In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2001.

Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications

Timothy Sherwood Erez Perelman Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,eperelma,calder}@cs.ucsd.edu

Abstract

Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to months to complete. To overcome this problem researchers choose a very small portion of a program's execution to evaluate their results, rather than simulating the entire program.

In this paper we propose Basic Block Distribution Analysis *as an automated approach for finding these small portions of the program to simulate that are representative of the entire program's execution. This approach is based upon using profiles of a program's code structure (basic blocks) to uniquely identify different phases of execution in the program. We show that the periodicity of the basic block frequency profile reflects the periodicity of detailed simulation across several different architectural metrics (e.g., IPC, branch miss rate, cache miss rate, value misprediction, address misprediction, and reorder buffer occupancy). Since basic block frequencies can be collected using very fast profiling tools, our approach provides a practical technique for finding the periodicity and simulation points in applications.*

1 Introduction

In order to evaluate new architecture features, detailed modeling of the pipeline, buses, and queuing delays are needed along with timing models and power estimation. Detailed simulation takes a great deal of processing power and time, and only a small subset of a whole program is often simulated. Many programs have wildly different behavior during different parts of their execution making the section of the program's execution simulated of great importance to the relevance and correctness of the study.

In [10], we found, when looking at architecture features, that most programs demonstrate cyclic behavior across many different metrics. These include IPC, branch prediction, value prediction, address prediction, cache performance, and reorder buffer occupancy. Cyclic (periodic) behavior of an application is defined as a repeatable pattern seen for the metric throughout the program's execution. For example, the SPEC95 program `wave`, shows two main phases to its cycle. It has an IPC of 3 during the 1st phase, and an IPC of 2 during the 2nd phase, and this repeats throughout its execution. The period is the length of time it takes to complete both phases of its cycle.

The main focus of our paper is to develop an automated, accurate, and efficient approach for determining the starting points in a program to simulate and the duration of the simulation. We focus on finding:

i. The end of the initialization part of the program, and the start of the cyclic part of the program.

ii. The period of the program. The period is the length of the cyclic nature found during a program's execution.

iii. The ideal place to simulate given a specific number of instructions one has time to simulate.

iv. An accurate confidence estimation of the simulation point.

To create a fast and efficient tool, we focused on an approach that does not use any knowledge of the architectural metrics for the program, but is instead highly correlated with the performance of those metrics.

We propose using *Basic Block Distribution Analysis* (BBDA) to calculate the above enumerated items. When running a program to completion, it will execute each basic block a certain number of times. Taking a snapshot of the number of times each basic block is executed provides us with a basic block fingerprint. We use basic block fingerprints gathered for small intervals of the program's execution to find representative areas of the program to simulate. This is done by finding the best match of these smaller basic block fingerprints to a basic block fingerprint representing the complete execution of the program.

A potential advantage of BBDA is that it only requires basic block profiles, which means a relatively fast basic block profiler is used (as opposed to slow timing simulation). In addition, many compilers already collect basic block and edge frequency profiling information for performance tuning, and to guide hot-path and code layout compiler optimizations.

The rest of the paper has the following organization. Section 2 details an example motivating why cyclic behavior exists in applications. Section 3 describes Basic Block Distribution Analysis, and how it is used to find the end of initialization and the period length in applications. Section 4 examines the cyclic behavior of programs in terms of architectural features and metrics, in order to see how they correlate to the basic block cyclic behavior found using BBDA. Section 5 presents results using BBDA for finding places in an application to simulate. It also analyzes the error in using BBDA for finding the representative part of the program to simulate across a range of architecture features (IPC, branch prediction, value prediction, address prediction, cache designs, and reorder buffer occupancy). Section 6 describes related work, and section 7 summarizes the results and contributions of our work.

2 Cyclic Behavior of Programs

Most programs do not execute in a steady state, even at a high level. Instead they tend to go through different stages of execution, starting with a setup phase which is used to initialize data structures and set up for the rest of execution. This start-up time can account for a significant amount of execution. For example, the SPEC95 program `wave` needs to execute for almost 7 billion instructions before it reaches the code that accounts for the bulk of the execution.

Once the initialization stage has been past and we are in the bulk of the execution, there are still execution phases to be found. Programs tend to be written in a modular fashion, often as a set of procedures contained in a loop, where each procedure is then another loop with more procedures. While this mode of execution is not representative of every important program written, it is the common case for compute bound applications, the type that we concern ourselves with when examining new architectural modifications. Applications, when written in this manner, have a very strong *periodic* behavior, alternating between completely different sections of code.

