THE FAST, EFFICIENT, AND REPRESENTATIVE BENCHMARKING OF FUTURE MICROARCHITECTURES

by

Jeffrey Stuart Ringenberg

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Professor Trevor N. Mudge, Chair
Associate Professor Steven K. Reinhardt
Associate Professor Dennis M. Sylvester
Assistant Professor Scott Mahlke

To Bridgette
For understanding.

ACKNOWLEDGEMENTS

My path to graduation took many different routes, some longer than others, and I owe much of my success to several, very important people.

My advisor, Trevor Mudge, started my journey in 1995 by granting me a summer undergraduate research position working with one of his graduate students, Nomik Eden. From our first Friday meetings at Ashley's to future lunches at Casey's, I enjoyed the freedom, flexibility, and guidance that I was given to pursue both my research and my teaching interests. Although I may have stayed around longer than usual, I would never have learned or discovered all that I have about my career, and myself, if not for the time that I was given. I will forever be indebted to them both for getting me started and to Trevor, especially, for all that he has done in keeping me on the path.

Another person of equal importance, David Oehmke, provided me with an incredible amount of assistance and guidance. How he managed to graduate so much earlier than I, I cannot fully comprehend. This may be due, quite possibly, to his wonderful wife, Cathy, and a desire to return to the real world. In the years that we worked together, he provided me with an incredible amount of insight and, more importantly, friendship. I, too, will always remember our culinary tour of San Diego. He is responsible for the original idea behind this dissertation and if it were not for this, you would soon be reading a completely different dissertation.

I would also like to thank my committee: Scott Mahlke, Steve Reinhardt, and

Dennis Sylvester. Their guidance following my thesis proposal helped me to focus my research and their comments refined my work.

Quite a few graduate students were pivotal in the successful completion of my work. Thanks to Ali Saidi, Geoff Blake, and Ron Dreslinski for providing me with several "eleventh hour" insights and suggestions that greatly impacted the usefulness of my research. In addition, thanks to Chris Pelosi for helping me create some of the tools that were used early in my work.

Several other people in the CSE department are deserving of my thanks. Denise DuPrie, Dawn Freysinger, Karen Liska, Stephen Reger, and Bert Wachsman all provided me with the inside information necessary to navigate the CSE, EECS, and Rackham buildings and bureaucracy. Were it not for them, I would still be figuring where to apply for positions, how to fill out the right information, and who I can talk to in order to stay employed and/or enrolled.

In addition to those people who helped with my research, there are many people I would like to thank in conjunction with my teaching at the University of Michigan. The first is my original inspiration, Elliot Soloway. His EECS Senior Design Projects course was a joy to behold and I was honored to be his Graduate Student Instructor for two years. I also whole-heartedly thank Dave Chesney for his guidance in teaching me how to handle a more traditional course and for introducing me to pedagogical research. I learned a great deal as his GSI in his software engineering course and I will always remember our dinner and conversation in Hawaii. Thanks also to Susan Montgomery for teaching her course on engineering education and for giving me a formalized introduction to the area. She was an excellent instructor and helped me to crystallize my decision to continue in teaching. Finally, I owe a great deal of gratitude to James Holloway and Toby Teory for continuing to give me the opportunity to teach

at the University. They have allowed me to pursue my true passion and this is a gift that I will forever cherish.

I also must thank my family and friends for always trusting that I made the right decision and never asking "So when are you going to graduate?" more than was absolutely necessary. To my parents, Joe and Diane Ringenberg, thank you for giving me the opportunity to start this whole journey back in 1996 and for always being interested and supportive of everything that I have done. Even though you never really understood what I was talking about, at least you listened. To my sister, Marta Ringenberg, thanks for the great conversations and for being who you are. Also, thanks for going into teaching. Maybe some day I will get one of your students in my class. To my friends, Pete Driver and Chris Rowland, thanks for the great lunches at Qdoba and for keeping the laughing to a minimum when I was still able to legally order the student drink. While I enjoyed sitting idly by, seeing Chris make the big bucks and Pete fly the friendly skies, I finally realized that I needed to join the World of Work too.

Finally, thank you to my partner, wife, and altogether better half, Bridgette Carr. She has supported me in more ways than I can describe and I will be forever indebted to her. From our first encounter on the sand courts of Slauson Middle School, her understanding nature, spirited conversation, and constant "encouragement" has provided me with a new perspective on life that reaches far outside the lab. We have literally traveled the world together and I don't ever plan to stop. I only hope that she is ready to continue that ride with me. Something tells me that she is...

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER I

Introduction

1.1 Motivation

Methods for properly simulating and testing new microarchitectural concepts have been proposed ever since computers were first designed. Most techniques require a large number of benchmark programs be run with many different input datasets for an equally large number of sample configurations of the design. Unfortunately, these requirements create a burdensome number of simulations that can significantly increase the design time of a product. For example, if 30 benchmarks were used with an average of five datasets per benchmark and there were 20 different configurations to be tested, this would require a total of 3,000 individual simulation runs. With each simulation potentially taking days or even weeks to complete, this amount of simulation quickly becomes a serious bottleneck. Compounding this problem is the fact that the most important and representative phases of a benchmark's code usually execute in the middle of the simulation and there is no easy way to execute only this portion without additional support.

Many different methods have been proposed to help alleviate these burdens; however, these methods often lead to decreased accuracy in the simulation or require that the simulator have additional functionality. For instance, if benchmarks or their re-

spective datasets are shortened or removed, the new set of benchmark and dataset combinations might no longer represent a proper workload profile for the design. In addition, methods such as checkpointing and fast forwarding meant to enable the simulation of only certain essential parts of the code require that the simulator have these abilities built in. Checkpointing requires the ability to capture the state of the system at some point in the simulation and later simulations then need the ability to load in the state and start again at that point. Fast forwarding requires a simulator to have a separate, faster mode of simulation that ignores many of the details of the design being simulated while at the same time guaranteeing its functional correctness. These features are not always available or may not be economical to implement if the simulator will have limited distribution or is meant for a small amount of use. Therefore, there is a strong need for new techniques that allow for the fast, efficient, and representative benchmarking of future designs.

1.2 Thesis Overview

This dissertation describes two different techniques that are 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 first technique, Intrinsic Checkpointing through Binary Modification (ICBM), augments a benchmarking program by directly inserting checkpointing code into the binary. When the augmented benchmark later executes, it begins with the checkpointing block of code with refreshes the system to a predetermined state, and then begins the execution of the original binary's code at a specific location. The second technique, InTrinsically Checkpointed assemblY (ITCY), also inserts checkpointing code into a benchmark. However, instead of modifying the original binary, the ITCY

method writes an entirely new program in assembly code composed of a subset of the static instructions from the benchmark and any necessary checkpointing and control code needed for proper execution. When the ITCY code is re-compiled, it will then only execute the code necessary to simulate a specific range of instructions from the original benchmark's dynamic instruction stream.

This dissertation makes the following contributions:

- The Intrinsic Checkpointing through Binary Modification (ICBM) technique is presented as a methodology that dramatically decreases the amount of time needed for simulation by analyzing and augmenting benchmark binaries to contain intrinsic checkpointing data. The newly modified binaries do not require re-compilation and allow for the rapid execution of only important portions of code thereby removing the need for fast-forwarding or explicit checkpointing support. In addition, the binaries have increased portability across multiple simulation environments and the ability to easily simulate only important parts of code in a highly parallel fashion.

- A technique for emulating the effects of system calls that are seen within an interval of execution is presented. This method inserts system call emulation code into a new benchmark eliminating the need of system call support for a simulator that runs the new program. As a result, the input dataset is effectively embedded in the new benchmark removing the need for file I/O and increasing its portability across multiple simulation environments.

- A method for combining multiple program fragments into one benchmark is presented. This technique can combine pieces of code from one benchmark, or from a variety of different benchmarks, into a single program. This new

program, if properly instrumented, can easily be moved between a variety of simulators and it can quickly simulate a large amount of representative code in a very short period of time.

- The InTrinsically Checkpointed assemblY (ITCY) technique is introduced as an additional method for creating fast and portable benchmarks. It extends on the work done with ICBM by extracting a representative portion, or portions, of a benchmarking program's assembly code and creating a new program. It incorporates the system call emulation and interval combining methods to create a new program that, when re-compiled, contains only a fragment of the original binary. When executed, the code fragments within the program run as if they were run from within their original benchmark in a fraction of the time and can serve as a replacement for the original benchmark. Once an ITCY benchmark is created, it does not need to be recreated if there is a change in the underlying microarchitecture since it is based purely on the original assembly code. In addition, the ITCY technique provides the framework for third parties to create microbenchmarks from their own internal benchmark sources. These microbenchmarks can then be released to the public without the concern of releasing any proprietary information since this sensitive information is effectively hidden inside the ITCY code.

1.3 Thesis Organization

This dissertation is organized as follows:

Chapter II provides background information on many different simulation techniques that aim to reduce the overall simulation time needed to test a new design.

Chapter III describes the Intrinsic Checkpointing through Binary Modification

technique along with providing an example of its use on the Alpha [13] architecture.

Chapter IV describes the InTrinsically Checkpointed assemblY technique and describes the method of system call emulation. It also presents the method for combining multiple intervals of instructions into a single benchmark.

Chapter V explains the experimental framework used to test the ICBM and ITCY techniques.

Chapter VI investigates the ICBM and ITCY techniques with respect to code overhead, performance modeling, effects on file size, and simulation speedup.

Chapter VII concludes the dissertation with a brief overview and a discussion of future directions.

CHAPTER II

Background

2.1 Overview

There are many different techniques 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 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 inter-

vals of the original benchmark have been identified, they may occur 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 executed up until the starting point, referred to as fast-forwarding, and then the 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 not only to refresh the architectural state of the system such as its memory and registers, but also its microarchitectural state such as its caches and branch predictors. The procedure of refreshing the microarchitectural state is referred to as *warmup*. Figure 2.1 summarizes these different techniques.

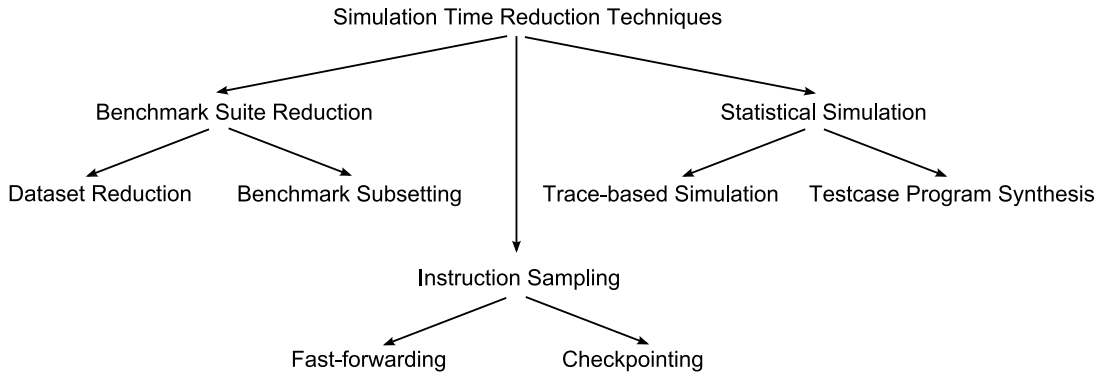

Figure 2.1: Summary of Simulation Time Reduction Techniques

This chapter is organized as follows. Section 2.2 discusses different approaches to benchmark suite reduction. Section 2.3 presents two main types of statistical simulation. Section 2.4 presents several different methods for instruction sampling and section 2.5 discusses a variety of different techniques for adding checkpoint and warmup information to the sample intervals. Finally, section 2.6 concludes with a summary.

2.2 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.

2.2.1 Dataset Reduction

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, when used to create the MinneSPEC [24] workloads, reduces the size of the input datasets to the SPEC CPU2000 benchmarks [23]. It has the advantage that, once the reduced workloads have been characterized, no analysis needs to be done on the program prior to simulation. The workload selection chooses smaller datasets through a process that systematically reduces larger workloads while still maintaining their simulation characteristics, however, this process requires many steps of manual tuning to avoid removing the representative nature of the datasets. In addition, the reduced workloads, if sufficiently small, can fit entirely within a typical cache and no longer test the functionality of the memory subsystem.

2.2.2 Benchmark Subsetting

In [48], 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 [36] proves to be the best with respect to accuracy versus cost.

The method in [36] makes use of a multi-step process that first chooses a group of microarchitectural-independent metrics with which to measure the performance of the programs within the benchmarking suite. The programs are run and their results for the chosen metrics are logged. Since this step in the process produces a large number of data points, the next step applies a technique called *Principal Component Analysis* (PCA) to the data points to reduce the dimensionality of the data. After the PCA step is run, a final step, *cluster analysis*, is run on the data to group program-input pairs into clusters that have similar behavior. One benchmark is then selected from each cluster and used to represent all the benchmarks in the cluster for subsequent runs of the suite. Their results show that from an initial set of 26 SPEC CPU2000 benchmarks, they only need to simulate 8 benchmarks to achieve a measure for Cycles Per Instruction (CPI) that is within a 5% error of the actual measurement for the entire suite.

In [37] and [38], the work done in [36] is extended to other benchmarking suites including SPEC CPU2006 [29], MiBench [19], and MediaBench [26]. In [37], performance counters are used to measure a set of metrics for the SPEC CPU2006 benchmarks. The standard PCA and cluster analysis in [36] is then carried out to select a subset of the benchmarks. The analysis identifies a subset of only 6 integer and 8 floating point benchmarks that are needed to represent most of the information found in the 12 integer and 17 floating point benchmarks that comprise the entire suite. In [38], an analysis similar to that proposed in [36] is done and it shows that only 5 benchmarks from the combined set of both MiBench and MediaBench are needed to achieve a measure for Instructions Per Cycle that is within a 5% error of

the actual measurement when both suites are combined. Table 2.1 lists the various subsets of benchmarks that were chosen where each is assumed to use the reference or large datasets.

SPEC CPU2006 Benchmarks			
Integer		**Floating Point**	
perlbench	mcf	cactusADM	leslie3d
libquantum	omnetpp	dealII	soplex
astar	xalancbmk	povray	calculix
		GemsFDTD	lbm
SPEC CPU2000 Benchmarks			
Integer		**Floating Point**	
gzip	gcc	applu	equake
mcf	twolf	fma3d	mesa
MiBench/MediaBench Benchmarks			
MiBench: susan1		MiBench: susan3	
MediaBench: djpeg		MiBench: adpcm decode	
MiBench: basicmath		MiBench: qsort	
MediaBench: ghostscript		MiBench: sha	

Table 2.1: List of Benchmark Subsets for SPEC2006, SPEC2000, and MiBench/MediaBench

2.3 Statistical Simulation

This section discusses techniques, referred to as statistical simulation, 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. This is done first by profiling the original benchmark to ascertain its execution characteristics. These characteristics are then input into algorithms that generate a statistically similar trace or program that will serve as a replacement for the benchmark with an execution time that is far less than the original.

Two types of statistical simulation are discussed below. 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, test-case program synthesis, generates an actual program binary from the initial profile that can be run on a traditional execution-based simulator.

2.3.1 Trace-Based Simulation

In [32], a basic method of trace-based simulation called HLS is proposed. It first executes the benchmark on a pair of simulators that measure several of the program's execution statistics. Statistics such as basic block size and distribution, dynamic instruction distance, instruction mix, cache performance, and branch predictor performance are logged and used to generate a profile of the benchmark. This profile is then input into a symbolic code generator that produces a sequence, or trace, of "instructions". Unlike regular instructions, these symbolic instructions contain special information that dictate their functional unit requirements, expected cache performance, and dynamic instruction distances. The instructions in the trace are then grouped into basic blocks which are linked together into a program flow-control graph. The basic blocks themselves are instrumented such that their branching behavior will produce branch prediction performance similar to that of the original benchmark. The execution of the trace is carried out by a special statistical simulator that is similar to a traditional simulator except that it interprets the statistical information contained inside the trace's symbolic instructions. The final simulation of the trace produced results that were within 5-7% for the SPECint95 benchmarks.

A technique for statistical simulation that is similar to HLS, is proposed in [31]. Unlike HLS, the technique in [31] moves the generation of the instruction trace into the statistical simulator. Therefore, the simulation takes as inputs an instruction profile and the set of cache and branch prediction models. This gives the simulator the flexibility to test different microarchitectural features without having to generate

a new trace of instructions beforehand. Experiments using this technique were carried out with varying levels of complexity with respect to the composition of the basic blocks. The simplest basic block composition that used a global mix of instructions per block and a limited amount of performance information, produced the fastest simulation and results that were within 8% for the SPECint95 benchmark baseline. The more complex basic block composition that incorporated variable basic block sizes, cache and brach prediction miss rates, and dependencies took three times as long to simulate as the simpler method, but produced results that were within 5% of the baseline.

A final technique for synthetic trace generation was first proposed in [16]. Instead of running the benchmark on a simulator to generate the program's profile, a trace of its execution is obtained from real hardware. This initial trace is then 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. The synthetic trace differs from the other methods discussed above with respect to how it represents instruction dependencies and allocates registers within the trace. Subsequent work done in [15] adds the ability to measure the expected power usage of a design by incorporating a power estimator into the statistical simulator. Results show that this model is capable of identifying a set of energy-efficient architectures that warrant further study. Later work done in [14] and [18] has expanded the capabilities of the method proposed in [16]. Control flow modeling is used in [14] to generate a control flow graph during the microarchitecture-independent profiling stage and improves the accuracy of the overall statistical simulation. Memory data flow modeling is incorporated in [18] for tracking statistics such as cache miss

correlations, load forwarding, and delayed hits. This method reduces the average performance prediction error from 10.7% down to 2.3%.

2.3.2 Testcase Program Synthesis

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 [7], the HLS methodology described in section 2.3.1 is modified slightly to include the concept of issue width and to more closely match the execution engine of SimpleScalar [1]. Since the SimpleScalar simulator will ultimately be executing the synthetic testcase, this modification is made to allow for more accurate modeling of program performance. The HLS framework, now referred to as S-HLS, is used to generate a profile of the benchmark. After the profile is generated, it is used to synthesize a program in C-code that is composed primarily of a subset of assembly language calls that map directly to the instructions in the basic blocks of the profile. The technique unfortunately suffers from an inability to properly model branch prediction and cache performance. This problem is reserved for future work and the technique is tested using technical loops as a proof of concept due to their high cache hit rates and high branch prediction rates.

Subsequent work done in [10] and [8] addresses the branch prediction and cache modeling issues in [7]. The difficulty of branch prediction modeling is alleviated by manipulating the branches that exit the basic blocks in the profile. By creating code that dictates whether or not a branch is taken or not taken, the branch prediction rate is able to be modeled effectively. To properly model D-cache behavior, a more detailed memory access model is developed. In addition, I-cache performance is more effectively predicted using a carefully tuned number of synthetic basic

blocks. After these problems were addressed, synthetic testcases for the SPEC2000 benchmarks were successfully generated and measurements for Instructions per Cycle (IPC) within 2.4% of the original benchmarks with similar workload characteristics were obtained. This work was extended further in [9] to do efficient power analysis and in [6] to do performance model validation of a PowerPC processor.

A detailed analysis was done in [5] on the source of the errors in testcase synthesis. Errors in the workload characterization phase were caused by the need to reduce the number of representative basic blocks due to size constraints and also the loss of information by only using a subset of assembly instructions to represent all the original instructions. In addition, small errors that still existed in the cache and branch prediction models were discussed. Other sources of error included problems with instruction dependencies, register assignment, and scaffolding code in the synthetic testcase. A final analysis of the errors showed that, while they do contribute to a loss of accuracy, the effects are small with respect to the performance of the original program with similar workload characteristics. However, it is noted that because of these errors, particularly those regarding cache and branch prediction performance, testcase synthesis should not be used as a replacement to detailed, application simulation. Rather, it should be used early in the design cycle to quickly evaluate the many different design choices that are available and provide insight into what configuration should be looked at more closely with detailed simulation.

2.4 Instruction Sampling

The methods in this section reduce simulation by using a technique called instruction sampling. These technique can achieve the highest level of accuracy with respect to performance prediction. However, they oftentimes result in a longer simulation

time 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.

2.4.1 SMARTS

The Sampling Microarchitecture Simulation framework (SMARTS) [47] is a simulation tool that provides quick and accurate results while only having to simulate in detail a small portion of the benchmarking program. It relies on strict statistical sampling and selects a subset of the program for detailed simulation whose results will adhere to an expected confidence interval. The program is divided into a series of equal length intervals each of which is further broken into a fast-forward component that stresses speed over accuracy, a warmup component that refreshes certain microarchitectural components such as caches and branch predictor structures that were neglected in prior fast-forwarding segments, and finally a detailed simulation component that runs with the most accuracy. After the user calculates, based on methodologies explained in [47], initial input values for warmup, detailed simulation, and overall interval lengths, SMARTS undergoes a series of tuning steps that refine the length of the fast-forward component to a point that will result in an appropriate amount of confidence for the benchmark being used.

After deriving all the necessary interval values, SMARTS starts the original benchmark binary in fast-forwarding, or functional simulation, mode and runs for an initial number of instructions provided by the user. At this point, the first interval begins the warmup period that updates the state of necessary components mentioned above. After the warmup period finishes, the detailed simulation segment begins and is followed by the fast-forwarding component that finishes the first interval. This system-

atic sampling of warmup, detailed simulation, and then fast-forwarding repeats itself until the benchmark completes. Figure 2.2 depicts this process graphically.

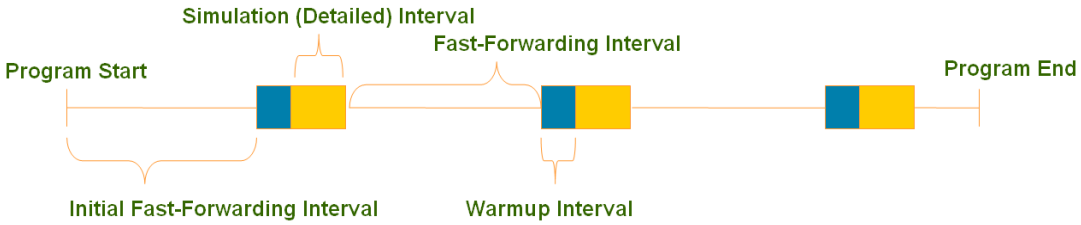

Figure 2.2: Sampleing Intervals for SMARTS

Simulations utilizing SMARTS were run on 8-way and 16-way superscalar processor models meant to represent current and future microarchitectures, respectively, using the SPEC CPU2000 benchmarks. CPI and Energy per Instruction (EPI) can be estimated within 3% with 99.7% confidence while measuring less than 50 million instructions for each benchmark. An additional amount of error, empirically set to roughly 2%, needs to be included due to unknown accuracy in the warmup routines. For the actual simulation runs themselves, SMARTS was able to reach an average error of only 0.64% for CPI and 0.59% for EPI with average speedups over detailed simulation of 35 for the 8-way model and 60 for the 16-way model.

2.4.2 SimPoint

Another tool that is useful for fast and accurate instruction sampling is SimPoint [41]. Like SMARTS, SimPoint only simulates in detail small portions of the benchmark leaving the rest for fast-forwarding. However, the methods it uses to decide which portions of the program will be used for the detailed simulation significantly differ from SMARTS. SimPoint makes use of the fact that most programs are comprised of repeated sections of execution referred to as phases. Since some of these phases look very much alike, SimPoint attempts to find them and then simulates only one of them from each set of similar phases. Results are then be extrapolated

based on the frequencies of the different phase sets in the code.

In order to identify phases in the code, the program is executed in a fast and functional manner where only the instructions are analyzed. In other words, it is executed independent of the underlying microarchitecture which allows for the final results to be carried across different microarchitectural configuration boundaries. As the program is being executed, frequency vectors meant to track various program structures are captured for contiguous intervals of size N. After the vectors are obtained for each interval of execution, their dimensionality is reduced, while still preserving commonality, by a process of random linear projection to allow for faster analysis in later steps. The reduced vectors are then input into a k-means clustering algorithm, using various values of k representing the number of clusters, to find a minimum number of clusters that satisfies a "goodness of fit" value calculated using the Bayesian Information Criterion on each clustering that is produced by the k-means algorithm. This final clustering will then contain a grouping of intervals into phases.

Once the clustering is found, it is input into a selection algorithm that will choose a representative interval from each cluster. To do this, the Euclidean distances between the vectors in the cluster are calculated and used to determine the similarity of each interval to one another. The interval that is most like the other intervals, referred to as the centroid, is then chosen as the most representative, assigned to be the simulation point for the cluster, and given a weight that corresponds to the overall number of instructions that it represents in the program. After a simulation point is found for each cluster, all simulation points from each cluster can then be simulated in detail.

When all the simulations have finished, it is a simple matter of multiplying the

metrics gathered from each interval by the weight assigned to that interval. These weighted metrics are then summed together to get the final metric used for the entire program. As an example, if a program consisted of two intervals where the first had a weight of .25 with a measured value of 5 and the second had a weight of .75 with a measured value of 10, the full execution metric would be estimated at .25(5) + .75(10) = 8.75.

Several modifications have been made to SimPoint that address some potential shortcomings of the original implementation. The first modification attempts to make SimPoint faster by introducing an additional variable into the simulation point selection algorithm [35]. Instead of simply using Euclidean distance similarity for selection, intervals are given higher priority if they occur early in the code as opposed to ones that appear later. This early SimPoint method makes it possible for the serial simulation to finish quicker since its points will occur earlier in the code. The second modification removes the restriction that each interval must be of equal length [25]. This allows for a more representative sample to be obtained since the intervals are not constrained to a specific length. Finally, additional improvements are made in the newest release of SimPoint, version 3.0 [21]. These improvements consist of faster and more efficient cluster identification, better clustering in general, the ability to handle large numbers of simulation points, and the ability to output only those clusters that account for the majority of the execution.

The results for SimPoint when run on the SPEC CPU2000 benchmarks show that an average error rate of 2.1% is achieved for the standard SimPoint algorithm and an average error rate of 3.5% is achieved for the early SimPoint algorithm. As expected, the early SimPoint algorithm completed in much less time, finishing 15 times faster than the standard algorithm. If each simulation point were to be

executed independently, the run time of the simulation would further improve and be merely a function of the number of instructions in each interval that are executed in detail combined with the amount of fast-forwarding needed to reach the interval. This amount of time would be orders of magnitude smaller since a typical interval has a length of 10 million instructions while whole programs often run into many billions of instructions.

2.4.3 Improving Sampled Simulation

Although SMARTS and SimPoint can greatly reduce the amount of time needed to simulate a new design, work done in [28] contributes several improvements to these and other instruction sampling techniques. First, it is shows that the number of instructions contained in each instruction sample can have an affect on its accuracy. In particular, smaller and more numerous samples lead to more accuracy in general than larger, less frequent samples when the simulation budget remains fixed. Second, the number of instructions necessary to estimate different performance metrics within a certain margin of error is shown to vary based on the metric. For example, if the only requirement for a simulation is to show the expected speedup of a design change, then 9X less instructions need to be sampled in order to remain within the same margin of error that would be required when estimating CPI instead. Finally, for techniques such as SMARTS that require several tuning runs of the benchmark in order to reach a certain confidence interval, it is shown that these tuning runs are no longer necessary when a proposed dynamic stopping technique is used in conjunction with online transactional processing benchmarks due to their unique execution patterns.

