Direct SMARTS: Accelerating Microarchitectural Simulation through Direct Execution

Shelley Chen

Submitted to the Department of Electrical and Computer Engineering

In partial fulfillment of the requirements for the degree of

Masters of Science

at

Carnegie Mellon University

May 2004

Committee Members

Babak Falsafi, Associate Professor of ECE (advisor)

James C. Hoe, Assistant Professor of ECE

Abstract

Due to growing complexity and costs of hardware systems, computer architects traditionally rely on software simulation to evaluate new designs. Although software simulation excels in convenience and flexibility, it suffers from prohibitively long turnaround time. Researchers are constantly searching for methods to accelerate software simulations. SMARTS is a framework that uses rigorous statistical sampling to accelerate simulation time without sacrificing accuracy. Its turnaround time is limited by the speed of the functional warming mode, which updates architectural state and select microarchitectural structures.

This paper presents direct warming as an efficient technique for accelerating functional warming. Direct warming extends direct execution, in which the simulated program code is executed natively on the host machine hardware rather than through emulation. To achieve identical simulation behavior to functional warming, direct warming integrates instrumentation code for record generation into the direct execution code. In this paper, we investigate and analyze several implementation alternatives to maximize the performance of direct warming by evaluating a collection of benchmarks on the RSIM simulator. On average, Direct SMARTS achieves a 96x speedup over full detailed simulation, with a maximum speedup of 134x. In addition, with the Direct SMARTS framework, we achieve an average error of 0.4%, with an upper bound of 0.7%.

Table of Contents

List of Figures

List of Tables

1 Introduction

Software simulation is necessary for microprocessor system design and analysis. Not only does software allow for the easy implementation of new ideas, but it facilitates the testing of new designs on many different configurations. Many times, a simple change in the configuration file or the initialization of a variable to a different value is sufficient. Unfortunately, the drawback to this design methodology is that simulation with software is orders of magnitude slower than the native runtime of the program on the actual hardware systems. The more detailed and accurate the simulator, the longer the total simulation time compared to native execution. With the complexity of current microprocessors growing dramatically and the instruction counts of common benchmark suites increasing with each new release, the execution time for accurate simulation of these microprocessors becomes progressively longer. It has become impractical to run full detailed simulations on the benchmark suites to model newly developed designs. Depending on the scale of the tested benchmarks, researchers can spend days and even weeks waiting for simulations to conclude.

As a result, researchers are constantly attempting to find alternative methods to shorten the runtimes of their software simulations. These methods, which include using truncated simulation runs and input sets that allow for smaller memory footprints, often lead to inaccurate measurements and misrepresented conclusions about the tested design [9]. Recently, proposals utilizing either statistical or trace-driven techniques have been introduced [3][12][15][7]. These techniques use functional simulation to skip to strategically chosen sections, or "sampling units," of the benchmark for measurements. Functional simulation emulates program behavior by updating the architectural state of the simulator and only necessary microarchitecture structures. By performing detailed simulation only on the chosen sampling units, researchers can reduce runtime considerably, while maintaining measurement accuracy. One technique, called the Sampling Microarchitecture Simulation (SMARTS) framework, has shown significant speedup with minimal error by applying systematic statistical sampling. However, the SMARTS simulation time is limited by the runtime of the functional simulator.

This thesis introduces a straightforward method for accelerating the functional simulation portion of the SMARTS simulator, called *direct warming*. The contributions of this thesis are as follows:

- *Direct Warming.* We explore a method of integrating direct execution in the functional warming mode of the SMARTS framework to further improve simulation turnaround time. Different schemes to optimize for performance and memory space are explored and evaluated.
- *General and Applicable.* This thesis presents results from integrating the SMARTS framework into another simulator with a different architecture and warming technique, verifying that the technique is general and applicable to different simulators.

1.1 Related Work

A simple and common technique used for dealing with prohibitively slow simulation times is to collect results based on abbreviated execution runs. In this technique, researchers bypass the initialization phase of the benchmark (e.g. usually 100 million instructions), which does not represent a program's typical behavior. Then, they continue with detailed simulation of the benchmark for a large number of instructions (e.g. 500 million instructions). However, drawing conclusions based on a single snapshot of the benchmark execution may be erroneous and misleading. Many programs go through different phases of execution, where each phase is repeated throughout the benchmark run. The behavior of a single phase of the benchmark may be vastly different from the other phases. More importantly, the observed behavior from the snapshot may not be representative of the program as a whole.

An alternative technique is to use reduced data sets. For example, the Spec2000 benchmark suite [11] provides test inputs, which have execution times that are a fraction of the reference input sets. However, the execution paths of these input sets are sometimes very different from those taken by the reference input sets. Therefore, behavior observed from the test inputs may not be representative of the reference input behavior. Hsu, et al. [9] report that test inputs of certain benchmarks in the Spec2000 suite have up to 300% error compared to the reference inputs when measuring the benchmark IPC (instructions/cycle).

Recently, researchers use more sophisticated trace-driven simulation methods to reduce execution time [3][12][15]. Specialized algorithms choose which instructions need to be simulated. Then, a trace is produced for these instructions and the detailed simulator is run according to the information in the trace. Since most programs have repeating phases, instructions in these phases need only be simulated once rather than many times. Not only does this method reduce the final execution time of the simulation, but since the traces are reusable, this method is very useful when running multiple simulations with the same simulator configuration. However, instruction traces can be extremely large and the overhead required to produce them prior to the actual simulation adds considerable time to the overall simulation time. Lauterbach [12] reduces trace creation overhead by using multiple computers to produce the traces in parallel, but this solution is very costly and most researchers have limited computing resources.

Several direct execution simulators have also been introduced. Embra, which is part of the SimOS project, uses dynamic binary translation to generate machine code for the native host [16]. Since machine code is generated during simulation, workloads are determined on the fly. Fujimoto and Campbell [10], on the other hand, make modifications directly to the benchmark binary by converting the binary into a higher level intermediate language, inserting timing code, and then retranslating the intermediate code for execution on the native host.

Sherwood, et al. [15] developed a tool called SimPoint, which utilizes Basic Block Vectors to automatically capture the behavior of programs over billions of instructions. Through a clustering algorithm, they determine the minimal sections of code which need to be executed in detail. Through the use of SimPoint, they achieve an average IPC error of 3%. Wunderlich, et al. [7] found that SimPoint implemented on SimpleScalar [1] is slightly less than twice as fast as the SMARTS framework, which is presented next, on the same simulator.

Wunderlich, et. al. [7] introduced an on-line algorithm for accelerating simulation time, utilizing both statistical sampling and *functional warming*. Functional warming is similar to functional simulation, except that the former only updates architectural state, while the latter also updates long history structures like branch predictors, caches, and TLBs. The basic concept behind this framework is to take measurements on a large number of sampling units, each consisting of roughly 1000 instructions. Functional warming is performed between sampling units. Before the start of the measurement phase of each sampling unit, only a minimal amount of detailed warming (detailed simulation with all measurement variables deactivated) is performed to bring the smaller structures (e.g. the load/store queue) up to date.

