Memory Access Optimizations in Instruction-Set Simulators

Mehrdad Reshadi
Center for Embedded Computer Systems (CECS)
University of California Irvine
Irvine, CA 92697, USA
reshadi@cecs.uci.edu

Prabhat Mishra
Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611, USA
prabhat@cise.ufl.edu

ABSTRACT

Design of programmable processors and embedded applications requires instruction-set simulators for early exploration and validation of candidate architectures. Interpretive simulators are widely used in embedded systems design. One of the key performance bottlenecks in interpretive simulation is the instruction and data memory access translation between host and target machines. The simulators must maintain and update the status of the simulated processor including memory and register values. A major challenge in the simulation is to efficiently map the target address space to the host address space. This paper presents two optimization techniques that aggressively utilize the spatial locality of the instruction and data accesses in interpretive simulation: *signature based address mapping* for optimizing general memory accesses; and *incremental instruction fetch* for optimizing instruction accesses. To demonstrate the utility of this approach we applied these techniques on SimpleScalar simulator, and obtained up to 30% performance improvement. Our techniques complement the recently proposed optimizations (JIT-CCS [1] and IS-CS [2]) and further improve the performance (up to 89%) on ARM7 and Sparc processors.

Categories and Subject Descriptors

I.6.5 [Simulation and Modeling]: Model Development; I.6.7 [Simulation and Modeling]: Simulation Support Systems; C.4 [Performance of Systems]: Modeling techniques.

General Terms

Algorithms, Measurement, Performance, Design.

Keywords

Instruction-set simulator, memory address-space mapping.

1. Introduction

Instruction-set simulators are essential tools for design space exploration as well as validation and evaluation of programmable architectures and compilers. The simulators run on a host machine and mimic the behavior of an application generated for a target processor. The simulators can be broadly classified into two categories: interpretive and compiled. Figure 1 shows the three main stages in a traditional interpretive simulation. In this approach, an instruction is fetched, decoded and executed at run time.

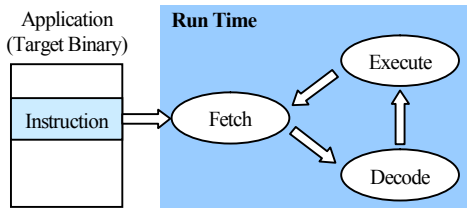

Figure 1- Three steps in interpretive simulation.

Interpretive simulators provide flexibility but are very slow. Compiled simulators achieve higher performance by moving the time consuming decoding step from the run time to compile time as shown in Figure 2. In this approach the application is decoded, and the optimized code for the host machine is generated at compile time.

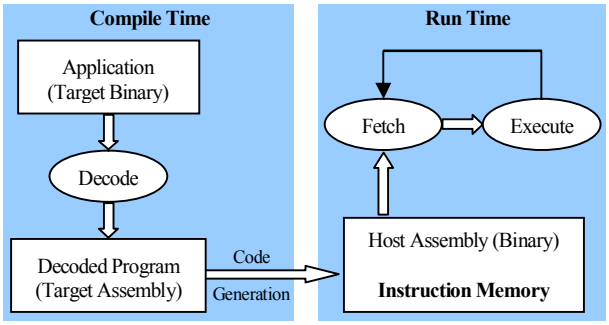

Figure 2- Traditional compiled simulation.

In spite of their performance advantage, the compiled simulators are not used in system-on-chip designs due to the assumption that the program does not change during run time. There are various application domains where programs are modified during run time for various reasons including power, performance and code size. For example, ARM processor uses two instruction sets (normal and reduced bit-width) and dynamically switches to different instruction-set at run time. Due to the restrictiveness of compiled simulation technique, interpretive simulators are widely used in embedded systems design flow. There are many techniques in the literature (JIT-CCS [1], IS-CS [2]) that combine the benefits of both simulation approaches by temporarily storing the decoded information and reusing them during run-time as well as optimizing the execution of each instruction. Since 10% of the code is executed 90% of the time,

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

the run-time decode is not necessary for majority of the dynamic instructions.