If, as computer architects, we are not cognizant of the fact that programs execute in distinct phases, we may be testing the performance of our machine on a single very unrepresentative section of execution such as the initialization phase, or at the very least we may be over-representing parts of the program.

Figure 1 shows the behavior of `wave` as it executes. Plotted on the graph are a variety of architectural metrics such as IPC and cache miss rates. The graph shows that `wave` has very distinct phases of execution, starting with an initialization phase that ends around 7 billion instructions. After this, the program enters into a series of cycles, each made up of two phases. In one phase an average of over three IPC is achieved every cycle, while in the other phase the IPC drops down to under two. Complete details for this graph are discussed in Section 4.

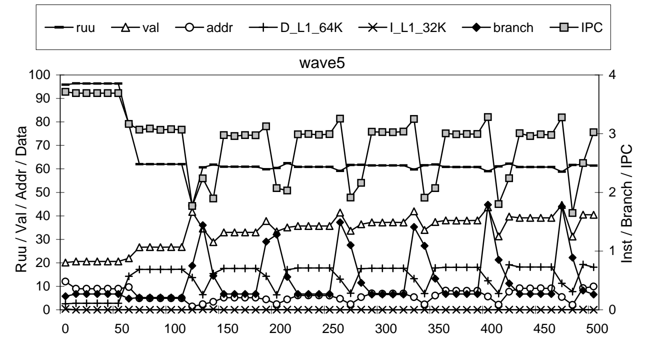

Figure 1: Time varying behavior for the SPEC95 program `wave`. Each unit of the X-axis represents 100 million committed instructions. Results are shown for IPC, reorder buffer (RUU) occupancy, value, address, and branch prediction miss rates, instruction and data cache miss rates using the Y-axis scale each metric is labelled upon.

The reason for this periodic behavior can be seen in the call graphs generated for `wave` shown in Figures 2 and 3. Figure 2 is the call graph generated for just the partial execution of `wave` during the sections of high IPC, while Figure 3 is the call graph for the sections of low IPC. The nodes on the graphs are procedures annotated with the number of times that they were called. The strong periodic behavior of `wave` is due to an outer function, not shown, calling two different routines in succession, `trans` and `field`. The call graphs show that `trans` and `field` do share some low level functions such as `__F_sqrt4` and `_OtsDivide32`, but the bulk of their execution occurs in different functions.

It is easy to see that if careful decisions are not made about where in a program's execution to simulate, we could easily see differences of a factor 2 in important metrics such as IPC. This example further demonstrates the fact that different phase behavior can be identified by examining the execution behavior of the code. This motivated us to develop a general automated technique for determining where to simulate based on the basic blocks of the program.

3 Basic Block Distribution Analysis

In this section we propose Basic Block Distribution Analysis as a generic way of determining the cyclic behavior of an application and finding preferred simulation points in the application in order to achieve a representative sample of its execution.

3.1 Basic Block Vectors

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies

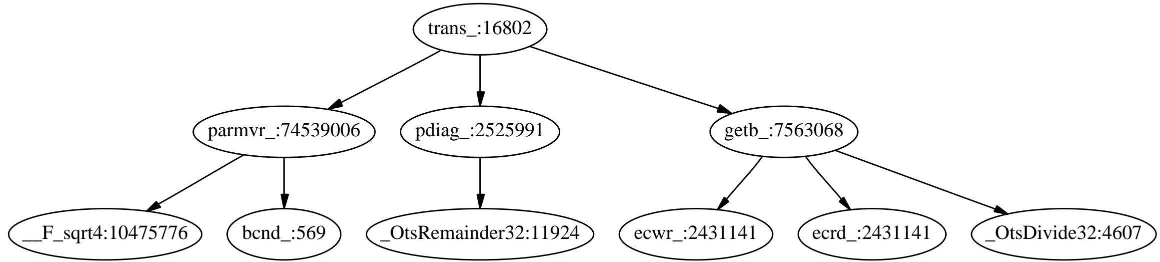

Figure 2: Call graph generated from `wave` for the phase of execution, where an average IPC of 3 is achieved.