Sherwood et al. [15] also suggest the use of check-pointing for the start of a simulation period. Information about both the architectural and microarchitectural state of the simulation environment is stored in these checkpoints. Thus, making functional warming unnecessary. Not only this, but the checkpoints are reusable if simulator configuration is constant. However, the overhead for creating these checkpoints is large.

1.2 Organization Outline

The remainder of this paper is organized as follows. Chapter 2 introduces background information required to understand the problem space, including a detailed description of the SMARTS framework and the RSIM simulator. Chapter 3 describes the components of Direct SMARTS and presents the trade-offs of the different performance optimizations which we analyzed. Chapter 4 concludes the paper, discussing possible extensions and future work.

2 SMARTS Background

This chapter gives the necessary background information to understand this thesis. Section 2.1 briefly covers the SMARTS statistical sampling framework. Section 2.2 describes the details of the modified RSIM simulator that we use as a foundation for our work. Section 2.3 provides a proof of concept for the SMARTS framework and evaluate the results presented in the SMARTS paper.

2.1 The SMARTS Framework

The Sampling Microarchitecture Simulation (SMARTS) framework [7] introduces an on-line simulation technique that utilizes rigorous systematic sampling to achieve fast and accurate simulation without the overhead of creating traces with the trace-driven techniques presented in Section 1.1. Statistical sampling states that information obtained from a subset of a population can be representative of the population as a whole. Systematic sampling spaces out the sampling units at regular intervals. We use systematic sampling to facilitate implementation of the framework. Well-established statistical principles are applied to obtain measurements from this subset of instructions, which are used to estimate measurements for the entire population [13].

Figure 1 shows the basic functional layout of the SMARTS framework. SMARTS can accurately estimate simulations by taking measurements on many small sampling units, evenly spread throughout the

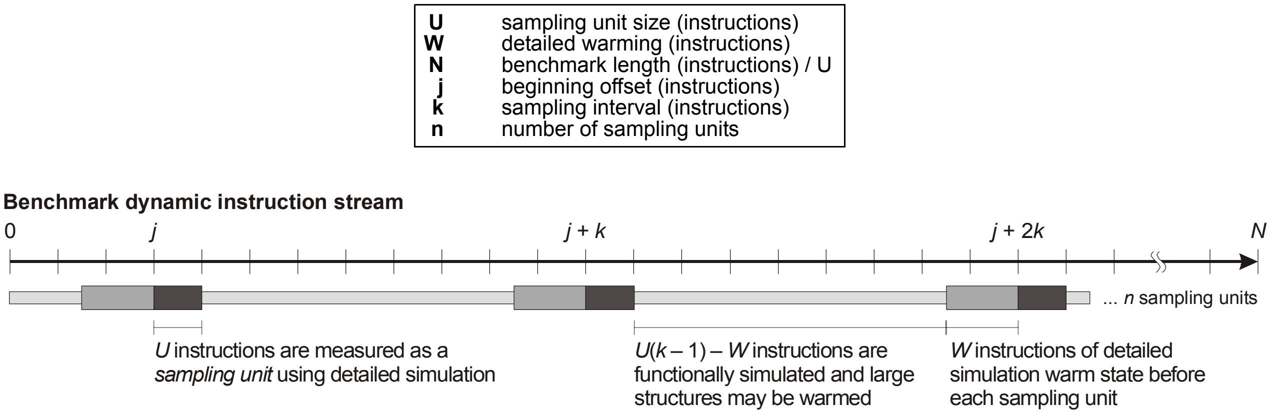
U sampling unit size (instructions)
W detailed warming (instructions)
N benchmark length (instructions) / U
j beginning offset (instructions)
k sampling interval (instructions)
n number of sampling units
Benchmark dynamic instruction stream
0
j
j + k
j + 2k
N
... n sampling units
U instructions are measured as a sampling unit using detailed simulation
U(k – 1) – W instructions are functionally simulated and large structures may be warmed
W instructions of detailed simulation warm state before each sampling unit

FIGURE 1: The SMARTS Framework. SMARTS switches between functional simulation of U(k-1) instructions and detailed simulation of U instructions. W instructions are needed for detailed warming. Figure above is courtesy of [7].

duration of the benchmark. Between sampling units, the simulator emulates the microarchitectural effects of the simulated instructions in a mode called *functional warming*, which we describe in the following section.

2.1.1 Warming

The most challenging part of the framework is to determine how to recreate the proper state of the simulator at the point immediately prior to the detailed simulation measurement period. In order to reproduce this state, detailed warming should be applied to a finite number of instructions prior to the measurement period. Detailed warming is equivalent to detailed simulation, without statistical counters activated. However, since detailed warming is as expensive as full detailed simulation, long intervals of detailed warming significantly add to the total simulation execution time. In addition, data structures with long histories, such as the caches, TLBs, and branch predictors, need large amounts of detailed warming. A less expensive functional warming mode executed continuously is more appropriate. Recall that functional warming is similar to functional simulation in that the simulator's architectural state is updated during emulation of the instructions. However, unlike functional simulation, functional warming also keeps long history structures like caches, TLBs, and branch predictors updated.

Wunderlich, et al. [7] determined that with the SMARTS framework, more than 99% of instructions are emulated in the functional warming mode, meaning that less than 1% of the instructions are executed in detail. Thus, the speed of the SMARTS simulator is dependent on the speed of the functional warming simulator. Therefore, optimizations to the functional warming mode of the simulator are expected to shorten the total execution time of the SMARTS simulator.

2.1.2 Switching

Another challenge is determining how to switch between the detailed and the functional warming modes of the simulator. Switching from the functional warming to detailed simulation mode is trivial because the functional warming mode executes instructions in order.

Switching from the detailed simulation mode to the function warming mode is complex. The detailed simulator can execute instructions out of order, which means that instructions may get reordered in the pipeline. In addition, instructions may take multiple cycles to complete, and all instructions in the pipeline must complete prior to the start of the functional warming mode. Thus, the entire pipeline and memory system need to be flushed before functional warming begins. This means that at the time when the functional simulation mode is about to begin, if there is a memory access request in the ports of the L1 data cache, the L1 cache needs to process this request, the L2 cache could need to process this request, and the directory may need to update an entry as well, all before the functional warming mode can begin.

2.2 The RSIM Simulator

We use the Rice Simulator for ILP Multiprocessors (RSIM) [14] in this study. RSIM is a shared memory, directory based, multiprocessor simulator. We conduct our experiments with a single processor. This serves as a proof of concept that the SMARTS framework is applicable to different architectures. In addition, having the SMARTS framework implemented on a multiprocessor simulator can initiate research to the extension of SMARTS to a multiprocessor environment without having to deal with the implementation details of sampling.

2.3 Applying SMARTS to RSIM

2.3.1 Machine Configuration and Benchmarks

Table 1 shows the machine configuration for RSIM used to perform testing and verification. We choose configuration parameters similar to those used by Wunderlich, et al. [7]. By choosing similar configurations, we expect to see benchmark behavior similar to the previous implementation.

We run simulations on a quad processor 480 MHz UltraSPARC-II system. Each CPU has a 16 KB L1 instruction cache, 32 KB data cache, and 8 MB L2 cache.