Figure 3 shows the flow of a fast and flexible interpretive simulation. The flexibility is maintained since the simulation engine still fetches and executes one target instruction at a time. The performance is achieved by reusing the previously decoded instructions. In case an instruction is modified during run-time, it needs to be detected and decoded.

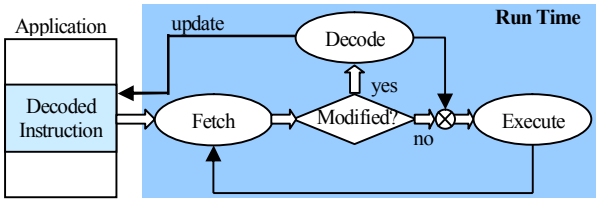

Figure 3- Fast and flexible interpretive simulation.

Irrespective of the simulation technique, the execute stage runs the code and updates the simulated processor status. In an instruction-set simulator, the processor status is the set of memory and register values in the target processor. Since the target processor can potentially access a very large memory, a major challenge during simulation is to efficiently map the target program memory address space to the simulator memory address space. In interpretive simulation, this address space mapping is needed in all three main stages of the simulation. In the fetch stage, the opcode of the instructions or their corresponding decoded information must be read from program memory or an instruction cache based on the value of the program counter. In the decode stage, if the decoded information needs to be reused, it must be stored in a location relative to the corresponding instruction address. In the execute stage, the load/store operations depend on the address space mapping to access the correct memory location. Similarly, in compiled simulation during execution of instructions the address mapping may be necessary.

The dynamic nature of indirect addresses, as in indirect jumps or indirect data memory accesses, requires the address space mapping to be done at run time. It is very similar to the virtual to physical memory address mapping in computer architecture except that the address space mapping in the simulator is performed sequentially in software without any hardware support. In simulators, the address space mapping is done via a typical hash function. To the best of our knowledge, there are no published results for optimizing this mapping.

In this paper, we present an efficient approach for address space mapping in interpretive simulators. We present *signature-based address mapping* for optimizing general memory accesses; and *incremental instruction fetch* for optimizing instruction accesses. The proposed techniques are fast, efficient and independent of the behavior of the simulated target instructions. Therefore, the techniques are also suitable for retargetable simulation. They can be applied both in interpretive and compiled simulators. The performance improvement is achieved through aggressive exploitation of spatial locality of both instruction and data accesses. To evaluate the effectiveness of our approach, we applied these techniques on SimpleScalar [7], a widely used interpretive simulator, and obtained up to 30% performance improvement. The experimental results (using ARM7 and Sparc processor models) demonstrate up to 89% performance improvement compared to the best known techniques ([1] and [2]) in interpretive simulation.

The rest of the paper is organized as follows. Section 2 presents related work addressing instruction-set simulation approaches. Our memory access optimization techniques are described in Section 3 followed by the experimental results in Section 4. Finally, Section 5 concludes the paper.

2. Related Work

There has been an enormous body of work done on different classes of instruction-set simulators such as trace driven ([14], [15]), interpretive ([1], [2], [7]) and compiled simulators ([3], [4], [5], [6], [12], [13]). While every simulator must deal with the address space mapping problem, to the best of our knowledge, very little has been reported that addresses this issue separately.

In compiled simulation techniques, a major part of the performance boost comes from the direct execution of instructions on the host machine: instead of mapping the address of each instruction, the simulator only needs to find the actual address of the beginning of a basic block and transfer the flow of execution to that block. Therefore the address space mapping is only performed for basic blocks and data memory read/write operations. In static compiled simulation, very large switch-case statements control the flow of execution based on the value of the program counter. In dynamic compiled simulation (binary translation), portions of the program are decoded into overlapping blocks, and program counter values are mapped to the memory addresses of the corresponding blocks.

Shade [3] has proposed a simulation technique that partially reduces the amount of address space mapping required for executing consecutive instructions. This approach has been adopted by other simulators ([4], [5], [6]) that use binary translation. In Shade, groups of instructions are decoded and stored in the memory blocks that have three sections: prolog, body and epilog. Every program counter value that has been the target of a jump is mapped to one such memory block and therefore the blocks may overlap. Every time a new block is generated, consecutive blocks are chained by adding a jump instruction in the epilog section of the proper block. The target instructions are directly mapped into host binary that is stored in the body section of the corresponding memory block. Prolog and epilog sections contain extra instructions for initializing and finalizing the execution of the instructions in the body section.