2.5 Techniques for Checkpointing and Warmup

After sample intervals are identified by methods such as those in section 2.4, they can be executed in a variety of ways. The first, and simplest, method simply runs the benchmark through a functional simulator that only updates the essential state of the system until it reaches the beginning of the interval. Then the interval's execution can begin in detail. Unfortunately, this technique, referred to as fast-forwarding, can take an incredible amount of time if the benchmark occurs late in the dynamic instruction stream of the benchmark. Alternatively, through a process called checkpointing, a copy of the system state, called a checkpoint, is gathered immediately prior to the execution of the interval. Then, instead of fast-forwarding to the start of the benchmark, this checkpoint data can be loaded into the simulator and the interval can immediately begin its execution.

Checkpoint data can be composed of many different elements meant to refresh the state of not only the architectural components of the system, but also its microarchitectural components. At the simplest level, the checkpoint must at least refresh the architectural state (i.e. the main memory and registers) used in the interval. However, if the sampled interval is to be completely simulated in detail on the microarchitectural level, there must be more detailed information contained in the checkpoint to refresh such components as the caches and branch predictor. The process of using this detailed information to update the microarchitectural state of the system is referred to as warmup. This section will describe several different methods of checkpointing and warmup in detail.

2.5.1 Direct Execution Checkpointing

The techniques in this section obtain checkpoint and warmup data 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. For systems that are in the early stages of development, a hardware implementation may not even exist. Regardless, they provide a very rapid methodology for obtaining checkpoint and warmup data.

The SimSnap tool [42] is a checkpointing approach that places the location where checkpoint data needs to be obtained directly in the source code of the program. The user has to modify the original source code of the benchmark to include checkpointing routines and also has to supply, on a function level, a location for where the code should be checkpointed. After the benchmark is compiled, it is run and at the predefined location in the code it will output checkpointing data to a separate file. Subsequent runs of the benchmark, when executed in a restore mode, will read in the data from the checkpoint file and resume operation where the benchmark left off.

Because of the fact that SimSnap relies heavily on the compiler and has to checkpoint not only the data required for the proper execution of the simulation interval, but the entire system state, it has several drawbacks. The first is that the checkpointing files can be quite large since they contain the entire state of the system. A second drawback is that SimSnap not only requires the user to make changes to the source code, but also that the source code itself be available. For benchmarks that do not supply source code, this will prevent them from utilizing the checkpointing features of SimSnap. A final drawback is that SimSnap will only work if the benchmark's native programming language is supported by a compiler. If the program was written in a propriety language or one that is no longer heavily used, then SimSnap

would not be able to generate checkpoint code.

In [34] and [40], direct execution is used to obtain instructions from within an instruction interval. In [34], the Pin [27] tool, which relies on direct execution, 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 [40], 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.

The Direct SMARTS technique described in [12], uses direct execution to progress the original benchmark between its intervals of detailed simulation. The checkpoint and warmup data that is generated by Direct SMARTS is fed directly into the simulator during the period of direct execution and is not stored for later use. This ability is made possible by running the benchmark inside of the RSIM simulator [33] which has the ability to switch back and forth between modes of direct execution and detailed simulation.

In [30], the Pin tool is again used to directly execute a benchmark. 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. This type of checkpointing is different than previous methods since it removes the need for a simulator to support system calls. This is important since popular simulators such as SimpleScalar must emulate the effects of system calls and many applications, particularly applications running in Linux, require system

calls that may not be emulated. Therefore, since the system calls have effectively been checkpointed, SimpleScalar can execute a broad range of applications that it previously could not support.

2.5.2 Simulator-Derived Checkpointing

Work done in [43] 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. TMI creates a list of memory addresses and data values that are used to refresh the architectural state of the system. TMI only contains stores to memory locations that are needed within the simulation interval and is, therefore, much smaller than a full system checkpoint. In addition, it uses several data packing strategies to reduce its overall size. 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 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 [43], [44] and [45] 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 since the size of each SMARTS interval is much smaller than the interval size in SimPoint, roughly 1000 instructions, the amount of checkpointing data needed per interval is very small. Unlike [43], however, caches are not the only microarchitectural element that are warmed up. The branch predictor is also warmed by storing a variety of configurations worth of checkpointing data for different branch predictor organizations in the checkpoint. The combination of the checkpoint and warmup data with

the interval of execution is referred to as a live-point and a library of live-points meant to checkpoint the entire SPEC2000 benchmarking suite occupies 12 GB of file space. Since each live-point is an individual simulation element, the live-points that comprise a full benchmark can be executed in parallel and in any order. In addition, if the live-points are randomly executed, then the confidence interval of the simulation can be tracked over time and when it reaches an acceptable level, the simulation can be terminated early. This addition of warmup and checkpointing data and the ability to leave a simulation early makes the live-points technique a fast and accurate simulation technique. Later work done in [46] extends live-points to cover multiprocessor server workloads and creates a very powerful simulation environment for future workloads.

In [4], [3], and [2], several additional methods are described that create checkpoint data that is similar to that generated in [43] and [45]. [4] presents a software structure called a memory timestamp record (MTR) that stores memory access patterns that can be used to refresh the state of a variety of cache configurations in a multiprocessor system. [3] presents a technique called branch predictor based compression (BPC) that creates a highly compressed representation of many different branch predictor configurations that can be used to warmup a variety of different branch predictors. Since MTR and BPC are both microarchitecture-independent methods, they offer a great deal of flexibility when warming up the state of a system.

2.5.3 Reuse Latency Warmup

Two techniques that offer very different warmup strategies than those described in the previous sections are MRRL [22] and BLRL [17]. Both techniques are based on what are referred to as *reuse latencies*, the distance in the dynamic instruction stream between two accesses to the same memory location. By analyzing these dis-

tances during an initial functional simulation, MRRL and BLRL identify a starting point in the benchmark that occurs prior to the beginning of a sampled instruction interval. This starting point is where execution should begin in order to effectively warmup the state of the system by the time the interval starts. This starting point identification is different than the previously discussed techniques since it does not require the discovery, storage, and loading of any checkpoint data and it is completely microarchitecture-independent. It simply tells the simulator to start executing the benchmark a certain number of instructions before the interval and then begin gathering statistics once the interval is reached. Since the introduction of MRRL preceded BLRL by several years, BLRL was able to improve upon the work in MRRL and decrease the size of the warmup interval by half over MRRL. Regardless, both are very useful methods for simply and effectively warming a system's state prior to the execution of a simulation interval.

2.6 Summary

This chapter presented a variety of techniques targeted at reducing the simulation time of a benchmarking program while still maintaining accuracy. Benchmark suite reduction locates redundancies in benchmarking suites and attempts to find the minimum subset of benchmarks and input datasets that are necessary to represent the entire benchmarking suite. Statistical simulation generates a profile of the original benchmark and then uses this to create a synthetic trace or program that can be used early in the design cycle to quickly sort out viable design choices that should be evaluated in more detail. Instruction sampling can be used to study a benchmark in much more detail by choosing representative intervals of instructions and using only those intervals for simulation. Checkpointing can be used to further decrease

the amount of simulation time needed to run intervals of instructions by allowing their immediate execution after checkpoint data has been handled. Finally, warmup can help to guarantee that the intervals' results will be as accurate as possible by refreshing the state of not only the system's architecture, but its microarchitecture as well. Table 2.2 compares 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.

Technique	Execution Time per B-mark	CPI Prediction Accuracy	Representativeness	Micro-architecture Dependent	Storage Req's	Flexibility
B-mark Suite Reduction [24]	Variable	Variable	Low	No	N/A	High
Statistical (Trace) [18]	approx. 1000x speedup	2.3%	Low	Yes	negligible	Low
Statistical (Testcase) [10]	approx. 1000x speedup	2.4%	Low	Yes	negligible	High
SMARTS [47]	5 hrs	0.64%	High	No	N/A	Medium
SimPoint [41]	2.8 hrs	3.7%	High	No	N/A	Medium
SimPoint Startup [43]	14 mins (serial) 1 min (parallel)	1.2%	High	Yes	4 GB for 20 SPEC2K b-marks	Low
LivePoints [45]	91 secs	1.6%	High	Yes	12 GB compressed all SPEC2K b-marks	Low

Table 2.2: Comparison of Several Popular Simulation Time Reduction Techniques

CHAPTER III

Intrinsic Checkpointing with Binary Modification

3.1 Overview

This chapter will present a technique called *Intrinsic Checkpointing through Binary Modification* (ICBM), first published in [39], which modifies a benchmark binary so that it will only execute a certain part of its code. This is done by first analyzing the code within a provided interval of execution and then generating a set of checkpointing and warmup instructions. These instructions, when inserted into the original binary, effectively recreate the system environment that the interval would see at its start had the binary been executed normally. The original binary must then be modified to start its execution at the beginning of these checkpointing instructions and then transfer control to the start of the interval instructions. Figure 3.1 represents these phases pictorially.

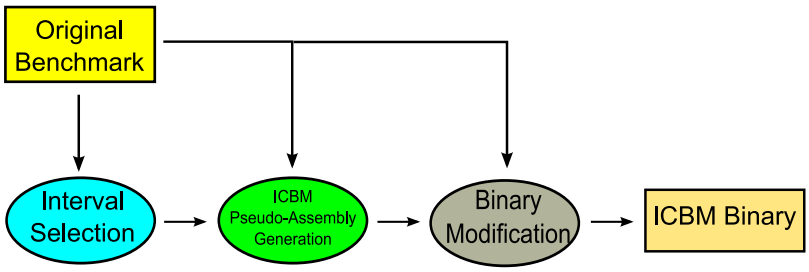

Figure 3.1: ICBM Process Flow Diagram

This technique creates a form of intrinsic checkpointing that removes the require-

ments of checkpointing and fast forwarding from the simulator and places it into the program itself. When combined with tools such as SimPoint [41] and BLRL [17], it becomes possible to properly benchmark a system in a fraction of the time compared to using the original benchmarking binary. For simulators such as RTL models where checkpointing and fast forwarding may be very difficult to implement, this technique allows for the simulation of benchmarks that in the past were severely hindered by their execution times. Figure 3.2 gives a graphical overview of how the various fast-forwarding, warmup, and detailed simulation intervals are combined into individual ICBM binaries.

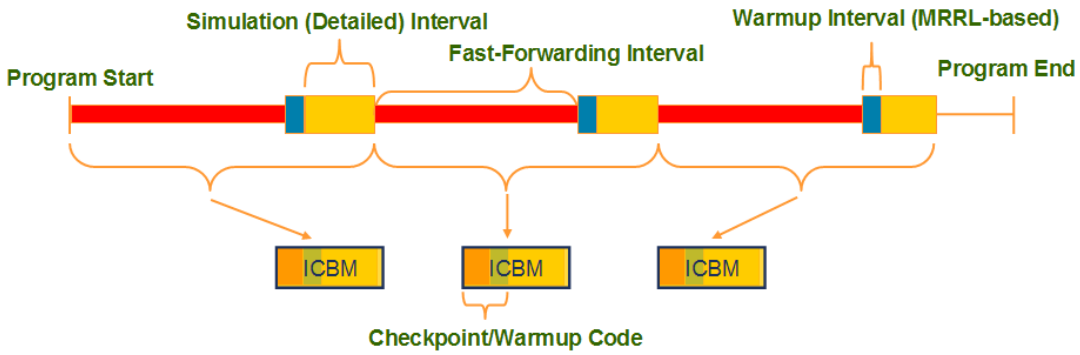

Figure 3.2: ICBM Interval Selection and Creation

This chapter is organized as follows. Sections 3.2, 3.3, and 3.4 will discuss how the register file, memory, and system calls are checkpointed, respectively. Section 3.5 with describe how the checkpointing code is inserted into the original benchmark through the binary modification process. Section 3.6 will describe how it is possible for the interval to exit its execution after the appropriate number of instructions and section 3.7 will list several implementation issues that are associated with ICBM. Section 3.8 will describe how the ICBM technique is verified and section 3.9 will illustrate the basic ICBM method with an example. Finally, section 3.10 will conclude with a summary.

3.2 Register Checkpointing

In order to create the necessary code to bring the simulation interval up to date, an analysis of the benchmark must be done to find out what system state needs to be restored for the proper execution of the simulation interval. The register file is by far the simplest component whose usage must tracked and restored since it only contains a limited number of architectural elements. To do this, a simple snapshot of the register file is taken immediately before the first instruction in the simulation interval and its current values are noted. These values are then restored with simple load instructions in the checkpoint code to bring the register file up to date.

3.3 Memory Checkpointing

Memory, unlike the register file, proves to be a much more troublesome component to intrinsically checkpoint, especially if there are many instructions within the simulation interval that depend on values in memory that were modified before the interval began. To track these modifications, a copy of the initial memory, INITMEM, is created when the program first begins its execution and the program is then allowed to fast-forward up to the simulation point in question. At that point, a copy of memory, CHECKMEM, is saved to hold any changes to the memory from the pre-interval code. Next, as the simulation interval progresses, each time a load is encountered within the simulation interval, if the value in INITMEM differs from the value in the current memory, CURRENTMEM, then it is known that some instruction in the pre-interval code stored a value to that memory location. Consequently, the memory location and its value from CHECKMEM are logged so that later it can be turned into a store in the checkpoint code. The memory location in INITMEM is then updated with the value from CURRENTMEM so that this situation will

not occur again. In the case of a store to memory within the simulation interval, the value is simply stored to both CURRENTMEM and INITMEM since this store effectively overwrites any loads that occurred in the pre-interval code.

3.4 System Call Checkpointing

System calls (syscalls) also create a rather difficult problem when handling the instructions prior to the interval's execution. File output syscalls are ignored since, unless the output file is used later in the interval as input, they won't affect the results of the program. File input, however, modifies file pointers in the program and the usage of all input files is logged so that all relevant pointers can be updated in the checkpoint code. This is done by noting the work that occurs in the simulation for file handling and keeping track of the various values that control file manipulation. Other system calls, such as those that change directories, modify file permissions, or open/close files also have these types of methods that track their usage. Code is inserted into the system call handlers of the simulator to have their requisite input values loaded using analysis similar to the register and memory analysis discussed above. Then, the actual system call is run from within the checkpoint code. Since there are only a few syscalls that can actually affect the system environment, the generated code for syscall checkpointing is small. The remainder of the syscalls that occur prior to the interval are either ignored or, if they modify memory or register state, the normal simulation interval analysis discovers these changes and generates the corresponding checkpoint code.

3.5 Interval Execution

Once the checkpointing data has been created and saved to a "pseudo-assembly" program, it must be converted into the appropriate machine instructions and inserted

into the original benchmark binary. To do this, the ICBM technique uses a Perl script that converts the pseudo-assembly into machine code, finds a place in the binary to insert the code, and then rewrites the binary. During insertion, it is important that the restoration of the registers occur at the end of the checkpointing code since the operations for restoring the memory need to make use of the registers. After the insertion, the new binary has its starting point set to the beginning of the checkpointing code. Next, at the end of the checkpointing code a final jump is inserted that will move the execution to the beginning of the simulation interval. Depending on the instructions available, this can be done with either an explicit jump to a PC provided in the instruction, or a jump to a PC value that is stored in a register. If a register is needed, as is the case with ICBM, then it will have to be one that is going to be overwritten before it is used inside the interval so that proper execution will be guaranteed. Figure 3.3 shows the instructions in a typical, pseudo-assembly checkpoint file prior to being converted into machine code and inserted into the binary. First, any syscalls that must be run prior to the interval are output, followed by the section to restore the memory, and then register restoration occurs. Finally, the jump to the start of the simulation interval is executed.

3.6 Exit Handling

A final issue that will need to be dealt with is the method by which the simulation ends following the execution of the interval. In a typical simulator, the preferable method is to simply tell the simulation when to stop by providing a count for the number of cycles to simulate. This, however, requires that the simulator have the ability to track the number of instructions on a per instruction basis from within the simulation. In addition, if the goal of creating a set of microbenchmarks is to

```
# <syscall stores>
st  <addr>, <imm>              # store imm value into mem @ addr
# <syscall register loads>
ldi  rX, <imm>                 # load imm value into register rX
SYSCALL(example)               # execute the syscall

# Memory Restoration
stb   <addr>, <8-bit imm>      # store a byte into mem @ addr
sth   <addr>, <16-bit imm>     # store a half into mem @ addr
stw   <addr>, <32-bit imm>     # store a word into mem @ addr
stqw  <addr>, <64-bit imm>     # store a quad into mem @ addr

# Register File Restoration
ldi  rX, <imm>                 # load imm value into Int register rX
ldi  rX, <imm>                 # load imm value into FP register rX

jmp   <interval_start>         # jump to the interval start
```

Figure 3.3: Pseudo-Assembly Template for ICBM code

be able to move them to native hardware, the hardware may not have the ability to track an instruction count. Thus, it will become necessary to insert a halt operation at a logical point in the code that will allow only the interval and its respective checkpointing code to be executed. Unfortunately, it is not as simple as replacing the instruction corresponding to the end of the interval with a halt instruction, because this instruction may be encountered prior to the interval's end due to things such as looping code. Therefore, there will need to be an additional level of analysis that occurs during the checkpoint code generation phase that identifies an instruction that can be replaced with a halt that will still preserve the most of the interval's behavior as possible. Figure 3.4 depicts a general view of an ICBM binary once it has been created.

3.7 Implementation Issues

There are several things that need to be explored before ICBM can be considered effective. The first of which is whether ICBM will be applicable across different benchmark suites. For example, memory intensive benchmarks such as large database

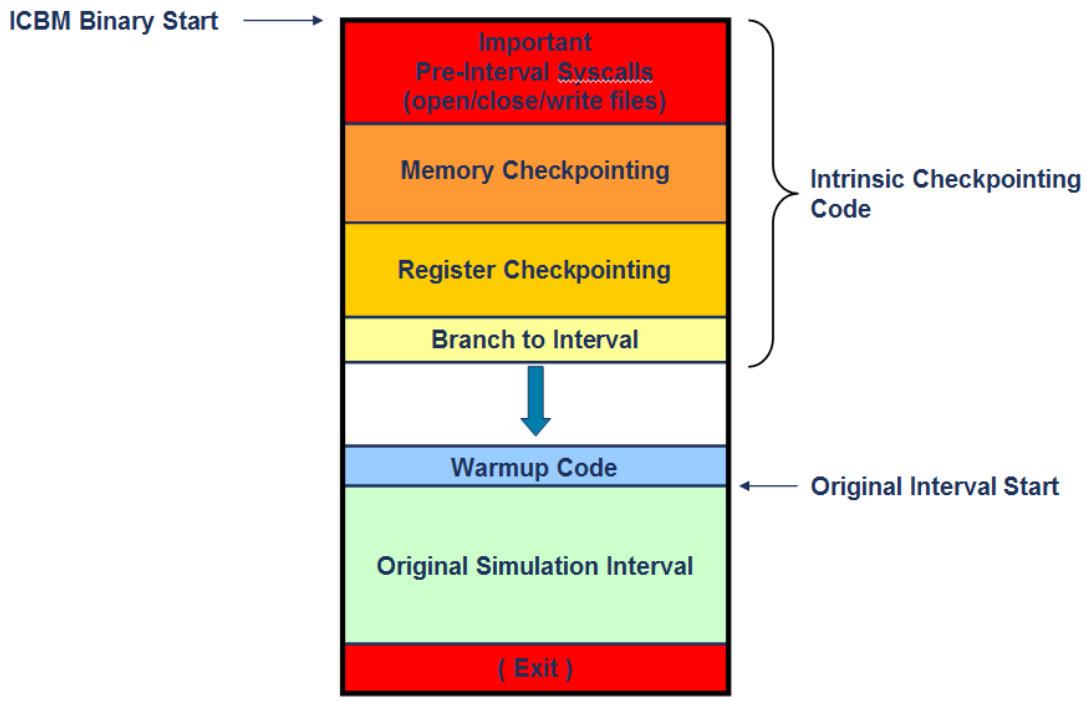

Figure 3.4: Anatomy of an ICBM Binary

transactions could work with an enormous amount of data that would need to be loaded into the memory before the simulation interval could be run. However, the worst penalty, measured in the number of loads necessary to bring the memory up-to-date, would be equal to a small multiple of the instructions in the interval since the instructions in the interval can only reference a finite amount of memory. This overhead, especially if the interval occurred late in the benchmark, would still be much less than the original method of fast-forwarding through all the pre-interval code. This same logic could be applied to the presence of large amounts of file manipulation or changes to the overall state of the system. Since there is only a limited amount of work being done in the interval, there should only be a limited amount of pre-interval work to be done.

Another point to consider is the importance of warming up the various microarchitectural elements before the detailed simulation interval can occur. This problem

is addressed in both SMARTS and SimPoint, and there are many different methods that can be used. A paper on SimPoint published in SIGMETRICS [20] summarized the various techniques including assuming all first accesses to important structures were hits, calculating the working set of the important structures in the interval and executing all instructions before the interval that will satisfy this working set [22] [17], or keeping certain structures "warm" during the fast-forwarding stage by updating only important parts [47]. For many benchmarks, the method that assumes hits on first accesses is fairly accurate, but the two other techniques provide the most accurate results. The technique that keeps the structures warm is obviously not a good candidate for ICBM since it requires the entire fast forwarding segment to be run. However, the technique that does a working set analysis on the structures, i.e. MRRL and BLRL, is very promising since the code to warm the structures can be built into the pre-interval code and not included in the statistics.

A final issue involves the simulation of multiple instruction intervals from one benchmark. Since the current ICBM technique can only handle one instruction interval at a time, the ICBM routines will have to be called repeatedly if a benchmark requires the execution of multiple intervals. If the user's goal is to create a large number of individual benchmark intervals that can be executed in parallel, then this is not an issue. However, it would be convenient if the intervals could be combined into a single benchmark. While it is possible to address this shortcoming by stringing multiple intrinsic checkpointing intervals together and jumping back and forth between checkpointing and interval code, it would involve large amounts of binary analysis and modification. Therefore, this is best solved another way and will be addressed later in chapter IV with the introduction of SuiteSpots.

3.8 Validation

Due to the nature of ICBM, if the intrinsically checkpointed benchmark begins its execution with the proper checkpoint data, it will generate the exact same simulation results as the instruction interval from the original binary, assuming no warmup has occurred in either case. This follows from the fact that all the code from the original binary remains intact throughout the checkpointing process and the only modification made to the binary occurs when the checkpointing data is inserted. For validation purposes, the simulation intervals for both the original binaries and the intrinsically checkpointed binaries for 19 of the SPEC2000 [23] benchmarks were simulated in detail and statistics were recorded for comparison. For each of the benchmarks, the measured IPC did not vary between the original benchmark and its intrinsically checkpointed counterpart. In addition to checking statistics, cycle-level register file comparisons were made between the ICBM binary and the original binary for several benchmarks to further validate the ICBM technique. Therefore, it can be concluded that each of the intervals were properly checkpointed.

3.9 Alpha Example

As an example of ICBM, the following section will present an analysis of the eon benchmark from SPEC2000. The SimpleScalar [1] simulator targeted at the Alpha ISA [13] was modified to do the analysis described in this chapter and to output a pseudo-assembly file that contains the essential syscalls, stores for memory restoration, and loads for register restoration all of which will comprise the checkpointing code that will need to be inserted into the original binary. Figure 3.5 lists a small part of the pseuso-assembly that was generated for syscall handling and figure 3.6 gives a sample of the loads and stores for memory and register restorations. For each

syscall, only the registers and memory that the syscall will use are restored prior to its execution. A good example of this is the write syscall where the data to write to the file is loaded into memory prior to being written to memory with the syscall.

```
...
...
ldi  r0,  0x000000000000002d
ldi  r16, 0x00000001200028f8
ldi  r17, 0x0000000000000000
SYSCALL(open)

ldi  r0,  0x0000000000000003
ldi  r16, 0x0000000000000006
ldi  r17, 0x0000000140039d08
SYSCALL(read)

stb  0x000000014003a200, 0x45
stb  0x000000014003a201, 0x6f
stb  0x000000014003a204, 0x20
ldi  r0,  0x0000000000000004
ldi  r16, 0x0000000000000001
ldi  r17, 0x000000014003a200
SYSCALL(write)
...
...
```

Figure 3.5: Pseudo-Assembly for Pre-Interval Syscall Checkpointing

```
# Stores for memory restoration
stqw 0x000000011ff95948, 0x3ff688818757f6a3
stw  0x000000011ff95914, 0x00000001
stw  0x000000011ff9599c, 0x0000001c

# Integer registers restoration
ldi  r0,  0x000000000000000c
ldi  r1,  0x000000014003f060

# Floating Point registers restoration
ldi  f0,  0x3fc7cd8dad5988b8
ldi  f1,  0x000000011ff95c20

# Special/Control registers restoration
ldi  FPCR, 0x0000000000000000
ldi  UNIQ, 0x0000000140008f08
```

Figure 3.6: Pseudo-Assembly for Register and Memory Restoration

As figure 3.6 shows, the pseudo-assembly consists of only one instruction for each memory location restoration. However, this pseudo-assembly cannot be directly

translated into Alpha machine language since the Alpha ISA only has 32-bit instructions. Therefore, the 64-bit addresses and data will need to be broken up into a series of optimized machine instructions that can produce the same effect. For example, all the 64-bit addresses and data that are needed in the checkpointing code for memory restoration could be written into a section of the binary set aside for data. A register could then hold the starting address of this section of the code and offsets could be used in conjunction with the register to load data into registers whose values could then be stored into the appropriate location in memory. Figure 3.7 gives an example of converting several 64-bit address/data stores into a series of machine instructions. As the figure shows, it is possible to convert one checkpoint store that requires both a 64-bit address and 64-bit data into a series of instructions that could take at most 7, or as little as 2, machine instructions depending on each store's address values. For register restoration, the same process could be used to restore a register in at most 5, or as little as 1, machine instruction.

3.10 Summary

This chapter has presented ICBM, a methodology that dramatically decreases the simulation time of a benchmarking binary by removing the need to fast forward through a large number of instructions without relying on the simulator to explicitly checkpoint the code. By analyzing the code of a desired simulation interval, ICBM generates a portion of checkpointing code meant to replicate the outcome of the instructions that were executed prior to the interval. This checkpointing code is then inserted at the start of the original binary. When the checkpoint code finishes its execution, control is transferred to the beginning of the simulation interval and execution can continue until the end of the interval.

```
#Pseudo Assembly Checkpoint Stores
#1 stqw   <Addr1 = 0xaaaabbbbccccdddd>,  <Value1 = 0xddddccccbbbbaaaa>
#2 stqw   <Addr2 = 0xbbbbccccdddd0000>,  <Value2 = 0xeeeeddddccccbbbb>
#3 stqw   <Addr3 = 0xbbbbccccdddd0008>,  <Value3 = 0xffffeeeeddddcccc>

#Data contained in the binary
.data:
0  : 0xaaaabbbbccccdddd (Addr1)
8  : 0xddddccccbbbbaaaa (Value1)
16 : 0xbbbbccccdddd0000 (Addr2)
24 : 0xeeeeddddccccbbbb (Value2)
32 : 0xffffeeeeddddcccc (Value3)

#Machine Code
.text:
## Load start address of .data section into r1 (4 instructions) ##
ldqw r2,0(r1)     # set up Addr1
ldqw r3,8(r1)     # set up Value1
stqw r3,0(r2)     # checkpoint store #1 Value1 @ Addr1
ldqw r2,16(r1)   # set up Addr2
ldqw r3,24(r1)   # set up Value2
stqw r3,0(r2)     # checkpoint store #2 Value2 @ Addr2
ldqw r3,32(r1)   # setup Value3
stqw r3,8(r2)     # checkpoint store #3 Value3 @ (Addr3 = Addr2 + 8)
```

Figure 3.7: Converting Pseudo-Assembly to Alpha Machine Instructions

CHAPTER IV

Intrinsically Checkpointed Assembly Code

4.1 Overview

The methodology presented in this chapter 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 linked together by branches. Figure 4.1 represents these phases pictorially.

