

A Technique to Exploit Memory Locality for Fast Instruction Set Simulation

Wei Qin
Boston University, USA
Email: wqin@bu.edu

Bo Hu
Fudan University, China
Email: bohu@fudan.edu.cn

Abstract—To verify software and system functionality during the design process of complex SoCs, designers routinely use instruction set simulators (ISSs) as high level processor models. Typically, an ISS needs to simulate billions of target instructions to verify a single task of the system in design, which is a very time-consuming process. Therefore, the simulation speed of ISSs has become one important concern of SoC designers. In this paper, we present a new technique to improve the speed of ISSs. Specifically, we focus on the translation of memory addresses between the target memory space and the host memory space. Such translation is performed by a hash table in many ISS implementations. To accelerate the translation, we introduce a software cache called translation buffer (TB) to cache the hashing results for quick future access. With properly chosen TB sizes, our implementation effectively exploits the abundant locality inherent in the memory footprints of real-world application programs and greatly reduces the overall address translation overhead. We integrated the technique into three popular ISSs and achieved a 15-32% improvement in simulation speed.

I. INTRODUCTION

A processor simulator is a piece of software that mimics the behavior of the target processor on the host computer. In general, processor simulators can be classified into two categories: instruction-set simulators (ISSs) and cycle accurate simulators (CASs). The former category serves to verify the functionality of the application programs; while the latter provides both functional and timing information. Typically, a program on an ISS runs 10 to 100 times slower than on a real machine, but 10 to 100 times faster than on a CAS. The choice of the simulator depends on the level of details and the simulation speed required by the users. In this paper, we will describe one optimization technique for ISSs.

Computer designers and compiler developers have long used ISSs to analyze processor performance and to verify the correctness of compilers and application programs. In recent years, as embedded processors become common building blocks for system-on-chips (SoCs), circuit designers start to use ISSs for hardware/software codesign tasks. For such tasks, the complete functional simulation of an application program often takes hours or even days. Therefore the speed of ISSs has become

a serious concern of many researchers in the electronic design automation (EDA) area.

Three classes of ISSs exist to date: interpreters, statically-compiled simulators and dynamically-compiled simulators. Interpreters are highly portable and easy to develop. Consequently they are most popular among SoC designers. In this paper, we will mainly discuss interpreters, although our approach is applicable to compiled simulators as well.

This paper presents a simulation technique to accelerate ISSs. The technique aims to optimize the simulation of memory operations by exploiting the abundant memory locality in real-world application programs. The paper is organized as follows. Section II introduces some background information regarding ISSs and recent related work. Section III proposes the new technique. Experiment results are presented in Section IV, after which Section V concludes the paper.

II. BACKGROUND

As the most popular type of ISS, interpreters for various instruction sets have been developed for different purposes. For example, popular tools such as SimpleScalar [1], MicroLib [6], the GNU Project Debugger (GDB) [2], and SimIt-ARM [7] all contains fast interpreters which allow programmers to conveniently verify and debug software. Interpreters are especially helpful for processors that are not yet manufactured as designers can use simulation results to guide the optimization of both hardware and software.

The typical structure of an interpreter is shown in

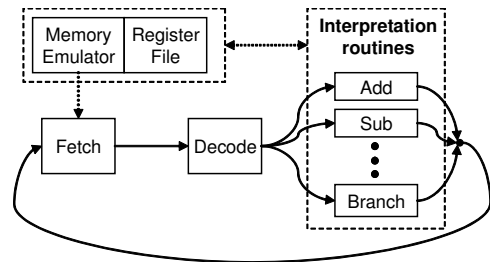


Fig. 1. Typical ISS Structure

Figure 1. Its execution kernel is a loop which repeatedly fetches instructions from the memory emulator, decodes and then dispatches them to their corresponding interpretation routines. The interpretation routines emulate the functionality of individual instructions. Their tasks include reading operands from the emulated register file and/or the emulated memory, performing computation per the semantics of the instructions, and writing back results to the register file and/or the memory. A common technique to emulate the target memory storage is to divide the memory space into pages and allocate storage space for a page only when it is used. All allocated pages are managed by a page table. The mapping from the target memory address to the actual storage address is called memory address translation, an approach similar to the virtual memory technique in computer systems.

