SuiteSpecks and SuiteSpots: A Methodology for the Automatic Conversion of Benchmarking Programs into Intrinsically Checkpointed Assembly Code

Jeff Ringenberg and Trevor Mudge
The University of Michigan
Electrical Engineering and Computer Science
{jringenb, tnm}@eecs.umich.edu

ABSTRACT

This paper introduces a methodology to reduce the overall simulation time of large benchmarking suites. Previous work shows that it is possible to simulate only small sections of a benchmark's dynamic instruction stream in detail without sacrificing accuracy in simulation results with respect to overall behavior. As benchmarking suites increase in size, many such techniques still require a great deal of simulation time to complete. The methods presented in this paper build on this previous work by converting representative sections of a benchmark's execution into intrinsically checkpointed assembly (ITCY) code that can serve as a replacement for the original benchmark. In addition, a methodology is proposed that creates new benchmark binaries that no longer need input files or system calls in order to execute properly. Simulations of the new benchmarks are much faster, require less overhead, and still properly represent the original benchmark's execution profile.

Results show that benchmarks created using these techniques can be very portable and accurately predict the performance of the original benchmark. An average error rate of less than 5% is achieved when compared to the original representative sections. In addition, a speedup of approximately 60x per benchmark is achieved over a standard set of SimPoints when the new benchmarks are executed serially and 1000x when executed in parallel. This translates into a reduction in simulation time from months to minutes and greatly decreases the amount of time necessary to test a new design.

1. INTRODUCTION

The process of properly testing and rating a new microprocessor design takes a great deal of time and effort. Large datasets, multiple configurations, and highly complex simulators all contribute to an ever increasing amount of time needed to measure performance. Many techniques have begun addressing this issue by tackling a variety of the factors that contribute to this increase in simulation time.

This paper describes a technique that is targeted at reducing the overall simulation time needed to test a new design while still maintaining an acceptable level of accuracy with respect to performance estimation. The technique first analyzes a set of predetermined segments of a benchmark's dynamic instruction stream to determine the checkpointing information that is needed were each segment to execute independently. This checkpointing information is then combined with the assembly code of the original benchmark to create an entirely new program written in assembly and composed of a subset of the static instructions from the benchmark along with any necessary checkpointing and control code needed for proper execution. This code, referred to as InTrinsically Checkpointed assemblY (ITCY) code, is re-compiled and executes only the code necessary to simulate a specific range of instructions from the original benchmark's dynamic instruction stream.

The rest of the paper is organized as follows. Section 2 will discuss several previously proposed simulation time reduction techniques. Section 3 will describe the ITCY code generation technique. Section 4 will outline the experimental framework and section 5 will provide results and analysis of the technique. Section 6 will conclude the paper along with providing some thoughts on future work.

2. PREVIOUS WORK

There are many different techniques, summarized in Figure 1, that can be used to reduce the simulation time of a new design while still maintaining an acceptable level of accuracy. They can be broken down into three main categories: *benchmark suite reduction*, *statistical simulation*, and *instruction sampling*. The first category decreases simulation time by reducing the size of the input data to the benchmarks, by simulating a subset of the original benchmarks, or by doing both. Since the benchmarks subsequently do not execute as many instructions as was originally intended, the overall execution time will be faster. The second category takes each benchmark and runs a set of profiling routines on it to extract its makeup and behavior. Once this information is obtained, a new, smaller benchmark

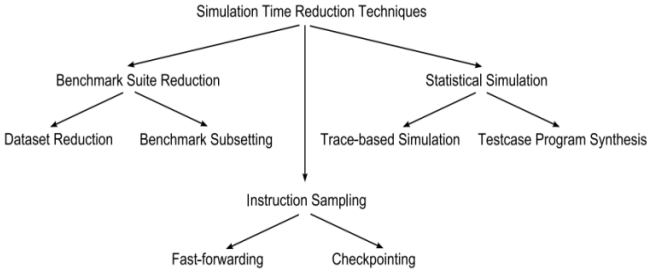

Figure 1: Summary of Simulation Time Reduction Techniques

binary or trace is created whose behavior is meant to mimic that of the original benchmark. The third category involves sampling intervals of instructions from the benchmark's dynamic instruction stream and then re-executing the intervals at a later time. The cumulative performance of these samples is intended to represent the overall performance of the benchmark if it were to be executed in its entirety.

A problem with instruction sampling, in particular, is that once the sample intervals of the original benchmark have been identified, they may start at any point in the benchmark's dynamic instruction stream. Reaching these starting points can be done in several different ways. The entire benchmark can be functionally executed up until the starting point, referred to as *fast-forwarding*, and then the detailed simulation of the sample can begin. Unfortunately, this method can take a long time to complete if the starting point occurs late in the benchmark's execution. Alternatively, the sample interval can use *checkpointing* to restore the state of the system corresponding to the start of the interval. The data used do to this, referred to as a *checkpoint*, can be used to refresh the architectural state of the system such as memory and registers, and also its microarchitectural state such as caches and branch predictors. The procedure of refreshing microarchitectural state is typically referred to as *warmup*.

2.1 Benchmark Suite Reduction

The techniques in this section reduce simulation time by altering the composition of the original benchmark suite. Even though great care goes into the creation of a benchmarking suite, there can still be a large amount of redundant behavior that is exhibited by its member programs and input datasets. The techniques in this section identify this behavior so that only those programs and input datasets that are necessary to properly represent the benchmarking suite can be simulated.

Dataset reduction reduces the amount of data that a program must process and therefore its execution will finish quicker than if the benchmark had massive amounts of data to handle. This method is used to create the MinneSPEC [11] workloads and reduces the size of the input datasets to the SPEC CPU2000 benchmarks [10]. Once the workloads are created, they can be used to quickly and repeatedly execute the benchmarks, however they are often so reduced that they no provide representative results.

In [24], several different approaches meant to reduce the number of benchmarks that need to be run from a given benchmarking suite are compared. These approaches advocate executing a reduced number of program-input pairs from the suite called a *subset* based upon the similarities between the pairs. Each approach uses a different method to choose a subset of the suite, however, the method detailed in [16] proves to be the best with respect to accuracy versus cost.