Although this approach results in very high performance, it has several drawbacks. First, directly generating host binary is a complex task that limits the portability and retargetability of the simulator. Second, generating overlapping blocks decreases the efficiency of block chaining and utilization of memory. Finally, when a self modifying program[1] changes the instruction memory, instead of re-decoding the modified instruction and

[1]Self-modifying programs occur often, e.g. in just-in-time compiler, dynamic linker and processor mode change.

updating it in the corresponding memory blocks, all of the memory blocks must be discarded because it is impossible or very expensive to find the containing blocks and the relevant portions to be updated.

In SimpleScalar [7], the address space mapping is done using a hash table. This approach imposes a constant cost on all address calculations and does not exploit the spatial locality of accessed addresses. In this paper we propose a simple and efficient approach that increases the performance of interpretive simulators by utilizing the spatial locality of accessed memory addresses. Our techniques can be effectively coupled with existing optimization techniques to further boost simulator performance.

3. Memory Access Optimization

A program generated for a target processor may expect specific information at specific memory addresses for correct execution. For example, addresses of functions, locations of global data and target addresses of branches are fixed in a program. During simulation, these addresses must be converted to valid addresses on the host machine that are legally accessible for the simulator. One way of doing this is to add an offset to all of the program addresses and map them to a continuous memory region allocated by the simulator on the host. For example, if *s* and *e* are respectively the smallest and the largest addresses that the program may access, the simulator needs to allocate (*e*-*s*+1) bytes of memory and then subtract *s* (as a negative offset) from every address accessed by the program at run time. However, the program may only access small and disjoint regions of its memory. In other words, while (*e*-*s*+1) can be a very large number, the program may actually consume much less memory. For example, the executable and data sections of a program are usually placed in non-contiguous sections mostly far away from each other in the memory. Also, a program may have access to a very large heap/stack area but may actually use a very small portion of it.

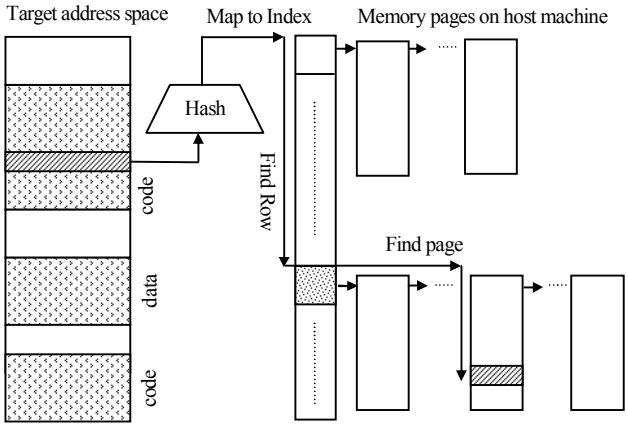

Figure 4- Address space mapping using typical hash table.

For better memory utilization, the simulator needs to allocate memory for the program only when it needs them. To do this, the memory space is partitioned into same sized pages and then a hash table maps the program addresses to the actual pages. Figure 4 shows this flow with a typical hash table. In a hash table, a hash function maps each input value to a unique row of the table. Since multiple input values may be mapped to the same row, the hash function needs to perform a search to find the corresponding page in the row.

After a program address is mapped to a memory page, the actual location (index) of the corresponding element in the page is calculated. Figure 5 shows the pseudo code that maps an address in the target processor address space to its corresponding address on the host machine. The *FindMemoryPage* function is the actual hash function that returns the memory page containing the *target address*. The *FindMemoryCell* function calculates the index of the corresponding cell in the mapped page.

```
MemoryCell* FindHostAddress(MemoryAddr target) {
  MemoryPage mp = FindMemoryPage( target);
  MemoryCell* mc = FindMemoryCell( mp, target);
  return mc;
}
```

Figure 5- Pseudo code for target to host address mapping.