To explain the memory address translation technique, here we use the implementation in the MicroLib [6] PowerPC interpreter as an example. Other popular interpreters use very similar implementations.

The PowerPC interpreter organizes memory storage in pages of 4096 bytes. It uses a two-level hash table as shown in Algorithm 1 to translate 32-bit target addresses to host addresses. Each entry of the first level table points to a second level table. Each entry of the second level table points to a list of conflicting pages. The 32-bit target memory address is divided into four portions. The lowest 12 bits form the page offset. The following 12 bits are used as the index for the first level table, while the next 4 bits are used as the index for the second level table. The highest 4 bits serve as the tag to identify a page match. The translation algorithm is shown as the pseudo-code below. The return value of the algorithm is the pointer to the head of a page. Adding the pointer to the page offset yields the final host address.

In the routine, the second level tables and the pages are allocated only when the target program first accesses corresponding addresses in the target address space. Therefore the algorithm efficiently consolidates the sparsely used target address space into the allocated host memory pages. When implementing such a memory translator, designers often face the trade-off between memory usage and translation speed. For the above example, it is possible to enlarge the second level tables so as to reduce or eliminate the chance of conflict. It is also possible to use a single-level table with 64K entries to perform the same translation. This trade-off regarding the right translation table structure is similar to the concerns related to the virtual memory system in a computer. In both cases, a two-level table is considered a proper choice to balance memory usage and performance.

In recent years researchers in the EDA community have shown renewed interest in developing fast and

Algorithm 1 The get_page() Routine

```

l1_entry = l1_table[target_addr[23:12]];
if l1_entry == NULL then
    l1_entry = allocate_l1_entry(target_addr);
    l1_table[target_addr[23:12]] = l1_entry;
end if
l2_entry = l1_entry→table[target_addr[27:24]];
page_pointer = l2_entry→page;
while page_pointer != NULL do
    if page_pointer→tag == target_addr[31:28] then
        return page_pointer;
    else
        page_pointer = page_pointer→next;
    end if
end while
page_pointer = allocate_page(target_addr);
page_pointer→next = l2_entry→page ;
l2_entry→page = page_pointer;
return page_pointer;

```

retargetable interpreters. Nohl et. al. proposed the just-in-time cache-compiled simulation (JIT-CCS) technique [5]. Its central idea is to buffer instruction decoding results using a direct-mapped software cache. Similarly, Reshadi et. al. proposed the concept of instruction-set compiled-simulation (IS-CS) to reduce instruction decoding time [8]. Their technique statically decodes the entire program during simulator building time, meaning that for each target program to simulate, a simulator instance needs to be generated. Instead of using static decoding or caching, Qin et. al. directly tackled instruction decoding cost in interpreters [7]. Their technique can synthesize efficient binary decoders with comparable performance to hand-coded ones.

III. PROPOSED TECHNIQUE

The above-mentioned techniques aim to reduce the decoding time of interpreters. In this paper, we propose a new technique to accelerate another part of interpreters, the memory emulator. The proposed technique is orthogonal to the above techniques and can be used in combination with any of them.

The translation routine in Algorithm 1 looks up the two level page table to get the page pointer. Although the look-up cost seems small by itself, but in an ISS whose simulation speed is of critical concern, the cost is considerable. Our profiling analysis shows that typically over 20% of the overall simulation time is spent in memory address translation. Hence it is worthwhile to optimize the memory translation speed.

By intuition, if many memory accesses fall into the same page, then the translation result of the first access

can be reused for the subsequent ones. This leads to the idea of exploiting the locality of memory accesses to reduce the number of full address translations. A straightforward measure is thus to use a software cache to buffer the translation results – the page pointer in Algorithm 1. We call such a software cache a translation buffer (TB). As address translation itself is analogous to the virtual memory technique in modern computer systems [3], the TB is analogous to the translation look-aside buffer (TLB) which is a hardware cache for accelerating virtual page lookup. However, unlike TLBs, the TB is to be implemented in software. Therefore it is important to carefully optimize its implementation to achieve real benefits. Two possible optimization schemes are to increase the hit rate and to reduce the TB access time. The former can be met by tuning the size and the associativity of the TB. However, for software implementation, an associative cache is slow since its sets need to be scanned sequentially on each access. Thus, in our implementation we use only direct-mapped caches. We will focus on tuning the size of the cache to achieve better hit rate.