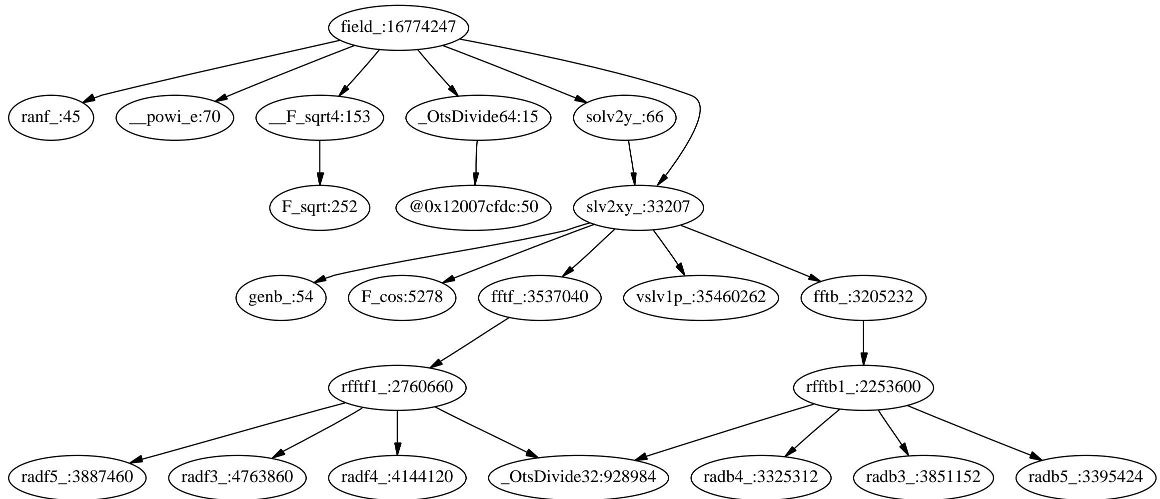

Figure 3: Call graph generated for the phase from `wave`, where there is an average IPC of 2.

with which basic blocks are executed as the metric to compare different sections of the application's execution. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing at that time, and basic blocks provide us with this information.

The program, when run for any interval of time, will execute each basic block in the program a certain number of times. Knowing this information provides us with a basic block *fingerprint* for that interval of execution, which tells where in the code the application is spending its time. The basic idea is to find a reasonable sized interval of time in the program's execution that has a basic block fingerprint similar to the full execution of the program. If we can find this, we know that both the full execution of the program and the interval we choose spends proportionally the same amount of time in the same code.

For the results in this paper, the basic block fingerprints are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, or *sample*, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next 100 million interval.

A *Basic Block Vector* (BBV) is a single dimensional array, where there is an element for each static basic block in the program. Each element in the array is the count of how many times a given basic block has been entered during an interval. It is sometimes useful to take Basic Block Vectors of varying size intervals. We say that a Basic Block Vector, which was gathered by counting basic block executions over an interval of N $times$ 100 million instructions, is a *Basic Block Vector of duration N*.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the *proportions* of basic block execution, a BBV is normalized by having each element divided by the sum of all the elements in the vector. This normalization ensures that the sum of all the

elements in the BBV is 1, which in turn allows us to compare vectors of different durations.

The Basic Block Vector that contains the normalized basic block frequencies for the entire execution of the program provides what we call the *target* BBV. It is the goal of the analysis that we present to find a Basic Block Vector, of small duration, that is very similar to the target BBV. In finding this we will have found a section of code that is representative of the whole.

In order to find a Basic Block Vector that is similar to the target BBV, we must first have some way of comparing two Basic Block Vectors. The operation we desire takes as input two Basic Block Vectors, and as output has a number which tells us how close they are to each other. To compute this function we take the element-wise subtraction of the two vectors. We then take the absolute value of each element, and sum all the elements together into a single number. This produces a number between 0 and 2, since each BBV sums to 1. We use this single number to tell us how closely related the two BBVs are. We call this the *difference* between the two BBVs. Now that we have a way of comparing two Basic Block Vectors, we can begin to look into how the execution of a program changes over time.

3.2 Creating a Basic Block Difference Graph

Before we can begin to understand how to find a representative interval of the program, we need to understand how the execution of a program changes over time. For this reason we create a Basic Block Difference Graph. The Basic Block Difference Graph is a plot of how well each individual sample in the program compares to the target Basic Block Vector created for the entire run to completion.

For each interval of 100 million instructions, we create a BBV of duration 1 and calculate its difference from the target BBV. Figure 4 shows the plot of all of the BBV differences across the entire execution creating a *Basic Block Difference Graph*. The x-axis is the number of instructions in 100 millions, and the y-axis is our measure of comparison between Basic Block Vectors discussed above, the BBV difference. A difference of 2 means that the two vectors are completely unrelated, while a deviation of 0 is the result of a perfect match between a BBV and the target vector.