Parameter	8-way Superscalar
Active List size	**128**
Memory System	**32 KB 2 way L1 I/D, 8 MSHRs** **1024 KB 4 way L2, 8 MSHRs**
ITLB/DTLB	**128 entries, fully associative**
L1/L2tag/L2data/mem latency	**1/6/12/1000 cycles**
Functional Units	**4 I-ALU, 2 FPUs, 4 address generation units**
Branch Predictor	**2 bit history, 2K tables, 11 shadowmappers**

TABLE 1. RSIM Machine Configuration.

Table 2 shows the benchmarks and input data sets that we use in our experiments. These benchmarks are from the Splash-2[8], Olden[2], and Spec2000[11] benchmark suites. The SMARTS framework is only beneficial when benchmarks are long. Otherwise, the difference in wait times between full detailed simulation and SMARTS is insignificant. The three benchmarks in Table 2 have been scaled to result in instruction counts between 1 and 10 billion instructions.

2.3.2 Verification of Optimal Parameter Values

We first need to determine the optimal sampling unit size. Figure 2 plots the coefficient of variation (V_{CPI}) for various sampling unit sizes for the selected benchmarks. The V_{CPI} determines the variance of CPI (cycles/instruction) in a benchmark. Benchmarks with higher V_{CPI} are harder to estimate with sampling because more instructions need to be executed in detail to achieve the desired accuracy. The figure shows that V_{CPI} starts out high, sharply decreases, and finally levels off. The cause of the sharp drop is due

Benchmark	Input Data Set
Barnes Hut	**4K particles**
Em3d	**32 nodes/thread, degree 8, 15% remote, span 8, 10000 steps**
Tomcatv	**array size 1024, 10 iterations**

TABLE 2. Benchmarks Used in Evaluation.

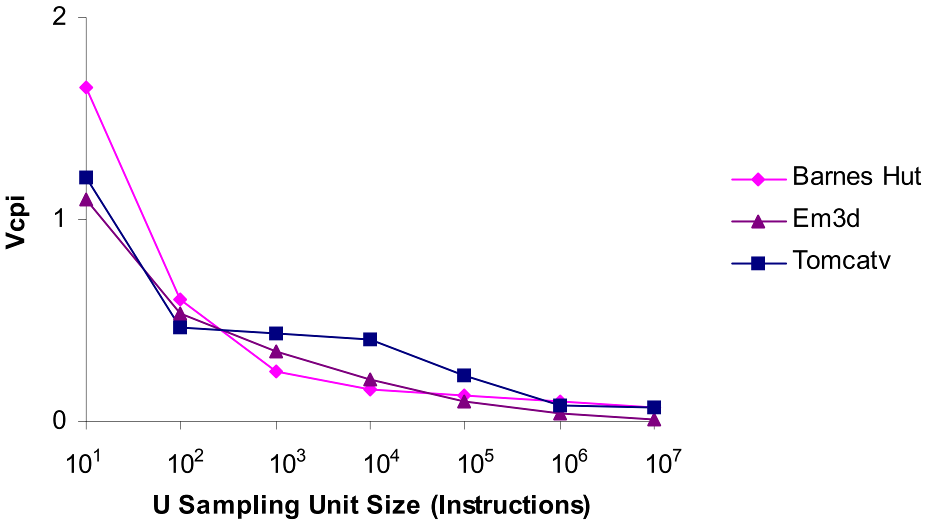

FIGURE 2: U vs. Coefficient of Variation of CPI (V_{CPI}).

to the large variation that individual instructions can have. CPI can be low when many instructions are being executed and the superscalar pipelines are full; yet CPI can be high due to long latency memory instructions. Thus, with a sampling unit size of 1, V_{CPI} is high. However, once the sampling unit size increases to multiple instructions, V_{CPI} drops significantly because the variations for each instruction average out. When the sampling unit size reaches 1000 instructions, short-term variations have been averaged out, so variations in CPI are actually due to overall benchmark behavior rather than short-term variations between instructions.

The optimal sampling unit size cannot simply be chosen at the point where the V_{CPI} stabilizes. We select the optimal sampling unit size so as to result in the smallest total number of instructions executed in detail as possible, where detailed simulation is the cycle-accurate timing model of the simulated architecture. Figure 3 shows the percentage of instructions executed in detail as a function of the sampling unit size. We calculate the minimum number of sampling units analytically (see Wunderlich, et al. [7] for more detailed information about calculation of minimum number of sampling units). We found that none of these benchmarks need more than 10,000 sampling units to reach the desired accuracy. Thus, we set n_{tuned}

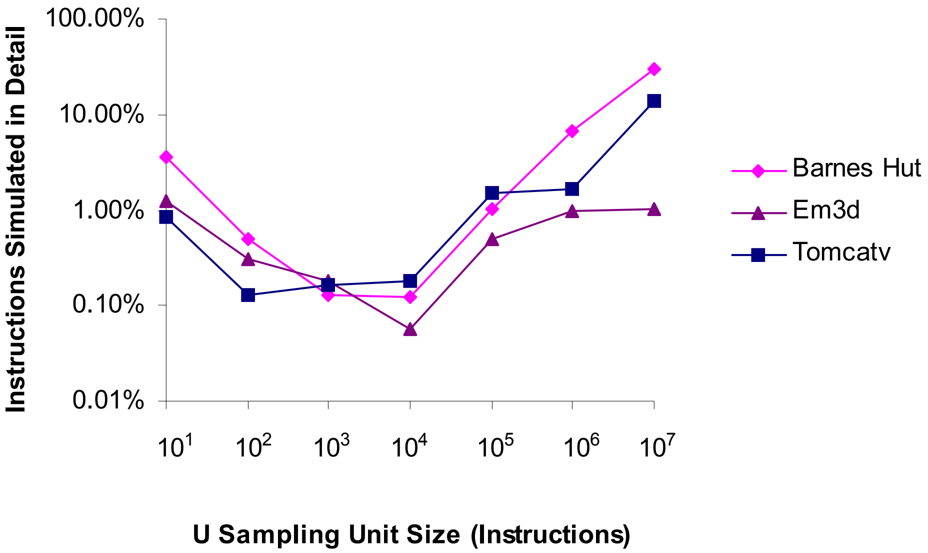

FIGURE 3: Instructions Simulated in Detailed with W = 2000 Instructions.

= 10,000. The dip in this graph is where the minimum number of instructions lies, which is near U=1000 instructions.

Figure 2 and Figure 3 confirm that the benchmarks used in our experiments have very similar behavior to those benchmarks presented in the previous SMARTS paper. It is reasonable to conclude that the same parameter value of U=1000 instructions works here as well. For Em3d, Figure 3 shows the minimum point to be at a sampling unit size of 10,000 instructions. However, through practice, we found that using a sampling unit size of 1000 instructions is sufficient for Em3d as well. In addition, through experimentation, we found that W=2000 instructions results in the desired accuracy. For a more detailed analysis and explanation see [7].