2.2 Statistical Simulation

This section discusses techniques that convert a benchmarking program into a trace or an entirely different program whose execution profile is intended to match that of the original benchmark. Two types of statistical simulation are discussed. The first, trace-based simulation, generates a trace of instructions from the initial profile which is run on a specially instrumented simulator capable of interpreting the trace. The second, testcase program synthesis, generates an actual program binary from the initial profile that can be run on a traditional execution-based simulator.

In [8], a method of trace-based simulation is proposed. First, a trace of the program's execution is obtained from real hardware. This initial trace is input into a set of three profiling tools, two of which model microarchitecture-dependent statistics (i.e. cache and branch performance) and the third which models microarchitecture-independent statistics. Once these statistics are gathered, a synthetic trace is generated. Later work done in [7] and [9] expanded the capabilities of the method proposed in [8] and reduced the average performance prediction error.

As an alternate to creating a synthetic trace, statistical simulation can be performed using a standalone program whose creation and execution profile are also based on statistical analysis. This method is referred to as *testcase program synthesis*. In [3], the HLS methodology described in [14] is modified and then used to generate a profile of the benchmark. After the profile is generated, a synthetic program, written in C, is created based on the profile. This technique unfortunately suffered from an inability to properly model branch prediction and cache performance and subsequent work done in [5] and [4] addressed these issues. A detailed analysis was done in [2] on the source of errors in testcase synthesis. It was shown that, while errors do contribute to a loss of accuracy, the effects are small with respect to the performance of the original program. However, because of these errors, testcase synthesis should not be used as a replacement for detailed, application simulation.

2.3 Instruction Sampling

The methods in this section reduce simulation time by using a technique called instruction sampling. These techniques can achieve the highest level of accuracy with respect to performance prediction. However, they oftentimes result in a longer simulation times than benchmark suite reduction and statistical simulation. The basic instruction sampling methods select intervals of instructions from the dynamic instruction stream and then re-execute those intervals at a later time. Depending on the size and location of the intervals, the reduction in simulation time can be quite dramatic.

The Sampling Microarchitecture Simulation framework (SMARTS) [23] is a simulation tool that uses statistical sampling to identify intervals of equal length within a benchmark's dynamic instruction stream. Each interval consists of an initial warmup phase, a detailed simulation phase, and a fast-forwarding phase. When the original benchmark is re-executed, it will fast-forward a preset number of instructions to the first interval and then begin the initial warmup phase. The detailed and fast-forwarding phases then

follow until the next interval is reached. Once the simulation completes, the results from the detailed phases can be used to estimate the overall performance.

Another popular instruction sampling tool is SimPoint [19]. Similar to SMARTS, it only executes a small amount of the original benchmark in detail, reserving the rest for fast-forwarding. Unlike SMARTS, however, SimPoint uses a technique to group intervals of the dynamic instruction stream into clusters based on their basic block profiles. Candidate intervals are then chosen from each cluster to be executed in detail. The overall performance of the benchmark is calculated using the results of each interval along with their frequencies in the original benchmark

2.4 Techniques for Checkpointing and Warmup

To enable a benchmark to execute an instruction interval in the middle of its dynamic instruction stream without executing all the instructions prior to the interval, checkpoint data must be used to refresh the state of the system. At the simplest level, the checkpoint must at least refresh the architectural state such as the memory and the register file. However, if the sampled interval is to be completely simulated in detail on the microarchitectural level, there must be more detailed information, referred to as warmup data, contained in the checkpoint to refresh such components as the caches and branch predictor.

One way to obtain checkpoint and warmup data is by directly executing the benchmark on real hardware. This can rapidly provide data, however, a system to directly execute the benchmark may not always be readily available. In [15] and [18], direct execution is used to obtain instructions from within an instruction interval. In [15], the Pin [12] tool produces a benchmark profile that is input into SimPoint to identify representative instruction intervals. Once these intervals are identified, they are compared against the original benchmark using Pin and a set of *PinPoints* is generated. The PinPoints can then either be used as an instruction trace of the interval or to dictate to an execution-driven simulator when it should switch between fast-forwarding and detailed execution modes. In [18], a completely new program is generated. When combined with a pre-loaded memory image, the program executes a set of instructions that represent the original interval of interest.

In [13], the Pin tool is again used to directly execute a benchmark on native hardware. However, the importance of Pin in this technique is that it is modified to capture system effects. These effects are stored in a system effect log and are later used during architecture simulations by having the simulator process the log. This type of checkpointing is different than previous methods since it removes the need for a simulator to support system calls.

Work done in [20] directly addresses checkpointing and warmup in SimPoint. It proposes two techniques, the *touched memory image* (TMI) and the *memory hierarchy state* (MHS), to refresh the system state prior to the execution of the simulation interval. The TMI creates a list of memory addresses and data values that are used to refresh the architectural state of the system and the MHS stores a cache state that is collected during a simulation of the memory hierarchy prior to the execution of the interval. Later, when the interval is simulated in detail, the microarchitectural information stored by the MHS is loaded and used to refresh the state of the cache, or any cache with a smaller size or associativity. Similar to the work done in [20], [21] and [22] propose techniques that are targeted at the rapid checkpointing and warmup of SMARTS. Again, state is only stored for the instructions that will be executed in each SMARTS interval, and unlike [20], caches are not the only microarchitectural elements that are warmed up. The branch predictor is also warmed by storing checkpoint data for a variety of configurations. The combination of the checkpoint and warmup data along with the interval is referred to as a *live-point*.

In [17], we presented an additional technique for generating simulator-derived checkpoint data called Intrinsic Checkpointing through Binary Modification (ICBM). This technique uses a method such as SimPoint to identify a set of intervals that represent the benchmark and each interval is then analyzed to create checkpoint data only for those instructions that are in the interval. Next, the checkpoint data is converted into a series of machine instructions and written directly into the benchmark's original binary. The original binary is also modified to begin its execution at the start of the checkpointing instructions and then transfer control to the interval once they have executed. The simulator ultimately exits the benchmark when the interval has completed the necessary instructions.