In the following sections we describe how we use the basic block difference graph to (1) find the initialization phase of the program, and (2) find the period for that program.

3.3 Finding the End of Initialization Phase

Execution during the initialization phase of programs is very different from the steady state behavior of the application. In a study on value prediction [2], we found that `tomcatv` saw a 68% execution speedup using value prediction when simulating the initialization phase of the program, in comparison to 5.8% speedup after fast forwarding past the initialization

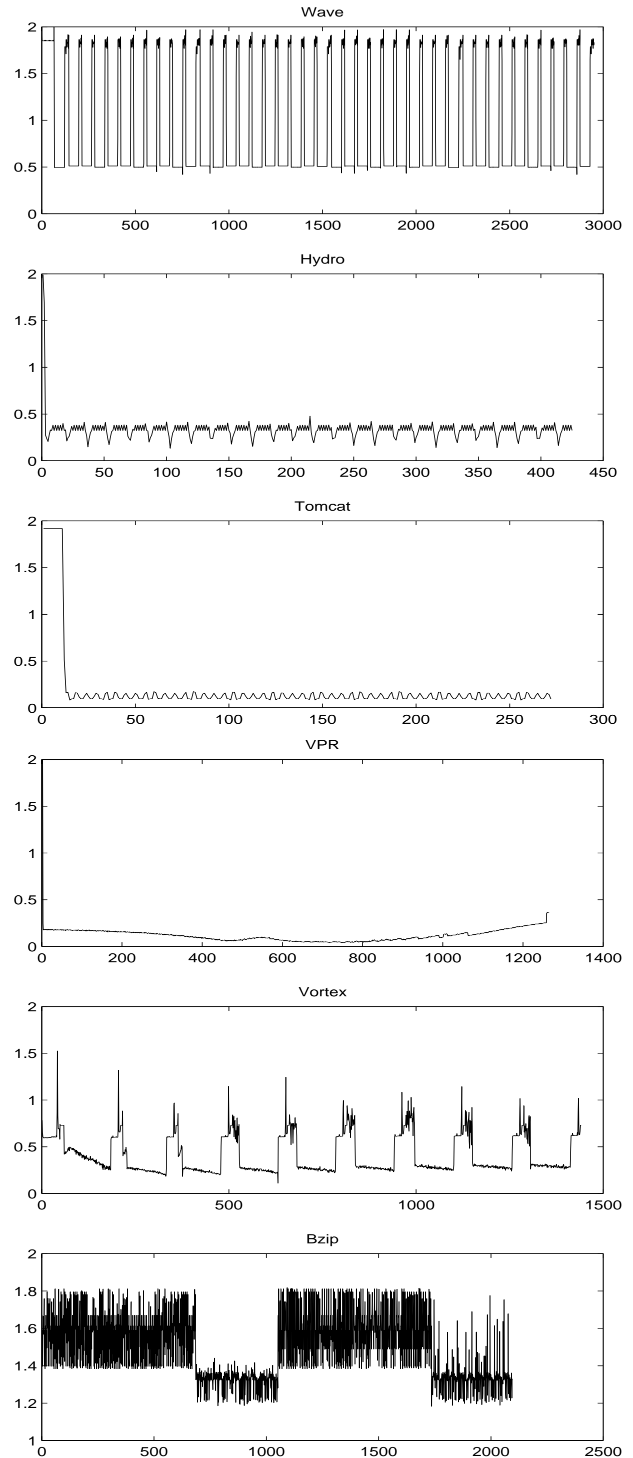

Figure 4: The basic block difference graphs. Each x-axis unit represents 100 million executed instructions. The graphs show the Basic Block Vector difference on the y-axis, which is calculated by comparing the target BBV with the BBV generated for each 100 million interval of executed instructions.

phase. In contrast, `vortex` saw an 11% execution speedup with value prediction in the initial part of the program, but saw a 27% execution speedup after fast forwarding. These results show that results generated for only the beginning of execution can be terribly misleading, and that it is very important to simulate representative sections of code.

The approach we use to determine the end of the initialization stage can be thought of as sliding a piece of jig-saw puzzle over the rest of the puzzle. Since the jig-saw piece will fit best at the spot it is removed from, the comparison at that point will show the least difference. However, as soon as it is shifted away from its space, the comparison with the underlying pieces will show a marked difference.