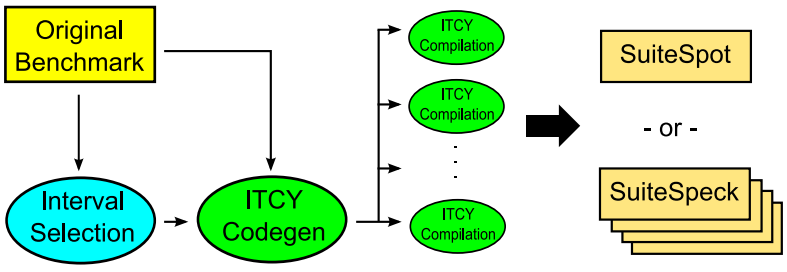

Figure 4.1: ITCY Code Generation Diagram

4.2 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. As tools such as SimPoint and SMARTS have shown, the entire dynamic instruction stream of a benchmark's execution does not need to be executed in order to properly simulate the behavior of the benchmark. In fact, they have shown that only a very 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. The remainder of the program can quickly be simulated on a functional level. However, this functional simulation can still take many hours to complete and the ITCY methodology removes the need for this functional simulation altogether.

4.3 ITCY Code Generation

4.3.1 Overview

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 toolset targeting the Alpha 21264 ISA. This ITCY code consists of three main parts: Intrinsic Checkpointing (IC) code meant to recreate the environment of the original benchmark, the original static instructions from the interval being converted, and special control code that is needed to handle various situations that will be described 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 effectively 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 that it contains will reference valid locations in its allocated memory space. Third, it must guarantee that the new dynamic instruction stream occurs 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. Each of these issues, along with the details of how system calls are emulated, require a number of modifications to the original simulation interval and will be discussed in further detail below. Figure 4.2 gives a graphical overview of how the various fast-forwarding, warmup, and detailed simulation intervals are combined, along with a depiction of how the above issues appear in the intervals, into either a SuiteSpot or a set of SuiteSpecks.

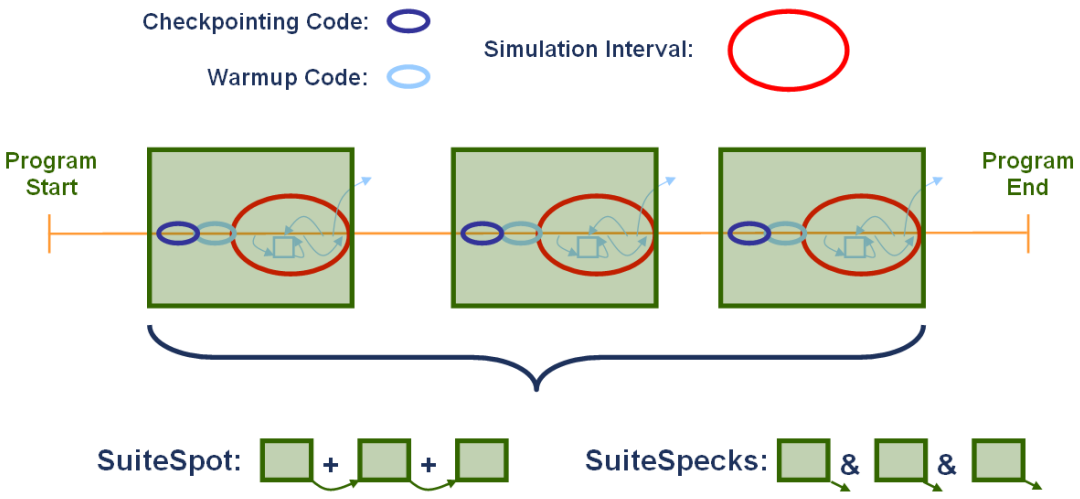

Figure 4.2: ITCY Interval Selection and the Creation of a SuiteSpot or SuiteSpecks

4.3.2 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 [39] with several modifications made to the way that memory is checkpointed. The intrinsic checkpointing of the register file remains unchanged.

In [39], several copies of the memory state were maintained: one from the start of the benchmark, one from the start of the interval, and one from the current point of execution. The memory from the start of the benchmark was compared against the current memory each time a load was encountered inside the simulation interval. If the values did not match, then an IC store was generated using the data from the memory at the start of the interval and placed at the beginning of the new interval.

The ITCY methods do not make use of multiple copies of memory 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 their values before the interval began. 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 later in the interval. 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 temporary list into larger multi-byte stores in order

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 [39], 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 a series of static load/store instructions that generated their addresses inline. Unfortunately, as the size of the interval grew, the number of these static instructions reached unwieldy proportions. To address this problem, the ITCY technique converts the static instructions into a loop that iterates over all the values in the .data section. This does, however, require the storage of not only the data values needed to checkpoint memory, but also the addresses themselves. This does require extra space in the .data section of the new binary, but it is offset by the fact that the new .text section no longer requires the large number of static instructions.

4.3.3 Ensuring Valid Memory Accesses

In order to allow the potential combination of multiple simulation intervals into a single, new benchmark, the location of the ITCY .text section can no longer be located in the same memory space as the original benchmark. Previous intrinsic checkpointing work 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 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.

The default memory layout for an Alpha binary places the .text section in a

memory region that precedes the .data section. The stack then starts at the beginning of the .text section and grows toward lower addresses. The heap, on the other hand, starts at the end of the .data section and grows toward higher addresses updating the value of the ld_brk_point (i.e. the top of the heap) whenever more space is needed. The ITCY technique leverages this information when creating the new ITCY binary and uses it to determine where in the new address space the sections of the code should be located. In addition, since the new binary utilizes the original binary's stack and heap locations for the execution of the interval code, it has no need for a stack or heap of its own. It simply requires a .data section with a predetermined size for storing IC data and a .text section which contains IC code that will not need a stack.

To ensure that the ITCY binary will have access to all the memory locations from the original interval's memory space, its .data section is placed in the addresses directly below the original stack and its .text section directly above the original heap. Then, immediately upon entering the initial IC code in the new binary, the IC code makes a single system call that sets the value of the ld_brk_point to be the maximum address that was accessed in the original heap. This sets the range of valid data addresses to begin at the start of the new .data section and end at the top of the original heap. In essence, the entire address space used by the original interval now comprises the heap of the new binary. This guarantees that all memory accesses to the original .text and .data sections will be valid. Figure 4.3 depicts these memory layouts.

4.3.4 Preserving Dynamic Instruction Stream Execution Order

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

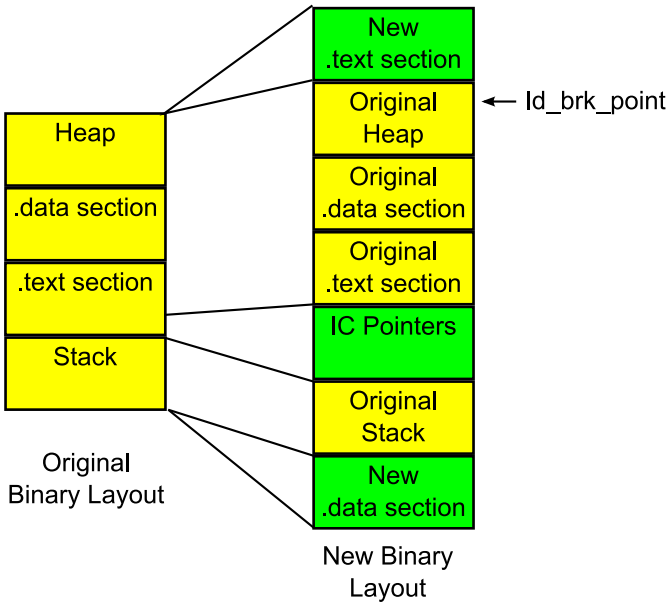

Figure 4.3: Original and New Binary Memory Layouts

follows the same execution order that was seen in its corresponding 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. Calculating these targets is done by giving every target instruction in the ITCY code a unique label that contains the original PC value. The compiler then automatically re-targets each branch when it is run. Figure 4.4 shows how this translation occurs.

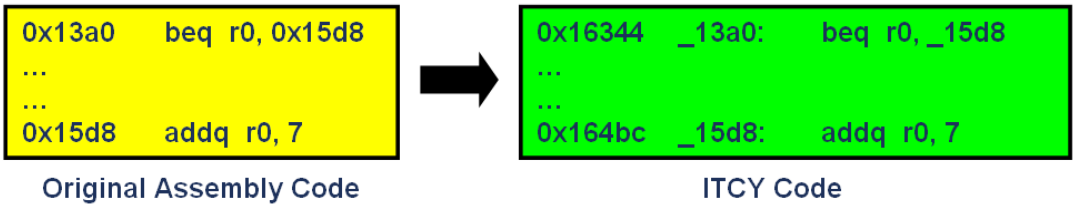

Figure 4.4: Retargeting Direct Branches to Maintain Proper Execution Order

In addition to handling the target of a branch, if a branch saved its return address to a register prior to its execution in the original interval, then this register must be explicitly loaded with the original return address prior to the execution of the new branch. The new branch must also be modified so that the return address assignment does not occur in the new binary since it would overwrite the work that was just done using the old return address. Figure 4.5 shows how this can be done in Alpha assembly code.

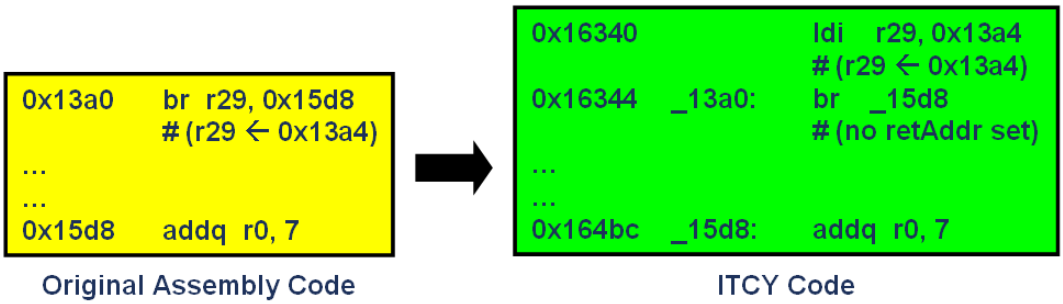

Figure 4.5: Explicitly Setting a Branch's Old Return Address Register

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 target will cause the program to transfer control into the old address space. This will violate the proper execution order of the new benchmark since the program no longer resides in the old address space. Therefore, indirect branches must be handled using a special block 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. It then writes these values into locations in the original address space during the pre-interval intrinsic checkpointing section of the ITCY code. The locations where these targets are written are not arbitrary, however. 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. Then, prior to the execution of the indirect branch in the ITCY code, the 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. Figure 4.6 shows this basic method for handling an indirect branch and its target using some Alpha-derived pseudo-code.

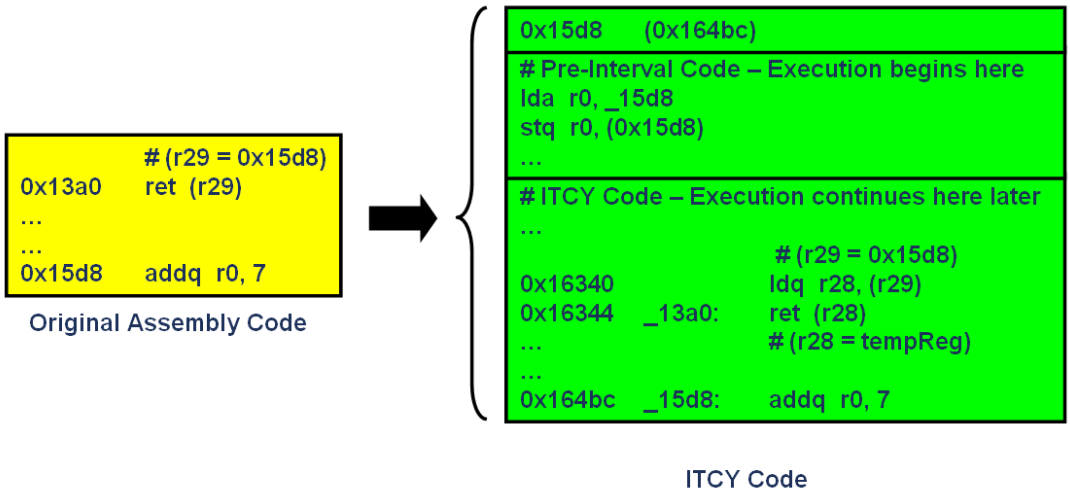

Figure 4.6: Using the Original Memory Location of an Indirect Branch to Store its New Target

However, since the Alpha ISA is used to implement these methods, one further situation must be accounted for in order for indirect branches to properly execute in all situations. The Alpha ISA has a 64-bit address space and instructions that are only 32-bits in length. Therefore, when a new indirect branch target's 64-bit PC value is written into the location of the original 32-bit branch target instruction, it essentially overwrites the contents of two Alpha instructions. A problem occurs when there are two indirect branch target instructions adjacent to each other in the old address space. If both new targets attempt to write their corresponding PC values

into these adjacent locations, then one of the branch target PC values will write over half of the other. Therefore, a different block of control code must be written to address this Alpha-specific situation.

This new block of special control code is only created for those indirect branches whose target PC values will be overwritten within the old address space by adjacent branch targets. If this situation is detected, one of the indirect branches will be selected for special handling and all of its targets will be stored in a special table in the .data section of the new binary. This indirect branch target table (IBTT) effectively contains a dynamic branch target trace of all indirect branches that require this special handling. Prior to one of these special branches executing, the IBTT is accessed using a special pointer stored in a reserved location in memory. The ITCY code generation process determines this memory location and guarantees that it will not be overwritten by any of the code in the new binary. The IBTT itself is dynamically generated when the ITCY code first begins its execution since it needs to make use of the target PC labels that were discussed earlier in this section. Figure 4.1 indicates the location of this pointer in the new memory layout and Figure 4.7 depicts how the IBTT is used to handle these special indirect branches using some Alpha-derived pseudo-code.

4.3.5 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

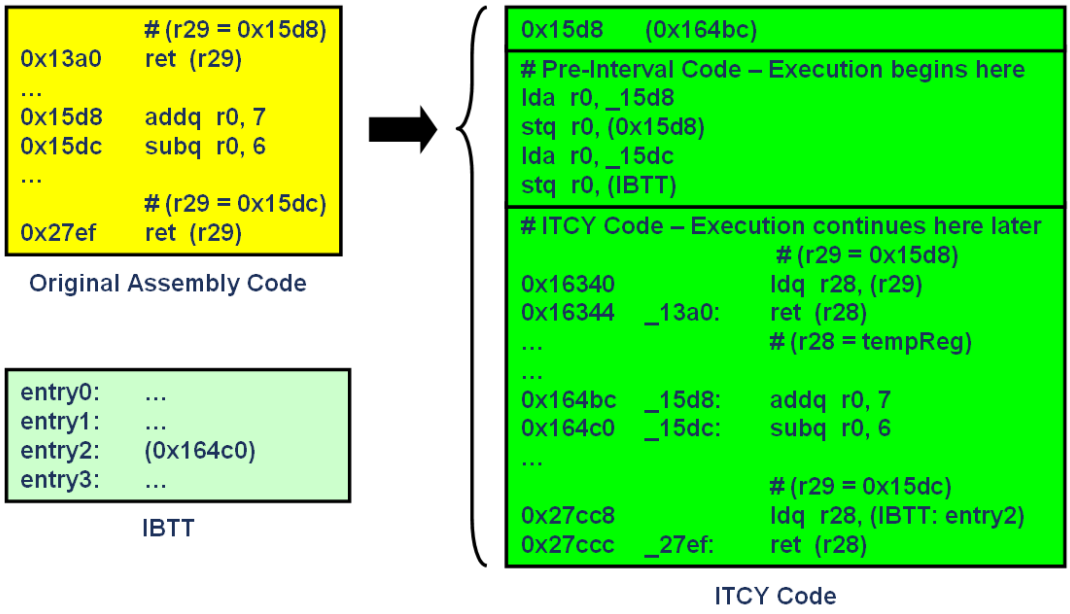

Figure 4.7: Using the IBTT to Handle Special Indirect Branch Targets

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 if they needed to be included. When applied to the interval as a whole, the resultant binary would contain a greatly reduced memory footprint. Therefore, the I-cache of the test system would perform much better when executing the ITCY code than when it was executing the original benchmark. Unfortunately, this would remove much of the representative nature of the ITCY code and, therefore, the ITCY code must be created such that its subsequent I-cache performance mimics that of the original benchmark.

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. These pads do not need to be as large as the original gap between the basic blocks. They simply need to be large enough

to place the basic blocks onto different cache lines (assuming that they were at least that far apart initially). A default cache line of 256-bytes is used as an upper limit. In addition, the ITCYgen routine attempts to place the instructions themselves on their appropriate cache line offsets so that their future performance behaves more realistically. Since the PC values of all the original static instructions are known, this method is relatively straightforward except when large amounts of control code are inserted within the ITCY code. If this occurs and the control code causes the instruction offsets to stray from their original values, the next basic block pair that was split will re-align all the instructions automatically. Figure 4.8 shows this use of instruction pads to recreate the original memory footprint in the new ITCY binary.

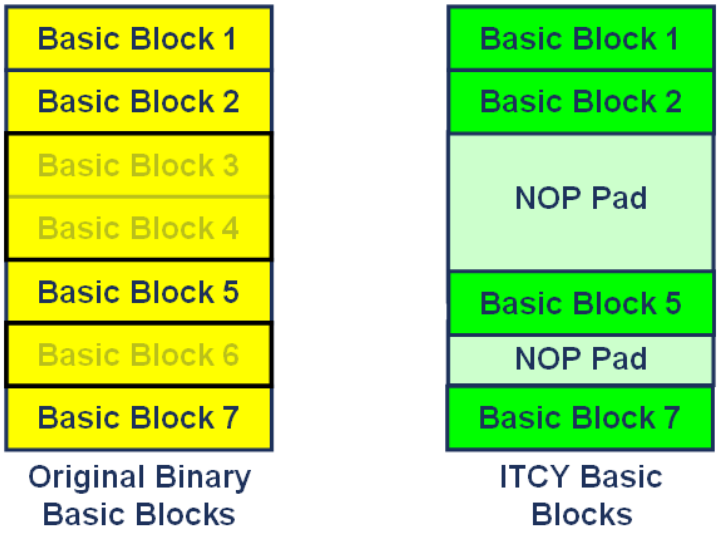

Figure 4.8: Using Instruction Pads to Recreate the Original Interval's Memory Footprint

4.3.6 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. Two different methods are proposed to handle this situation. The first technique logs the value of the PC that corresponds

to the last instruction of the interval in addition to the number of times that it is executed. SimPoints uses a similar technique to determine when to transfer to detailed simulation after fast-forwarding through a benchmark. Once the PC and count are known, special annotative instructions are added to the IC code that load the value of this exit PC along with its execution count into two predefined registers. In addition, to handle the transfer of control to additional intervals when creating a SuiteSpot, the starting address of the next interval is logged and loaded into a third register. A special signal instruction is then added to the IC code that will alert the simulator to process the three registers.

When the ITCY code is later executed and the simulator encounters the signal instruction, it reads the values of the exit PC, the occurrence count, and the starting address of the next interval from the three registers and stores them into its internal state. If the address for the next interval is equal to zero, this indicates that the final interval has been reached in a SuiteSpot or that a single SuiteSpeck is being simulated. As the detailed execution of the current interval proceeds, 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 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 technique must be used whereby a termination routine is inserted into the binary itself. This was briefly mentioned in section 3.6. 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 handled differently.

Initially, the code generator looks for a singly-executed instruction that is within a certain dynamic instruction distance from the final instruction. This instruction can occur before or after the original final instruction as long as its distance to the last instruction in the dynamic instruction stream falls within a given threshold provided by the user. If this instruction is found, then the code generator replaces it with an exit syscall or a branch to the next interval. If it is not found, then a special section of control code is added in front of the original last instruction that will execute a specific number of times before branching to an exit syscall or a branch to the next interval. This is made possible by maintaining an exit counter in a special memory location adjacent to the pointer to the IBTT. Each time the control code executes, it checks the value of the exit counter and if it is equal to zero, it branches to the exit of the interval. If it is not equal to zero, the code decrements the counter and executes the original instruction. A sample of this special control code can be seen in Figure 4.9. This technique can, however, have an undesirable effect on the overall representativeness of the ITCY code if the exit routine is executed many times inside of a loop. Therefore, the first technique discussed above where the occurrence count and PC addresses are stored in the IC code is the preferable method.

4.3.7 System Call Emulation

The final step of the code generation process involves the removal of system calls (syscalls) from the final ITCY code. Using methods similar to intrinsic checkpointing, a system call can be replaced with a branch to a special section of code that emulates

```
#set $21=0x120000018    # use register $21, original inst overwrites it
       lda $21, 3($31)
       ldah $21, 9216($21)
       sll $21, 3, $21
       stq $1, 0($21)          # save temp register $1 to 0x120000018
       ldq $1, 8($21)          # load counter from 0x120000020
       bne $1, continue        # test counter, branch past exit if not done
 #set $0=0x1                   # load $0 and $16 with exit values
       lda $0, 0x1($31)
 #set $16=0x4d
       lda $16, 0x4d($31)
       call_pal 0x83           #  call exit system call
continue:
      subq $1, 1, $1           # decrement counter
      stq $1, 8($21)           # store counter
      ldq $1, 0($21)           # restore temp register $1
      cmple $18,3,$21          # execute original instruction
```

Figure 4.9: Special Exit Handling Control Code

the effects of the original system call. The actual checkpointing data that the code section must load into the system is determined by the system call handling routines built into SimpleScalar. These routines essentially mimic the execution of the system calls using the native system call handlers of the host system running SimpleScalar.

Just like with IC, not all of the work done by a system call must be emulated. Instead, only those data values that are used by loads later in the ITCY code must be written to memory. In addition, if a store that needs to be emulated for the syscall references a memory location that was not used prior to the syscall's execution in the interval, then the store is moved back to the initial IC code block. This minimizes the effects of the syscall emulation's execution on the final detailed simulation. When these values are written to memory, their memory flags are cleared just like a normal write inside of the ITCY code. Finally, any changes to the register file will still occur whether they are used or not.

If a static system call executes more than once inside the interval, it will not produce the exact same results each time it is called. Therefore, its emulation code

must be unique each time that it is executed. This is accomplished by using a technique similar to the Indirect Branch Target Table discussed in section 4.3.4. When the ITCY code first begins executing, it creates a System Call Emulation Table (SCET) in the .data section of the binary that holds the addresses from the .text section of each system call emulation block. These emulation block starting addresses are again obtained through the use of assembly instruction labels. Since the execution of each emulation block will occur in a predetermined order, their code can be stored in the .text section in chronological order and their corresponding starting addresses can be similarly loaded into the .data section. Figure 4.10 depicts how the SCET can be used to store the addresses of system call emulation blocks and Figure 4.11 provides a specific example of how an emulation block is reached from inside the ITCY code.

Figure 4.10: Using the SCET to Store the Starting Addresses of System Call Emulation Blocks

4.4 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. An advantage of using a SimPoint over multiple SuiteSpecks is that later intervals inside a SimPoint may not need to intrinsically checkpoint as much of their state as a SuiteSpeck. Since the state of the simulation is affected by the execution of

```
.data
__IC_SYS__NUM_0__:
_0_syscall_0:  .quad _IC_0_SYS_EMUL_0
_0_syscall_1:  .quad _IC_0_SYS_EMUL_1
…
.text
# setup pointer to syscall emulation block list
  lda $1, __IC_SYS__NUM_0__
# set $2=0x120000010
  lda $2, 1($31)
  ldah $2, 4608($2)
  sll $2, 4, $2
  stq $1, 0($2) # store pointer
….
_0_1201cb218:
#set $28=0x120000010
  lda $28, 1($31)
  ldah $28, 4608($28)
  sll $28, 4, $28      # put ptr addr in $28
  ldq $0, 0($28)       # load pointer into $0
  addq $0, 8, $0       # update ptr for later
  stq $0, 0($28)       # store the next ptr
  ldq $0, -8($0)       # load current ptr
  jmp ($0)             # jump to emulation block
```

```
_IC_2_SYSCALL_EMUL_0:
#set $1=0x11ff9e578
  lda $1, 15535($31)
  ldah $1, 9215($1)
  sll $1, 3, $1
#set $0=0x0
  bis $31, $31, $0
  stb $0, -32712($1)
#set $1=0x0          # restore $1
  bis $31, $31, $1
#set $0=0x0          # restore $0
  bis $31, $31, $0
  br _0_1201cb218
_IC_2_SYSCALL_EMUL_1:
#set $1=0x11ff9e578
  lda $1, 15535($31)
  ldah $1, 9215($1)
  sll $1, 3, $1
#set $0=0x1
  lda $0, 0x1($31)
  stb $0, -32782($1)
#set $1=0x0          # restore $1
  bis $31, $31, $1
#set $0=0x0          # restore $0
  bis $31, $31, $0
  br _0_1201cb218
```

Figure 4.11: System Call Emulation Code Sample

each of its intervals, later intervals can take advantage of the fact that some of the system state has already been checkpointed by earlier intervals. Therefore, these intervals may not need to generate as much intrinsic checkpointing code as their earlier counterparts.

SuiteSpecks, on the other hand, must rely entirely on intrinsic checkpointing to bring the state of the system up to date. This state is still drastically smaller than traditional checkpointing methods, however. The main advantage of SuiteSpecks over SuiteSpots is that they can all be executed in parallel and will provide results much faster. However, this requires more simulation resources.

The compilation of ITCY code into SuiteSpecks or a single SuiteSpote proceeds in a very straightforward fashion. The code generation phase in section 4.3 auto-

matically generates a Makefile that can be used to compile the code. The Makefile must be created in order to tell the compiler where it should place the .data and .text sections of the new ITCY code as was described in section 4.3.3. Figure 4.12 depicts a general view of an ITCY binary once it has been created.