3.1 Signature Based Address Mapping (SAM)

Due to the spatial locality of memory accesses in the simulated program, many consecutive memory accesses may be mapped to the same memory page. In other words, the hash function returns the same result for many consecutive calls. Therefore, by detecting this situation, we can avoid the overhead of hash function, and directly map the address to its corresponding page. In order to detect whether a new address maps to the same page as the previous one, we need to calculate a signature for the addresses and compare them. The least significant bits of an address is usually used for indexing in a page hence two addresses that reside in the same page may differ only in these bits. A shift right operation extracts the constant portions of these addresses which we use as their signature.

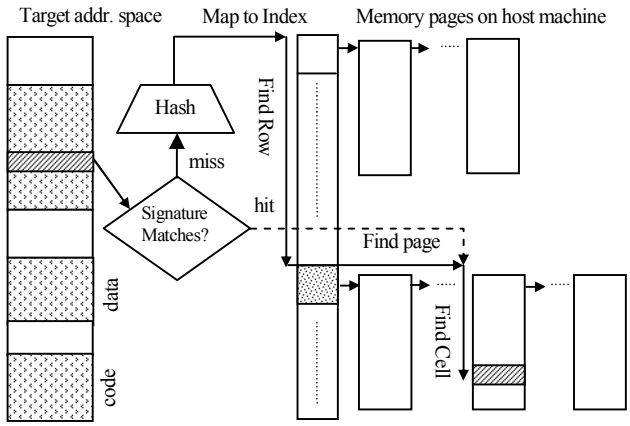

Figure 6- Signature based Address Mapping optimization.

Figure 6 shows the new adaptive hash function. If the signature of the new address to be mapped matches with that of the previous one (a hit), the previous result is reused; otherwise (a miss) the normal hash function is called to calculate the mapping. The new adaptive hash function executes faster whenever address signatures match. The signatures must be so that all of the addresses that map to the same page have the same signature.

Also the calculation and comparison of the signatures must have less overhead than calling the hash function.

Figure 7 shows the modified pseudo code for address mapping. In this code, whenever the original hash function (*FindMemoryPage*) is called to find a new page, the result page and the signature of the addresses that map to it are stored. As long as the signatures of the addresses in the next calls match with the existing signature, the stored memory page is used. In this way, the cost of finding consecutively accessed pages is decreased but the cost of finding a new page is slightly increased. Overall it has a considerable positive impact on the performance of the simulator because the number of page hits is significantly higher than the number of page misses as demonstrated in Section 4.2.

```
MemoryCell* FindHostAddress(MemoryAddr target)  {
  MemoryPage mp;
  if ( signature( target) == lastSignature )
    mp = lastMemoryPage;
  else  {
    mp = FindMemoryPage(target);
    lastMemoryPage = mp;
    lastSignature = signature(target);
  }
  MemoryCell* mc = FindMemoryCell( mp, target);
  return mc;
}
```

Figure 7- Signature based Address Mapping optimization.

The *FindMemoryPage* function performs more computations than a single shift right that is needed to calculate the signature of the addresses. It also needs to access internal data structures in the host processor memory to find the corresponding page. These extra host memory accesses may remove simulation information from the cache of host processor and hence degrade the cache performance of the simulator. Therefore, by using signature based address mapping and avoiding the *FindMemoryPage* function, the simulation engine executes fewer operations and shows a better cache performance on the host machine.

3.2 Incremental Instruction Fetch (IIF)

In any instruction-set simulator, as in real hardware, the program counter indicates the address of the instruction that must be executed next. Figure 8 shows the simulation loop: the opcode (or the decoded information) of the next instruction is read from the memory and the program counter is incremented before executing the instruction.

```
while ( not end of program )  {
  hostInstAddress = FindHostAddress (programCounter);
  increment (programCounter);
  execute instruction at hostInstAddress;
}
```

Figure 8- The execution loop of the simulator.

A branch instruction may change the value of the program counter and the sequential flow of the program execution. In other words, as long as the program counter is not changed by an instruction, we can calculate the address of next instruction to be executed by incrementing the address of current instruction. Therefore, instead of calling the mapping function for every program counter value, it is invoked only when an instruction changes the sequential flow of execution (PC changed by that instruction) as shown in Figure 9. The special case occurs when the address of next instruction does not reside in the current memory page. This is similar to a program counter change and is handled in the same manner.