Wunderlich, et al. [7] verifies the accuracy of the SMARTS framework in SimpleScalar[1], a superscalar uniprocessor simulator. We already determined the optimal parameters. Now, we need to check that the accuracy of the implementation of Direct SMARTS is within acceptable bounds. The details of implementing Direct SMARTS will be discussed in the next chapter, but to complete the verification section, the

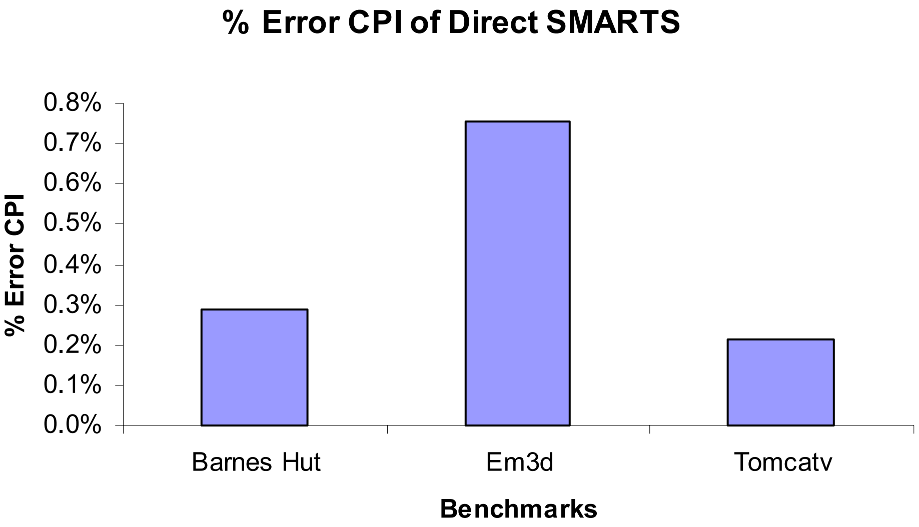

FIGURE 4: Accuracy of Direct SMARTS with U=1000 Instructions and W=2000 Instructions.

resulting accuracy is presented here. As seen in Figure 4, the average CPI measurement has an error of less than 0.75%. We present optimal speedup numbers in the next chapter.

3 Direct SMARTS

3.1 RSIM's Direct Execution Mode

The RSIM simulator has a direct execution mode which executes dynamically generated machine code instructions [4][1]. RSIM has the ability to switch from the detailed simulation mode into the direct execution mode, and vice versa. Figure 5 shows the functional flow diagram of the direct execution mode of RSIM. The direct execution mode takes as input RSIM formatted instructions and does a lookup in the translated cache to see if the corresponding basic block has previously been translated into machine code. There is a translated cache, which maps basic block PCs to the corresponding location in memory of their translated machine code. If the basic block entry already exists, then the corresponding machine code is executed natively on the host machine. Otherwise, the simulator exits from direct execution, translates the

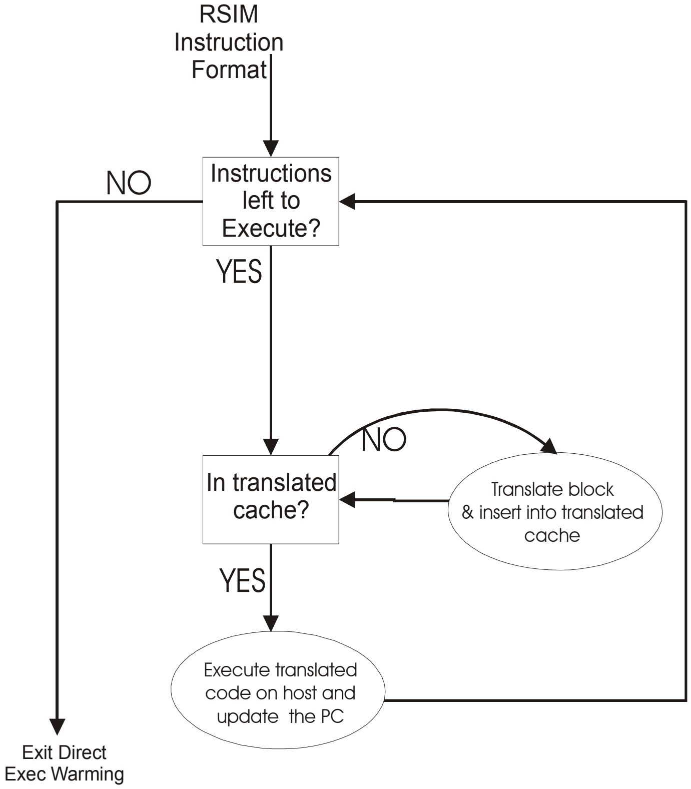

FIGURE 5: Functional Block Diagram of Direct Execution Mode of RSIM.

[1] courtesy of Professor Vijay S. Pai of Electrical and Computer Engineering at Rice University

block, and creates a new entry in the translated cache. The simulator then re-enters direct execution mode, executes the most recently translated basic block, and does a lookup for the PC of the next basic block. If the simulator continues to hit in the translated cache, RSIM stays in direct execution mode until the end of the program.

3.2 Speedup Opportunity of Direct Execution

Historically, direct execution means the execution of the exact same binary in the simulator as you would execute on the native host [5]. Usually, the binary is uncompiled into a higher level language, instrumentation is inserted, and the code is recompiled for execution on the native host. Thus, the register file and memory of the native host are updated during simulation. However, much like Embra [16], RSIM executes its own dynamically generated machine code. Although we are "directly executing" machine code on the host machine, we are making updates to the simulated environment, not the native environment.

In [7], functional warming was limited to an average of 35x speedup over detailed simulation. However, in functional warming, we still emulate instructions. Thus, considerable overhead is incurred compared to direct execution of the benchmark. Table 3 shows the speeds of the various modes of RSIM compared to detailed simulation. For a benchmark which runs for an hour on the native machine, it would take an average of 118 days to complete the same benchmark with detailed simulation. With functional execution, the execution time is reduced to approximately 8.6 days. However, running direct execution on the same benchmark takes only about 8 hours. By integrating direct execution into the SMARTS framework, we can speed up the execution time of SMARTS by an order of magnitude over functional warming.

Benchmark	Func Exec	Direct Exec	Native Exec
Barnes Hut	**15.15**	**100.04**	**2487.29**
Em3d	**11.54**	**803.78**	**2826.33**
Tomcatv	**14.15**	**175.85**	**1257.14**

TABLE 3. Speedup of Different Simulation Modes vs. Detailed Simulation.

3.3 Implementing Direct Warming

After seeing the speedup opportunity available with direct execution, we need to determine the best way to integrate the needed warming instrumentation into direct execution while minimizing the overhead.

We have two separate records, which keep the information for cache accesses and for branch predictor updates. Table 4 shows the information that needs to be collected to warm each structure. The d- and i-cache accesses are kept in the same record to ensure that the interleaving of the accesses are conserved, which can influence the state of the L2 cache.

To perform an i-cache update, we need a flag to indicate that this entry is an i-cache access (as opposed to a d-cache access) and the instruction PC. For a d-cache access, however, we need the flag to indicate a d-cache access, the memory address being accessed, and the flag indicating a read or write. Having a separate flag to indicate a d-cache or an i-cache access simplifies code translation when updating an entry, resulting in a more optimized simulator.

Another problem to resolve is determining exactly when to perform the warming updates. Ideally, we want to perform the updates when the corresponding instructions are executed during direct execution.

Structure	Parameters Needed for Warming
Memory System	Instruction PC Memory Address Data Cache Access? Write?
Branch Predictor	Instruction PC Unconditional Branch? Branch Hint Next PC

TABLE 4. Warming Parameters.