To find the end of the initialization phase we treat the BB difference graph as a signal. We take the first quarter of the BB difference graph (signal), which we call the *Initialization Representative Signal* (IRS), and we use this to search for the end of the initialization. We take the IRS and slide it over the BB difference signal looking for the first peak where the IRS differs from the BB difference signal. In this way we treat the BB difference graph as the puzzle, and the IRS as the piece of the puzzle we are sliding across.

We chose IRS to be the first quarter of the BB difference signal to capture the majority if not all of the initialization stage. This is based on the assumption that the initialization phase will be shorter than half the length of the entire execution.

We compare the IRS at every point across the first half of the original BB difference signal. A signal starting at each point in the BB difference graph equal in length to the IRS signal, is compared to the IRS. To compare these two sub-signals we take the absolute difference of each point of the two sub-signals, and summarize the resulting differences into a single number. This number represents how close these two signals match up. This is done for every point within the first half of execution in the BB difference graph resulting in a new graph, which we call the *Initialization Difference Graph*. These are shown in Figure 5.

The graphs can be categorized into two observable behaviors. A periodic pattern as seen with `wave`, `vortex`, and `bzip`, is due to the IRS containing the initialization stage as well as some cyclical behavior from the execution. This is enough to manifest the cyclical behavior during the remainder of the comparison past the initialization stage. A steep incline with a plateau is seen with `hydro`, `tomcat` and `vpr`. The plateau is explained by the initialization part of these programs not having any overlap with the rest of the program after the initialization phase is completed.

From the programs we examined, the initialization stage is complete at the first peak or corner in the initialization difference graph. When the initialization representative signal finally reaches the end of the initialization stage on the BB difference signal, the difference is maximized since there is no more of the initialization phase left to compare to.