While the techniques presented in this paper are similar to [17] in some aspects, the methods in this paper have addressed several shortcomings that were discussed in [17]. First, ICBM code requires the simulator to count the number of instructions in the modified binary during its execution and exit when it has executed a predefined number of instructions. Second, system calls must still be supported by the simulator and cannot be removed from the modified binary. This requires that input files still be made available to the modified benchmark. Finally, ICBM code only works with single simulation intervals and cannot combine multiple intervals into a single, new benchmark.

3. INTRINSICALLY CHECKPOINTED ASSEMBLY CODE

The methodology presented in this section is the focus of this paper and consists of three distinct phases that transform a large benchmarking binary into one or more intrinsically checkpointed binaries. The first phase selects representative intervals of the original benchmark's dynamic instruction stream using a tool such as SimPoint or SMARTS. The second phase converts these intervals into InTrinsically Checkpointed assemblY (ITCY) code by running the original benchmark through a modified functional simulator. Finally, the third phase compiles the ITCY code into either a set of *SuiteSpecks*, independent code segments that can be executed in parallel, or a *SuiteSpot*, a grouping of ITCY code segments

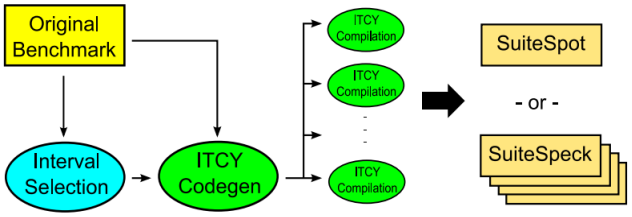

Figure 2: ITCY Code Generation Diagram

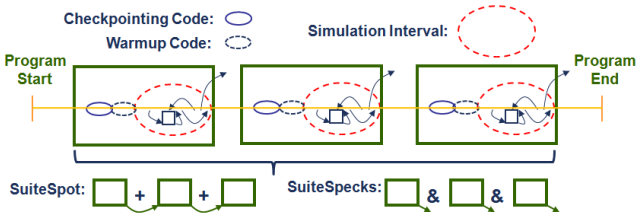

Figure 3: ITCY Interval Selection and the Creation of a SuiteSpot or SuiteSpecks

linked together by branches. Figure 2 represents these phases pictorially.

3.1 Simulation Interval Selection

During this phase, the original benchmark is analyzed to select regions of its dynamic instruction stream that can be used for detailed analysis using a tool such as SimPoint or SMARTS. As these tools have shown, only a small subset of a program needs to be simulated in detail in order to obtain a representative sample of the overall behavior of the benchmark and the remainder of the program can quickly be simulated on a functional level. However, this functional simulation can still take many hours to complete.

3.2 ITCY Code Generation

The ITCY code generation phase takes the selected dynamic instruction intervals and converts them into ITCY code using an augmented functional simulator from the SimpleScalar [1] toolset targeting the Alpha 21264 ISA [6]. The ITCY code consists of three main parts: intrinsic checkpointing (IC) code meant to recreate the environment of the original benchmark through checkpointing, the original static assembly instructions from the interval, and special control code that is needed to handle various situations discussed below.

This conversion process must address several issues in order for the new code to execute properly. First, it must ensure that the initial state of the original interval when executed in the new binary matches the state that it possessed when it began its execution in the original benchmark. Second, it must set up the new binary such that any memory accesses it contains will reference valid locations in its allocated memory space. Third, it must guarantee that the new dynamic instruction stream runs in the same order as when it was first executed. Fourth, it must recreate the static instruction footprint in such a manner that the cache access patterns of the new binary mimic those of the original binary. Finally, the new code must exit after the correct number of dynamic instructions from the original interval have occurred. In addition to the five issues mentioned above, system calls (syscalls) encountered inside each simulation interval will be emulated using code similar to that used to checkpoint the interval itself. Figure 3 gives a graphical overview of how the various fast-forwarding, warmup, and detailed simulation intervals are combined into either a SuiteSpot or a set of SuiteSpecks.

3.2.1 Initial State Intrinsic Checkpointing

In order for the new ITCY code to begin its execution properly, its initial state must be restored to that which was seen when it was first executed inside of the original benchmark. The methods used to accomplish this are based on the techniques outlined in [17] with several modifications made to the way that memory is checkpointed. The ITCY methods do not make use of multiple copies of memory, as was seen in [17], and instead mark memory usage prior to the interval's execution using flags. Any stores encountered prior to the execution of the interval mark each byte of the memory locations that they modify indicating that the locations have changed. In addition, any memory locations changed due to syscalls are similarly flagged. Then, whenever a load occurs inside the interval, the simulator checks the flags on all the bytes that will be read. If any of the bytes have been modified, a temporary IC byte store is generated for that byte using the value in the current memory. The memory location's flag is then cleared so that it will not generate any more IC stores. In addition, any stores that occur inside the interval clear their associated memory flags since they will still execute in the new binary. After the interval's analysis is complete, the simulator attempts to compact any adjacent byte stores in this list into larger multi-byte stores to compress the size, and reduce the execution time, of the final ITCY code.

In addition to the modifications of how IC store instructions are generated, the ITCY technique changes the methods that manipulate the data used for the intrinsic checkpointing of memory. In [17], memory was checkpointed using values that were stored in the .data section of the new binary. These values were stored into their respective addresses using static load/store instructions that generated their own addresses. Unfortunately, as the size of the interval grew, the number of these static instructions grew too large. ITCY code, however, converts the static instructions into a loop that iterates over all the values in the .data section. This requires the storage of the data values needed to checkpoint memory and also the addresses where the data will be stored. This requires extra space in the .data section of the new binary, but it is offset by the new .text section not requiring large numbers of static instructions.