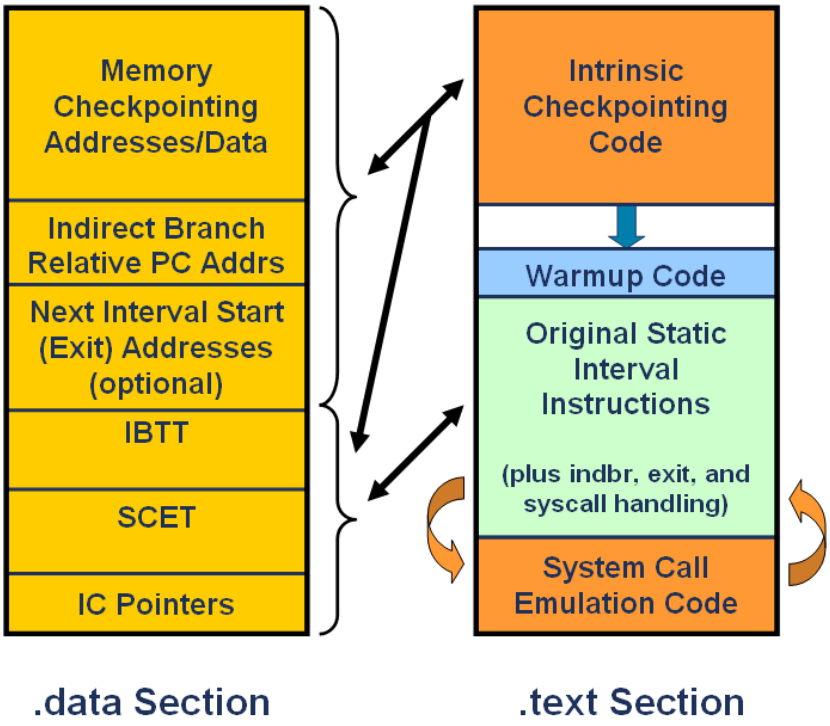

Figure 4.12: Anatomy of an ITCY Binary

4.5 ITCY Code Execution

After the ITCY code is compiled, it can be executed just like any other benchmark. As an additional benefit, input datasets are no longer needed since all system calls are now emulated inside the benchmark and any file input that the original benchmark needed is checkpointed within the state of the ITCY code. On 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 special handling is required.

This special SuiteSpot handling can done in two different ways. The first, and simplest, method adds signal instructions into the ITCY code that have the ability to reset and print the current set of statistics for the simulator. At the boundary of an interval's intrinsic checkpointing and SimPoint interval code, an instruction is inserted to signal the simulator that it needs to reset all of its statistics. Then, at the end of the detailed interval a similar instruction is used to tell the simulator to print its current statistics. This print signal is only inserted if the exit handling technique that uses special termination code is being used. If the technique that counts the exit PC is being used, then at the point when the exit count reaches its appropriate value, the simulator will need to output the current set of statistics prior to moving to the new interval. The multiplication and summation of the SuiteSpot statistics then proceeds as usual.

The second method for handling differently weighted intervals within a SuiteSpot builds upon the first method and actually inserts the functionality directly into the new benchmark, completely removing the need for post-simulation weight handling. Using a procedure identical to that used to load the exit PC of an interval, the weight of the interval can also be loaded into the internal state of the simulator. When the end of the detailed interval is reached, this weight can be accessed and applied to all the current statistics. These internally weighted statistics are then set aside within the simulator for later summation. The procedure of resetting the statistics prior to entering the detailed portion of the interval remains the same. When the final

SuiteSpot interval finishes its execution, its weighted statistics are then added to all the previous interval's statistics to produce a final set of properly weighted and combined statistics. Incidentally, this method of incorporating the weight into the interval can also be used with SuiteSpecks and would remove the need of multiplying the performance metrics by their weight prior to their summation.

4.6 Validation

To verify that the ITCY code is executing the proper instructions from the original SimPoint interval, a trace of the register file can be maintained on a cycle level basis. This trace can then be compared to a similar trace that was created when the ITCY code was first generated. If the traces match, it can be assumed that the code is executing the proper instructions in the proper order. Due to the large amount of file overhead that this technique produces, it was only run on relatively small intervals to stress test the correctness of the ITCY technique. Another more rapid, but slightly less accurate, technique for testing whether an interval executes properly loads a special exit code into the input argument register of the exit syscall just prior to its execution at the end of the benchmark. If the exit syscall is reached, this exit code will be output to the screen. This code can then be combined with the number of executed instructions to verify that the interval executed the expected number of instructions. This technique ensures that the execution of the benchmark follows the correct path to its exit, however, it cannot guarantee that the exact same instructions are executed. Regardless, it does serve as an indicator that the program at least reached its exit properly and since this execution path greatly depends on the proper execution of the interval's instructions, it can be assumed that the benchmark executed the correct instructions.

4.7 Additional Features

The ITCY technique presented in this chapter provides the groundwork for the creation of a variety of different benchmarks that can incorporate many useful features. Many of these features that pertain directly to the ITCY method have already been discussed. However, there are additional features that the technique can provide. These will be discussed in this section.

4.7.1 Intrinsic Warmup

Microarchitectural warmup, as was discussed in section 2.5, is an essential component of any checkpointing methodology. Many techniques require the creation of special data structures that hold large amounts of this warmup information. This large amount of information is needed because different warmup data is needed for different microarchitectural configurations. In addition, simulators must be modified in order to load this information into their microarchitectural components. With the ITCY technique, just like with ICBM, warmup is a straightforward application of MRRL or any other similar technique such as BLRL. Since these reuse latency techniques provide a specific number of instructions that need to be executed prior to the detailed interval, the instructions can simply be included in the ITCY code to effectively warmup the simulator's microarchitectural state provided that statistics gathering is not enabled until after the warmup interval has occurred. Therefore, special warmup data structures are not needed, the simulator or system that runs the benchmark does not need to be modified, and each ITCY benchmark is microarchitecture-independent.

4.7.2 Multi-suite Benchmarks

Another advantage of the ITCY technique is that proprietary benchmarks can easily be distributed by corporations that do not want to release the full version of their software to the public. ITCY benchmarks can be created using these "in-house" programs without the fear of releasing too much information while still providing a useful and representative benchmark. In addition, entire suites of programs could be combined to create single, multi-benchmark binaries that could simulate a large number of applications at once. The procedure for creating a SuiteSpot is general enough that, as long as the state of the machine is restored properly, entire benchmarks can be combined into one SuiteSpot. There are certain limitations when using this approach; however, the ability exists to accomplish this goal.

4.7.3 Fine-grained Statistics Control

A final feature of the ITCY technique makes it possible to insert any number of special purpose instructions into the simulation interval. As an example, fine-grain statistics control instructions can be added to start and stop the logging of statistics. When placed around ITCY checkpointing and control code, these instructions can allow the simulator to control whether or not the ITCY instructions will affect the statistics of the simulation. These are not an all-purpose solution to controlling the effects of the ITCY code on the performance of the original benchmark, and as Chapter VI will show, they are not really necessary. However, they serve as an example of a feature that can easily be added to ITCY code to increase its functionality.

4.8 Summary

This chapter has presented the InTrinsically Checkpointed assemblY (ITCY) technique, a methodology for the creation of fast and flexible benchmarks. While similar to ICBM code in some aspects, ITCY code has several added benefits. The code is highly portable to a variety of simulators and it is microarchitecture-independent with respect to both the simulation interval that it executes and also any warmup that it requires. System calls are removed from the original simulation interval, so the new binary no longer requires a simulator that supports them and, therefore, all input data is embedded within the benchmark. In addition, multiple simulation intervals can be combined into a single benchmark whose performance can represent that of all its component intervals.

CHAPTER V

Experimental Framework

5.1 Target Architecture

The ICBM and ITCY techniques were both tested using the Alpha [13] architecture as a reference. The sim-safe functional simulator from the SimpleScalar version 3.0d [1] simulation infrastructure was modified to do the analysis and code generation for both techniques. Other simple modifications were made to SimpleScalar simulators to allow for the execution of the special control instructions that were created for certain configurations of ITCY code. In particular, special handling needed to be added to allow the simulator to do the following: read the occurrence count and PC address of each interval's exit instruction and exit or transfer control to the next interval when necessary, read the starting addresses of subsequent intervals inside a SuiteSpot and transfer control accordingly, and finally handle any statistics control instructions including the printing and resetting of statistics. Most of these modifications are not necessary if the new ITCY binary handles interval exits by inserting special exit handling code. However, as was discussed previously in section 4.3.6, this exit handling code can have an undesirable impact on the accuracy of the ITCY code if it is executed many times. Therefore, modifying the final detailed simulator to handle annotated ITCY code is the preferred method.

5.2 Code Compilation

The compilation of the generated code from each technique varied. For ICBM code, a special Perl script was written that was given the pseudo-assembly file containing all the intrinsic checkpointing instructions and also the original binary. The original binary was then opened and analyzed to find a location to instert the checkpointing code. Once this location was found, the pseudo-assembly code was parsed, optimized, converted into a set of 32-bit machine instructions, and placed into a .text section of the original binary. In addition, any values needed for memory restoration were written into a .data section. For ITCY code, the majority of this work was already done in the code generation phase, so its Makefile simply needs to be run to produce the new binary or binaries. The actual compilation occurred on a native Alpha compiler since no cross-compiler is easily obtainable that will output binaries for SimpleScalar. This, unfortunately, created a large bottleneck in the entire process since each new ITCY binary needed to be compiled serially on a relatively old and slow machine.

5.3 Benchmarks Used

The benchmarks used for this dissertation were taken from the SPEC CPU2000 benchmarking suite [23]. The ICBM techniques were tested using 19 of the benchmarks with the reference inputs. The actual benchmarks used can be seen in chapter VI. For the ITCY method, all 26 benchmarks with reference inputs were used including a variety of different input data sets. This resulted in a total of 41 different benchmark/dataset pairs.

5.4 SimPoint Intervals

Since the techniques discussed in this dissertation are best used in conjunction with an instruction sampling technique, SimPoint was used to provide the sample intervals since it does well at identifying phases inside a program. For ICBM, a SimPoint interval of 100 million instructions was used for each benchmark since it is only able to handle a single instruction interval at a time. For ITCY, the majority of the results that are presented use thirty, 10 million instruction intervals per benchmark since work done in [28] indicates that smaller and more frequent intervals produce more accurate results. Other studies were done with a variety of different interval sizes to test different aspects of the ITCY technique. These will be described in more detail in chapter VI.

CHAPTER VI

Results and Analysis

This chapter presents the results when testing both the ICBM and ITCY checkpointing methodologies. The results can be broken down into a set of four broad categories: code overhead, performance modeling, effects on file size, and simulation speedup. The ICBM results were first published in [39] and will be expanded upon. The ITCY method will be explored in detail paying particular attention to its effects on code overhead and performance modeling.

6.1 Intrinsic Checkpointing with Binary Modification

This section presents results for the ICBM method when used on 19 of the SPEC CPU2000 benchmarks targeted at the Alpha ISA using the reference input datasets. The exact benchmarks used can be seen in Table 6.1. The simulation interval used was obtained using SimPoint with a specified interval length of 100 million instructions. Since the current ICBM method only creates code for a single interval, SimPoint was limited to only choosing one interval that would attempt to represent the entire benchmark. The processor configuration used for the detailed sim-outorder simulator from the SimpleScalar toolset can be seen in Table 6.2.

Benchmark Name	Input Dataset	Type
ammp	ref	FP
applu	ref	FP
apsi	ref	FP
art	ref: 110	FP
bzip2	ref: source	INT
eon	ref: cook	INT
equake	ref	FP
fma3d	ref	FP
galgel	ref	FP
gap	ref	INT
gcc	ref: 166	INT
gzip	ref: graphic	INT
lucas	ref	FP
mcf	ref	INT
mesa	ref	FP
mgrid	ref	FP
parser	ref	INT
twolf	ref	INT
wupwise	ref	FP

Table 6.1: Benchmarks used for ICBM Results

6.1.1 Code Overhead

Code overhead generated when using the ICBM method can be described in several different ways. The first simply analyzes the generated pseudo-assembly instructions upon their creation and breaks them down into several different categories based on their utility for the upcoming interval. Alternatively, the pseudo-assembly instructions can be converted into their subsequent Alpha instructions and analyzed in a simulator. Both of these methods of analysis will be presented below. The amount of increase in the number of instructions will also be presented since the conversion of the pseudo-assembly will result in a larger number of Alpha instructions.

Pseudo-Assembly Code Overhead

One way of analyzing the code overhead of the ICBM method divides the initial pseudo-assembly intrinsic checkpointing code into four different categories: syscall

Simulator Parameter	Parameter Value
Instruction Fetch Queue Size	4 instructions
Issue/Decode/Commit Width	4 instructions
Branch Predictor	Bimodal, 2048 entries
Return Address Stack Size	8 instructions
Branch Target Buffer	512 sets, 4-way associativity
Out-of-Order Execution	Enabled
Wrong-path Execution	Enabled
Branch Mis-prediction Latency	3 cycles
Register Update Unit Size	16
Load/Store Queue Size	8
L1 I-Cache	512 sets, 32-byte blocks, Direct Mapped
L1 D-Cache	128 sets, 32-byte blocks, 4-way associativity, LRU
L2 Unified Cache	1024 sets, 64-byte blocks, 4-way associativity, LRU
L1 I-Cache Hit Latency	1 cycle
L1 D-Cache Hit Latency	1 cycle
L2 Unified Cache Hit Latency	6 cycles
I-TLB	16 sets, 4096-byte blocks, 4-way associativity, LRU
D-TLB	32 sets, 4096-byte blocks, 4-way associativity, LRU
I/D-TLB Miss Latency	30 cycles
Main Memory Access Latency	18 cycles first chunk, 2 cycles inter-chunk
Memory Access Bus Width	8 bytes
Functional Units	4 integer ALUs, 1 integer multiplier 4 floating point ALUs, 1 floating point multiplier
Memory System Ports	2

Table 6.2: Baseline Configuration for ICBM Results

	SpecInt Avg		SpecFP Avg		Spec2K Avg	
	dowrites	nowrites	dowrites	nowrites	dowrites	nowrites
syscall loads	2,035	884	16,400	9,974	10,351	6,146
syscall stores	37,565	57	6,998	49	19,868	52
interval loads	68	68	68	68	68	68
interval stores	485,417	485,417	1,792,717	1,792,717	1,242,275	1,242,275

Table 6.3: Breakdown of ICBM Pseudo-Assembly Checkpointing Instructions

loads for syscall register restoration, syscall stores for syscall memory restoration, interval loads for main register restoration, and interval stores for main memory restoration. The details of these categories can be found in sections 3.2 through 3.4. The results are further broken down into whether or not the possibly unnecessary file output syscalls encountered prior to the simulation interval are included in the ICBM code, referred to as "dowrites" if file output is included and "nowrites" if not. The syscall data corresponds to the number of loads and stores that must be run

in order for the essential, pre-interval syscalls to provide proper program behavior. The interval data lists how many loads and stores are required to checkpoint the simulation interval. Note the number of loads needed for register restoration of the simulation interval is always constant since there are only a finite number of registers needing restoration. Table 6.3 shows the breakdown with the results averaged for integer, floating point, and all SPEC2000 benchmarks. As the table shows, register restoration for the interval is inconsequential compared to the remainder of the checkpointing code. Syscalls play a larger part in the checkpoint code if file output is included; however, they still only comprise 2.4% of the total checkpointing code on average. In addition, the results for the integer benchmarks have much less checkpointing code due to the fact that they are less focused on memory activities and will, therefore, require less code to restore it.

Alpha Machine Code Overhead

An alternative method for analyzing the ICBM code overheard involves converting the pseudo-assembly into Alpha instructions and then running the code on a simulator. The next set of results displays how many Alpha instructions the ICBM code needs to execute compared to the number of instructions that would need to be executed in both the original simulation interval, and also the fast-forwarding section prior to the interval, if intrinsic checkpointing were not available. As before, the results are broken into whether or not file output is included in the syscall statistics. As Figures 6.1 and 6.2 show, the checkpointing code represents only a small part, roughly 2% on average, of the executed interval code and an extremely small part, less than 0.005%, of the fast-forwarding interval. Therefore, it is expected that the execution of the code with checkpointing included will be much faster than having to fast-forward to the simulation interval itself.

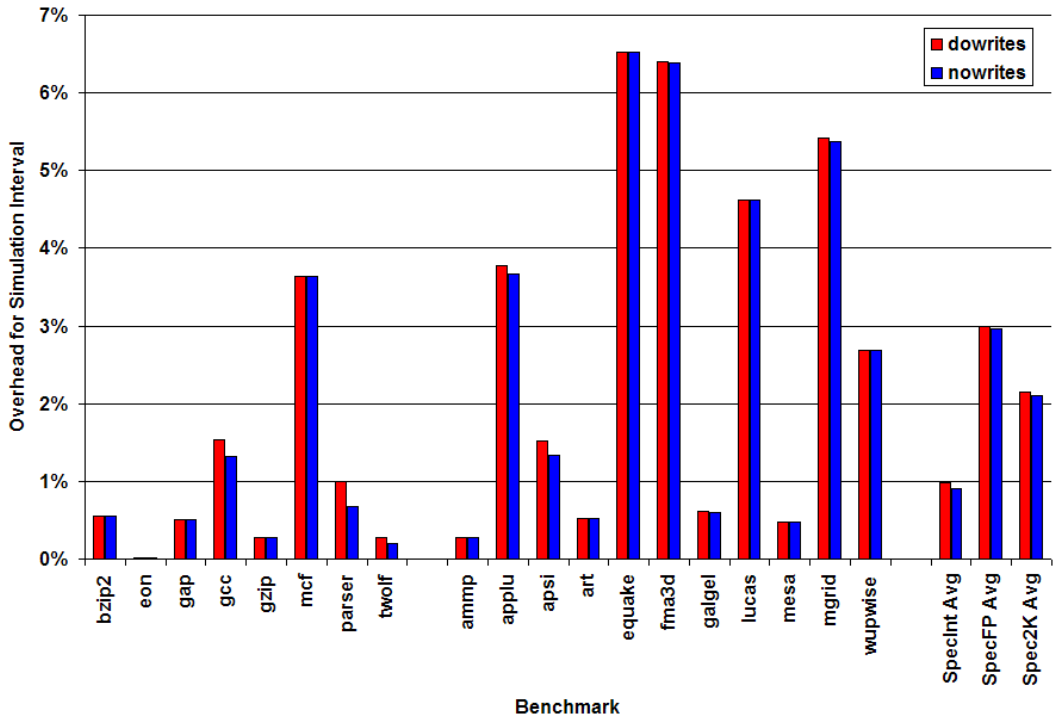

Figure 6.1: Overhead of ICBM Checkpointing Code Compared to Simulation Interval

The conversion of pseudo-assembly into Alpha machine instructions does come at a cost, however. Figure 6.3 shows the increase in the number of instructions when the conversion takes place. As section 3.9 mentioned, anywhere from 1 to 7 Alpha instructions may be needed to represent a single pseudo-assembly instruction, therefore, the number of instructions is expected to increase. However, as the figure shows, there is only a 180% increase on average in the number of instructions which shows that the worst case of 7x instructions rarely occurs and the average conversion takes less than 3 instructions.

6.1.2 Performance Modeling

Since the ICBM technique directly modifies the benchmarking binary to insert its checkpointing code and then executes the original interval of instructions, it does not effect the performance modeling of the new binary with respect to the technique that

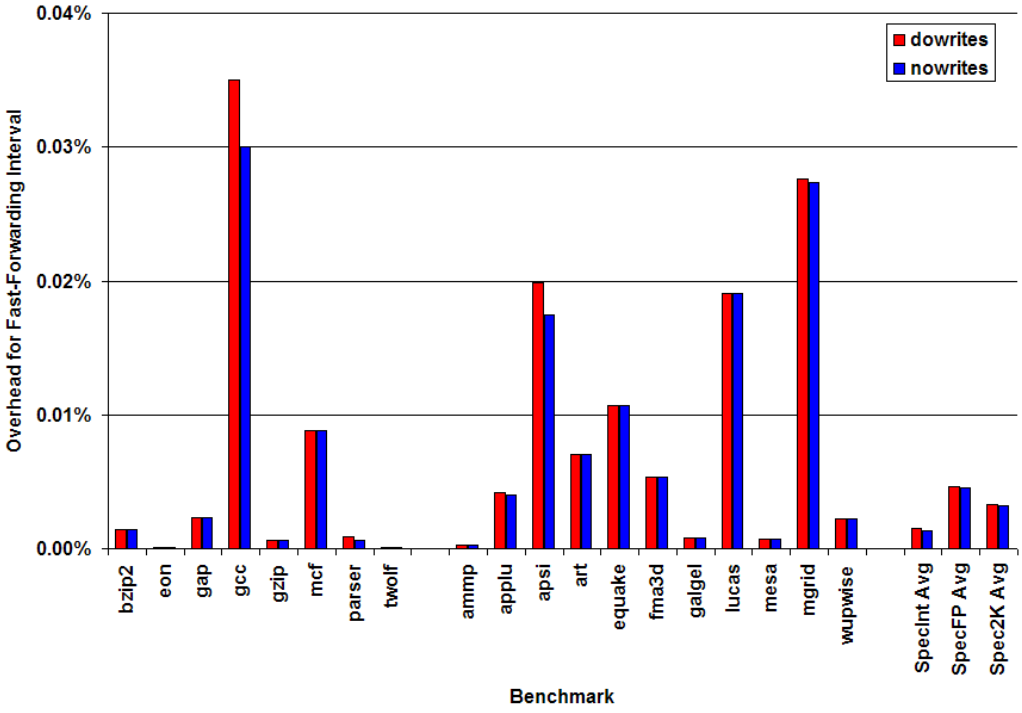

Figure 6.2: Overhead of ICBM Checkpointing Code Compared to Fast-Forward Interval

chooses the simulation interval. All of the original instructions execute in the same order and with the same behavior as if they were reached through fast-forwarding prior to the binary's modification. This was verified by simulating the corresponding instruction intervals from the original binaries and the ICBM binaries in detail and noting that there was no variation in the IPC and similar metrics. Therefore, ICBM binaries are purely at the mercy of the accuracy of the simulation interval selection technique. For the case of a single, 100 million instruction SimPoint interval with no warmup, it was shown that the average percent error for SimPoint was 2.12%. Therefore, it can be expected that this average percent error will also apply to an ICBM benchmark that does not include any warmup code.

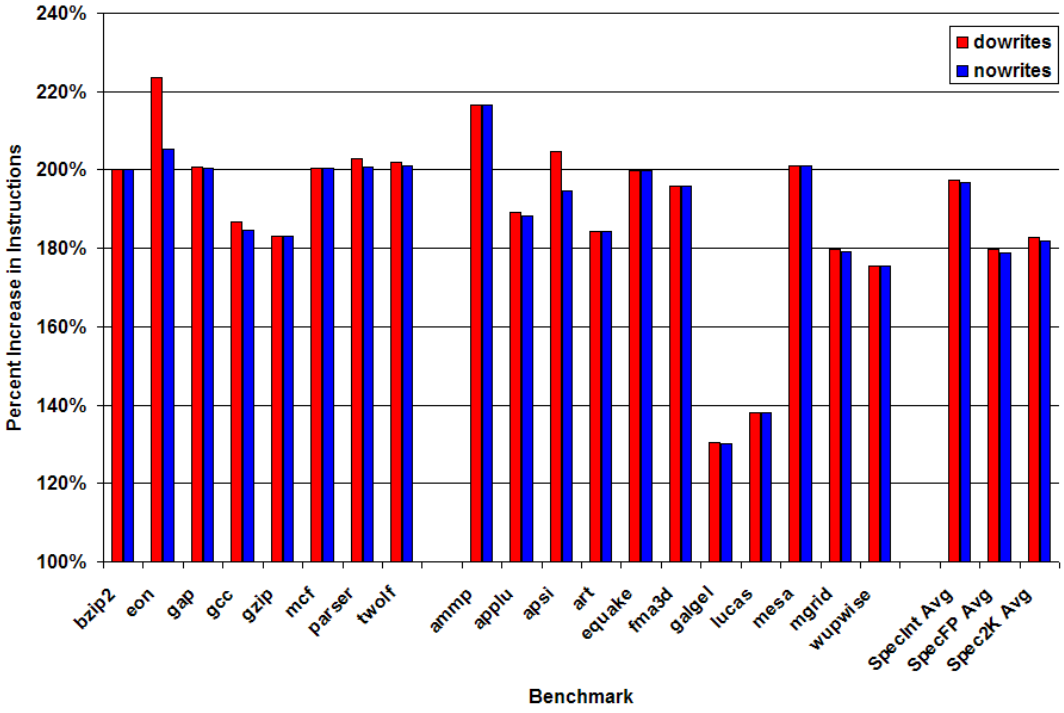

Figure 6.3: Increase in Number of Instructions when Converting to Alpha Code

	Initial Benchmark	dowrites	nowrites
SpecInt Avg	1,120,256	8,551,333	8,075,642
SpecFP Avg	1,506,583	26,719,511	26,502,684
Spec2K Avg	1,343,919	19,069,751	18,743,930

Table 6.4: Increase in ICBM File Size (measured in bytes) for INT, FP, and SPEC2000

6.1.3 Effects on File Size

Table 6.4 and Figure 6.4 show the increase in size measured in bytes of the benchmarking binary when the ICBM code is inserted. Due to the fact that the memory restoration requires 64-bit data both for the values and addresses, it is expected that there will be a fairly significant increase in size since 64-bit values and 64-bit addresses that are not close to each other in the address space must each occupy 8 bytes of code in the data section. In addition, highly data intensive benchmarks such as the scientific computation based floating point benchmarks of SPEC2000 will show a much larger increase in code size, approximately 25 megabytes on average,

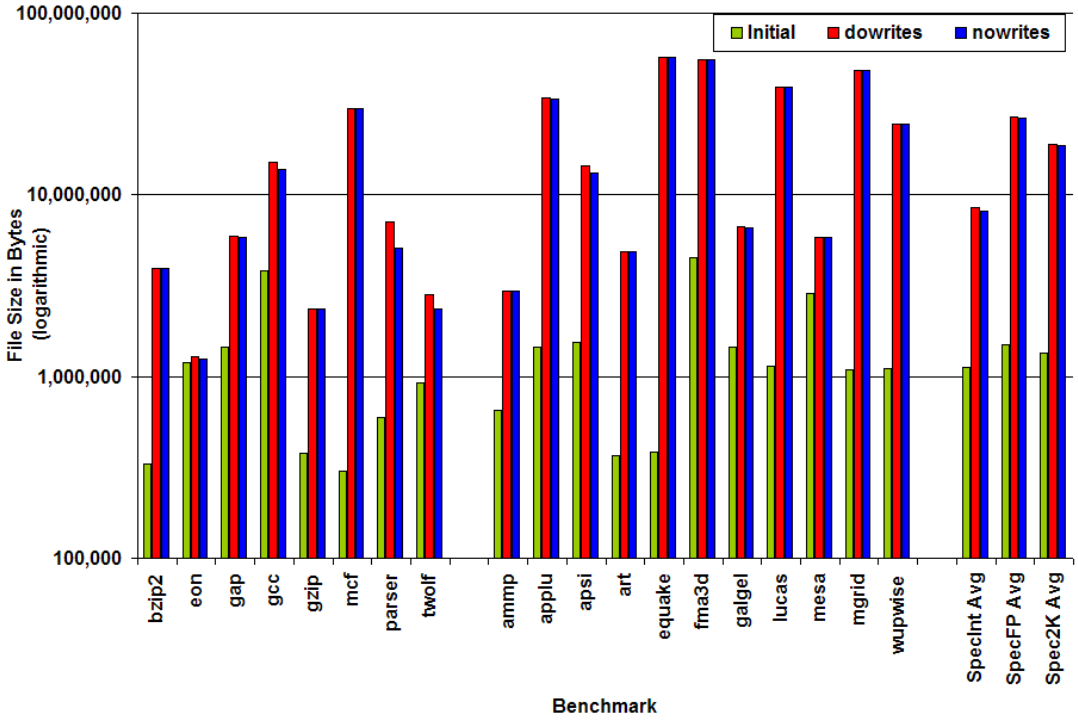

Figure 6.4: Increase in ICBM File Size (measured in bytes) Overall

since it will take more to set up the memory that they will be using for their computations. However, for the integer benchmarks, the increase in size is rather modest, roughly 7 megabytes on average, when compared to other forms of checkpointing that checkpoint the entire system state. In addition, it is always possible to further decrease the final code size by decreasing the size of the simulation interval.