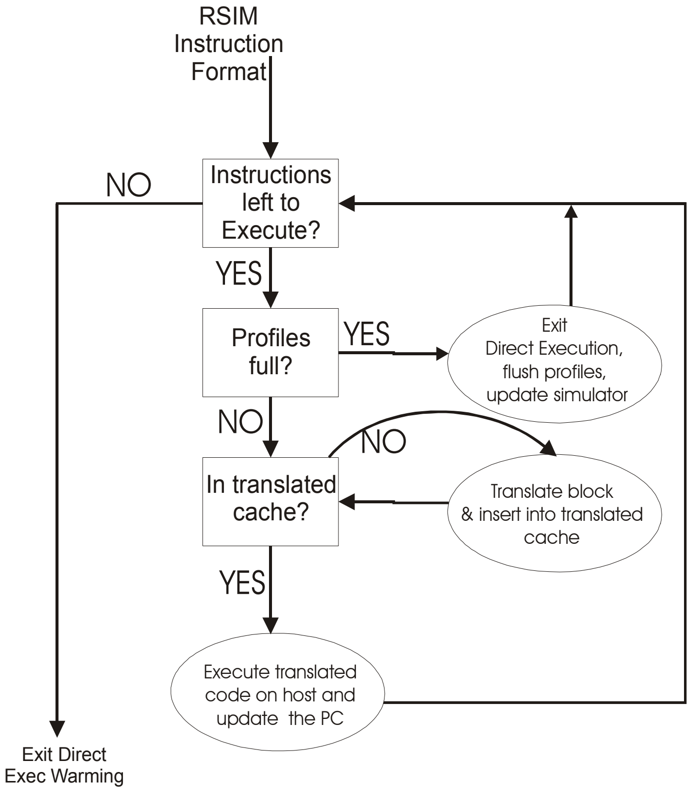

FIGURE 6: Functional Block Diagram of Direct Warming mode.

However, this is impractical. Searching through large data structures is tedious when programming in machine code. At a certain point, we realize that the compiler can create better optimized code than a person can. To achieve on-line updates with high-level code, we need to exit from direct execution every time a memory system or branch predictor update is performed. However, this is not ideal since switching between direct execution mode and the simulator comes with a 164 instruction overhead for setting up the different environments. We conclude that the best solution is to construct a record of memory system updates and a second record of branch predictor updates. After exiting from direct execution, we initiate the post-processing to update the caches, TLBs, and branch predictors to complete the direct warming phase. Figure 6 shows the functional block diagram for direct warming.

Figure 7 depicts the base configuration layout of records for the branch predictor and memory system. Each field of each entry in the records is an integer field, or 4 bytes. This is necessary for the address fields since they are 32-bits, but the other fields are just boolean values. We later explore the use of bitmaps to

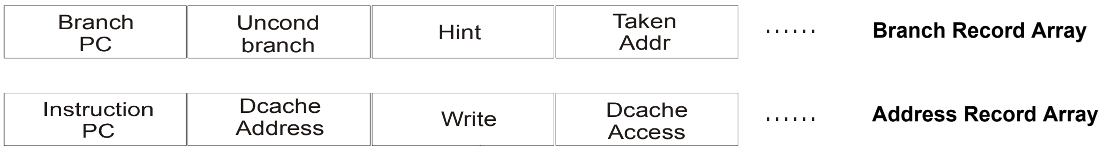

FIGURE 7: The Base Case Configuration.

minimize space overhead. Since absolute speed is our main goal, minimizing the total instruction count is more important than space conservation. A single record for both d- and i-cache accesses is needed to ensure that we conserve the interleaving of the accesses, which affects the accuracy of the unified L2 cache at the start of each measurement cycle. In addition, implementing a generic memory access entry rather than different types of entries for d- and i-cache accesses simplifies the processing needed for the records.

3.4 Optimizations

In this section, we evaluate the different optimizations described in the previous section.

3.4.1 Record Size Sensitivity Analysis

First, we want to determine the number of entries in each record. It is impossible to let direct warming run indefinitely because there is no upper bound on the size to which the records may grow. It is possible to dynamically reallocate memory for the records each time they fill; however, this involves considerable overhead and is not a practical solution. Dynamically increasing the record size can cause them to eventually grow large enough to fill the host caches, creating slowdown due to thrashing. A more efficient solution is to break out of direct execution when a record fills up and conduct the post-processing for memory system and branch predictor updates at this time.

Because it is impractical to test all the permutations of every size for each record, we evaluate three representative sizes: a small size (8 entries for cache,16 entries for branch predictor), a large size (4096 entries for cache, 4096 entries for branch predictor), and a middle size which we predict to be the optimum (512 entries for cache, 512 entries for branch predictor). For the large record size, we choose a size that is large enough to cause thrashing in the L1 cache of the host machine (4096 entries for each record, resulting

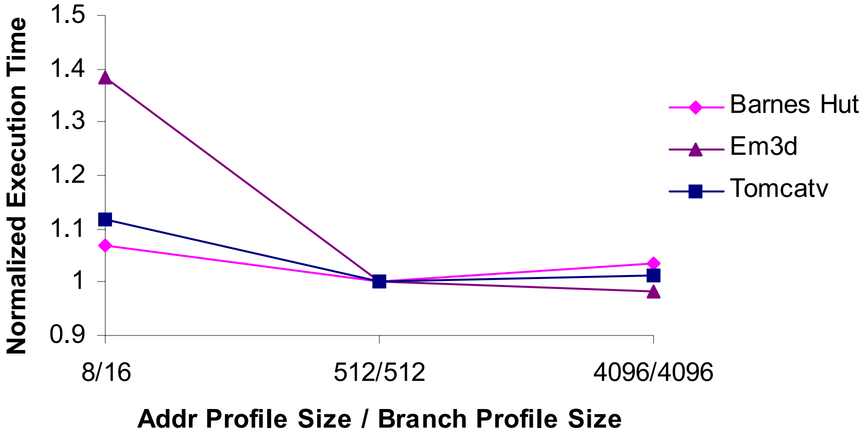

FIGURE 8: The Sensitivity of Record Size. Results normalized to 512/512 record size.

in 128KB of memory, on a host machine with a 32KB L1 cache). We used the base configuration record layout described in the previous section to evaluate sensitivity of record size.

Figure 8 shows select results from the described sensitivity analysis. Note that the results show normalized execution times. Although we did perform the sensitivity analysis on all permutations of the three sizes for each of the records, many of these results are uninteresting as they all lie within the bounds depicted above. We see that 512 entries for each record is, as predicted, the optimal size for the sensitivity analysis, although 4096 entries is not much slower. Em3d is actually faster for the 4096/4096 record configuration. This is because the benchmark contains many tiny loops and hits frequently in the translated cache so it stays in direct warming for long periods of time. This fills up the records. Thus, increasing the size of the records does benefit the benchmark. The smaller size records are significantly slower because they cause direct warming to break out of direct execution too frequently.

3.4.2 Quick Hits Array

We evaluate the implementation of a quick hits array, which is an array that keeps a record of the most recently touched address for each set in the cache [6]. If the next address is the same as the last touched address of the set, the next access is not recorded into the record. Programs with high temporal locality benefit the most from this optimization.