3.2.2 Ensuring Valid Memory Accesses

In order to allow the potential combination of multiple simulation intervals into a single, new benchmark, the

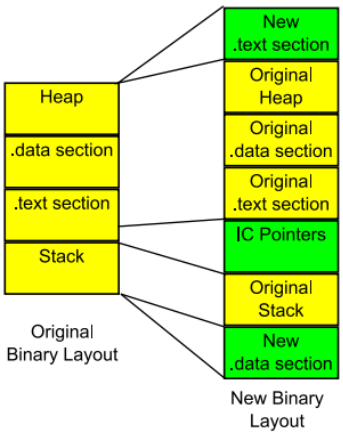

Figure 4: Original and New Binary Memory Layouts

location of the new ITCY .text section can no longer be located in the same memory space as the original benchmark. Previous intrinsic checkpointing work in [17] did not suffer from this requirement since all modifications were done to the original binary which retained its initial location in memory. However, this prevented the creation of a new benchmark that contained code from different intervals in the same benchmark or from different benchmarks altogether. To allow for this flexibility, the ITCY technique reorders the sections of the new binary such that new segments of code can easily be added without affecting the values in the original benchmark's memory space. Figure 4 depicts these memory layouts.

3.2.3 Preserving the Dynamic Instruction Stream

Since ITCY code contains the static, and not dynamic, instructions of the original interval, special care must be taken to ensure that the execution of the ITCY code follows the same execution order that was seen in the original interval's execution. This is especially the case when branches are executed within the interval's code. Since the ITCY code will no longer occupy the same PC addresses within the memory's address space, branches need to be handled by either replacing the original branch target with the PC address of the new target or by using special control code. For conditional branches, the original branch target can simply be replaced with the new target in the assembly code. Calculating these targets is done by giving every target instruction in the ITCY code a unique label that contains the original PC value as part of the label. The compiler then automatically re-targets each branch when it is run. If a branch saved its return address to a register prior to its execution, then this register must be explicitly loaded with the original return address prior to the execution of the new branch in order to maintain proper execution semantics.

Indirect branches, however, are more difficult. Since the execution of the original interval's instructions inside the ITCY code will generate the exact same register values that were seen in the original benchmark, using a register with its original value to supply an indirect branch with its target will cause the program to transfer control into the old address space. This violates the proper execution order of the new benchmark. Therefore, indirect branches must be handled using special blocks of control code.

The method for handling these problematic indirect branches first finds all the PC addresses of the new indirect branch targets using their target PC labels. These PC values are then written into locations in the original address space during the pre-interval section of the ITCY code. Each new target PC address must be written to the address that corresponds to where the old target instruction resided in the original address space. Prior to the execution of the indirect branch in the ITCY code, a special block of control code uses the value in the register that contains the original target address as a memory location to read in the new branch target address into a temporary register. This temporary register is then used by the indirect branch in the ITCY code for the new target PC value. The ITCY code generation phase guarantees that these temporary registers do not interfere with the correct execution of the program by either finding a register that will be overwritten in the future prior to being read or by setting up a block of control code to save the register to a special location in memory before it is used and then restoring it afterwards.

3.2.4 Preserving Cache Access Patterns

Not only must the ITCY code perform functionally correct, but it must also maintain a certain level of accuracy with respect to the underlying microarchitecture. Cache performance typifies this requirement since the cache performance of the ITCY code can be drastically different than the original benchmark depending on how the ITCY code is created. Since only those static instructions that are executed in the original interval are output into the ITCY code, any code that was not executed is omitted from the new binary. If this omitted code separated basic blocks that were spaced far apart in the original memory space, the exclusion of this code would cause the basic blocks to appear next to each other in the memory space of the new binary thus creating a totally new memory footprint. To help the ITCY code maintain as much representative cache behavior as possible, "pads" of no-op instructions are incorporated in between basic blocks that have had their original separating code removed.

3.2.5 Exit Handling

When the ITCY code finishes its execution of a simulation interval, it needs to be able to exit the interval code since it does not contain any code from the original binary beyond what was seen in the interval. To handle this situation, two different methods can be used. The first method logs the value of the PC that corresponds to the last instruction of the interval along with the number of times that it is executed. In addition, to handle the transfer of control to any additional intervals when creating a SuiteSpot, the starting address of the next interval is also kept. Once these values are known,

special annotative instructions are added to the IC code so the simulator can load them into its internal state for later use. During the execution of the ITCY code, every time the simulator encounters the exit PC, it increments the current execution count until it matches the initial count. At this point, if the next interval address is equal to zero, the execution terminates. If the next address is not equal to zero, then the simulator moves to the start of the next interval.

If it is not possible to add this tracking functionality to the simulator, a second method can be used whereby a termination routine is inserted into the binary itself. Since the ITCY code will be executing static instructions from the original interval, if a termination routine is to be added, it will need to insert one or more instructions into the set of static instructions at the exact location that the last instruction in the interval was executed. If this last instruction is only executed once, it can simply be replaced with an exit system call or a branch to the next interval. If, however, the static instruction that corresponds to the last dynamic instruction is executed more than once, then it must be replaced with a block of special control code that internally tracks the number of times that it is executed so that it can finish at the appropriate time. By using this second method, the ITCY code can be run directly on an unmodified simulator, or possibly on real hardware, and it will exiting properly.

3.2.6 System Call Emulation

The final step of the code generation process involves the removal of system calls from the final ITCY code. Using methods similar to intrinsic checkpointing, a syscall can be replaced with a branch to a special section of code that emulates the effects of the original system call. The actual checkpointing data that the emulation code must load into the system is determined by the system call handling routines built into SimpleScalar. These routines mimic the execution of system calls using the native system call handlers of the host machine. Once the emulation code completes, control is transferred back to the interval via an unconditional branch.

One side effect of this emulation is that the code will only recreate the effects of the syscall when it was executed during the ITCY code generation. This prevents syscalls that do not have repeatable behavior, such as gettimeofday(), from executing properly when the ITCY code is later used. However, a benchmark with these syscalls is less desirable since its results would not be predictable. Instead, by embedding this temporal information into the ITCY benchmark, this variability is removed. In addition, the emulation of system calls removes the need for input files because any file reads that occurred in a system call have been effectively checkpointed inside the new binary's system call emulation code.

3.3 ITCY Code Compilation

The final phase of converting a benchmark compiles the ITCY code into either a set of SuiteSpecks or a single SuiteSpot. As was mentioned previously, a SuiteSpot combines multiple simulation intervals into one single binary. Each interval is linked to the next interval by using branch instructions in the place of an exit syscall. The compilation of ITCY code into SuiteSpecks or a single SuiteSpot proceeds in a very straightforward fashion since the code generation phase in section 3.2 automatically generates a Makefile that can be used to compile the code. This Makefile must be created in order to tell the compiler where to place the .data and .text sections of the new ITCY code as was described in section 3.2.2.