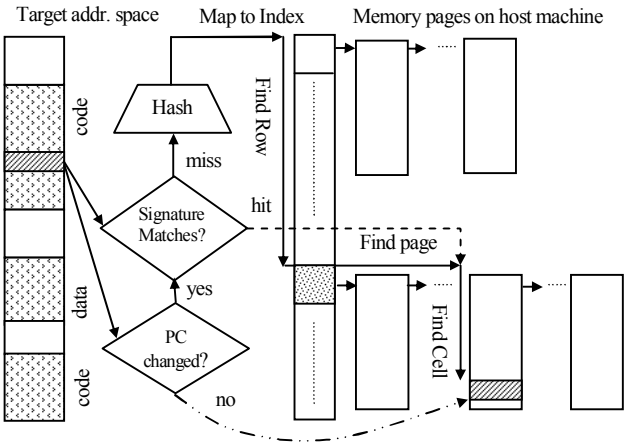

Figure 9- Incremental Instruction Fetch optimization.

Figure 10 shows the optimized version of the simulation execution loop. This algorithm does not require the semantics of the executed instructions. Therefore, the simulation engine can be completely independent of the behavior of the simulated processor instruction set. This optimization will improve the performance only when the cost of incrementing the instruction address and detecting the program counter change is less than that of mapping the address through a hash function call. Its efficiency also depends on the size of the basic blocks in the program. Simulation of longer sequential codes will benefit more speedup because they require less hash function calls.

```
while ( not end of program )  {
  hostInstAddress = FindHostAddress(programCounter);
  while ( ( programCounter is not changed ) and
          ( hostInstAddress is in the current page ) )  {
    increment ( programCounter );
    execute instruction at hostInstAddress;
    increment ( hostInstAddress );
  }
}
```

Figure 10- Pseudo code for Incremental Instruction Fetch.

4. Experiments

We evaluated the applicability of our memory access optimization techniques using various processor models. In this section, we present simulation results using two popular processors, ARM7 [10] and Sparc [11].

4.1 Experimental Setup

We implemented our technique on two simulators. First, we used the *SimpleScalar* [7] for ARM processor to demonstrate the usefulness of our approach on a popular interpretive simulator. Second, we developed a fast and flexible interpretive simulation framework [16] for ARM and Sparc processors that uses the

recently proposed optimization techniques on reuse of decoded instructions (JIT-CCS [1]) and improving instruction execution speed (IS-CS [2]). In this paper, we refer the second simulator as *Base Simulator*. All of the experiments are performed on a Pentium 4, 2.4 GHz with 512 MB RAM running Windows XP. The application programs are taken from MiBench (bluefish, crc), MediaBench (adpcm, epic, g721) and SPEC95 (compress, go) benchmark suites.

4.2 Results

Figure 11 shows the performance of the SimpleScalar before and after applying the *SAM* and *IIF* optimizations. Higher spatial locality and higher reduction in hash function calls result in better performance using these optimizations. The results demonstrate 13% to 30% performance improvement in SimpleScalar.

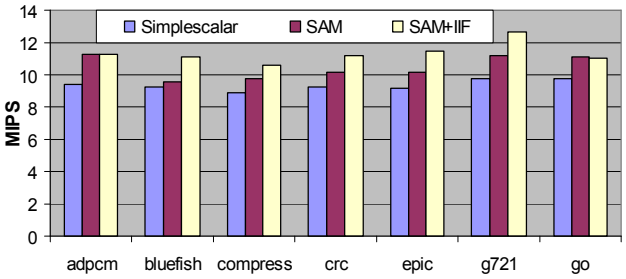

Figure 11- SimpleScalar simulator for ARM processor.

Figure 12 shows the utilization of our techniques on SimpleScalar. The first bar shows the hit rates after applying *SAM* optimization. A hit means that the signature of the address to be mapped is equal to that of the previous address and therefore the previous mapping result can be reused. For example, in case of *adpcm* benchmark, 78% of the time the signatures were identical; therefore the hash function was not called. Less number of hash function calls (higher hit rate) implies better performance.