3.4.3 Reducing Storage Overhead

As mentioned previously, there are several fields in each record entry that hold boolean values. These can be stored in a bitmap to save significant storage space. With bitmaps, each entry is about half the size of the record entry from the base case. However, this scheme requires additional processing time to execute. The number of instructions to create the mapping and modification of the bitmap value may outweigh the benefits of space conservation.

The advantage of the base case profiling scheme is that the record arrays are cache line aligned. However, this is not true in the bitmap scheme. To simplify the structures, we decided to separate the bitmaps from the integer fields of the records. The layout for the bitmap scheme is shown in Figure 9. Having separate arrays for each entry also increases processing time by introducing the possibility for multiple cache misses per entry. Although each boolean field only requires a single bit to store (rather than a 32-bit integer used in the base case), the bookkeeping of the bitmaps during direct execution and the post-processing following direct execution requires more instructions. To update one bit in the bitmap, a new bitmap needs to

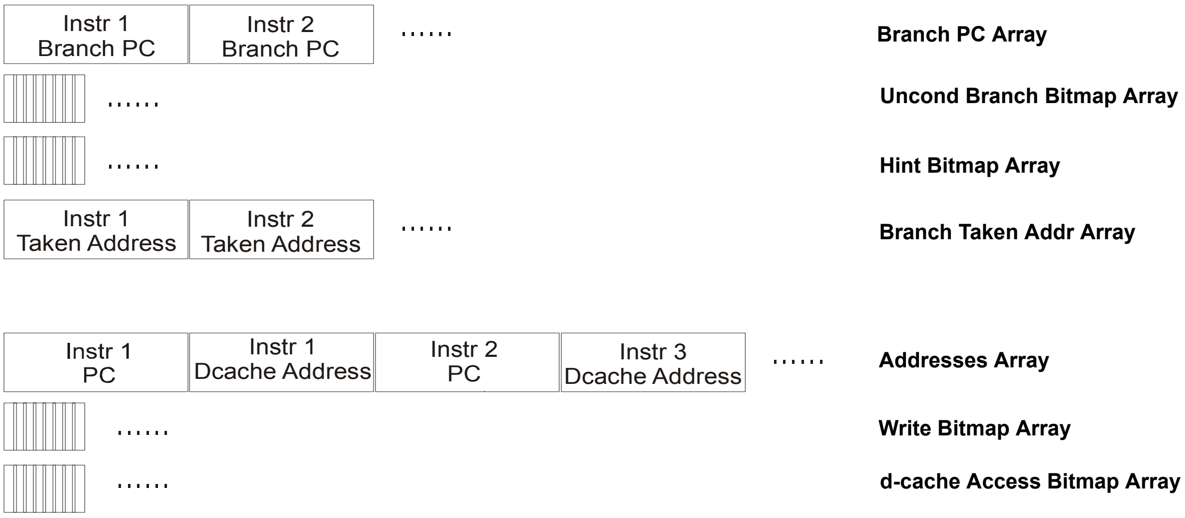

FIGURE 9: The Bitmap Array Scheme.

be created, the original bitmap value needs to be loaded, modified, and stored. This results in significantly more machine code instructions generated per RSIM instruction compared to the base case.

3.4.4 Results/Analysis

Figure 10 shows the effects on execution time for the different optimizations just described. We decided to evaluate not only the 512 entry record size, which was the optimal record size, but also the 4096 entry record size, in which the bitmap array scheme should be beneficial.

As predicted, the quick hits array scheme contributes a considerable amount of performance benefit for benchmarks with many small loops like Em3d, which showed almost a 7x speedup. On average, adding a quick hits array resulted in a 3.3x speedup for the benchmarks shown.

For the bitmap array scheme, we initially transferred the branch record into the bitmap array to observe the effects of reducing the memory space on only that record. As we already concluded that the quick hits array is beneficial, we implemented the bitmap scheme with the quick hits array optimization. These are the results shown in Figure 10. From just implementing bitmap arrays for the branch record, the processing overhead is already large enough to slow down the total execution time compared to the quick hits optimization. Based on these results, we find it unnecessary to transfer the memory access record into the bitmap

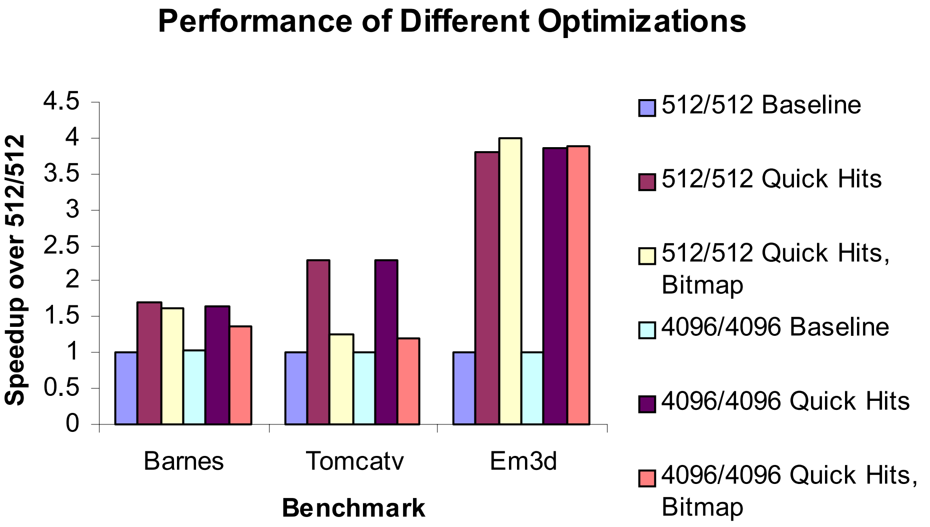

FIGURE 10: Performance of Different Optimizations. ***X*/*Y* = *X* addr record entries / *Y* branch record entries.**

array scheme. This would only increase the processing time overhead, slowing down the overall runtime of the application even more.

Thus, the optimal configuration for direct warming is the quick hits scheme, which has a clear advantage over the bitmap scheme. The additional amount of processing that is needed for the bitmap scheme outweighs the benefits of the compression. The more memory instructions or branch instructions that are being executed, the larger the slowdown of the bitmap array scheme. Appendix 1 in Chapter 6 shows the optimal code translation of a memory instruction with direct warming for RSIM.

3.5 Speedup of Direct SMARTS

After optimizing the direct warming mode, we integrate it into the Direct SMARTS framework. We showed previously that the evaluated benchmarks have similar characteristics to the Spec2000 benchmarks from [7]. We set the parameters as in the original paper: U=1000 instructions, W=2000 instructions, and n_{tuned} = 10,000 instructions.