3.4 ITCY Code Execution

After the ITCY code is compiled, it can be executed just like any other benchmark. Input datasets are no longer needed since all system calls are now emulated by the benchmark and any file input data that the original benchmark needed is checkpointed within the state of the ITCY code. One concern that needs to be addressed, however, is the handling of interval weights. Since each interval may not represent an equal amount of the original benchmark's execution profile, the performance metrics for an interval may need to be offset to reflect this discrepancy. If SuiteSpecks are used, their performance metrics simply need to be multiplied by their respective weights before they are summed together. However, if a SuiteSpot is used to represent many different intervals each with a different weight, then signal instructions must be added to the ITCY code to alert the simulator when a new interval begins so that it can handle the statistics.

3.5 Validation

To verify that the ITCY code was executing the proper instructions from the original SimPoint interval, a trace of the register file was maintained on a cycle level basis and compared to a similar trace that was created during the ITCY code generation. If the traces matched, it was assumed that the code was executing the proper instructions. Due to large overhead, this was only done for relatively small intervals. A more rapid, but less accurate, technique loaded a special exit code into the input argument register of the exit syscall just prior to its execution at the end of the benchmark. When the exit syscall was reached, this exit code was output to the screen and checked by inspection.

4. EXPERIMENTAL FRAMEWORK

The ITCY technique was tested using the Alpha [6] architecture and the configuration in Table 1 with out-of-order execution. The sim-safe functional simulator from the SimpleScalar version 3.0d [1] simulation infrastructure was modified to do the interval analysis and code generation. Other simple modifications were made to the simulators to handle the parsing of any special ITCY control instructions that were previously discussed.

Simulator Parameter	Parameter Value
Instruction Fetch Queue Size	32 instructions
Issue/Decode/Commit Width	8 instructions
Branch Predictor	Combined, 8192 entry meta-table Bimodal Part: 8192 entries 2-Level Part: 8192 l1/l2 table entries, 13-bit history
Branch Target Buffer	512 sets, 4-way associativity
L1 I-Cache	128 sets, 32-byte blocks, 2-way associativity, LRU
L1 D-Cache	128 sets, 32-byte blocks, 4-way associativity, LRU
L2 Unified Cache	4096 sets, 64-byte blocks, 4-way associativity, LRU
I-TLB	32 sets, 4096-byte blocks, 8-way associativity, LRU
D-TLB	32 sets, 4096-byte blocks, 8-way associativity, LRU

Table 1: Baseline Configuration

The compilation of the generated ITCY code was straightforward since the majority of the work was already done in the code generation phase. Each Makefile simply needed to be run to produce the new binary or binaries. The actual compilation occurred on a native Alpha compiler since no cross-compiler was easily obtainable that would output binaries for SimpleScalar. All 26 benchmarks from the SPEC CPU2000 benchmarking suite with reference inputs were used including a variety of different input data sets. This resulted in a total of 41 different benchmark/dataset pairs. SimPoint was used to provide the sample intervals and the majority of the results used thirty, 10 million instruction intervals per benchmark. These intervals were then converted into a set of SuiteSpecks for each benchmark. SuiteSpots were not used due to size constraints and also due to the fact that they are functionally equivalent to their SuiteSpeck counterparts when they are executed serially. Next, individual benchmark results were calculated as the weighted average over the SimPoint intervals. Each SuiteSpeck fast-forwarded through the intrinsic checkpointing code prior to the start of the interval.

5. RESULTS AND ANALYSIS

This section presents the results when testing the ITCY checkpointing methodology. The results can be broken down into a set of four broad categories. These categories are as follows: code overhead, performance modeling, effects on file size, and simulation speedup.

5.1 Code Overhead

The code overhead for the ITCY method can be quite large due to the nature of the technique. Since each simulation interval is removed from the original benchmark and converted into ITCY code, it will need special instructions inserted into the interval to handle indirect branching and exit handling. In addition, system call emulation code will need to be inserted. The instructions that are inserted into the new binary consist of those that occur prior to the start of the interval, referred to as *pre-interval instructions*, and those that occur within the interval, referred to as *intra-interval instructions*. All of this new code will add to both the overhead and size of the new benchmark, however, it allows for a greater deal of flexibility and portability for the new binary insomuch that it can be moved between different microarchitectural configurations of the same base architecture (i.e. the Alpha ISA).

	SpecInt Avg	SpecFP Avg	Spec2K Avg
Mem/Reg Checkpointing	574,751	4,524,063	2,019,621
Indbr/Syscall/Exit Handling	1,418	168	960
Total Pre-Interval Dynamic Insts	576,169	4,524,231	2,020,582
Total Pre-Interval Static Insts	271	459	340

Table 2: Breakdown of ITCY Pre-Interval

5.1.1 Required Pre-Interval Instructions

For the instructions that occur prior to the interval, they primarily consist of those needed to checkpoint the memory and the register file. The remainder of the pre-interval instructions set up the .data section for indirect branch and syscall handling as described in sections 3.2.3 and 3.2.5, set up the exit handling described in section 3.2.6, and handle any other miscellaneous tasks described in section 3.

Table 2 shows the dynamic instruction counts for the integer (Int), floating point (FP), and all SPEC2000 benchmarks separated out into memory/register checkpointing and the remainder of the pre-interval instructions. It also shows the total number of static instructions in the pre-interval code. From the results, it is clear that more than 99% of the pre-interval dynamic instructions are devoted to memory/register checkpointing and that those instructions come from a very small number of static instructions. This indicates that a great deal of looping is occurring inside the code as was described in section 3.2.1. The table also shows that there is nearly an order of magnitude difference in the number of dynamic memory/register checkpointing instructions needed for the floating point benchmarks since they traditionally use more memory and will require more checkpointing. Overall, this translates into an increase in the number of dynamic instructions of 40% on average for the floating point benchmarks and only 5% for the integer benchmarks.