6.1.4 Simulation Speedup

To measure the decrease in simulation time, the original code and the ICBM code were both fast-forwarded to the start of the simulation interval and then the interval was simulated in detail using the default configuration of the SimpleScalar sim-outorder simulator described above. In the case of the original binary, a considerable amount of time was needed for fast-forwarding, whereas for the ICBM binary little time was needed. The performance of the underlying simulator can have an effect,

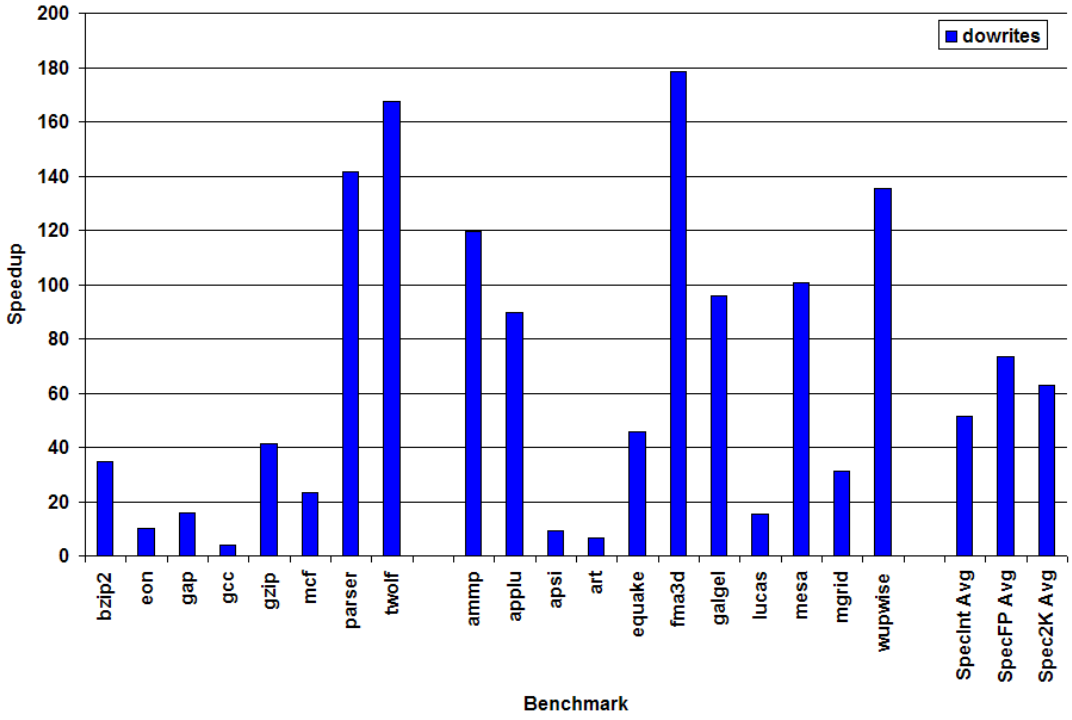

Figure 6.5: Speedup of ICBM Code over Initial Benchmark

however, on the overall speedup of both the ICBM and ITCY speedup results. If the simulator that does the initial fast-forwarding is slow, then the speedup results could be unnecessarily inflated. The SimpleScalar simulator used in this study has a fast-forwarding rate of roughly 4.5 million instructions per second when run on a modern, 3 GHz desktop computer with 1 GB of RAM. A comparable simulator to SimpleScalar that targets the Alpha ISA, M5 [11], has a fast-forwarding rate of approximately 3.5 million instructions. Therefore, it is safe to assume that the SimpleScalar simulator is not incorrectly representing the speedup potential of the ICBM and ITCY techniques. If the simulator being used does not support fast-forwarding, such as with a Verilog or a generic RTL model, the overall speedup would be orders of magnitude better since, as was demonstrated in Figure 6.2, an ICBM binary has drastically fewer instructions to execute prior to the interval.

Figure 6.5 shows the speedup in runtime of the ICBM binary, which includes

pre-interval syscall writes, over the initial binary when run on a Pentium4 2.4GHz+ processor with 1 GB of RAM. The speedup ranges from roughly 5x for gcc up to nearly 180x for fma3d with an average of roughly 60x. The relatively small speedup of gcc is due to the fact that the fast-forwarding interval prior to its simulation interval contains less than 5 billion instructions whereas for fma3d it contains 184 billion instructions. In terms of wall clock time, gcc took 3.7 minutes when using ICBM and 14.5 minutes when not and fma3d took 3 minutes when using it and 8.9 hours when not. On average, the runtime of an ICBM binary was 3 minutes as opposed to 3.13 hours when not. To measure the potential speedup if fast-forwarding is not available, the runtime of each ICBM binary is used as a rough estimate for the amount of time it would take the simulator to simulate in detail 100 million instructions of the benchmark. If this were the case, then the non-ICBM versions of gcc and fma3d would take 2.7 hours and 2.7 days, respectively, to finish the execution of the SimPoint interval. The average time to finish the SimPoint interval for a benchmark if fast-forwarding is not available is 1.5 days and the twolf benchmark would take a maximum of 5.9 days.

6.2 Intrinsically Checkpointed Assembly Code

This section presents results for the ITCY method when used on all 26 of the SPEC CPU2000 benchmarks targeted at the Alpha ISA using the reference input datasets. Several benchmarks contain multiple reference datasets and are treated as separate benchmarks. The combination of benchmarks and the various input datasets resulted in 41 unique benchmark/dataset pairs. The exact benchmarks used can be seen in Table 6.5.

For the majority of the experiments, 30 SimPoint intervals per benchmark were

obtained each with an interval length of 10 million instructions and an associated weight. These intervals were then converted into a set of thirty SuiteSpecks for each benchmark. Next, individual benchmark results were calculated as the weighted average over the 30 SimPoint intervals. Each SuiteSpeck handles its exiting by utilizing the the exit PC counting method described in section 4.3.6 and the simulator fast-forwards through the intrinsic checkpointing code that occurs prior to the start of the interval. The processor configuration used for the detailed sim-outorder simulator from the SimpleScalar toolset was obtained from the SimPoint website and can be seen in Table 6.6. Any changes made to the number of intervals, benchmark set, simulation method, and/or processor configuration will be explicitly noted where appropriate.

Benchmark Name	Input Dataset	Type
ammp	ref	FP
applu	ref	FP
apsi	ref	FP
art	ref: 110 and 470	FP
bzip2	ref: graphic, program, and source	INT
crafty	ref	INT
eon	ref: cook, kajiya, and rushmeier	INT
equake	ref	FP
facerec	ref	FP
fma3d	ref	FP
galgel	ref	FP
gap	ref	INT
gcc	ref: 166, 200, expr, integrate, and scilab	INT
gzip	ref: graphic, log, program, random, and source	INT
lucas	ref	FP
mcf	ref	INT
mesa	ref	FP
mgrid	ref	FP
parser	ref	INT
perlbmk	ref: perfect	INT
sixtrack	ref	FP
swim	ref	FP
twolf	ref	INT
vortex	ref: one, two, and three	INT
vpr	ref: route	INT
wupwise	ref	FP

Table 6.5: Benchmarks used for ITCY Results

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 Pred. Speculative Update	Occurs in Instruction Decode
Return Address Stack Size	64 instructions
Branch Target Buffer	512 sets, 4-way associativity
Out-of-Order Execution	Enabled
Wrong-path Execution	Enabled
Branch Mis-prediction Latency	14 cycles
Register Update Unit Size	128
Load/Store Queue Size	32
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
L1 I-Cache Hit Latency	2 cycles
L1 D-Cache Hit Latency	2 cycles
L2 Unified Cache Hit Latency	20 cycles
I-TLB	32 sets, 4096-byte blocks, 8-way associativity, LRU
D-TLB	32 sets, 4096-byte blocks, 8-way associativity, LRU
I/D-TLB Miss Latency	30 cycles
Main Memory Access Latency	151 cycles first chunk, 2 cycles inter-chunk
Memory Access Bus Width	2 bytes
Functional Units	8 integer ALUs, 2 integer multiplier 2 floating point ALUs, 2 floating point multiplier
Memory System Ports	4

Table 6.6: Baseline Configuration for ITCY Results

6.2.1 Code Overhead

The code overhead for the ITCY method, unlike for ICBM, can be much larger due to the nature of the technique. Since each simulation interval is removed from the original benchmark and converted into intrinsically checkpointed and annotated assembly code, it will need special instructions inserted into the interval to handle indirect branching and exit handling. In addition, in order to remove system calls from the interval, system call emulation code will also need to be inserted into the new binary. 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 itself, 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. This was discussed previously in sections 4.3 and 4.7.

6.2.2 Required Pre-Interval Instructions

For the instructions that occur prior to the interval, they primarily consist of those needed to checkpoint the memory used by the interval's instructions 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 4.3.4 and 4.3.7, set up the exit handling PC counting procedure described in section 4.3.6, and handle any other miscellaneous tasks described in chapter IV.

	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 6.7: Breakdown of ITCY Pre-Interval Instructions

Table 6.7 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 for 10 million instruction intervals. 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 instuctions come from a very small number of static instructions. This indicates that a great deal of looping is occuring inside the code to checkpoint the memory as was described in the latter part of section 4.3.2. In addition, the table shows that there is almost an order of magnitude difference in the number of dynamic memory/register checkpointing instructions needed for the floating point benchmarks. This is due

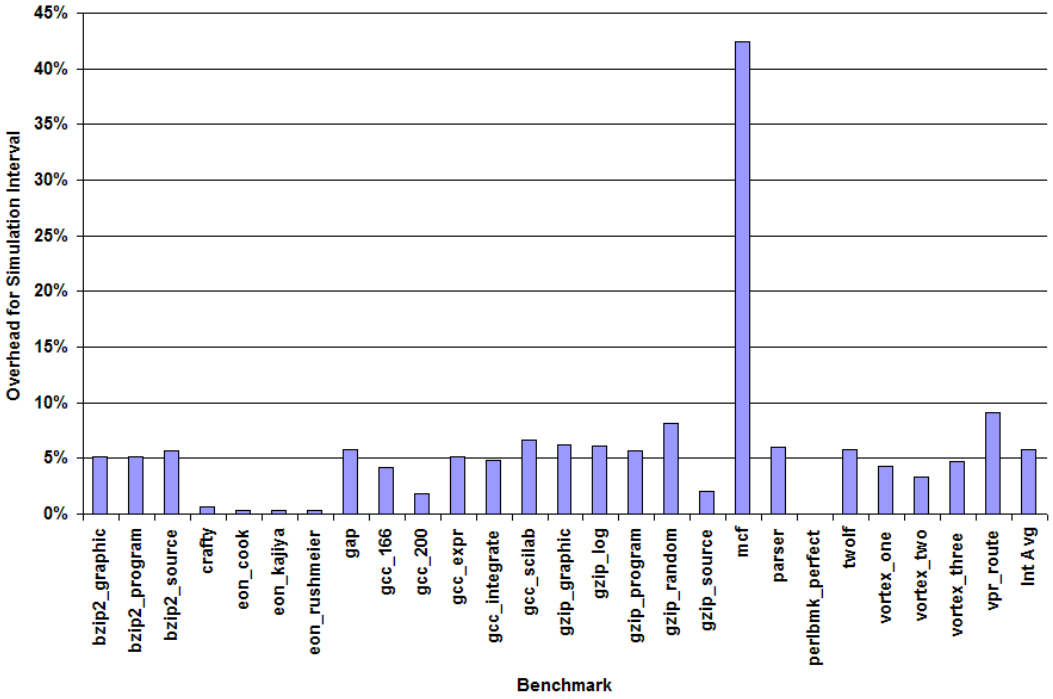

Figure 6.6: Overhead of ITCY Pre-Interval Code Compared to Simulation Interval: INT

to the fact that floating point benchmarks traditionally make much greater use of memory and, therefore, require a great deal more checkpointing. This result is also reflected in Figures 6.6 and 6.7 where it is shown that the floating point benchmarks increase the number of dynamic instructions in the new binary by an average of 40% and the integer benchmarks by only 5%. The overhead for the integer benchmarks would be even lower if not for the inclusion of mcf which increases the overall integer average by over 1%.

When contrasting Figures 6.6 and 6.7 with 6.1, the results for the ITCY technique show a greater number of instructions devoted to pre-interval intrinsic checkpointing. This is expected since the interval size for the ITCY simulations was only 10 million instructions whereas for ICBM it was 100 million. Since a longer interval will need less checkpointing for the instructions that occur later in its execution, the amount of checkpointing overhead compared to the overall interval size will decrease

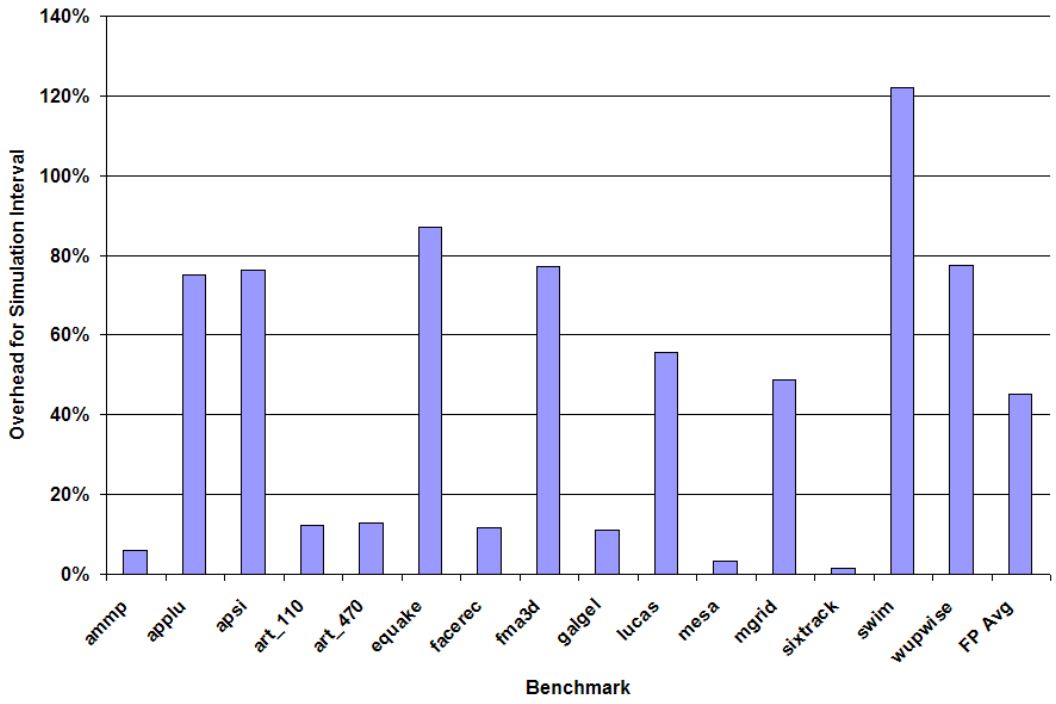

Figure 6.7: Overhead of ITCY Pre-Interval Code Compared to Simulation Interval: FP

as the interval size increases. The particularly large difference in the floating point benchmarks can also be attributed in part to the addition of the swim benchmark which increases the floating point average by over 5%.

Required Data Entries

	SpecInt Avg	SpecFP Avg	Spec2K Avg
Required .data quads	165,444	1,292,521	577,789

Table 6.8: 8-byte .data Entries Needed for ITCY Code

Not only do the floating point benchmarks require a larger number of pre-interval IC instructions, but they also need more entries in the .data section of the new binary. This is reflected in Table 6.8 where the floating point benchmarks again need nearly an order of magnitude more 8-byte .data entries than the integer benchmarks. This size is related to the larger number of IC instructions since these instructions use the values in the .data section to checkpoint the memory of the interval prior to its

execution.

Number of Indirect Branches

	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 6.9: Number of Syscalls and Indirect Branches Seen in ITCY Interval

Another interesting result observed in Table 6.7 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 6.9 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 on average and, therefore, require a larger amount of static instructions in the pre-interval code.

6.2.3 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, unless the simulator is instrumented to turn stats on and off at the instruction level as mentioned in section 4.7.3. 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 as was discussed in sections 4.3.4, 4.3.6, and 4.3.7.

Indirect and Unconditional Branch Handling Overhead

Benchmark	Store	Load	Integer	Benchmark	Store	Load	Integer
bzip2_graphic	0	64,671	64,671	ammp	0	16,211	16,253
bzip2_program	0	56,928	56,928	applu	0	956	1,131
bzip2_source	0	62,020	62,020	apsi	0	8,593	8,593
crafty	0	122,580	122,583	art_110	0	11,223	21,513
eon_cook	0	263,910	474,443	art_470	0	1,454	1,459
eon_kajiya	0	264,948	481,414	equake	0	96,366	99,282
eon_rushmeier	0	267,290	477,865	facerec	0	38,454	38,473
gap	0	261,221	641,047	fma3d	0.17	59,124	61,161
gcc_166	1,914	32,440	33,858	galgel	0	272	272
gcc_200	1,977	76,095	78,421	lucas	0	7,790	7,791
gcc_expr	2,690	82,366	85,930	mesa	0.20	156,901	215,603
gcc_integrate	1,661	49,885	51,459	mgrid	0	158	158
gcc_scilab	1,799	65,327	68,907	sixtrack	0	12,374	12,498
gzip_graphic	0	64,937	65,012	swim	0	156	156
gzip_log	0	25,977	25,996	wupwise	0	31,233	31,233
gzip_program	0	30,073	30,164	FP Avg	0.02	29,418	34,372
gzip_random	0	59,092	59,098				
gzip_source	0	33,250	33,278				
mcf	0	142,343	144,311				
parser	73	194,317	194,340				
perlbmk_perfect	65	325,465	669,244				
twolf	2,559	81,694	83,686				
vortex_one	0	179,677	180,837				
vortex_two	0	191,888	192,957				
vortex_three	0	177,267	178,159				
vpr_route	6	75,953	76,051				
Int Avg	490	125,062	178,180				

Table 6.10: Intra-Interval Indirect Branch ITCY Code Overhead Breakdown

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 6.9. 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. Table 6.10 shows the breakdown of the intra-interval indirect branch handling code into the number of store, load, and integer instructions necessary to implement the techniques discussed in section 4.3.4. The majority of the benchmarks do not require any stores since stores are only necessary if special indirect branch handling is needed that makes use of the current pointer into

Benchmark	Integer	Benchmark	Integer
bzip2_graphic	194,013	ammp	49,147
bzip2_program	170,784	applu	2,555
bzip2_source	186,062	apsi	25,779
crafty	310,841	art_110	13,087
eon_cook	522,282	art_470	4,348
eon_kajiya	510,332	equake	283,969
eon_rushmeier	541,464	facerec	94,126
gap	130,649	fma3d	206,395
gcc_166	66,976	galgel	716
gcc_200	171,978	lucas	23,369
gcc_expr	179,235	mesa	279,431
gcc_integrate	110,131	mgrid	309
gcc_scilab	148,287	sixtrack	25,990
gzip_graphic	194,663	swim	356
gzip_log	77,893	wupwise	93,696
gzip_program	90,037	FP Avg	73,551
gzip_random	177,264		
gzip_source	99,695		
mcf	422,111		
parser	631,623		
perlbmk_perfect	211,040		
twolf	245,773		
vortex_one	600,121		
vortex_two	625,546		
vortex_three	596,532		
vpr_route	233,030		
Int Avg	286,475		

Table 6.11: Intra-Interval Unconditional Branch ITCY Code Overhead

the IBTT. In addition, benchmarks that have roughly the same number of integer and load instructions correspond to those indirect branches that do not write the old return address into a register prior to the execution of the branch. An indirect branch that stores its return address into a register prior to its execution needs to make use of a number of integer instructions in order to load the return register with the expected return address from the old interval. Unconditional branches also must makes use of integer instructions to store their old return addresses into a register when necessary. The number of these instructions can be seen in Table 6.11.

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. In terms of individual dynamic indirect branches, only 2.4 and 2.2 extra instructions are needed to handle the average integer and floating point indirect branch, respectively. The small number of instructions needed per branch indicates that the method of loading in the new target PC by using the old target PC location to hold the address is an effective strategy that leads to a minimal number of instructions being inserted into the interval code.

System Call Emulation Overhead

System call handling, unlike indirect branch handling, contributes very little to the intra-interval code overhead. Since Table 6.9 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 as can be seen in Figures 6.8 and 6.9. Several benchmarks do exhibit a great deal more syscall handling code than the average, however. 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. However, 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. In terms of the amount of code needed per syscall, Table 6.12 shows the average number of instructions needed to emulate a single syscall for both the integer and floating point benchmarks. Overall, an average floating point syscall requires more than double the number of instructions

as an integer syscall. This is consistent with the behaviour seen in section 6.2.2 that showed floating point benchmarks require much more checkpointing information due to the nature of the data that they use.

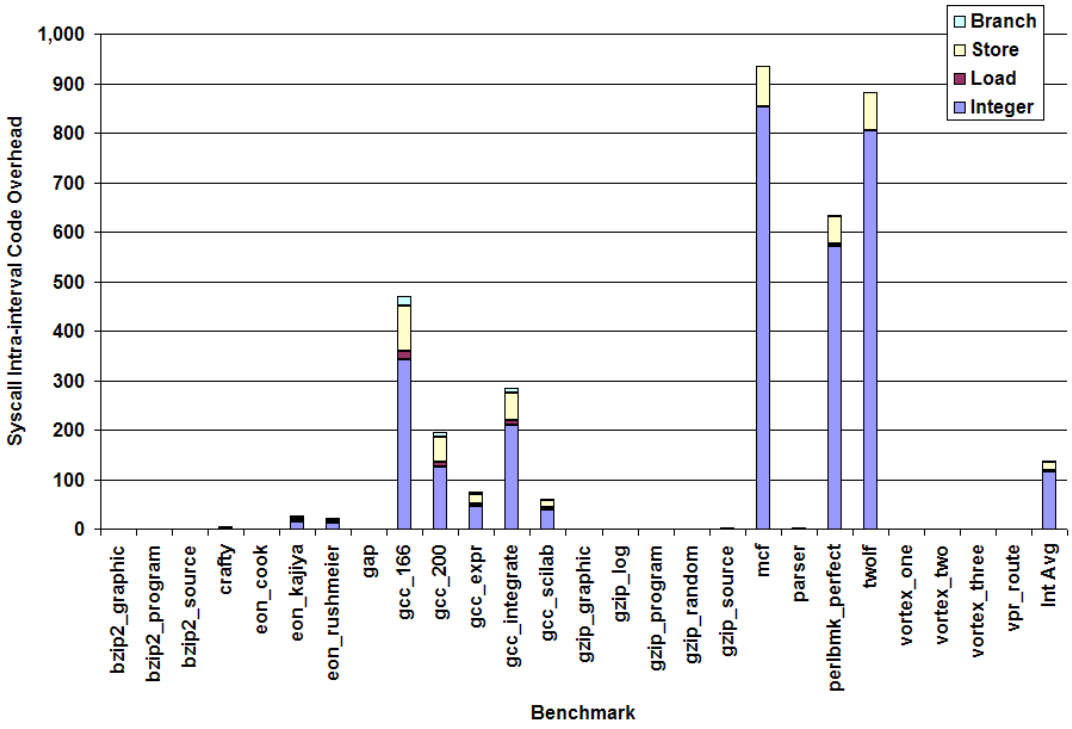

Figure 6.8: Overhead of ITCY Intra-Interval Syscall Emulation Code: INT benchmarks

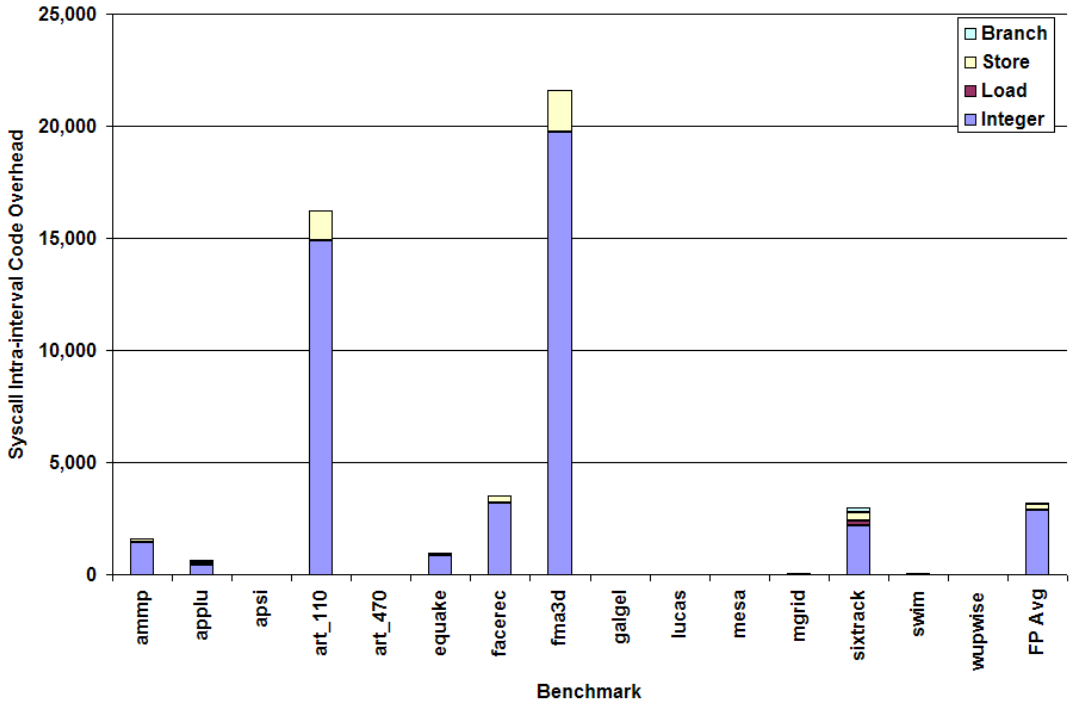

Figure 6.9: Overhead of ITCY Intra-Interval Syscall Emulation Code: FP benchmarks

Benchmark	Size	Benchmark	Size
bzip2_graphic	0	ammp	3,663
bzip2_program	0	applu	18
bzip2_source	0	apsi	0
crafty	11	art_110	15,694
eon_cook	0	art_470	0
eon_kajiya	11	equake	5,607
eon_rushmeier	11	facerec	922
gap	0	fma3d	5,941
gcc_166	54	galgel	0
gcc_200	39	lucas	13
gcc_expr	33	mesa	12
gcc_integrate	58	mgrid	10
gcc_scilab	36	sixtrack	28
gzip_graphic	0	swim	11
gzip_log	0	wupwise	10
gzip_program	0	FP Avg	297
gzip_random	0		
gzip_source	13		
mcf	1,754		
parser	11		
perlbmk_perfect	432		
twolf	3,310		
vortex_one	12		
vortex_two	13		
vortex_three	0		
vpr_route	13		
Int Avg	120		

Table 6.12: Average Syscall Emulation Block Size (in Instructions)

Exit Handling Overhead

Since the current ITCY code experiments use the exit PC counting strategy, they don't contribute any intra-interval code overheard. They do contribute a small amount of pre-interval code, as was described in section 4.3.6. However, once the interval is entered, the simulator is responsible for tracking the exit PC, counting its occurrences, and then exiting when the occurrence count reaches the value that was read at the end of the pre-interval code.