Figure 12 shows the speedups achieved with Direct SMARTS. The number of instructions run in detailed simulation is constant: (2000 instructions in detailed warming +1000 instructions in detailed sim-

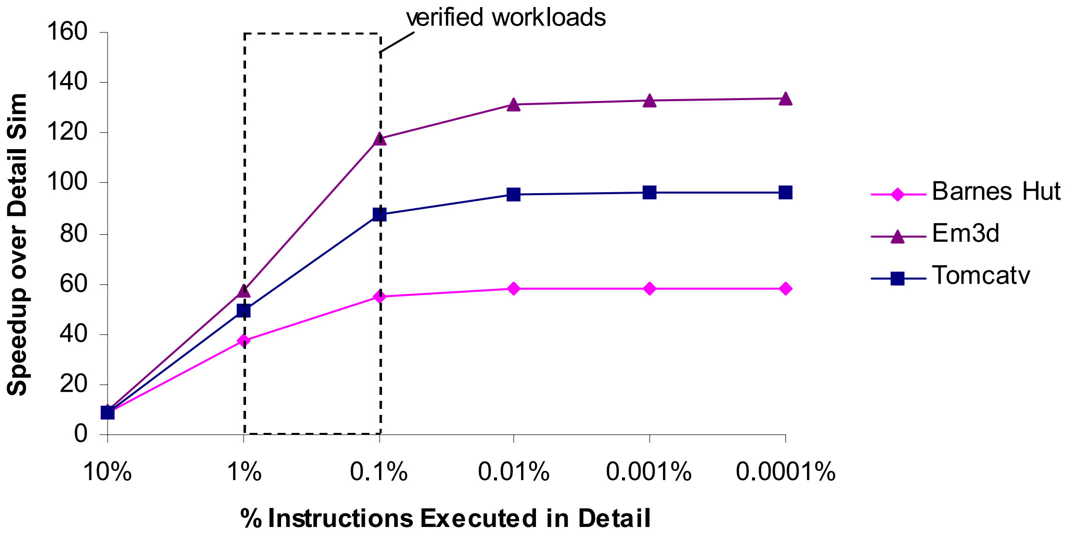

FIGURE 11: Optimal Speedups For Select Benchmarks. The boxed area represents the verified speedup numbers.

Benchmark	Func Warm	Func Exec	Direct Warm	Direct Exec	Native Exec
Barnes Hut	**13.43**	**15.15**	**58.32**	**100.04**	**2487.29**
Em3d	**10.43**	**11.54**	**133.42**	**803.78**	**2826.33**
Tomcatv	**12.54**	**14.14**	**96.29**	**175.85**	**1257.14**

TABLE 5. Speedup vs. Detailed Simulation Including Direct Warming and Function Warming Modes.

ulation) x 10,000 samples = 30,000,000 instructions. Thus, the higher the total number of instructions in a program, the higher the speedup benefits we see from using Direct SMARTS. The boxed area in Figure 12 shows the range of instruction counts that we verified. The average speedup is 95x, with a maximum speedup of 132x compared to the full detailed simulation runs of the benchmarks. We use the results obtained from these experiments to calculate the speedups for larger input sets. It is impractical to spend the time to actually run these long programs because it takes considerable time to run the full detailed simulations on these longer applications. From our analysis, maximum speedup is achieved when only 0.001% of the benchmark is run in detail, which in this case means a total instruction count larger than 3 trillion instructions, taking 774 days on average to complete with full detailed simulation.

Table 5 shows the speedups of the various modes of RSIM, including the warming modes, over detailed simulation. Although direct warming is significantly faster than functional warming, it is still considerably slower than direct execution. Direct warming is 3.4x slower than direct execution, but functional warming is only 1.12x slower than functional execution. There are multiple reasons for this. First, direct warming has an initialization phase, in which the basic blocks are first translated into machine code. The larger the code footprint, the more time lost to initialization. Thus, benchmarks with small memory footprints benefit most from direct warming. In addition, the number of instructions that need to be recorded into the records also affects the resulting speedup of direct warming. A single RSIM memory instruction translates into approximately 35 native instructions and a single RSIM branch instruction translates into approximately 17 native instructions. Therefore, benchmarks with a higher number of memory and branch instructions will exhibit smaller speedup because of the additional overhead of record generation.

4 Conclusions

Software simulation is essential in the design of hardware systems due to its convenience and flexibility. However, with the increasing complexity of many hardware designs, the runtime of these software simulations have become extremely time consuming.

We presented a simple and straightforward technique called direct warming, which is a faster alternative to functional warming. We analyzed the effects of different optimizations on execution time of the chosen benchmarks. We found that the quick hits array scheme has the largest performance benefits. The bitmap array scheme reduces space usage, but its added processing time is too expensive.

After the development of the direct warming mode, we evaluated the performance benefits of the integration of direct warming into the Direct SMARTS framework. We implemented the framework on the RSIM simulator and tested it on several benchmarks. The results of the experiments demonstrate an average of 0.4% error, with an upper bound of 0.7%. In addition, we were able to achieve speedups of 96x speedup, with a maximum speedup of 134x.

In addition to the evaluation of more benchmarks, we plan on expanding the speedup optimization to increase the performance of the direct warming mode of RSIM. For example, certain checks are precautionary rather than necessary. In addition, any optimizations to the translated code also reduces overhead for direct execution. Since the instruction translations are run so frequently, small optimizations have a significant impact on overall runtime. Further research can also explore ways of implementing Direct MultiSMARTS, the direct execution, multiprocessor version of SMARTS. Multiprocessor research with this statistical sampling technique becomes more complex due to interactions between processors. Speedup is limited by the amount of communication between processors, which is by itself an uncertain quantity. The amount of warming required is also unclear because of the behavior of the network, in that messages can be lost or stalled indefinitely.