Another interesting result observed in Table 2 is the higher number of "other" instructions that are required for the integer benchmarks compared to the floating point benchmarks. Since the memory checkpointing instructions occur in a loop, they only contribute a constant number of static instructions to the pre-interval code. However, the instructions that handle indirect branch targets and syscall emulation are entirely static and increase in amount whenever the number of indirect branches or syscalls increase. As Table 3 shows, the number of syscalls that need emulation will have little effect in this case, however, the integer benchmarks have over 4 times as many indirect branches as the floating point benchmarks.

	SpecInt Avg	SpecFP Avg	Spec2K Avg
Syscalls	1.1	10.7	4.6
Indirect Branches	124,101	29,417	89,460
Unconditional Branches	286,475	73,551	208,576

Table 3: Number of Syscalls and Indirect Branches in ITCY Interval

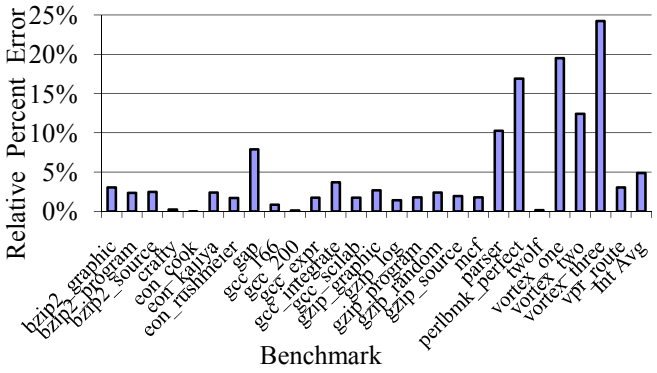

Figure 5: Relative Percent Error compared to SimPoints When Predicting CPI - Integer Benchmarks

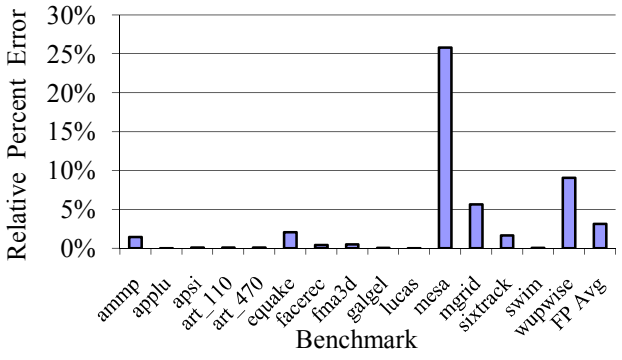

Figure 6: Relative Percent Error compared to SimPoints When Predicting CPI - Floating Point

5.1.2 Required Intra-Interval Instructions

The instructions that occur in the pre-interval code can be fast-forwarded through at the start of the simulation and ignored. However, the instructions that occur within the interval must be executed during the detailed simulation. These intra-interval instructions can negatively affect the results of the simulation if they are too numerous and must be carefully handled to minimize this risk.

Indirect and unconditional branch handling contributes the largest amount of intra-interval code overhead since these branches occur quite frequently in the interval as was seen in Table 3. However, the table only gave information about the number of branches that were in the interval and not the effects of the code that handles their proper execution. On average, indirect branch handling increases the length of an integer benchmark interval by 3% and a floating point benchmark interval by 0.6%. Unconditional branch handling increases the length of an integer benchmark interval by 2.8% and a floating point benchmark interval by 0.7%. The smaller increase for floating point benchmarks is likely due to the fact that they traditionally spend a great deal of time inside of looping code where conditional branches, not indirect branches, control the flow of the program.

System call handling, unlike indirect branch handling, contributes very little to the intra-interval code overhead. Since Table 3 showed an average of only 4.6 syscalls per benchmark interval, it is expected that there will be little handling code needed. This is indeed the case. However, several benchmarks do exhibit a great deal more syscall handling code than the average. This is likely due to an interval containing a large amount of file input which would result in a substantial amount of checkpointing code being introduced to emulate the input. This is particularly the case for mcf, twolf, art_110, and fma3d. On average, syscall emulation only increases the length of an integer benchmark interval by 0.001% and a floating point benchmark interval by 0.03% due to the very small number of syscalls seen in each interval.

5.2 Performance Modeling

Performance modeling using ITCY code can suffer from the effects of the intra-interval code insertion. In addition, the relocation of the original interval into a new location in memory and the use of only those static instructions that are executed from the original benchmark can also have an effect as was discussed in sections 3.2.2 and 3.2.4. The effects of ITCY code on I-cache performance are expected to be the most severe due to the usage of only those static instructions seen within the interval for the new ITCY binary. Fortunately, the previously discussed technique of inserting instruction pads into the interval to simulate the spatial separation of basic blocks addresses this problem to a certain extent. However, since the number of misses with respect to the overall number of I-cache accesses can be very small in some cases, extra misses can greatly affect the relative error rate of the overall I-cache miss rate.

D-cache effects are expected to be much smaller than those seen with the I-cache. Since the same memory locations will be accessed at roughly the same points in time as the original interval, the D-cache performance will only be affected by those intra-interval instructions that access memory. The majority of these instructions do not do so except for the instructions that are inserted to handle indirect branch targets. Since these accesses are to locations in memory that were never used in the initial binary except to store the program code, they will affect the D-cache performance.

Unlike the effects on cache performance, branch prediction performance for ITCY benchmarks should remain nearly unchanged. This is due to the fact that all branches are guaranteed to follow their prescribed paths from the original interval since the register values used for conditional branching are the same. In addition, the proper performance of various branch prediction structures such as the Return Address Stack is ensured by outputting all the original branch assembly opcodes and not replacing them with different opcodes. Overall, there was a change of less than a 1% for all benchmarks and 0.2% on average with respect to branch predictor performance.