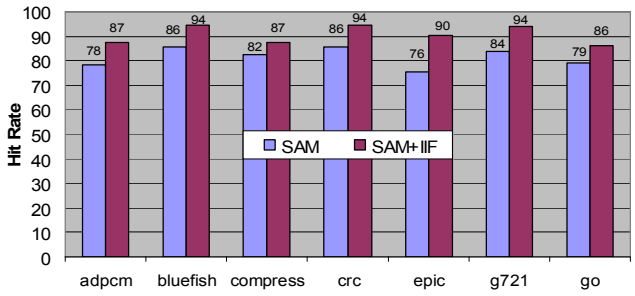

Figure 12- SAM and IIF on Simplescalar ARM simulator.

The second bar in Figure 12 shows the hit rates after applying both *SAM* and *IIF* optimizations. As mentioned in Section 3.2, when PC is not changed by an instruction (hit for IIF), the address of current instruction is incremented to find the memory address of next instruction instead of calling the hash function. For example, in *adpcm* benchmark, the hit rate is 87% using both SAM and IIF optimizations. These techniques drastically reduce the number of hash function calls (up to 94%) and thereby generate improved simulation performance (up to 30%).

Figure 13 and Figure 14 show the performance of Base Simulator for ARM and Sparc processor models respectively. In these figures, the first bar shows the performance of Base Simulator that implements the JIT-CCS and IS-CS optimizations. The second and third bars show the simulation performance with only *SAM* optimization, and with both *SAM* and *IIF* optimizations respectively. The results demonstrate 20% to 89% performance improvement on top of the best known techniques (JIT-CCS and IS-CS) in interpretive simulation.

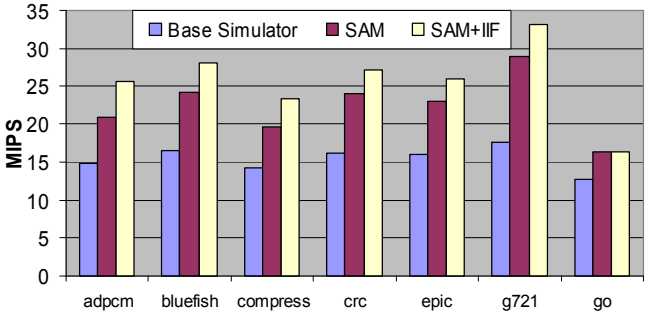

Figure 13- Performance of base simulator for ARM.

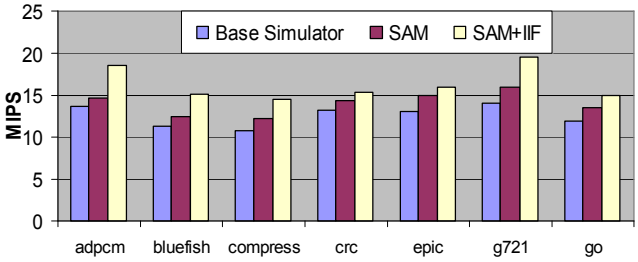

Figure 14- Performance of base simulator for Sparc

Clearly, our techniques generated higher performance improvement when applied with recent optimizations (Base Simulator) than when applied to Simplescalar. There are two primary reasons for this difference:

- First, the data and executable sections in a typical program are usually placed far away from each other in the memory. The executable sections are accessed when the instructions are read and decoded while the data sections are accessed during execution of load/store instructions. Simplescalar simulator accesses both the data and the executable sections in the same iteration of the execution loop. This reduces the spatial locality of the memory accesses. However, in the base simulator, as long as decoded instructions are reused (using JIT-CCS or IS-CS), only the data sections of the memory are accessed and hence the spatial locality can be exploited more effectively.
- Second, Base Simulator performs fewer operations (due to JIT-CCS and IS-CS) than SimpleScalar to simulate the same application. Although, our techniques generated almost same reduction of hash function calls in both simulators, SAM and IIF optimizations are more effective in Base Simulator since it results in a higher percentage of total reduction in the number of executed operations.