As an illustrating example, Algorithm 2 shows the translation algorithm with a 32-entry TB and 4K-byte pages. Given a target address, it first looks up the corresponding TB entry. If the tag matches, then the entry contains the pointer to the host page. Otherwise, the TB misses the address and a full translation is performed. The translation result is then written back into the TB. One distinction between a TB a normal cache is that the TB does not have a valid bit for each entry. On simulation start, the entries are initialized with some valid page pointers and remain valid thereafter. This omits the overhead of checking the valid bit in software.

In case when the TB size is 1, the algorithm can be further simplified since there is no need to compute the index and the entry address. For a proper TB implementation, such simplification can be done automatically by the C/C++ compiler through constant propagation.

Algorithm 2 The `get_page_fast()` Algorithm

```

tag = target_addr[31:17];
index = target_addr[16:12];
if tag==buffer[index].tag then
    return buffer[index].page;
else
    buffer[index].tag = tag;
    buffer[index].page = get_page(target_addr);
    return buffer[index].page;
end if

```

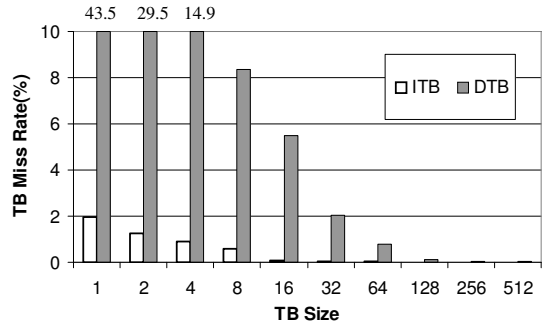


Fig. 2. ITB and DTB Miss Rates

IV. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the proposed technique, we integrated it into three popular open-source ISSs, the SimpleScalar Alpha ISS [1], the MicroLib PowerPC ISS [6] and the SimIt-ARM ISS [7]. To minimize the influence of coding style (e.g. use of macros or functions), we followed the same style of the original ISS in each case study. The ISSs are compiled by GCC 3.3.3 with optimization switches `-O3` and `-fomit-frame-pointer`. All experiments were performed on an unloaded P4 2.8GHz workstation with 2GB memory.

We use a mix of SPEC CPU95 [9] and MediaBench [4] benchmarks for the SimIt-ARM ISS since those are typical embedded applications for an ARM processor. The baseline ISS without any TB has an average simulation speed of 29.4 MIPS (Million Instructions per Second).

We first evaluate the miss rate under different TB sizes. Since instruction fetching and data memory access have very different access patterns, we consider them separately. We use two buffers, an ITB for instruction memory access, and a DTB for data memory access. We vary the size of each buffer from 1 to 512 and calculate the average buffer miss rates. The results are given in Figure 2. For instruction access, a single-entry buffer achieves low miss rate and further increasing the buffer size only slightly improves. For data access, a buffer size of 32 or larger can achieve low miss rate.

As previously explained, when the TB size is 1, the code in the Algorithm 2 can be optimized by the compiler. Because of this and the already low miss rate under a single-entry ITB, our experiments indicated that an ITB size of 1 yields faster simulation speed than other sizes. So we fixed the ITB size to 1 for all rest experiments. When we enable the ITB but not DTB, the average simulation speed is 32.8 MIPS, reflecting a 11% improvement over the baseline case. We then enable both the ITB and the DTB. Our simulation results indicate that when the DTB size goes beyond 32, the speed variation becomes negligible. This meets our expectation based on the DTB miss rates in Figure 2. With an ISS containing a single-entry ITB and a 256-entry DTB, we can achieve

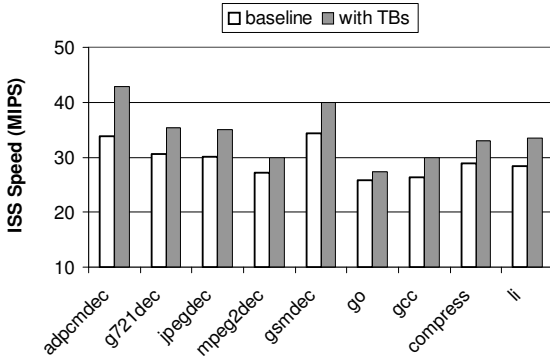


Fig. 3. SimIt-ARM Speed for Benchmarks

an overall 15% performance improvement compared to the baseline ISS. Figure 3 compares the simulation speed for individual benchmarks. It demonstrates that our technique is useful for all the benchmarks.