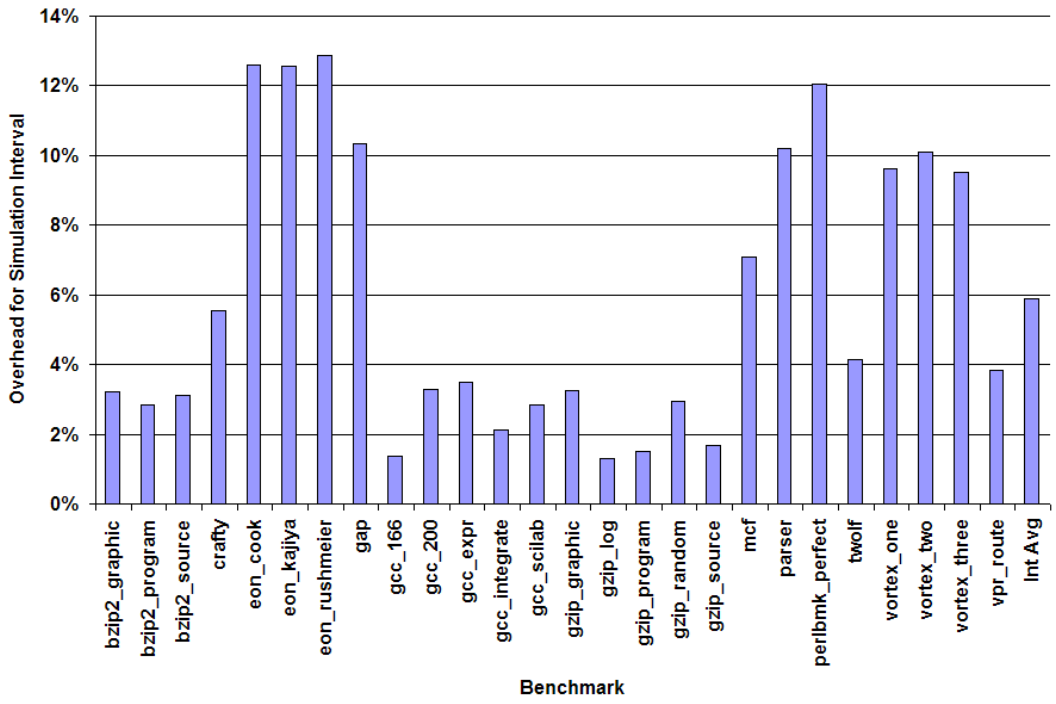

Figure 6.10: Overhead of ITCY Intra-Interval Code Compared to Simulation Interval: INT

Overall Intra-interval Code Overhead

The overall increase in the number of dynamic instructions in the simulation interval due to intra-interval code can be seen in Figures 6.10 and 6.11. As expected, the floating point benchmarks require less code with an average interval size increase of only 1.4%. The integer benchmarks, however, due to their increased number of branches require more intra-interval code and their average interval size increases by roughly 6%.

Intra-interval Instruction Pads

	SpecInt Avg	SpecFP Avg	Spec2K Avg
Required Inst Pads	11,086	3,174	8,191

Table 6.13: Instruction Pads Needed for ITCY Code

A final source of intra-interval code overhead comes from the insertion of static

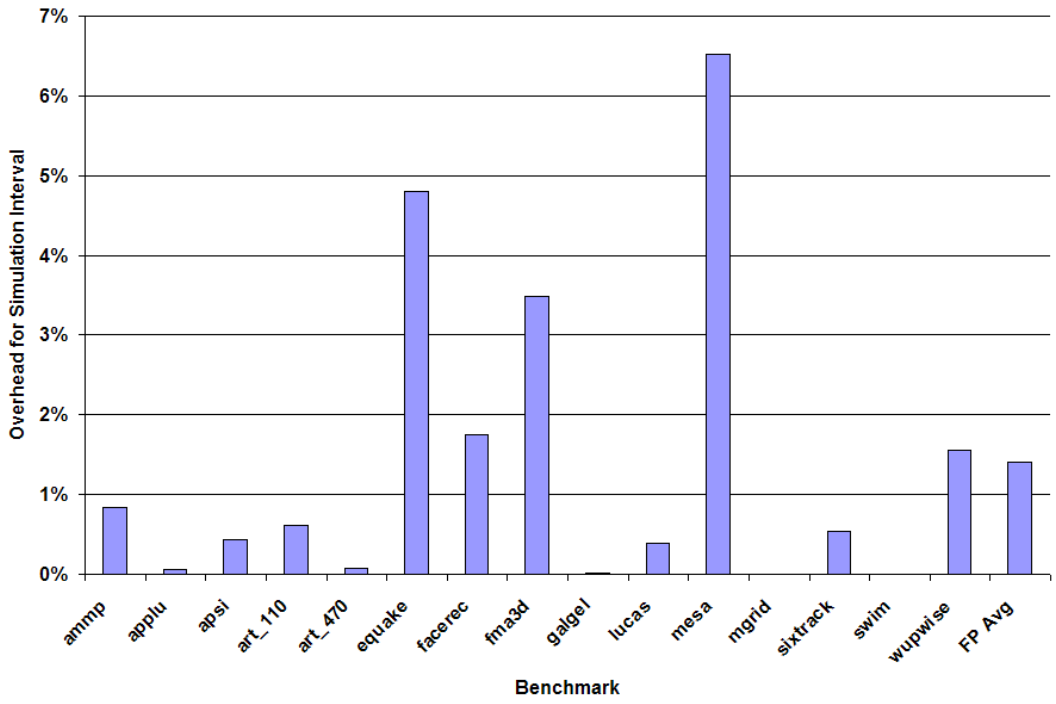

Figure 6.11: Overhead of ITCY Intra-Interval Code Compared to Simulation Interval: FP

instruction pads into the ITCY binary based on the methods described in section 4.3.5. These pads separate blocks of code taken from the original benchmark that were originally not located next to each other in the binary's address space. Depending on the dynamic execution of the interval, the blocks that were originally far apart may end up adjacent to each other when they are inserted into the new ITCY binary since only executed static instructions are output to the ITCY code. Fortunately, the code inside of these instruction pads is never executed because it surrounds basic blocks that transfer control away from a pad prior to reaching it. Table 6.13 lists the average number of instruction pads needed for the integer, floating point, and overall SPEC2000 benchmarks. The floating point benchmarks use less than a third of the number of pads that the integer benchmarks do. Since floating point benchmarks traditionally consist of a large amount of looping code, their static code footprint within an interval may be smaller. Thus, they will use less instruction padding to

separate out basic blocks that do not belong next to each other in the new ITCY binary. Since Alpha instructions are 4 bytes in length, the insertion of instruction pads increases an integer benchmark's size by roughly 44 KB on average and a floating point benchmark's size by 12 KB on average. Compared to the file size increases seen in sections 6.1.3 and 6.2.8, these numbers are quite small.

6.2.4 Cache Performance Modeling

Performance modeling using ITCY code, unlike with ICBM, 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 4.3.3 and 4.3.5. This section will detail the effects that ITCY code can have on various microarchitectural metrics, namely cache performance, branch prediction accuracy, and CPI.

I-Cache Effects

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. The previously discussed technique of inserting instruction pads into the interval to simulate the spacial 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, extra misses can greatly effect the relative error rate of the overall miss rate. These effects are reflected in Tables 6.14 and 6.15 especially for the benchmarks bzip2_source, ammp, facerec, and wupwise. It should be noted that prior to the addition of instruction pads into the new ITCY binary, the relative error rates for I-cache misses were orders of magnitude

greater since nearly all the ITCY instructions fit inside the I-cache. This lead to a very small number of misses for all benchmarks. Therefore, the instruction pads are an effective method for ensuring more realistic I-cache performance, however, certain situations can still result in high relative error rates.

Benchmark	I-Cache Miss Rate		Percent
	ITCY	Baseline	Relative Error
bzip2_graphic	0.000	0.000	5.38%
bzip2_program	0.000	0.000	4.38%
bzip2_source	0.000	0.000	595.10%
crafty	0.052	0.051	2.07%
eon_cook	0.032	0.031	3.83%
eon_kajiya	0.026	0.026	0.03%
eon_rushmeier	0.026	0.024	7.51%
gap	0.016	0.018	13.69%
gcc_166	0.008	0.007	10.41%
gcc_200	0.018	0.018	4.83%
gcc_expr	0.022	0.020	7.84%
gcc_integrate	0.014	0.013	7.90%
gcc_scilab	0.024	0.023	5.62%
gzip_graphic	0.000	0.000	5.50%
gzip_log	0.000	0.000	4.04%
gzip_program	0.000	0.000	6.00%
gzip_random	0.000	0.000	7.07%
gzip_source	0.000	0.000	2.86%
mcf	0.000	0.000	44.01%
parser	0.003	0.005	40.57%
perlbmk_perfect	0.002	0.007	67.85%
twolf	0.016	0.017	1.21%
vortex_one	0.036	0.026	39.54%
vortex_three	0.038	0.026	47.70%
vortex_two	0.040	0.032	22.83%
vpr_route	0.000	0.000	16.54%
Int Avg			37.47%
Int Avg (w/o bzip2_source)			15.17%
ammp	0.000	0.000	93.65%
applu	0.008	0.008	0.00%
apsi	0.029	0.029	0.31%
art_110	0.000	0.000	0.85%
art_470	0.000	0.000	2.03%
equake	0.001	0.001	18.92%
facerec	0.000	0.000	128.16%
fma3d	0.058	0.059	1.84%
galgel	0.000	0.000	94.84%
lucas	0.000	0.000	4.29%
mesa	0.033	0.024	39.52%
mgrid	0.011	0.007	45.74%
sixtrack	0.008	0.008	7.61%
swim	0.001	0.001	9.02%
wupwise	0.006	0.002	235.48%
FP Avg			45.49%
FP Avg (w/o wupwise)			31.91%

Table 6.14: I-Cache Miss Rate Performance of ITCY Code Compared to Baseline

Benchmark	I-Cache Accesses			I-Cache Misses		
	ITCY	Baseline	% RelError	ITCY	Baseline	% RelError
bzip2_graphic	16,532,783	16,034,716	3.11%	72	66	8.65%
bzip2_program	18,037,513	17,483,965	3.17%	82	76	7.69%
bzip2_source	19,245,317	18,552,882	3.73%	646	90	621.04%
crafty	16,331,032	15,611,439	4.61%	845,976	792,276	6.78%
eon_cook	16,474,037	14,947,079	10.22%	531,829	464,723	14.44%
eon_kajiya	21,548,352	19,407,674	11.03%	556,218	501,124	10.99%
eon_rushmeier	19,275,712	17,470,313	10.33%	494,885	417,191	18.62%
gap	19,581,036	18,344,298	6.74%	310,545	337,071	7.87%
gcc_166	11,745,036	11,564,597	1.56%	97,220	86,698	12.14%
gcc_200	15,280,134	14,627,103	4.46%	280,535	256,162	9.52%
gcc_expr	14,424,118	13,993,886	3.07%	317,697	285,809	11.16%
gcc_integrate	13,047,768	12,708,024	2.67%	178,855	161,446	10.78%
gcc_scilab	15,207,028	14,581,659	4.29%	364,922	331,284	10.15%
gzip_graphic	19,058,912	18,297,197	4.16%	514	468	9.89%
gzip_log	16,277,334	16,379,361	0.62%	325	341	4.64%
gzip_program	18,658,446	18,856,409	1.05%	276	263	4.89%
gzip_random	17,175,312	16,407,557	4.68%	348	311	12.08%
gzip_source	18,579,310	18,780,793	1.07%	282	277	1.75%
mcf	22,246,846	21,082,706	5.52%	6,547	4,308	51.96%
parser	20,064,486	18,469,452	8.64%	54,547	84,486	35.44%
perlbmk_perfect	26,856,371	19,394,387	38.47%	59,604	133,878	55.48%
twolf	20,280,900	19,089,435	6.24%	334,014	318,254	4.95%
vortex_one	11,755,339	10,650,926	10.37%	427,240	277,404	54.01%
vortex_two	12,075,299	10,861,731	11.17%	481,056	352,279	36.56%
vortex_three	11,763,010	10,636,655	10.59%	450,550	275,827	63.35%
vpr_route	15,518,110	14,980,535	3.59%	4,349	3,602	20.72%
Int Avg			6.74%			42.52%
ammp	12,753,883	12,717,078	0.29%	376	5,903	93.63%
applu	11,106,712	11,106,088	0.01%	93,745	93,735	0.01%
apsi	10,615,232	10,567,809	0.45%	307,664	307,252	0.13%
art_110	11,962,854	12,017,730	0.46%	378	377	0.39%
art_470	12,058,421	12,065,938	0.06%	410	402	1.97%
equake	10,829,118	10,352,651	4.60%	11,402	9,166	24.40%
facerec	11,144,164	10,969,937	1.59%	5,471	2,360	131.78%
fma3d	11,345,901	11,040,351	2.77%	654,297	648,601	0.88%
galgel	10,147,794	10,146,161	0.02%	437	224	94.87%
lucas	10,013,683	10,003,102	0.11%	51	49	4.40%
mesa	12,815,195	11,895,370	7.73%	422,364	280,994	50.31%
mgrid	14,946,643	14,604,747	2.34%	158,158	106,040	49.15%
sixtrack	10,412,433	10,391,348	0.20%	78,190	84,458	7.42%
swim	11,030,151	11,027,323	0.03%	7,889	8,669	9.00%
wupwise	10,885,351	11,081,405	1.77%	67,315	20,426	229.55%
FP Avg			1.49%			46.53%

Table 6.15: I-Cache Accesses and Misses of ITCY Code Compared to Baseline

D-Cache Effects

D-cache effects are 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, however, the instructions that are inserted to handle indirect branch targets access the original interval's address space in order to load in the new 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 effect the D-cache performance of the ITCY code. As can be seen in Figures 6.12 and 6.13, their effects are minimal in most situations. For those situations were there is a high relative error for the D-cache miss rate, this is attributable, based on the results seen in Table 6.10, to the increased number of indirect branch handling instructions from the intra-interval code. The integer benchmarks, especially eon and perlbmk, clearly show this sensitivity to the number of inserted instructions.

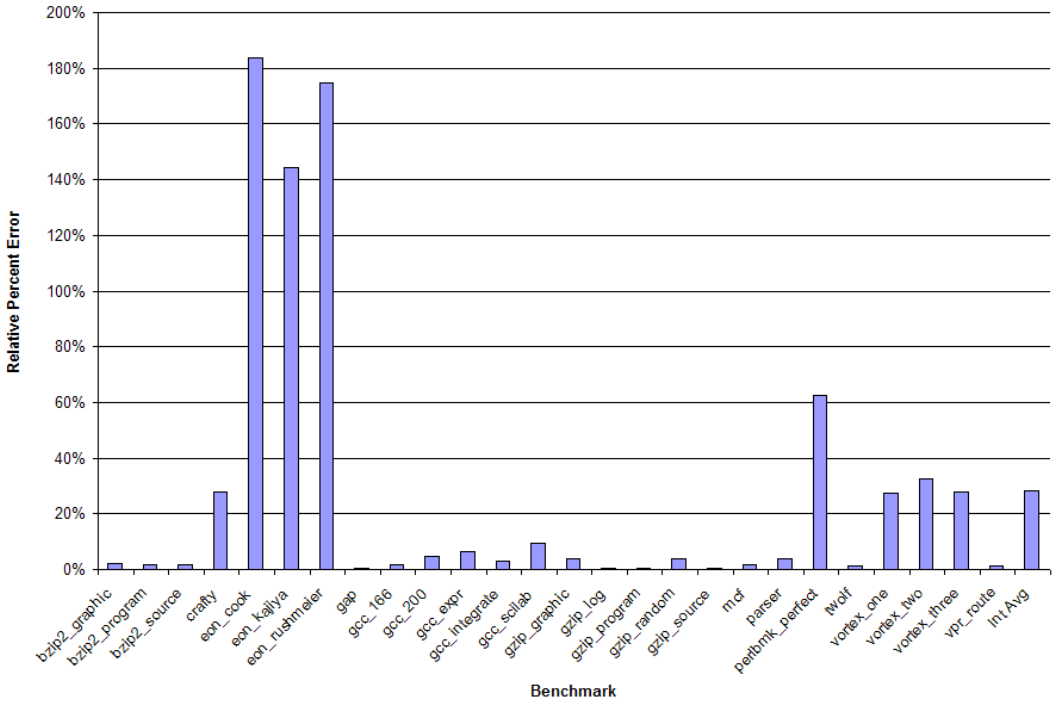

Figure 6.12: L1 D-Cache Miss Rate of ITCY Code - Integer Benchmarks

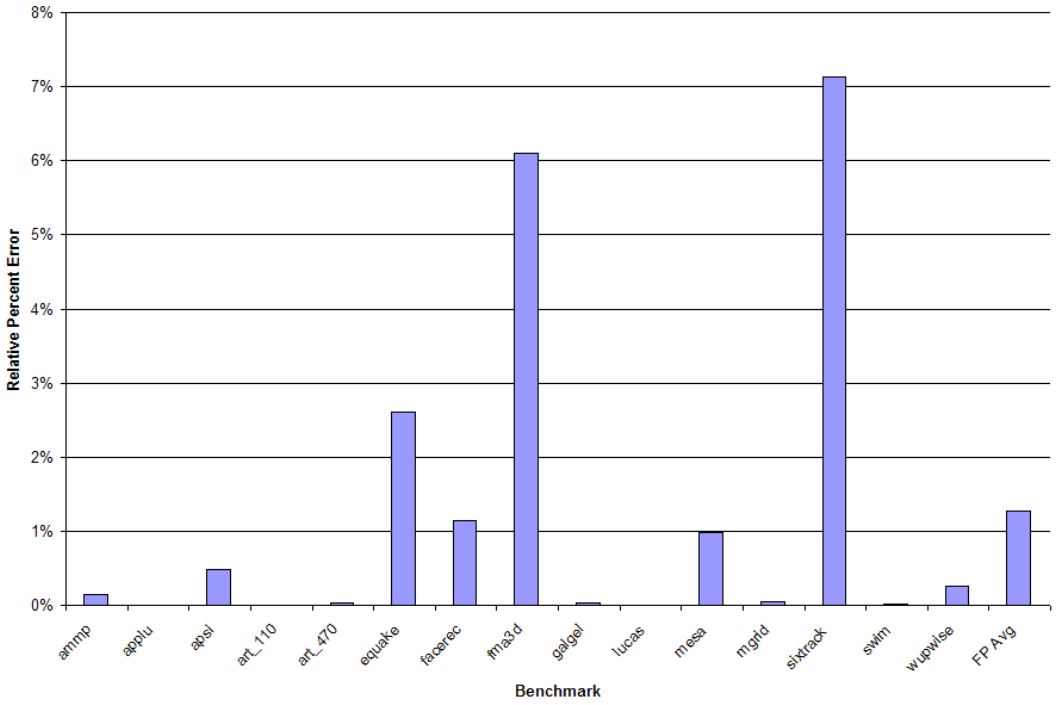

Figure 6.13: L1 D-Cache Miss Rate of ITCY Code - Floating Point Benchmarks

L2 Cache Effects

The effects of the ITCY code on the L2-cache can be seen in Figures 6.14 and 6.15. The effects on the integer benchmarks, with the exception of perlbmk, are quite small. Incidentally, if perlbmk were removed from the average, the average relative error for the L2-cache performance on the integer benchmarks would drop from 16.7% to 6%. The poor relative error rate for perlbmk is attributable to the combination of the ITCY code's effects on both the I-cache and the D-cache where perlbmk performs quite poorly for both. The floating point benchmarks, again with a few exceptions, also show a low average relative error, however, it is greater than its average error for D-cache performance. This is due to the effects of its high average error rate with respect to the I-cache.

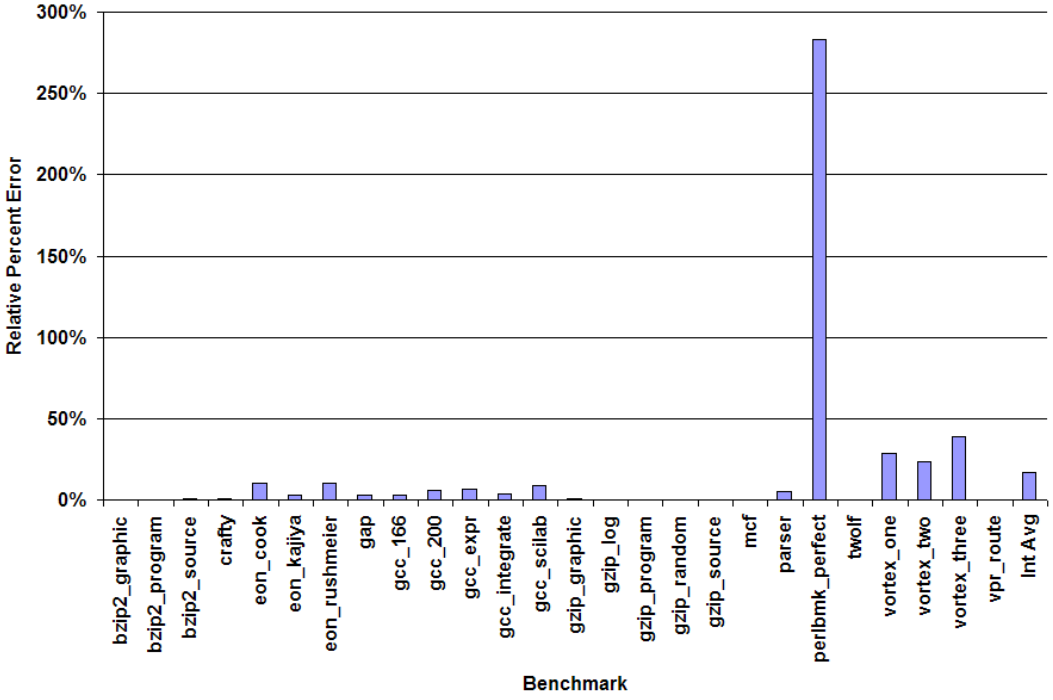

Figure 6.14: L2 Cache Miss Rate of ITCY Code - Integer Benchmarks

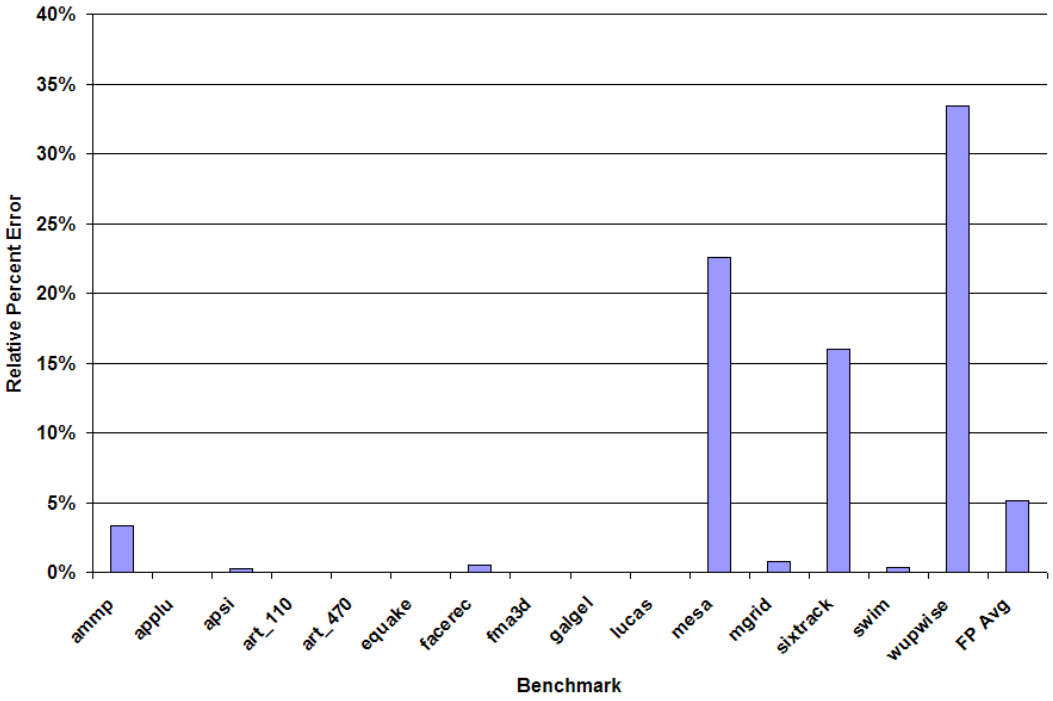

Figure 6.15: L2 Cache Miss Rate of ITCY Code - Floating Point Benchmarks

6.2.5 Branch Prediction Performance Modeling

Unlike the effects on cache performance, branch prediction performance for ITCY benchmarks remains 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. Figures 6.16 through 6.19 show that, for both the direction and the address prediction rates, there is a change of less than a 1% for all benchmarks and 0.2% on average. This indicates that the branch prediction behavior of the ITCY code is nearly identical to that of the original interval.