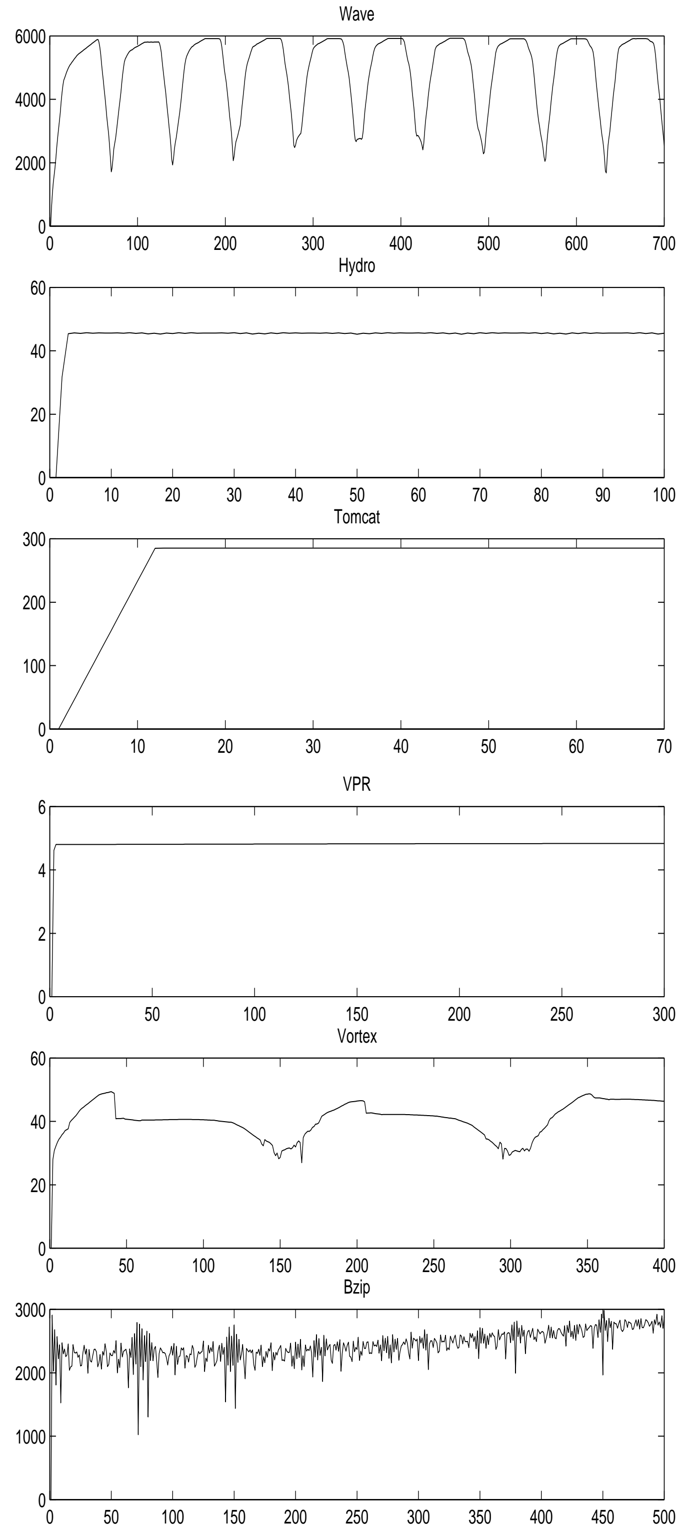

Figure 5: The initialization difference graphs. The x-axis units are in terms of 100 million instructions. The y-axis for each x-axis value represents the signal difference between the IRS and the original basic block difference signal *starting* at that x-axis value.

Mathematically, a peak or a corner in a graph represents the point where the slope is changing the fastest. The second derivative is a function of the rate of change of the slope, and is used in our algorithm to determine this point marking the end of the initialization. The first column in Table 2 shows the end of initialization points that are automatically found using the above analysis.

3.4 Finding the Period

To find the period we form a *Period Representative Signal* (PRS) from the BB difference graph starting at the precomputed end of the initialization phase found in the previous section. The PRS we use is one quarter the length of the program's execution. We found that duration to be sufficient to capture periods of length (duration) comprising up to half of the program's execution.

To find the period we slide the PRS across half the entire BB difference graph, starting at the end of the initialization stage. We perform the same comparisons for each x-axis value as above for finding the initialization stage, resulting in *Period Difference Graphs* shown in Figure 6.

The period graph shows all of the points where the PRS matched the sub-signals from the original signal (BB difference graph). After shifting the PRS over the BB difference graph, the resulting calculations close to zero represent a match of the PRS to the original sub-signal. The time duration between each match represents the period for the program. Therefore, all of the local minimums from shifting the PRS are used to calculate the period. The period is calculated by taking these minimum Y-axis points in the period graph, and calculating the length in instructions (X-axis) between these minimums. This length is the period of the signal, and the period of the application. The second column in Table 2 shows the periods that are automatically found using the above analysis.

The two programs that do not fit cleanly into our description for finding periodic behavior are `vpr` and `bzip`. `Vpr` does not have a very visible period, and its behavior is not very repetitive. However, we still find very good representative points for simulation for `vpr` as is shown using the analysis we present in section 5.

`Bzip` on the other hand has multiple periods. The first and largest period has a duration of 1046 as seen in Figure 4, which consists of 2 cycles over the complete execution of `bzip`. In looking at Figure 7, we can see how the behavior is captured when creating a BB difference graph using different BBV durations. Results are shown for using basic block vectors with duration of 6, 12, and 52 (hundred of million instructions) to create the BB difference graph. For a vector duration of 6, we find that the next period to be found has a duration of 78, and the smallest period is of size 9. These figures also show that using larger durations of a BBV creates a BB difference graph that emphasizes the larger periods.