The main test of the ITCY method's ability to model the original benchmark's performance is done by quantifying its

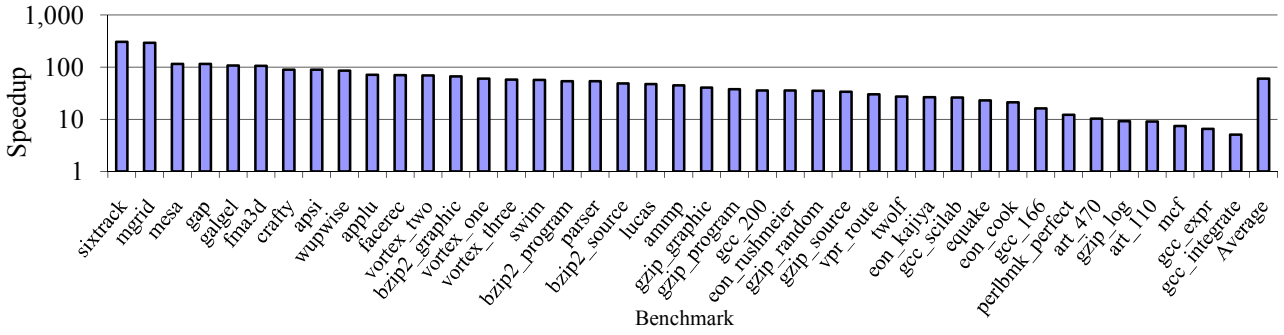

Figure 7: Speedup of ITCY Code Executed Serially

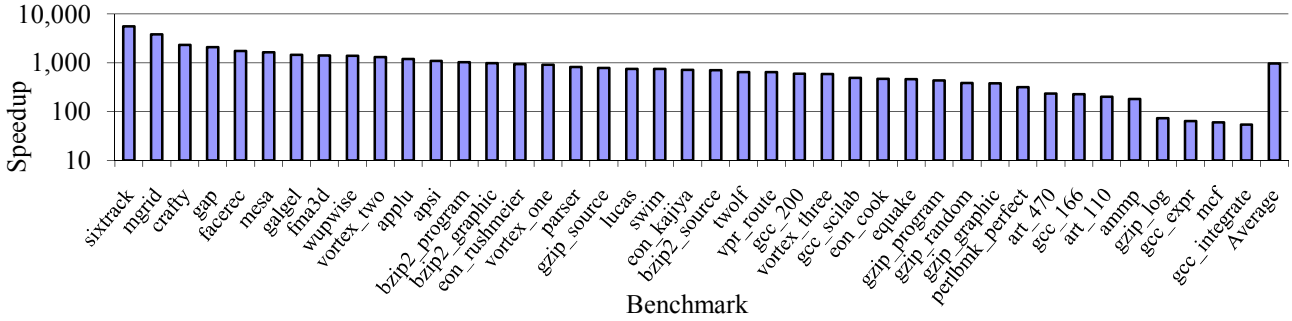

Figure 8: Speedup of ITCY Code Executed in Parallel

effects on CPI when compared to using SimPoints alone. Since CPI incorporates many different microarchitectural elements, its prediction accuracy serves as a good indicator of whether ITCY is a useful tool for predicting the performance of a system overall. Figures 5 and 6 show that the average CPI for all benchmarks is within 5% relative error compared to only using SimPoints. Several benchmarks (e.g. parser, perlbmk, vortex, and mesa) are more sensitive than others to fluctuations in performance due primarily to the effects of intra-interval branch handling, therefore, their performance predictions are less accurate. The integer benchmarks also show a slightly increased average relative error and this is likely attributable to their increased number of indirect branches. A final comparison between ITCY code and a full detailed simulation showed that the integer benchmarks were within 7.5% average relative error and the floating point benchmarks within 6%. Overall, based on the ITCY technique's ability to accurately predict performance, it is quite useful for predicting the performance of a design.

5.3 Effects on File Size

The increase in the file size when a benchmark interval has been converted into ITCY code can vary greatly based upon the size of the interval chosen, the amount of intrinsic checkpointing code needed, and the amount of code that must be inserted into the interval. Individual SuiteSpecks have an average file size for the integer and floating point benchmarks of 1.48 MB and 10.04 MB, respectively. When SuiteSpecks are summed together for each benchmark to simulate the size of a SuiteSpot, the average file size for the integer and floating point benchmarks is 44.3 MB and 279 MB, respectively. Combining all the benchmarks resulted in a total of 1.15 GB for the integer benchmarks and 4.18GB for the floating point benchmarks.

5.4 Simulation Speedup

To measure the speedup of the ITCY code compared to the original benchmark, both the original benchmark and the ITCY binary were fast-forwarded to the start of the simulation interval and then allowed to finish the detailed execution of the interval. The ITCY binary's runtime was then compared to the original benchmark's runtime. Since each benchmark of the ITCY code is broken into 30 SuiteSpecks, the results are presented not only for the situation when the intervals would be executed serially in Figure 7, but also if they were executed in parallel in Figure 8. The serial case provides a rough estimate for how long a SuiteSpot would take to complete since it would essentially be executing the same amount of code as the 30 SuiteSpecks. From the results, it can be seen that there is an incredible amount of speedup achievable when using ITCY code. With a serial speedup of 60x, this translates from hours to minutes. The greatest speedup, however, can be seen in the parallel speedup with an average of nearly 1000x speedup and a maximum speedup of over 5500x for sixtrack. It should be noted that the parallel speedup is not simply 30 times faster than the serial speedup since some intervals take much longer to run than others due to their late locations in the dynamic instruction stream. However, an average parallel speedup of 1000x clearly shows the potential of using ITCY code to rapidly and efficiently

Technique	ExecutionTime per Benchmark	CPI Prediction Accuracy	Representativeness	Microarchitecture Dependent	Storage Requirements	Flexibility
B-mark Suite Reduction [11]	Variable	Variable	Low	No	N / A	High
Statistical (Trace) [9]	~ 1000x speedup	2.30%	Low	Yes	Negligible	Low
Statistical (Testcase) [5]	~ 1000x speedup	2.40%	Low	Yes	Negligible	High
SMARTS [23]	5 hours	0.64%	High	No	N / A	Medium
SimPoint [19]	2.8 hours	3.70%	High	No	N / A	Medium
SimPoint Startup [20]	14 mins (serial) 1 min (parallel)	1.20%	High	Yes	4 GB for 20 SPEC2K b-marks	Low
LivePoints [22]	91 secs	1.60%	High	Yes	12 GB compressed for all SPEC2K b-marks	Low
ICBM [17]	3 mins	2.12%	High	No	365 MB for 19 SPEC2K b-marks	Medium
ITCY	16 mins (serial) 32 secs (parallel)	7%	High	No	5.3 GB for all SPEC2K b-marks	High

Table 4: Comparing ITCY to Several Popular Simulation Time Reduction Techniques

simulate the entire SPEC2000 benchmarking suite. One final important note to remember is that the speedup numbers presented here are with respect to fast-forwarding to the start of the simulation interval. If the execution times of the ITCY code were compared to a full detailed simulation of each benchmark, the speedup would be orders of magnitude greater.