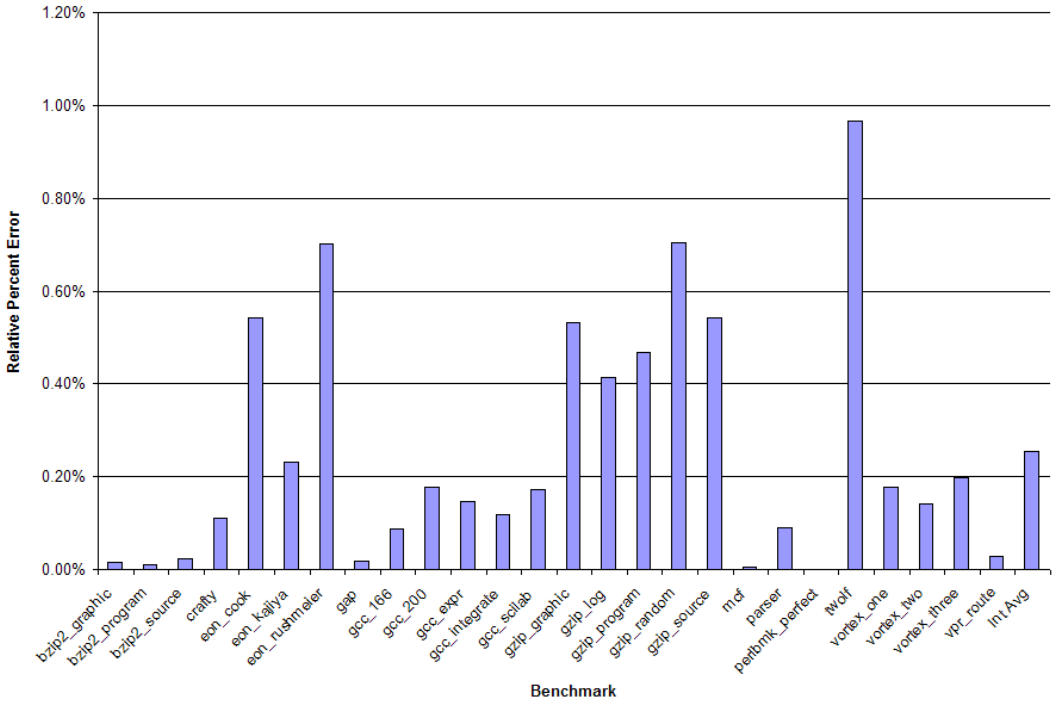

Figure 6.16: Branch Direction Prediction Rate of ITCY Code - Integer Benchmarks

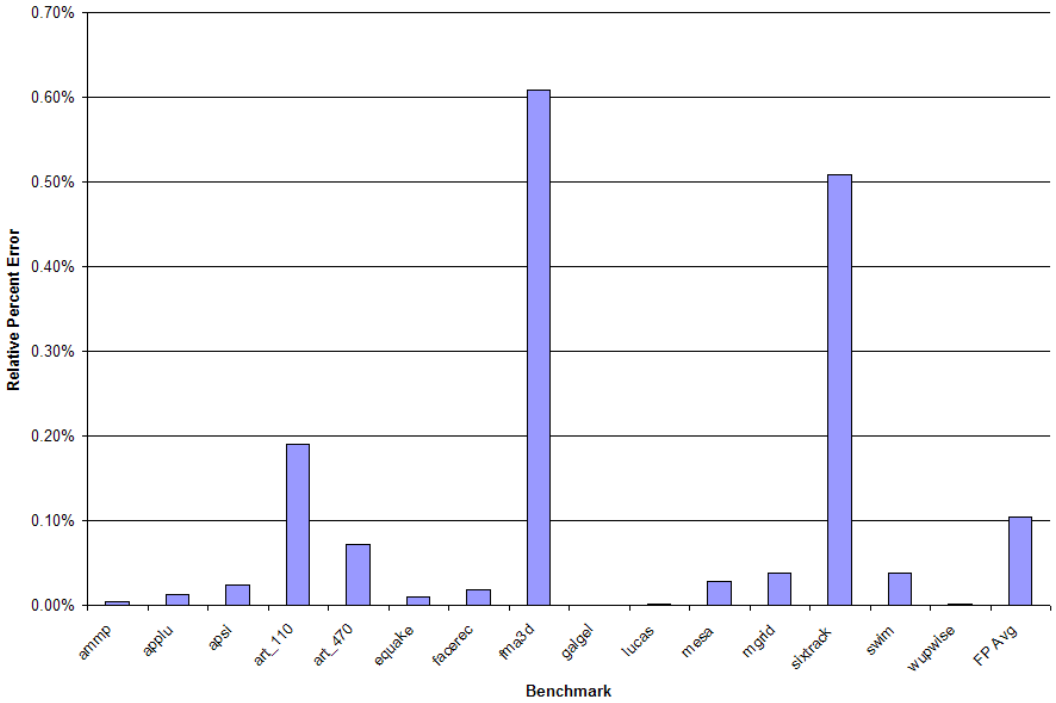

Figure 6.17: Branch Direction Prediction Rate of ITCY Code - Floating Point Benchmarks

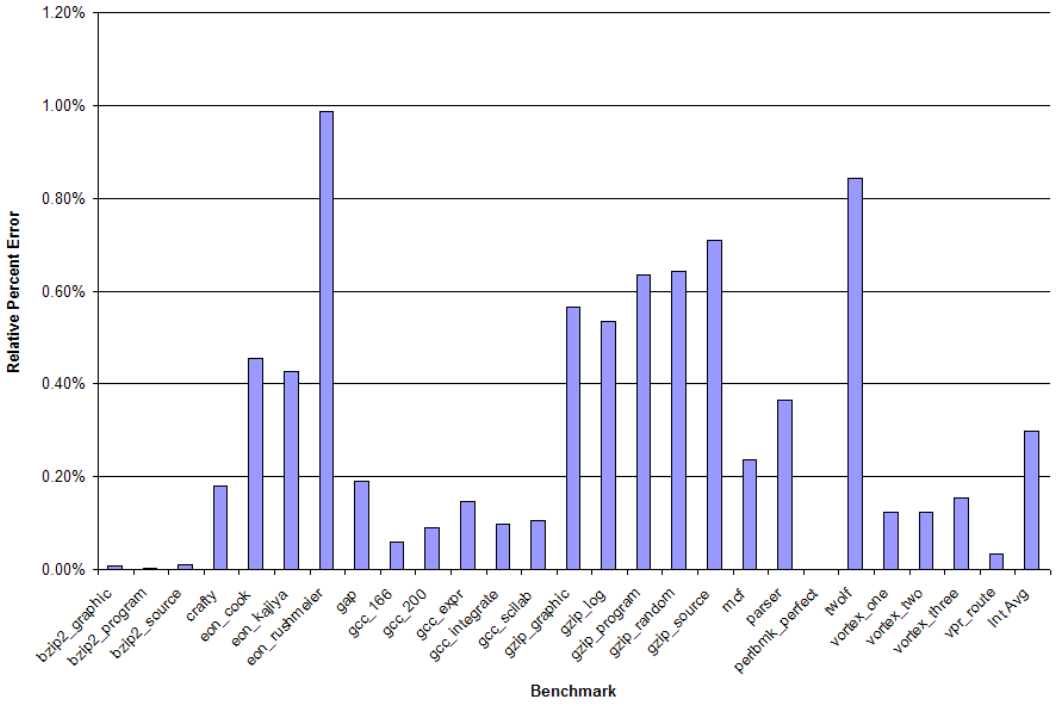

Figure 6.18: Branch Address Prediction Rate of ITCY Code - Integer Benchmarks

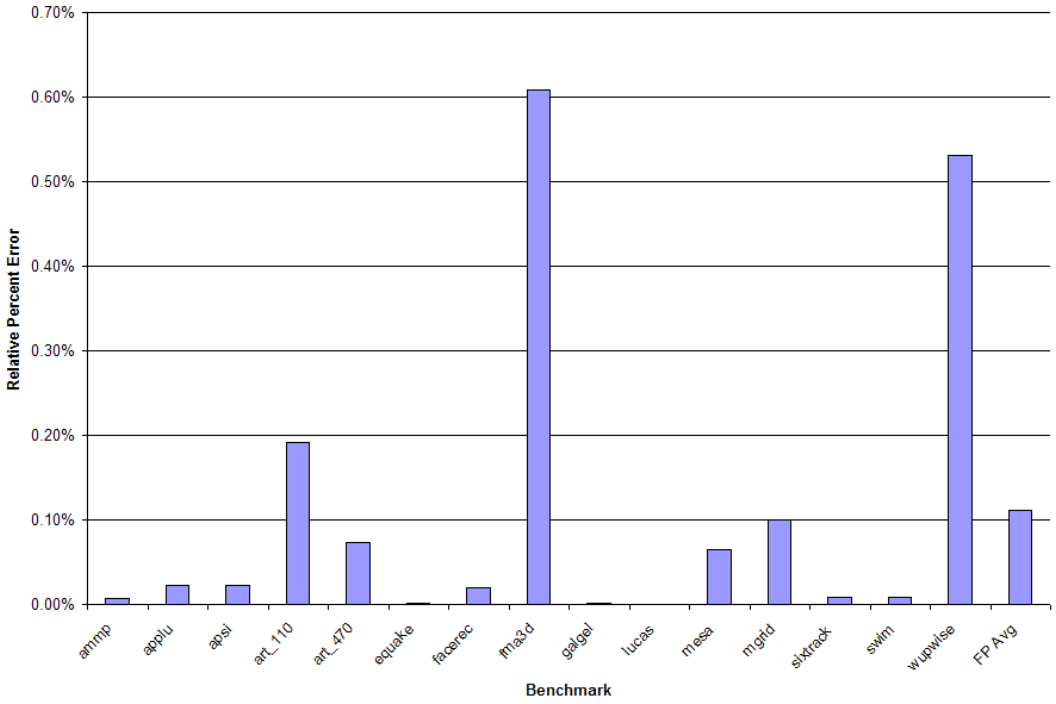

Figure 6.19: Branch Address Prediction Rate of ITCY Code - Floating Point Benchmarks

6.2.6 CPI and IPC Performance Modeling

The next measurement of the ITCY methods's ability to model the performance of the original benchmark is done by quantifying its effects on CPI. Since CPI is affected by many different microarchitectural elements, the accuracy of its prediction serves as a good indicator of whether the ITCY method is a useful tool for predicting the performance of a system. From the different studies done above, it is expected that the effects of the D-cache will have the most significant impact on the performance estimates of CPI since the branch prediction was nearly unchanged and the I-cache performance did not include enough misses to dramatically affect the overall performance of the system. Other factors that can contribute to the performance of the system such as functional unit usage, should remain unchanged due to the fact that the vast majority of the instructions that comprise ITCY code are those that originate from the initial interval. Figures 6.20 and 6.21 confirm these expectations and show that the average CPI for all benchmarks is within 5% relative error. The integer benchmarks do show a slightly increased relative error and this is more that likely attributable to their increase relative error with respect to D-cache performance.

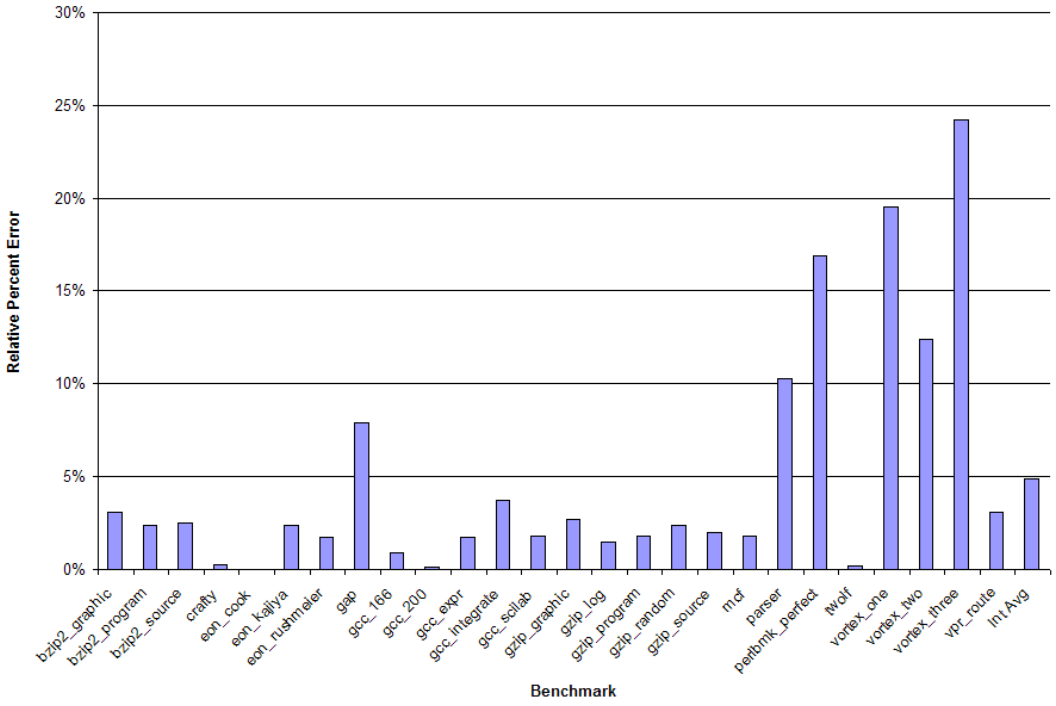

Figure 6.20: CPI of ITCY Code - Integer Benchmarks

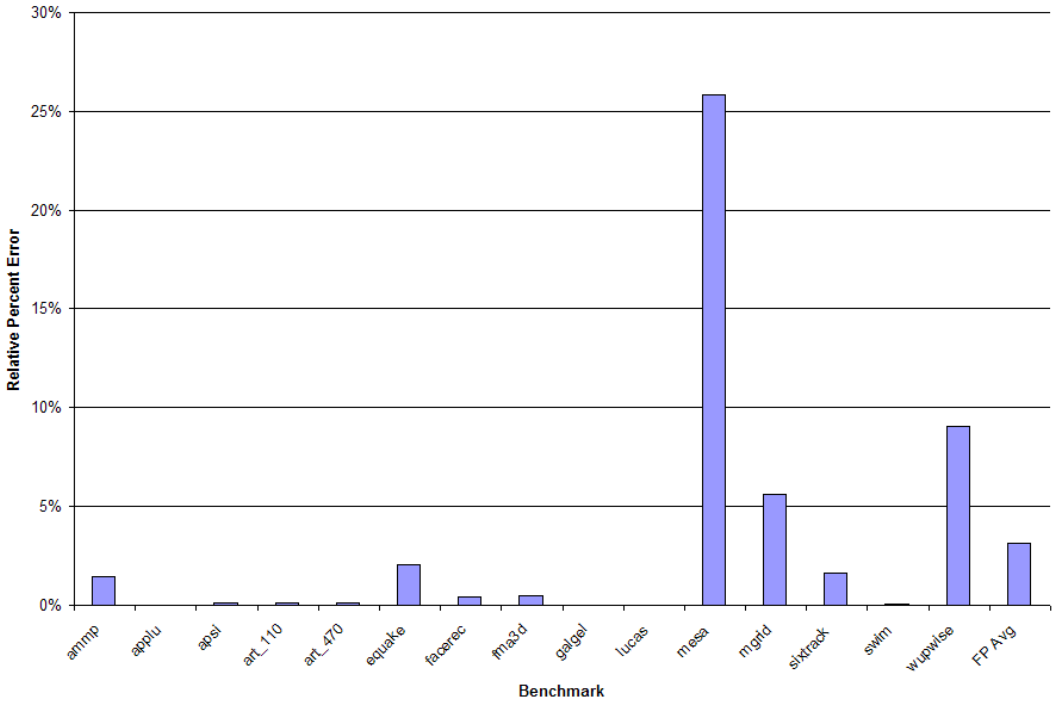

Figure 6.21: CPI of ITCY Code - Floating Point Benchmarks

A final measurement of the effects of ITCY code is done on by comparing against the IPC of a full detailed simulation of the SPEC2000 benchmarks. The IPC results were taken from the SimPoint website and were done using the exact same baseline configuration. Based on the results in figures 6.22 and 6.23, it is clear that the ITCY code still has a reasonable average error rate with respect to the IPC of a full detailed simulation. The integer benchmarks are within 7.5% average relative error and the floating point benchmarks within 6%.

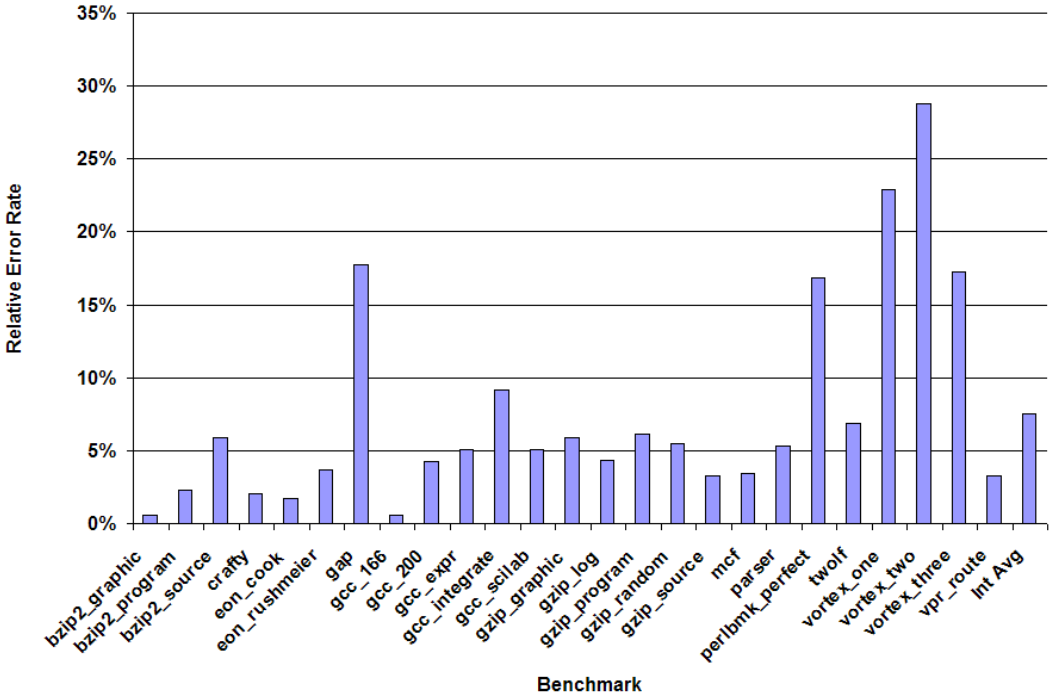

Figure 6.22: IPC of ITCY Code Compared to Detailed Simulation - Integer Benchmarks

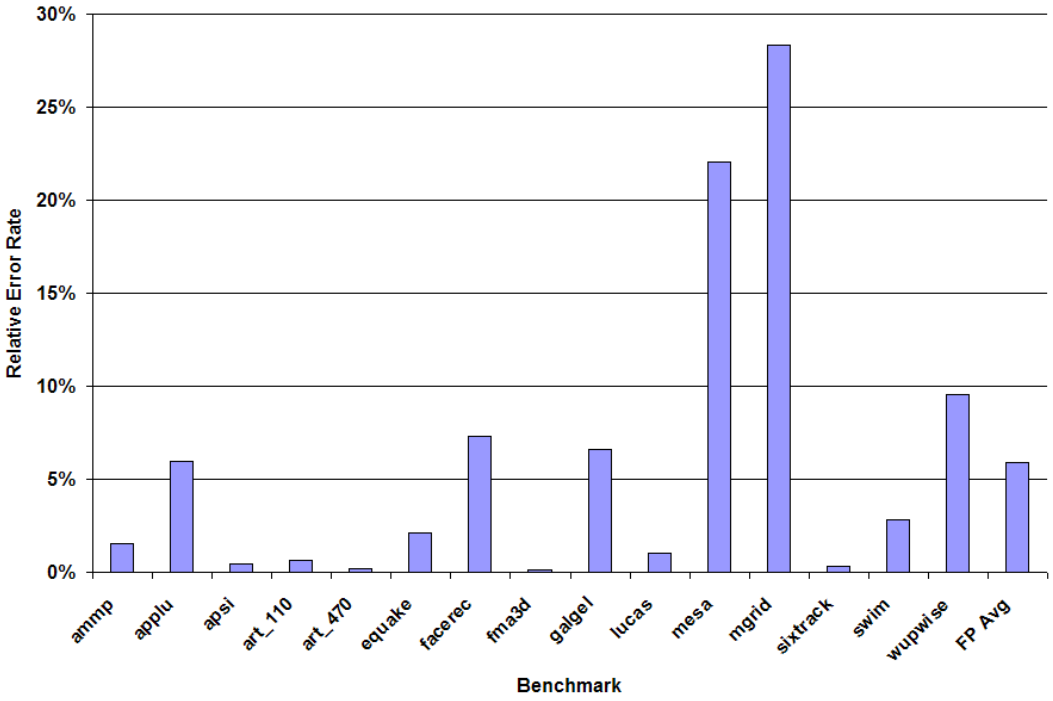

Figure 6.23: IPC of ITCY Code Compared to Detailed Simulation - Floating Point Benchmarks

6.2.7 Tracking Design Changes

Although the ITCY technique does have the ability to predict performance quite well on average, it did not predict the CPI for several benchmarks very well. For those benchmarks where ITCY fails to properly predict absolute performance, it can still prove useful when predicting the change in performance when a change is made to the underlying microarchitecture. For instance, if the user wanted to see the effects on performance of increasing the size of the I-cache, the relative change in performance from the original configuration to the new configuration is a more important metric than the absolute performance accuracy. If the original benchmark showed a decrease in CPI of a certain value and the ITCY benchmark showed roughly the same change, even though the ITCY binary did not predict the same absolute performance, the results will still be useful. This type of change tracking study was done in [10] and showed that while a technique may not predict the absolute performance of a benchmark properly, it can still be effective when used to measure the effects of changing an underlying microarchitectural component.

To test the usefulness of using ITCY binaries to do change tracking analysis, the two worst performing benchmarks were selected from both the integer and the floating point benchmarks. Changes were then made to the cache sizes and pipeline widths of the baseline configuration to measure the relative change in CPI. Both the I-cache and the D-cache had their sizes increased and decreased by a factor of two and the issue/commit/decode (ICD) widths of the pipeline were also increased and decreased by two. As Figure 6.24 shows, the ITCY binaries were able to track the change in performance even though they did not accurately predict the absolute performance. In fact, mesa, which showed a 26% relative error when predicting CPI in section 6.2.6, was still able to effectively predict the change in CPI when the

various design changes were made. Therefore, even though some ITCY binaries may not be useful for predicting the absolute performance of every benchmark, they are still able to predict the change in performance when a modification is made to the overall design.

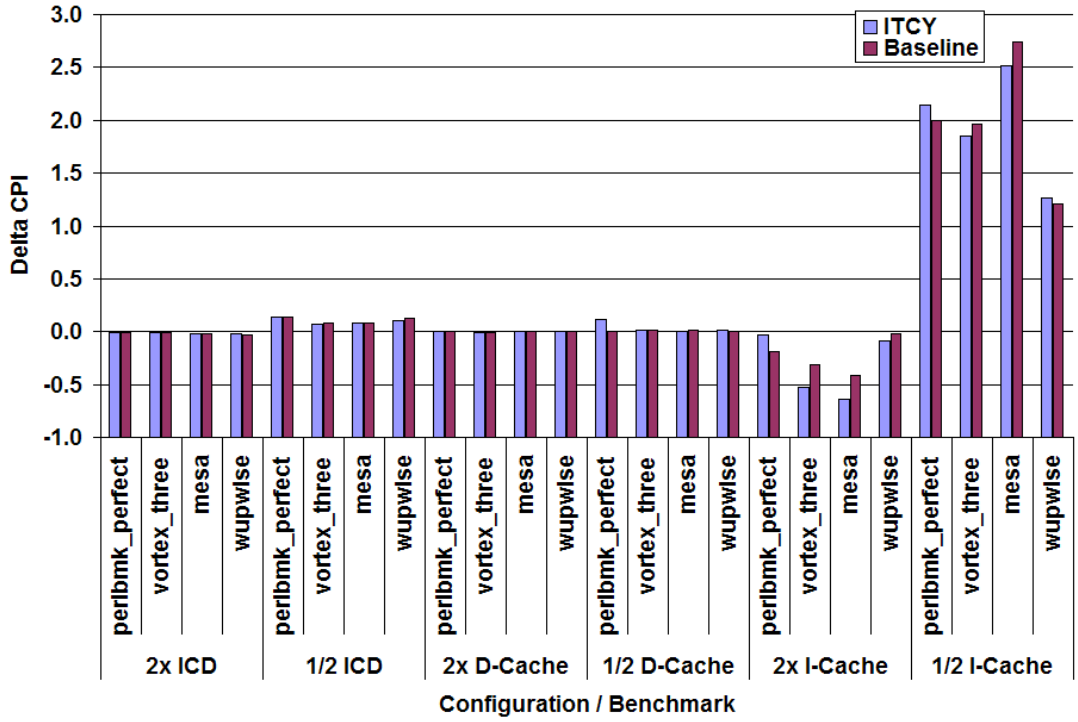

Figure 6.24: Tracking Design Changes using Delta CPI - ITCY versus Baseline

6.2.8 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. Unlike binaries produced using the ICBM method, ITCY code has the potential to be smaller in size if the interval chosen is small and does not require a significant amount of checkpointing. However, if larger intervals are chosen, then the file size will eventually become greater than ICBM code due to the increased amounts

of intra-interval code necessary. Since the intervals for the ITCY study contain only 10 million instructions and those for the ICBM study contain 100 million, the average SuiteSpeck is expected to be smaller than the average ICBM benchmark. This is the case in both Figures 6.25 and 6.26. Alternatively, if you include the file size for all 30 intervals, the ITCY code is larger than the ICBM code as in Figure 6.27 and 6.28. For the floating point benchmarks, this is more apparent than for the integer benchmarks. However, since the ITCY code for each benchmark actually consists of 300 million instructions, a proper comparison with ICBM should divide the sum of the 30 intervals by three to get a rough comparison of the amount of ITCY code needed for a 100 million instruction interval. For the integer benchmarks, this results in roughly a 15 MB ITCY file compared to an 8 MB ICBM file and for the floating point benchmarks a 93 MB ITCY file compared to a 27 MB ICBM file. A final summation of all the ITCY benchmarks is shown in Figure 6.29 indicating that a full set of replacement binaries for the SPEC2000 benchmarking suite would require 5.3 GB of storage.