It is worth noting that the effect of the ITB is more significant than that of the DTB. This is due to the fact that the ITB is used more frequently than the DTB – every simulated instruction accesses the ITB while only load and store instructions access the DTB.

We use SPEC CINT2000 benchmarks [9] for the MicroLib PowerPC ISS and the SimpleScalar Alpha ISS (sim-fast). We augment the PowerPC ISS with a single-entry ITB and a 256-entry DTB. We then compare the simulation speed of the original ISS and the augmented one. The results are shown in Figure 4. On average, the proposed technique achieves a 32% speed improvement. The SimpleScalar memory emulator heavily utilizes C macros and is very difficult to modify. So we only implement a single-entry ITB with 20 lines of new code in its memory emulator. Figure 5 compares the simulation speed before and after the modification. Our technique yields an average improvement of 29% for all the benchmarks.

V. CONCLUSIONS

In this paper we presented a new technique to accelerate memory address translation in many ISSs. The technique utilizes a software cache to buffer the translation results for quick future access. It can effectively exploit the locality in the memory footprints of programs and reduce the total number of address translations. With small amount of effort we are able to achieve a 15-32% improvement in simulation speed for three popular ISSs. The technique is orthogonal to existing techniques that accelerate decoding and can be used together with them.

Though it is not demonstrated by our experiments, we believe that the technique is also useful for compiled ISSs. In compiled ISSs, there is no need for an ITB since target instructions are compiled into host ones and

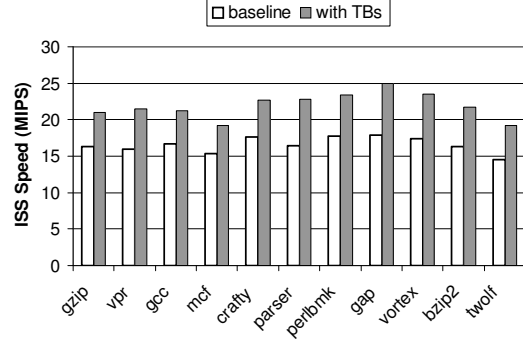


Fig. 4. PowerPC ISS Speed for Benchmarks

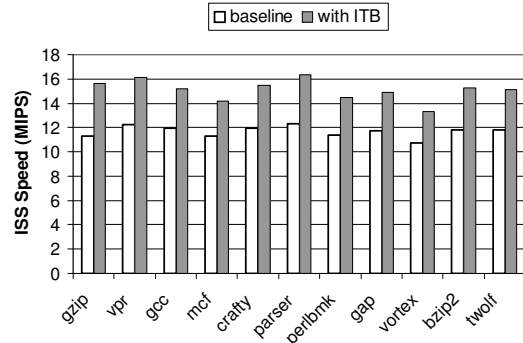


Fig. 5. SimpleScalar Speed for Benchmarks

therefore not fetched during simulation. But the DTB becomes more beneficial since compiled ISSs are so fast that any improvement in the memory emulator will have a notable impact on the overall simulation speed.

REFERENCES

- [1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, Feb 2002.
- [2] Free Software Foundation. <http://www.gnu.org/software/gdb/gdb.html> (current May 2005).
- [3] J. Hennessy, D. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 3rd edition, 2002.
- [4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of International Symposium on Microarchitecture*, pages 330–335, 1997.
- [5] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of Design Automation Conference*, pages 22–27, 2002.
- [6] D. G. Perez, G. Mouchard, and O. Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of International Symposium on Microarchitecture*, pages 43–54, 2004.
- [7] W. Qin and S. Malik. Automated synthesis of efficient binary decoders for retargetable software toolkits. In *Proceedings of Design Automation Conference*, pages 764–769, 2003.
- [8] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings of Design Automation Conference*, pages 758–763, 2003.
- [9] Standard Performance Evaluation Corporation. <http://www.spec.org> (current May 2005).