Figure 15 compares the performance of SimpleScalar ARM with our final ARM simulator that implements all of the optimizations (JIT-CCS, IS-CS, SAM and IIF). The final simulator is up to 3.4 times faster than SimpleScalar ARM.

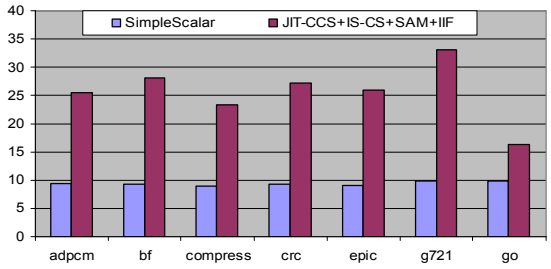

Figure 15- SimpleScalarARM vs. all optimizations

In this section we demonstrated that our techniques complement the recently proposed optimizations and further improve the performance (up to 89%) of the simulator. Our techniques improved the performance (up to 30%) of Simplescalar, a widely used interpretive simulator, which does not use any recent optimizations.

5. Conclusions

Instruction-set simulators are an integral part of today's processor and software design process. Fast and flexible interpretive simulators are widely used in embedded systems design. One of the key performance bottlenecks in instruction-set simulators is the instruction and data memory access translation between host and target machines. This paper presented an efficient approach for optimizing memory accesses in instruction-set simulation. We proposed two techniques to exploit the spatial locality of memory accesses: s*ignature based address mapping* for optimizing both data and instruction accesses; and i*ncremental instruction fetch* for reducing the overhead of the instruction accesses. We applied these optimizations on two different simulators: SimpleScalar for ARM processor; and Base Simulator that models both ARM and Sparc processors and also implements recent optimization techniques. Our experimental results demonstrated up to 30% performance improvement in SimpleScalar, and up to 89% performance improvement in Base Simulator on top of the existing optimizations. The better performance improvement in Base Simulator is due to the use of decoded instruction cache and a lighter execution loop.

The proposed techniques are general and can be applied to any simulation framework including both compiled and interpretive simulators. The efficiency of these techniques depends on the spatial locality of memory accesses and the average size of basic blocks in the simulated program. Future work will focus on application of these techniques on further real-world processors.

6. Reference

[1] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, Andreas Hoffmann. "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation". DAC, 2002.

[2] Mehrdad Reshadi, Prabhat Mishra, Nikil Dutt. "Instruction-Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation". DAC, 2003.

[3] Robert. F. Cmelik, David Keppel. Shade: A fast instruction set simulator for execution profiling. Measurement and Modeling of Computer Systems, 1994.

[4] Emmett Witchel, Mendel Rosenblum. "Embra: Fast and Flexible Machine Simulation". MMCS, 1996.

[5] Eric Schnarr, James R. Larus. "Fast Out-of-Order Processor Simulation using Memoization". PLDI, 1998.

[6] Eric C. Schnarr, Mark D. Hill, James R. Larus. "Facile: A language and compiler for high-performance processor simulators". PLDI, 2001.

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

[8] David Keppel. "A Portable Interface for On-The-Fly Instruction Space Modification". ASPLOS, pp. 86-95, 1991.

[9] David Keppel, Susan J. Eggers, and Robert R. Henry, "A Case for Runtime Code Generation". University of Washington Comp. Science and Engg. Technical Report 91-11-04, 1991.

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

[11] Sparc Version 7 Instruction set manual.

[12] Gunnar Braun, Andreas Hoffmann, Achim Nohl, Heinrich Meyr. "Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description". ISSS, 2001.

[13] J. Zhu, Daniel Gajski. "A Retargetable, Ultra-fast Instruction Set Simulator". DATE, 1999.

[14] R. Uhlig, "Trap-Driven Memory Simulation". Ph.D Thesis, Dept. of EECS, University of Michigan, Ann Arbor 1995.

[15] R. A. Uhlig and T. N. Mudge. "Trace-driven memory simulation: A survey". ACM Computing Surveys, 29(2):128–170, 1997.

[16] M. Reshadi, N. Bansal, P. Mishra, N. Dutt, "An Efficient Retargetable Framework for Instruction-Set Simulation", CODES+ISSS, pp. 13-18, 2003.