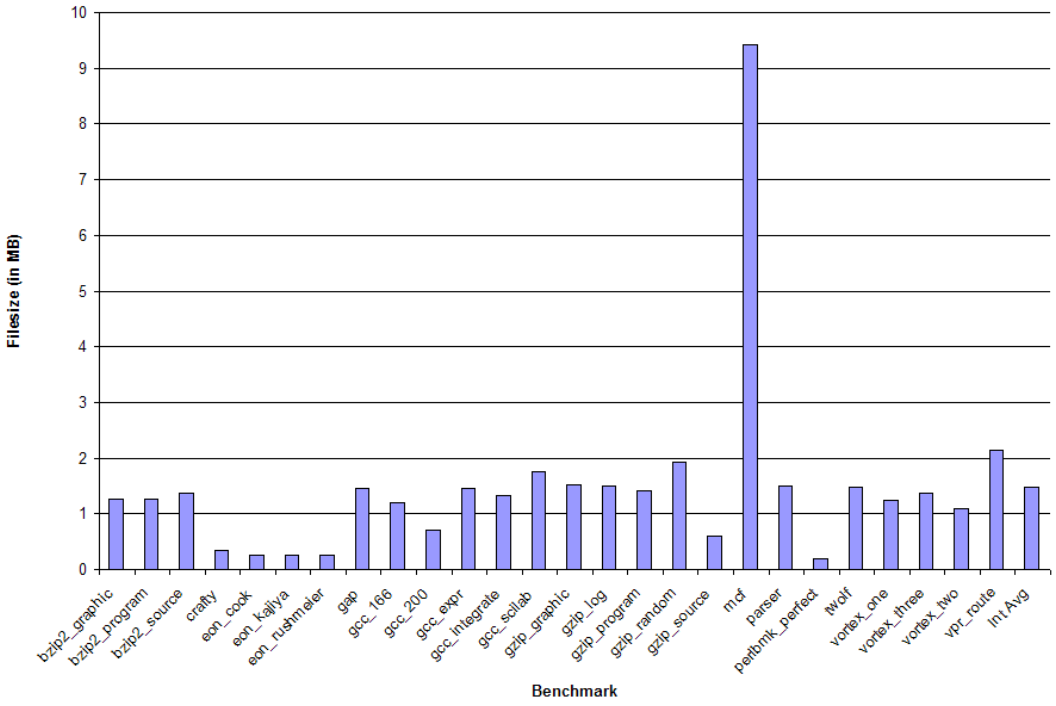

Figure 6.25: File Size of ITCY Code (per Interval Average) - Integer Benchmarks

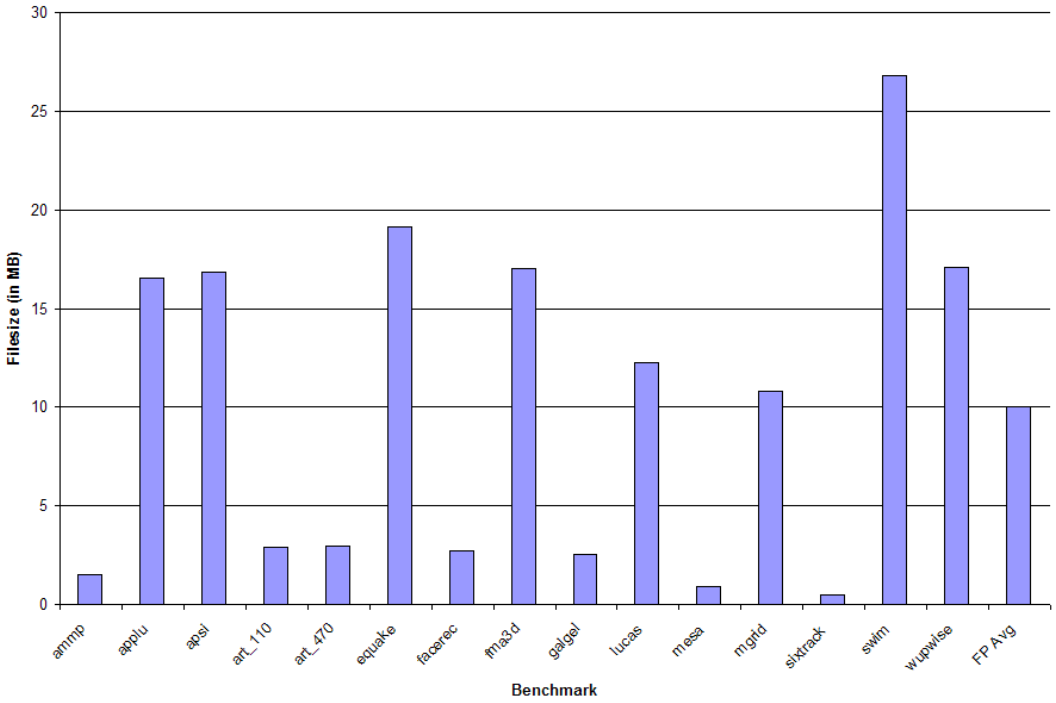

Figure 6.26: File Size of ITCY Code (per Interval Average) - Floating Point Benchmarks

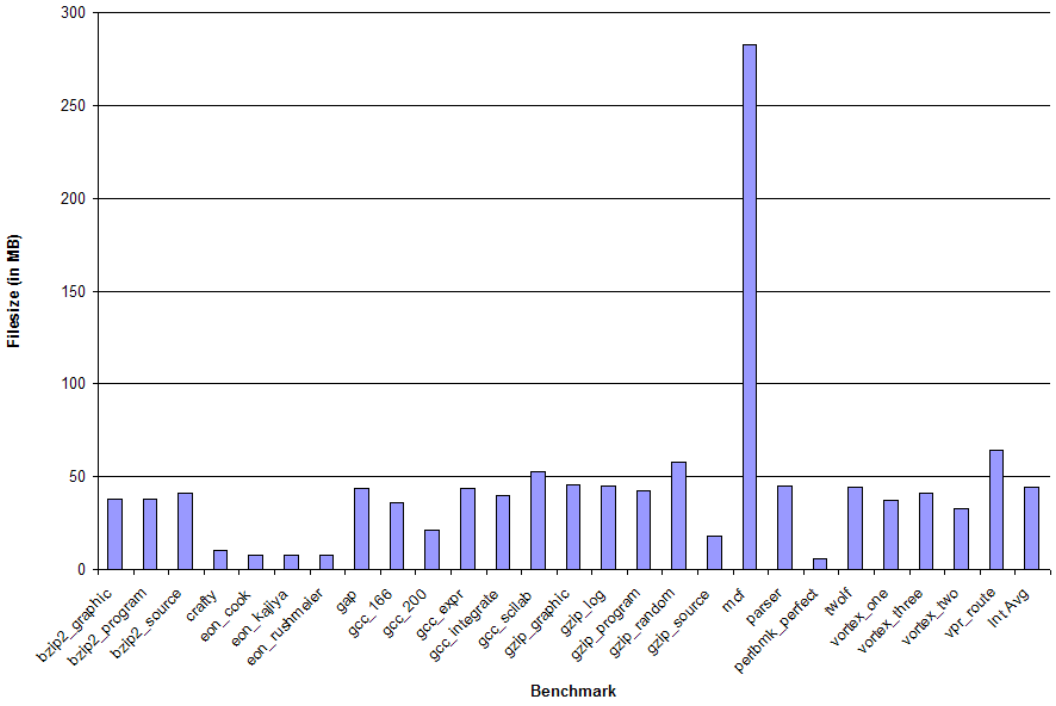

Figure 6.27: File Size of ITCY Code (all Intervals Totaled) - Integer Benchmarks

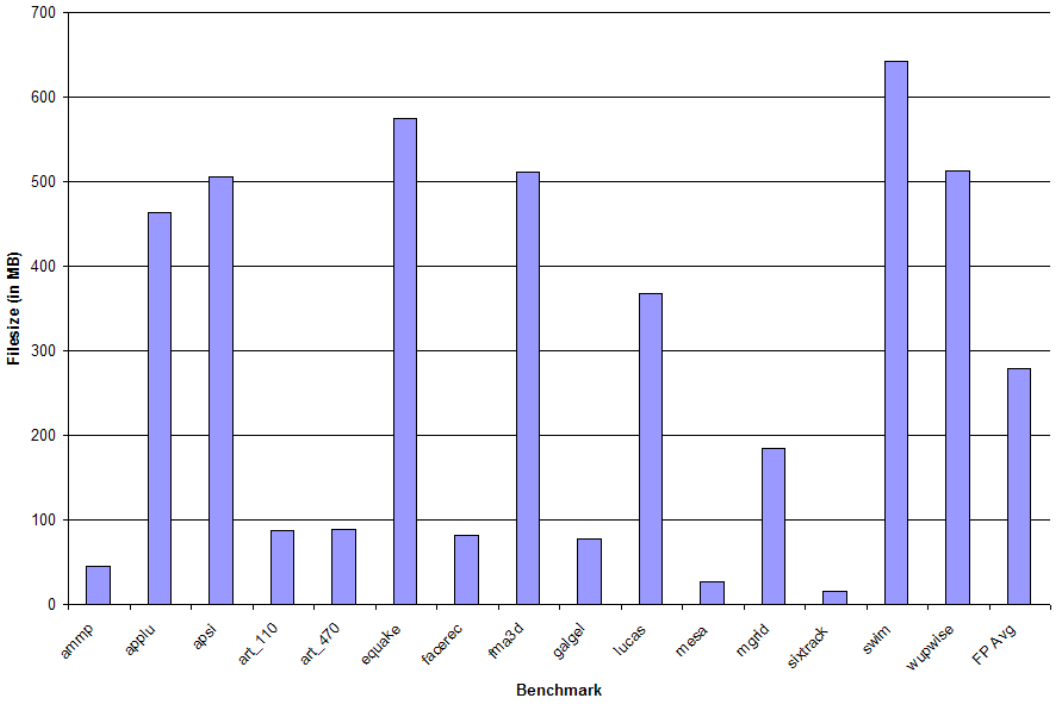

Figure 6.28: File Size of ITCY Code (all Intervals Totaled) - Floating Point Benchmarks

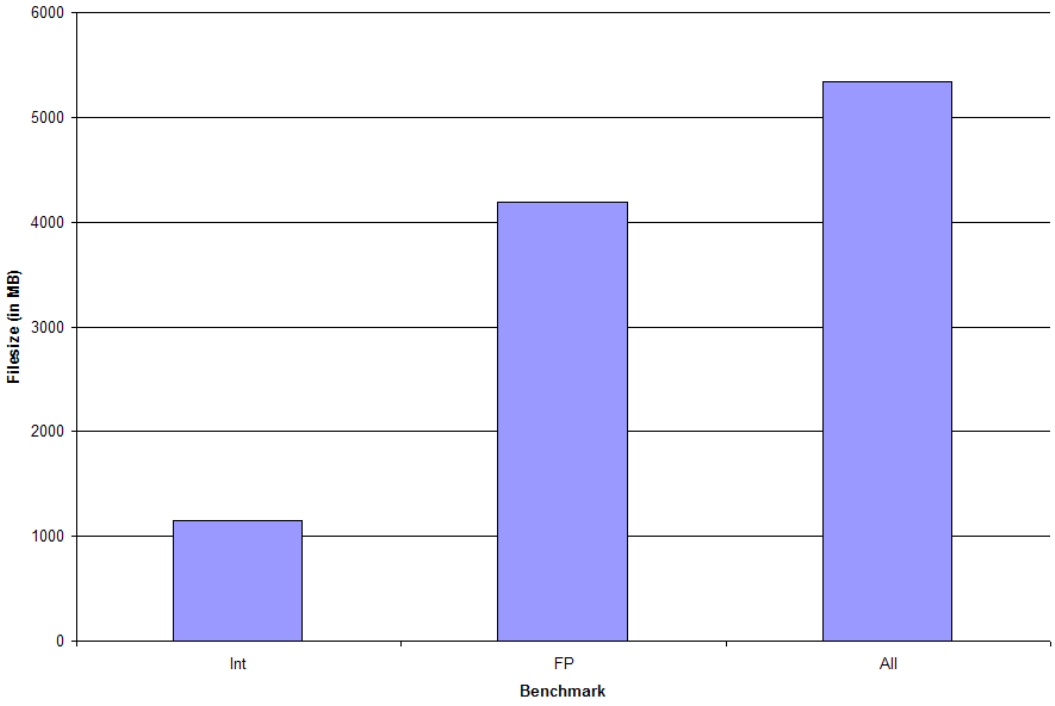

Figure 6.29: File Size of ITCY Code - All Benchmarks Totaled

6.2.9 Simulation Speedup

To measure the speedup in simulation time of the ITCY code compared to the original benchmark, an analysis was done similar to that done in section 6.5 for the ICBM code. However, since each benchmark of the ITCY code is broken into 30 pieces, the results are presented not only for the situation when the intervals would be executed serially for each benchmark in Figure 6.30, but also if they were all done in parallel in Figure 6.31. 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. The serial speedup is nearly identical to the ICBM speedup at 60x since both are executing roughly the same number of instructions. The greatest speedup can be seen in the

parallel speedup with an average of nearly 1000x speedup for the entire SPEC2000 suite 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 longer to run than other. However, a parallel speedup of 1000x clearly shows the potential of using ITCY code to rapidly and efficiently 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 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.

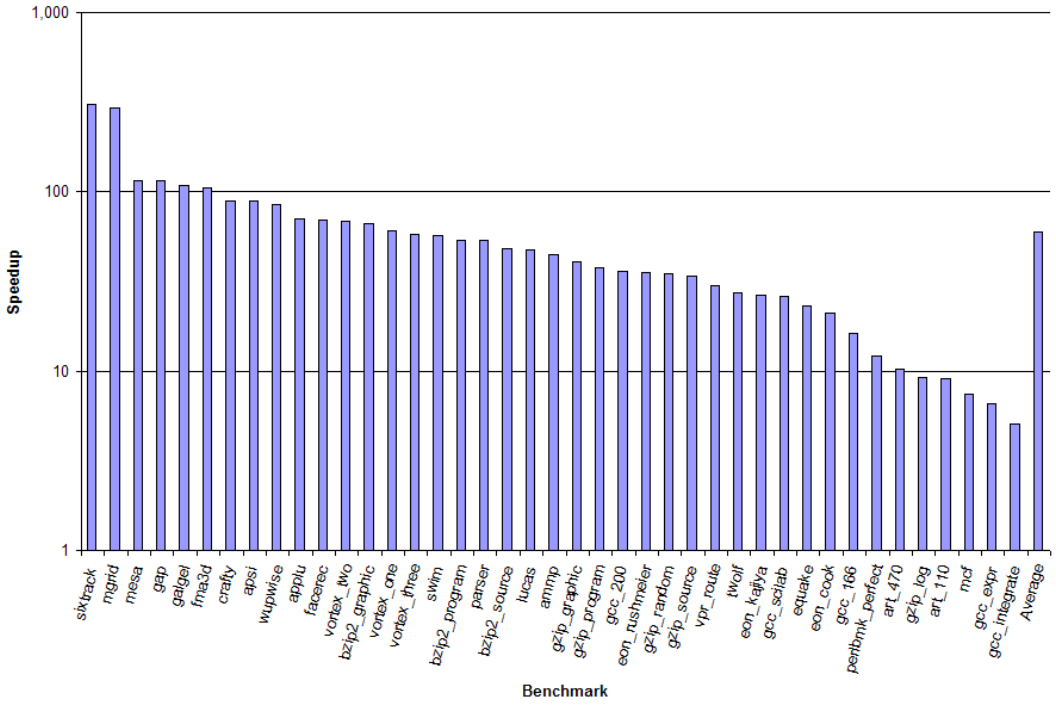

Figure 6.30: Speedup of ITCY Code Executed Serially

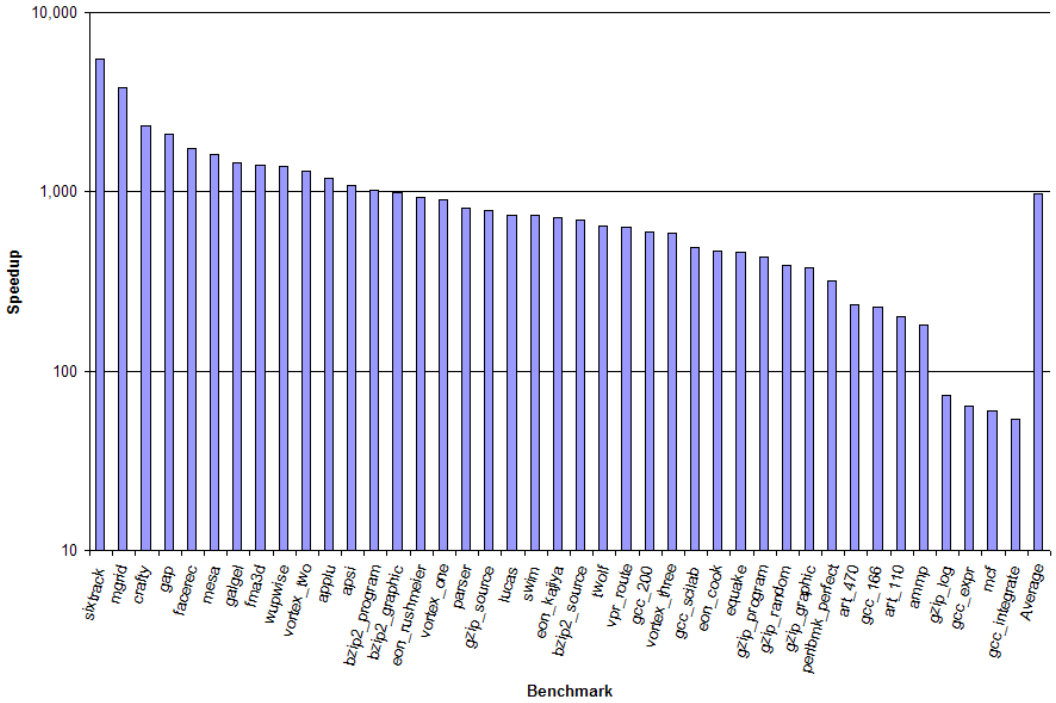

Figure 6.31: Speedup of ITCY Code Executed in Parallel

6.3 Discussion

The results presented in this chapter have shown the great potential of the ICBM and ITCY techniques 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 two techniques, and other current simulation time reduction methods, that need to be discussed. To aid in the discussion, Table 6.16 adds the ICBM and ITCY techniques to Table 2.2 that was seen in section 2.6. It should be noted that the CPI prediction accuracy is with respect to full detailed simulation and not the simulation intervals used to generate the ICBM and 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.

One major difference between the two techniques is with respect to the accuracy of their results. Since the ICBM technique uses the original binary to execute the chosen interval, the interval's execution profile will be exactly the same as when it was first run from within the original binary. Unfortunately, this binary reuse does not allow the ICBM technique to easily incorporate multiple intervals of instructions without requiring the creation of multiple new benchmarks. The only way to increase the prediction accuracy is to create more ICBM binaries to represent more intervals of execution. The ITCY technique, however, relocates the static instructions from the original binary into a new location in memory and then creates an entirely new binary by recompiling the interval code along with its intrinsic checkpointing and special control code. This allows the ITCY technique to, among other things, combine multiple simulation intervals into a single benchmark and remove syscall

Technique	Execution Time per B-mark	CPI Prediction Accuracy	Representa-tiveness	Micro-architecture Dependent	Storage Req's	Flexi-bility
B-mark Suite Reduction [24]	Variable	Variable	Low	No	N/A	High
Statistical (Trace) [18]	approx. 1000x speedup	2.3%	Low	Yes	negligible	Low
Statistical (Testcase) [10]	approx. 1000x speedup	2.4%	Low	Yes	negligible	High
SMARTS [47]	5 hrs	0.64%	High	No	N/A	Medium
SimPoint [41]	2.8 hrs	3.7%	High	No	N/A	Medium
SimPoint Startup [43]	14 mins (serial) 1 min (parallel)	1.2%	High	Yes	4 GB for 20 SPEC2K b-marks	Low
LivePoints [45]	91 secs	1.6%	High	Yes	12 GB compressed all SPEC2K b-marks	Low
ICBM (Single, 100M inst interval)	3 mins	2.12%	High	No	365 MB for 19 SPEC2K b-marks	Medium
ITCY (Thirty, 10M inst intervals)	16 mins (serial) 32 secs (parallel)	7%	High	No	5.3 GB for all SPEC2K b-marks	High

Table 6.16: Comparing ICBM and ITCY to Several Popular Simulation Time Reduction Techniques

calls from the code. While this affords a great deal of flexibility when creating multi-purpose benchmarks, the relocation of the interval instructions and the insertion of control code can adversely affect certain performance metrics of the new binary. Benchmarks that contain a great deal of indirect and unconditional branching can require large amounts of extra code to ensure their proper execution and the effects of this additional code can affect the accuracy of the new binary's results. In addition, benchmarks that have a very low I-cache miss rate can result in inaccurate results when only a few additional I-cache misses are introduced into the simulation. Many of these inaccuracies can be addressed by a judicious selection of instruction intervals

that limits the occurrence of these situations. However, if the interval selection is unavoidable or the accuracy of the new binary is the most important feature required, then the ICBM technique will need to be used.

Another difference between the proposed techniques and other simulation time reduction techniques pertains to their dependence on an underlying microarchitecture. Other checkpointing techniques, in order to refresh the microarchitectural state of the system, must store multiple sets of data that can be applied to a variety of different target microarchitectures. This can lead to a large amount of data being stored for each benchmark. The ICBM and ITCY techniques, however, require no such storage of microarchitectural information and simply require the execution of some pre-interval, warmup code identified using a reuse latency technique such as BLRL. In addition, since this extra information does not need to be stored, the resultant size of an ICBM or ITYC benchmarking set can be smaller. For example, a live-points library [45] for the SPEC2000 benchmarking suite requires 12 GB of storage whereas the ITCY technique requires only 5.3 GB. The live-points technique does produce a smaller average error rate, however, as was previously mentioned, the ICBM technique can be used if accuracy is most important. In addition, the live-points method requires simulator support to load in the checkpointing and warmup data thereby limiting its flexibility, whereas the ICBM and ITCY techniques do not.

CHAPTER VII

Conclusion

7.1 Thesis Summary

This dissertation presents two techniques that dramatically reduce the simulation time of a benchmarking program by allowing the rapid execution of only representative portions of code. Both techniques create highly portable benchmarks that can be moved between many different simulation environments. The first technique, Intrinsic Checkpointing through Binary Modification (ICBM), augments a benchmarking program by directly inserting checkpointing code into the binary. The new program does not need to be recompiled and is able to rapidly execute a fragment of the original program without requiring any special checkpointing or rapid simulation support on the part of the simulation environment. The second technique, InTrinsically Checkpointed assemblY (ITCY), also inserts checkpointing code into a benchmark. Unlike ICBM, the ITCY technique creates an entirely new assembly program comprised of the static instructions from one, or many, locations from within the original benchmark. The ITCY methods also remove system calls from the original program and convert their effects into emulation code that are 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. This

flexibility provides the framework for third parties to create microbenchmarks from their own internal benchmark sources. These microbenchmarks can then be released to the public without the concern of releasing any proprietary information since this sensitive information is effectively hidden inside the ITCY code. The end result of both techniques 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.

7.2 Future Directions

There are many different directions that the techniques in this dissertation can explore. Some of them are detailed below:

- *Emerging Workloads*: The applications focused on in this dissertation were those typically found in a standard computing environment. However, many different benchmarking suites exist for a variety of different and unique computing applications. Database workloads, online transactions, and JAVA applications are a few of the many different types of benchmarking applications that could be used in conjunction with the proposed techniques. Further study is needed to decide which suites would benefit more than others.

- *Embedded Applications*: Embedded applications often suffer from a limited amount of memory and running useful benchmarks can often be a problem. The techniques in the dissertation 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. In addition, the lack of a need for system call handling could be used to an advantage if the embedded device did not have such functionality.

- *Multiprocessor Workloads*: Multiprocessor workloads and designs are quickly becoming a part of the mainstream. However, little work has been done to move simulation time reduction techniques over to the multiprocessor domain. While it may be difficult to intrinsically checkpoint per-processor architectural features, it would be possible to easily checkpoint the main memory of a multiprocessor system using the proposed techniques. If the simulator had the ability to interpret processor-specific signal instructions and only update certain processors with data contained within a piece of ITCY code, then this could be used to update the state of individual processors as well.

- *SuiteMarks*: As was previously discussed, ITCY code has the ability to combine multiple intervals of code from different benchmarking applications into a new, single benchmark. The basic technique was used on intervals within the same benchmark to create SuiteSpots, however, it was not used with intervals from different benchmarks altogether. This method, if used properly, could potentially create individual benchmarks meant to represent the execution of an entire benchmarking suite. This is particularly the case for benchmarking suites that represent a large amount of redundant information. These "SuiteMarks" could then be gathered together to create one large repository of benchmarking suites that could rapidly and accurately simulate a vast quantity of benchmarks in a very short period of time.

- *ArchMarks*: A final area of future work could leverage the ability of the ITCY technique 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 architecture. 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.

BIBLIOGRAPHY

BIBLIOGRAPHY

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

[2] K.C. Barr. *Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots*. PhD thesis, Massachusetts Institute of Technology, September 2006.

[3] K.C. Barr and K. Asanovic. Branch trace compression for snapshot-based simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 25–36, Austin, TX, March 2006.

[4] K.C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 66–77, Austin, TX, March 2005.

[5] 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.

[6] R.H. Bell, Jr., R.R. Bhatia, L.K. John, J. Stuecheli, J. Griswell, P. Tu, L. Capps, A. Blanchard, and R. Thai. Automatic testcase synthesis and performance model validation for high-performance PowerPC processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 154–165, Austin, TX, March 2006.

[7] 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.

[8] 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.

[9] R.H. Bell, Jr. and L.K. John. Efficient power analysis using synthetic testcases. In *Proceedings of the International Symposium on Workload Characterization*, pages 110–118, Austin, TX, October 2005.

[10] 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, Cambridge, MA, June 2005.

[11] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, Jul/Aug 2006.

[12] S. Chen. Direct SMARTS: Accelerating microarchitectural simulation through direct execution. Master's thesis, Carnegie Mellon University, June 2004.

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

[14] 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.

[15] L. Eeckhout and K. De Bosschere. Early design phase power and performance modeling through statistical simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 10–17, Tucson, AZ, June 2001.

[16] 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*, pages 1–6, Austin, TX, April 2000.

[17] L. Eeckhout, Y. Luo, K. De Bosschere, and L.K. John. BLRL: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48:451–459, 2005.

[18] 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*, pages 87–96, Cairns, Queensland, June 2006.

[19] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*, pages 3–14, Austin, TX, December 2001.

[20] G. Hamerly, E. Perelman, and B. Calder. How to use SimPoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):25–30, March 2004.

[21] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation*, Madison, WI, June 2005.

[22] J. Haskins and K. Skadron. Memory Reference Reuse Latency: Accelerated sampled microarchitecture simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 195–203, Austin, TX, March 2003.

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

[24] 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.

[25] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 135–146, Austin, TX, March 2005.

[26] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, December 1997.

[27] 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.

[28] Y. Luo. *Improving Sampled Microprocessor Simulation*. PhD thesis, The University of Texas at Austin, August 2005.

[29] H. McGhan. SPEC CPU2006 benchmark suite. *Microprocessor Report*, October 2006.

[30] 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.

[31] S. Nussbaum and J.E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Barcelona, Spain, September 2001.

[32] 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.

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

[34] 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.

[35] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, October 2003.

[36] 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.

[37] A. Phansalkar, A. Joshi, and L.K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the International Symposium on Computer Architecture*, pages 412–423, San Diego, CA, June 2007.

[38] A.S. Phansalkar. *Measuring Program Similarity for Efficient Benchmarking and Performance Analysis of Computer Systems*. PhD thesis, The University of Texas at Austin, May 2007.

[39] 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.

[40] 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.

[41] 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.

[42] P.K. Szwed, D. Marques, R.M. Buels, S.A. McKee, and M. Schulz. SimSnap: Fast-Forwarding via native execution and application-level checkpointing. In *Proceedings of the Workshop on the Interaction between Compilers and Computer Architectures*, pages 65–74, Madrid, Spain, February 2004.

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

[44] 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.

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

[46] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26:18–31, July-August 2006.

[47] 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.

[48] 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.