in *Appl*
*template* = DetermineTemplate(*inst*, *maskTable*)
$template_{inst}$ = CustomizeTemplate(*template*, *inst*)
newStr = "InstMemory[*addr*] = **new** $template_{inst}$"
*TempProgram* = AppendInst(TempProgram, newStr)
**endfor**
*DecodedProgram* = Compile(*TempProgram*)
**End**

**Algorithm 2**: *DetermineTemplate*
**Inputs**: Instruction *inst* (Binary), and Mask Table *maskTable*.
**Output**: Template.
**Begin**
**foreach** entry $< mask, template >$ in *Mask Table*
**if** *mask* matches *inst* **return** *template*
**endfor**
**End**

**Algorithm 3**: *CustomizeTemplate*
**Inputs**: Template *template*, Instruction *inst* (Binary).
**Output**: Customized Template with Parameter Values.
**Begin**
**switch** instClassOf(*inst*)
**case** Data Processing:
**switch** (inst[31:28])
**case** 1110: *condition* = *Always* **endcase**
**case** ....
...
**endswitch**
**switch** (inst[24:21])
**case** 0100: *opcode* = ADD; **endcase**
**case** ....
...
**endswitch**
.....
**return** $template < condition, opcode, ..... >$
**endcase** /* Data Processing */
**case** Branch: ... **endcase**
**case** LoadStore: ... **endcase**
**case** Multiply: ... **endcase**
**case** Multiply LoadStore: ... **endcase**
**case** Software Interrupt: ... **endcase**
**case** Swap: ... **endcase**
**endswitch**
**End**

We illustrate the power of our technique to generate an optimized decoded instruction using a single data processing instruction. We show the binary as well as the assembly of the instruction below.

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

Assembly: ADD           r1, r2, r3 LSL #10
         (op{<cond>}{S} Rd, Rn, Rm shift #<immed>)
```

The *DetermineTemplate* function returns the *DataProcessing* template (shown in Example 1) for this binary instruction. The *CustomizeTemplate* function generates the following customized template for the execute function.

```
void DataProcessing<Always, Add, False,
                    SftOper<Reg, ShiftLeft, Imm>>::execute()
{
    if (Always::execute()) {
       _dest = Add::execute(_src1, _sftOperand.getValue());
       if (False::execute()) {
          // Update Flags
          ...
       }
    }
}
```

After compilation using a C++ compiler, several optimizations occur on the *execute()* function. The *Always::execute()* function call is evaluated to *true*. Hence, the check is removed. Similarly, the function call *False::execute()* is evaluated to *false*. As a result the branch and the statements inside it are removed by the compiler. Finally, the two function calls *Add::execute()*, and *_sftOperand.getValue()* get inlined as well. Consequently, the *execute()* function gets optimized into one single statement as shown below:

```
void DataProcessing<..skipped..>::execute() {
    _dest = _src1 + _sftOperand._operand << 10;
}
```

Furthermore, in many ARM instructions, the shifter operand is a simple register or immediate. Therefore, the shift operation is actually a *no shift* operation. Although the manual says that the case is equivalent to shift left zero, we use a *no shift* operation that enables further optimization. In this way, an instruction similar to the above example would have only one operation in its *execute()* method.

## 3.3 Simulation Engine

Due to compile time decoding and our *instruction abstraction* technique, the simulation engine is fast and simple. In this section we briefly describe the three basic steps in the simulation kernel viz., fetch, decode (if necessary), and execute.

The simulation engine fetches one decoded instruction at a time. As mentioned earlier, each instruction entry contains two fields viz., binary and pointer to the optimized functionality for the instruction. Before executing the fetched instruction, it is necessary to verify that the current instruction is valid i.e., this instruction is not modified during run time. The simulation engine compares the binary part of the current instruction having address *addr* with the binary instruction of the application program stored in memory at address *addr*. If they are equal then the decoded instruction is valid and the engine executes the optimized functionality referenced by the instruction.

However, if the instruction is modified then the modified binary is re-decoded. This decoding is similar to the one performed in the compile time decoding of instructions except that it uses a pointer to an appropriate function. While we develop the templates for each class of instructions, we also develop one function for each class. The mask table mentioned in Section 3.1 maintains the mapping between a mask for every class of instruction and the function for that class. The decoding step during run time consults the mask table and determines the function pointer. It also updates the *instruction memory* with the decoded instruction i.e., it writes the new function pointer in that address.

The execution process is very simple. It simply invokes the function using the pointer specified in the decoded instruction.

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

# 4. EXPERIMENTS

We evaluated the applicability of our IS-CS technique using various processor models. In this section, we present simulation results using a popular embedded processor, ARM7 [17], to demonstrate the usefulness of our approach.

## 4.1 Experimental Setup

The ARM7 processor is a RISC machine with fairly complex instruction set. We used *arm-linux-gcc* for generating target binaries for ARM7. Performance results of the different generated simulators were obtained using Pentium 3 at 1 GHz with 512 MB RAM running Windows 2000. The generated simulator code is compiled using the Microsoft Visual Studio .NET compiler with all optimizations enabled. The same C++ compiler is used for compiling the decoded program as well.

In this section we show the results using two application programs: adpcm and jpeg. We have used these two benchmarks to be able to compare our simulator performance with previously published results [1].

The *arm-linux-gcc* with *-static* option generates approximately 50K instructions for the benchmarks. When all optimizations are enabled in the MS VC++ compiler, it takes about 15 minutes to compile and generate the decoded programs.

## 4.2 Results

Figure 4 shows the simulation performance using our technique. The results were generated using an ARM7 model. The first bar shows the simulation performance of our technique with run-time program modification check enabled. Our technique can perform better if it is known prior to execution that the program is not self modifying. The second bar represents the simulation performance of running the same benchmark by disabling the run-time check. We could achieve upto 9% performance improvement by disabling the instruction modification detection and updation mechanism.

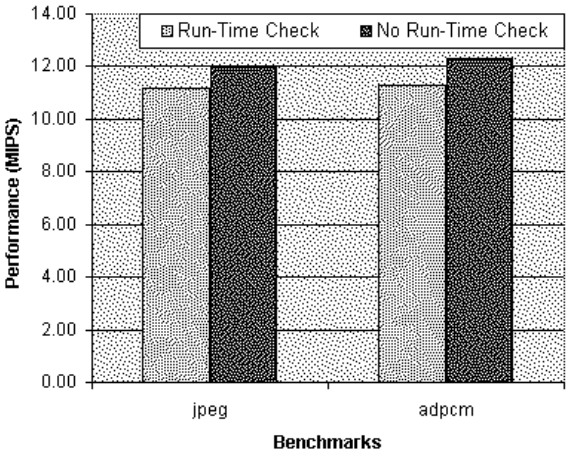


**Figure 4: Instruction Set Compiled Simulation**

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

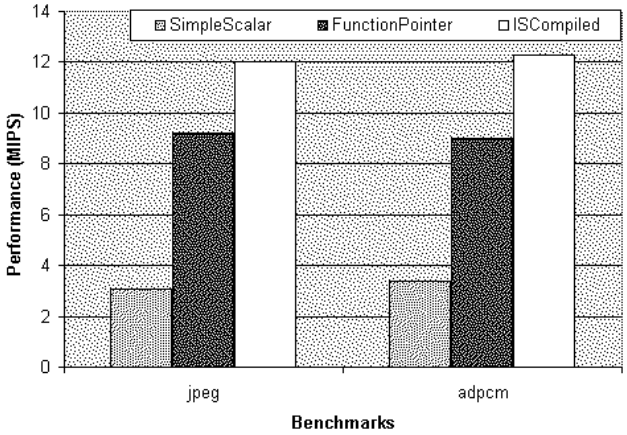


**Figure 5: Effect of Different Optimizations**

There are two reasons for the superior performance of our technique: moving the time consuming decoding out of the execution loop, and generating aggressively optimized code for each instruction. The effects of using these techniques are demonstrated in Figure 5. The first bar in the chart is the simulation performance of running the benchmarks on an ARM7 model of Simplescalar [3] that does not use any of these techniques. The second bar shows the effect of do-

ing the decoding process at compile time and using function pointers during execution. The use of function pointer in the decoded instruction is similar to [1]. We are able to achieve better result than JIT-CCS [1] even in this category because of the fact that JIT-CCS technique performs decoding of instruction during run-time (at least once) while we are doing it during compile time. Besides, they use a software caching technique to reuse the decoded instruction but we do not. The last bar is our simulation approach that uses both techniques: compile-time decode and using templates to produce optimized code.

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

## 5. SUMMARY

In this paper we presented a novel technique for instruction set simulation. Due to the simple interpretive simulation engine and optimized pre-decoded instructions, our instruction set compiled simulation (IS-CS) technique achieves the performance of compiled simulation while maintaining the flexibility of interpretive simulation. The performance can be further improved by disabling the run-time change detection which is suitable for many applications that are not self modifying.

The IS-CS technique achieves its superior performance for two reasons: moving time-consuming decode to compile time, and using templates to produce aggressively optimized code for each instance of instructions. We demonstrated performance improvement of upto 40% over the best published results on an ARM7 model.

Future work will concentrate on using this technique for modeling other real world architectures using an architecture description language to demonstrate the retargetability of this approach.

## 6. ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to acknowledge members of the ACES laboratory for their inputs.

## 7. REFERENCES

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

[2] S. Pees et al. Retargeting of Compiled Simulators for Digital Signal Processors using a Machine Description Language. *DATE*, 2000.

[3] Simplescalar Home *www.simplescalar.com*.

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

[5] B. Cmelik et al. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *ACM SIGMETRICS Performance Evaluation Review*, Volume 22(1), Pages 128-137, May 1994.

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

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

[8] E. Schnarr et al. Fast Out-of-Order Processor Simulation using Memorization. *PLDI*, 1998.

[9] E. Schnarr et al. Facile: A Language and Compiler for High-Performance Processor Simulators. *PLDI*, 1998.

[10] E. Witchel et al. Embra: Fast and Flexible Machine Simulation. *MMCS*, 1996.

[11] F. Engel et al. A Generic Tool Set for Application Specific Processor Architectures. *HW/SW Codesign*, 1999.

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

[13] Y. Futamura. Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler. *Systems, Computers, Controls*, Volume 2(5), Pages 45-50, 1971.

[14] G. Hadjiyiannis et al. ISDL: An Instruction Set Description Language for Retargetability. *DAC*, 1997.

[15] P. Mishra et al. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. *ISSS*, 2001.

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

[17] The ARM7 User Manual *www.arm.com*.