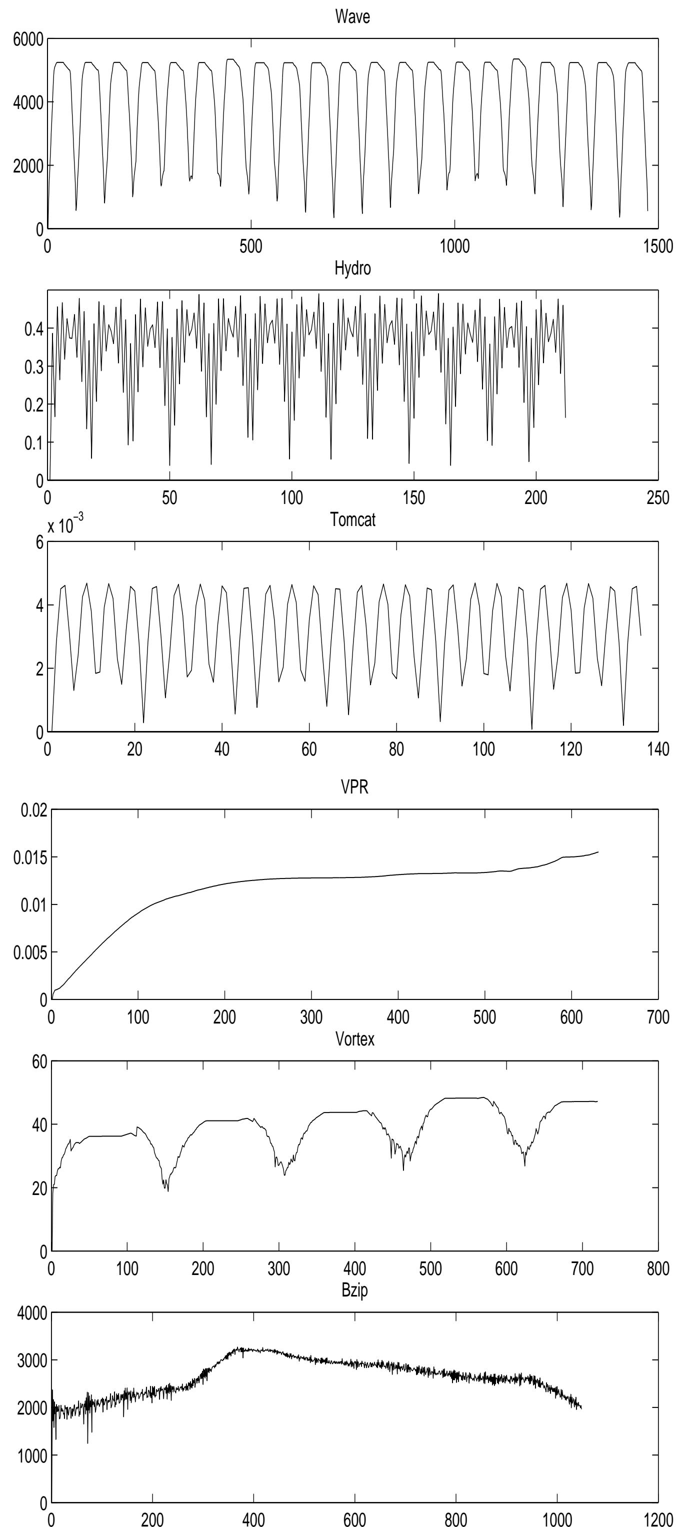

Figure 6: The period difference graphs. The x-axis units are in terms of 100 million instructions. The y-axis for each x-axis point represents the signal difference between the PRS and the original basic block difference signal *starting* after the end of the initialization phase.

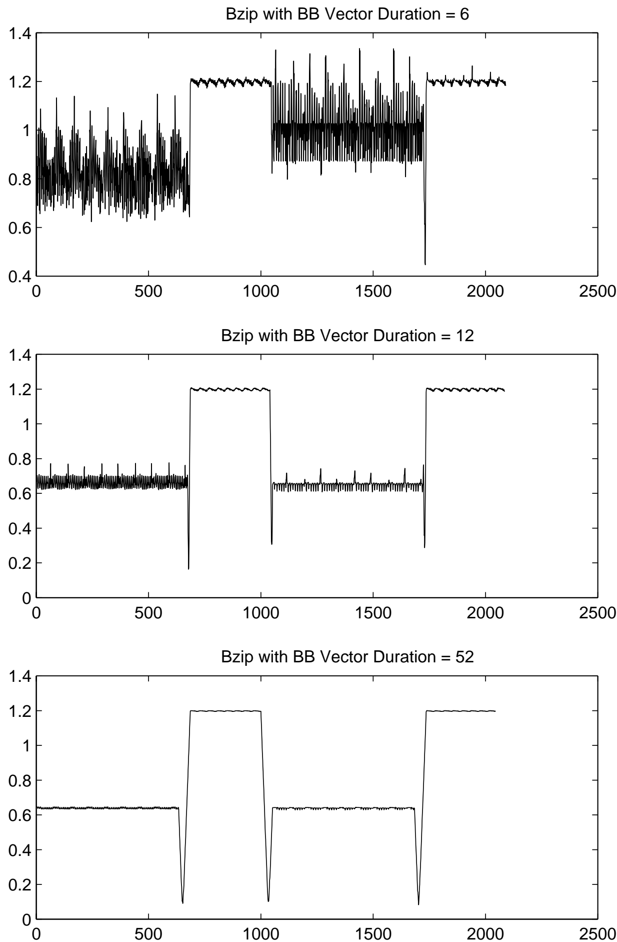

Figure 7: The basic block difference graphs for `bzip` varying duration of the basic block vector when creating the graphs. The results show that using a larger BB vector creates a BB difference graph that emphasizes the larger periods.