5 Appendix

```
# check if there's enough room in the addresses array.
sethi REG_SPARE5, num_addresses_touched
setlo REG_SPARE5, num_addresses_touched
lduwREG_SPARE5, 0(REG_SPARE4)
sethi REG_SPARE2, MAX_DE_INSTS-2
setlo REG_SPARE2, MAX_DE_INSTS-2
sub REG_SPARE2, REG_SPARE4, REG_SPARE2
srl REG_SPARE2,31,REG_SPARE2
int branch_blockindexDestAddr = remapped_blockindex;
brz_PT REG_SPARE2 # sign bit was 0 (pos.)
nop

# array is too large. Need to exit direct exec and flush the array
(FailureCase)

# put PC value into spare register
sethi REG_SPARE1, pc << 2
setlo REG_SPARE1, pc << 2

# check for an i-cache hit.
srl REG_SPARE1,captr_block_bits,REG_SPARE2
sethi REG_SPARE1,FastSimICacheReadHits
setlo REG_SPARE1,FastSimICacheReadHits
andi REG_SPARE2,NUM_HITARRAY_ENTRIES-1,REG_SPARE3
sll REG_SPARE3,2,REG_SPARE3
# load tag in that hit position
lduw REG_SPARE1,REG_SPARE3,REG_SPARE3
sub REG_SPARE2,REG_SPARE3,REG_SPARE3
retval = remapped_blockindex;
brz REG_SPARE3
# update the quick hits tag array.
andi REG_SPARE2,NUM_HITARRAY_ENTRIES-1,REG_SPARE3
sll REG_SPARE3,2,REG_SPARE3

# i-cache MISS:
# increment the counter
addi REG_SPARE4, 1, REG_SPARE2
stw REG_SPARE5, 0, REG_SPARE2

# record the pc of the instruction
sethi REG_SPARE1, pc
setlo REG_SPARE1, pc
sethi REG_SPARE2, addresses_touched
setlo REG_SPARE2, addresses_touched
sll REG_SPARE4, 4, REG_SPARE4
stw REG_SPARE2, REG_SPARE4, REG_SPARE1

# mark entry in d-cache access as zero as well
addi REG_SPARE4, 12, REG_SPARE4
stw REG_SPARE2, REG_SPARE4, 0

# enter displacement if i-cache HIT.
branch_blockindexIC |= remapped_blockindex-branch_blockindexIC

sethi REG_VTLBCHECK,&FastSimVTLBCheck
setlo REG_VTLBCHECK,&FastSimVTLBCheck
lduw REG_VTLBCHECK,0,REG_VTLBCHECK
sethi REG_VTLB,&FastSimVTLB
setlo REG_VTLB,&FastSimVTLB
lduw REG_VTLB,0,REG_VTLB

# First, copy the operation and change it to an add (for addr. gen.)
addi r2, imm, REG_SPARE1

# manual handling of semi-aligned references
andi REG_SPARE1,READ
branch_blockindexADDR = remapped_blockindex;
brnz REG_SPARE2

srl REG_SPARE1,captr_block_bits,REG_SPARE2 # get line #
sethi REG_SPARE1,FastSimDCacheReadHits
setlo REG_SPARE1,FastSimDCacheReadHits
lduw REG_SPARE1,0,REG_SPARE1
andi REG_SPARE2,NUM_HITARRAY_ENTRIES-1,REG_SPARE3
sll REG_SPARE3,2,REG_SPARE3

lduw REG_SPARE1,REG_SPARE3,REG_SPARE3 # load tag in that hit
position

sub REG_SPARE2,REG_SPARE3,REG_SPARE3 # match if 0
```

```
retval = remapped_blockindex;
brnz REG_SPARE3 # branch around on success

# update the quick hits tag array. make this functional first, then cut down
the runtime.
andi REG_SPARE2,NUM_HITARRAY_ENTRIES-1,REG_SPARE3 #
index into hit-check array
sll REG_SPARE3,2,REG_SPARE3

stw REG_SPARE1,REG_SPARE3,REG_SPARE2 # update QuickHitsTa-
gArray

# recompute address since RabbitInlinedCacheAccess destroyed
addi r2, imm, REG_SPARE1

# Record the address accessed
sethi REG_SPARE5, num_addresses_touched
setlo REG_SPARE5, num_addresses_touched
lduw REG_SPARE5, 0, REG_SPARE2
addi REG_SPARE2, 1, REG_SPARE4# incr num_addresses_touched
stw REG_SPARE5, 0, REG_SPARE4

# the block_cache if d-cache MISS:
sethi REG_SPARE4, addresses_touched
setlo REG_SPARE4, addresses_touched
sll REG_SPARE2, 4, REG_SPARE2

# store address into the addresses_touched array
addi REG_SPARE2, 4, REG_SPARE2 # goto 2nd spot in entry
stw REG_SPARE4, REG_SPARE2, REG_SPARE1

# also update the write_addresses array if this is a write.
addi REG_SPARE2, 4, REG_SPARE2 # goto 3rd spot in entry
stw REG_SPARE4, REG_SPARE2, REG_SPARE1

# mark this as a d-cache access
addi REG_SPARE2, 4, REG_SPARE2
stw REG_SPARE4, REG_SPARE2, REG_SPARE1

# enter displacement if d-cache HIT.
branch_blockindexDC |= remapped_blockindex-branch_blockindexDC

# recompute address since RabbitInlinedCacheAccess destroyed
REG_SPARE1
addi r2, imm, REG_SPARE1

# Now, calculate the mapping page #
srl REG_SPARE1,12,REG_SPARE1
# now index into the VTLB-Checker
andi REG_SPARE1,VTLBSIZE-1,REG_SPARE2
sll REG_SPARE2,2,REG_SPARE2
lduw REG_VTLBCHECK,REG_SPARE2,REG_SPARE3
# now check if success
SUB_3REGS(REG_SPARE1,REG_SPARE3,REG_SPARE3
branch_blockindexVTLB = remapped_blockindex;
brz REG_SPARE3
lduw REG_VTLB,REG_SPARE2,REG_SPARE2 # do VTLB lookup for
success path
branch_blockindexADDR |= remapped_blockindex-
branch_blockindexADDR # displacement of address fault branch

# NOW DEAL WITH THE FAILURE CASE
remapped_block[branch_blockindexDC |= (remapped_blockindex-
branch_blockindexDC # displacement of d-cache miss branch

(Failue Case)

# NOW CONTINUE THE SUCCESS CASE
# load the page-mapping from the VTLB
remapped_block[branch_blockindexVTLB |= (remapped_blockindex-
branch_blockindexVTLB # displacement of success branch

# recompute address
addi r2, imm, REG_SPARE1

# get offset within page
andi REG_SPARE1,4095,REG_SPARE1

# actual execution of the instruction
LDUW r1, imm(r2)
```

Appendix 1: Code Translation for LDUW r1,imm(r2).

6 Acknowledgements

I would like to thank my advisor, Babak Falsafi, who has guided and supported me these past two years. I would not have been able to get to where I am without his help and leadership. In addition, I would like to thank Roland Wunderlich and Tom Wenisch for countless discussions throughout the semester. In addition, special thanks needs to be given to Vijay S. Pai from Rice University for allowing us access to the Direct Execution simulator for RSIM.

I would also like to thank my parents, Joe and Jessie Chen, for the endless love and support throughout my studies. In addition, I'd like to thank my friends from A-level with whom I have spent so much time with during my stay at CMU.

One final note is that I need to thank EGO for having extremely cheap soda very accessible from my cubicle. Definitely I would not have been able to get through those endless nights without the help of countless cans of cherry coke.

References

[1] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, Computer Sciences Technical Report, 1997.

[2] M. C. Carlisle. Olden: Parallelizing programs with dynamic data structures on distributed memory machines, June 1996.

[3] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 468–477, 1996.

[4] C. J. Hughes et al. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, February 2002.

[5] R. C. Covington et al. The Rice parallel processing testbed. In *Proceedings of the 1988 ACM Sigmetrics conference of Measurement and Modeling of Computer Systems*, 1988.

[6] R. R. Kessler et al. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium of Computer Architecture*, pages 131–139, 1989.

[7] R. Wunderlich et al. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 84–95, June 2003.

[8] S. Woo et al. The Splash-2 programs: Characterizations and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[9] W. C. Hsu et al. On the predictability of program behavior using different input data sets. In *Workshop on interaction between compilers and computer architectures, (INTERACT-6) held with HPCA-8*, February 2002.

[10] R. M. Fujimoto and W. B. Campbell. Direct execution models of processor behavior and perfor-

mance. In *Proceedings of the 1987 Winter Simulation Conference*, pages 751–758, 1987.

[11] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.

[12] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *Hawaii International Conference on System Sciences*, volume Volume 1: Architecture, pages 205–210, January 1994.

[13] P. S. Levy and S. Lemeshow. *Sampling of Populations: Methods and Applications*. John Wiley & Sons, Inc., 1999.

[14] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual, version 1.0. Technical Report 9705, Rice University, Department of Electrical and Computer Engineering, 1997.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 45–57, 2002.

[16] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.