5.5 Comparison with Other Techniques

The results presented in this section have shown the great potential of the ITCY technique for dramatically reducing the simulation time of future designs while still maintaining a high degree of accuracy. However, there are several trade-offs and differences between the techniques and other simulation time reduction methods. Table 4 compares ITCY to several of the more popular techniques by contrasting their reported speedup, accuracy with respect to full detailed simulation, representativeness, microarchitecture dependence, storage space requirements, and flexibility with respect to the subsequent simulation environment.. It should be noted that the CPI prediction accuracy is with respect to full detailed simulation and not the SimPoint intervals used to generate the ITCY binaries. Since these binaries use a simulation interval selection technique that produces its own share of relative error, their performance prediction accuracy is partly due to the selection technique used.

6. CONCLUSION AND FUTURE WORK

This paper presents a technique that dramatically reduces the simulation time of a benchmarking program by allowing the rapid execution of only representative portions of code and creates highly portable benchmarks that can be easily moved between many different simulation environments. This technique creates an entirely new assembly program comprised of the static instructions from one, or many, locations from within the original benchmark. System calls are also removed from the original program and their effects are converted into emulation code that is inserted into the assembly of the new benchmark. A method is also proposed that allows for the combining of multiple ITCY code segments into a single benchmark. The end result of the technique is the creation of a set of programs that facilitate the fast, efficient, and representative benchmarking of future designs without the need for a complex simulation environment.

There are several different directions that the techniques in this paper can explore. For example, embedded applications often suffer from a limited amount of memory and running useful benchmarks can often be a problem. The techniques in this paper could be applied to large benchmarking workloads to create applications that would be easier to use with an embedded device and would allow the rapid simulation of complex benchmarks in a fraction of the time. The lack of a need for system call handling could also be used to an advantage if the embedded device did not have such functionality. In addition, ITCY code has the ability to combine multiple intervals of code from different benchmarking applications into a new, single benchmark. The basic ITCY technique proposed using intervals from within the same benchmark to create SuiteSpots, however, it is entirely possible to use intervals from different benchmarks altogether. This method, if used properly, could potentially create individual benchmarks meant to represent the execution of many different applications at once. Finally, the ITCY technique could be leveraged to pull out specific pieces of assembly code from a given benchmark and create new benchmarks that are meant to stress particular parts of an underlying architecture. For example, intervals of code that stress a certain part of the pipeline could be isolated and re-compiled into a benchmark that would focus only on that part of the microarchitecture. Alternatively, benchmarks could be created that execute a specific region of code that uses a greater than normal amount of power and then the design could focus on reducing the power usage of the processor with the benchmark representing a worst case scenario.

REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, February 2002.

[2] R.H. Bell, Jr. *Automatic workload synthesis for early design studies and performance model validation*. PhD thesis, The University of Texas at Austin, December 2005.

[3] R.H. Bell, Jr. and L.K. John. Basic block simulation granularity, basic block maps, and benchmark synthesis using statistical simulation. Technical Report TR-031119-01, The University of Texas at Austin, November 2005.

[4] R.H. Bell, Jr. and L.K. John. The case for automatic synthesis of miniature benchmarks. In *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation*, pages 88–97, Madison, WI, June 2005.

[5] R.H. Bell, Jr. and L.K. John. Improved automatic testcase synthesis for performance model validation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 111–120, 2005.

[6] Compaq Computer Corporation. *Alpha 21264 microprocessor hardware reference manual*, July 1999.

[7] L. Eeckhout, R.H. Bell, Jr., B. Stougie, K. De Bosschere, and L.K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the International Symposium on Computer Architecture*, pages 350–361, Munich, Germany, June 2004.

[8] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2000.

[9] D. Genbrugge, L. Eeckhout, and K. De Bosschere. Accurate memory data flow modeling in statistical simulation. In *Proceedings of the 20th Annual International Conference on Supercomputing*, Cairns, Queensland, June 2006.

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

[11] A.J. KleinOsowski and D.J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, pages 10-13, June 2002.

[12] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[13] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 216–227, Saint Malo, France, June 2006.

[14] M. Oskin, F.T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the International Symposium on Computer Architecture*, pages 71–82, Vancouver, British Columbia, June 2000.

[15] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the International Symposium on Microarchitecture*, pages 81–92, Portland, OR, December 2004.

[16] A. Phansalkar, A. Joshi, L. Eeckhout, and L.K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 10–20, Austin, TX, March 2005.

[17] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic Checkpointing: A methodology for decreasing simulation time through binary modification. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 78–88, Austin, TX,March 2005.

[18] M. Sakamoto, L. Brisson, A. Katsuno, A. Inoue, and Y. Kimura. Reverse Tracer: A software tool for generating realistic performance test programs. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 81–91, Cambridge, MA, February 2002.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, San Jose, CA, October 2002.

[20] M. Van Biesbrouck, B. Calder, and L. Eeckhout. Efficient sampling startup for SimPoint. *IEEE Micro*, 26(4):32–42, 2006.

[21] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, and J.C Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report CALCM 2004-3, Carnegie Mellon University, November 2004.

[22] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, and J.C. Hoe. Simulation sampling with live-points. In Proceedings *of the International Symposium on Performance Analysis of Systems and Software*, pages 2–12, Austin, TX, 2006.

[23] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, pages 84–95, San Diego, CA, June 2003.

[24] J.J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D.J. Lilja, and L.K. John. Evaluating benchmark subsetting approaches. In *Proceedings of the International Symposium on Workload Characterization*, pages 93–104, San Jose, CA, October 2006.