3.5 Fourier Analysis

We initially tried to use Fourier analysis to discover the period of a signal. We convolved the signal with itself to smooth out the basic block difference graph, emphasizing the frequencies with larger amplitude. The convolution accentuates the periodic behavior of the original signal, but this new signal still had to be analyzed to find the period. For `vortex`, the new signal was actually inferior to the original, since `vortex` has a slightly varying period throughout its entire execution. The convolution did not work well for signals that did not have static period lengths.

The Fourier analysis potentially could have benefits when dealing with certain types of execution. Our period algorithm currently computes a single period for the entire execution. This is not always the optimal period, because there could locally be periodic behavior throughout the execution. `Bzip` is a prime example of this as described above. `Bzip` has two distinct phases, and each phase has its own periodic behavior but the signal is very noisy. Fourier analysis could potentially provide information about all the periodic behavior in a signal, and extracting this to optimize our current approach is part of future work.

Instruction Cache	32k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	64k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 12 cycle latency
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

4 Cyclic Behavior of Architectural Metrics

In this section we examine the time varying behavior of applications in terms of architectural features and metrics. We show the correlation between the periodic behavior found via BBDA and the architectural features and metrics examined during simulation.

4.1 Methodology

To examine these architecture features and metrics, we collected information for three of SPEC95 programs (`tomcatv`, `hydro`, and `wave`) and three of the SPEC 2000 programs (`bzip`, `vortex`, `vpr`) for their reference input sets. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, C++, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used was derived from the SimpleScalar 3.0a tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. The simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction. The baseline microarchitecture model is detailed in Table 1. We modified the 3.0a release of SimpleScalar, so that the memory hierarchy buses were pipelined, with a transfer width of 8 bytes per cycle.

4.2 Collecting Time Varying Behavior

To show the time varying behavior of the programs, SimpleScalar was modified to output and clear its statistics after every 100 million *committed* instructions. Only the statistic counters are cleared between intervals, and the state of the machine (e.g., cache and branch prediction tables) are *not* cleared between intervals. This eliminates any cold-start error from being added into the experiment.

Results are then graphed for every 100 million committed instructions for the programs examined. This should yield a clear picture of the large scale runtime behavior exhibited by each application as well as indicating which sets of instructions are more indicative of the execution as a whole. It is, however, of small enough granularity that it provides useful information about program start up times and can be easily simulated on any machine. Each program was run until completion, but we only graph enough intervals to show the cyclic nature for each program.

The following summarizes the data graphed:

- **Instructions Per Cycle**. This is the number of instructions that are committed in each sample, which is always 100 million, divided by the number of simulated cycles that it took to execute those instructions.

- **Percent RUU Occupancy**. SimpleScalar uses a unified Register Update Unit (RUU) to model its reorder buffer and reservation stations [12]. In our simulations we used a 128 entry RUU, and report results in terms of the percent of the RUU entries used on average during a 100 million instruction sampling period.

- **Cache Miss Rate**. Cache miss rates are shown for a 32 KB 2-way associative instruction cache, and a 64 KB 4-way associative data cache. Both caches have 32 byte lines.

- **Branch Prediction Miss Rate**. We used McFarling's bi-modal gshare branch predictor [7]. An 8K entry 2-bit chooser table is used to choose between an 8K entry 2-bit bi-modal branch predictor and an 8K entry gshare table. A 256 entry 4-way associative branch target buffer is used to provide the predicted addresses, and a 32 entry return address stack is used to predict return instructions. The branch misprediction rate over all the types of executed branch instructions is shown.

- **Address Prediction Miss Rate**. Miss rates are shown for 2-delta stride address prediction for an infinite sized table (each load gets its own entry) [5, 9]. The 2-delta address predictor will only change it's prediction if the stride is seen two times in a row. Miss rates are shown for only applying address prediction to load instructions.

- **Value Prediction Miss Rate**. Miss rates are shown for 2-delta value and address prediction for an infinite sized table [4, 13]. The 2-delta value predictor will only change the stride if seen two times in a row. Miss rates are shown for only applying value prediction to load instructions.

Note, address and value prediction were not used for architectural optimizations in gathering these results, only their miss rates were gathered. Therefore, they do not affect the IPC, branch or cache miss rate results being shown.

4.2.1 Cyclic Architecture Results

Figure 8 and Figure 1 show the time varying behavior of the 6 SPEC programs we examined. The legend is at the top of each figure. For each program, the results for IPC, average percent RUU occupancy, percent branch miss rate, percent value miss rate, percent address miss rate, and percent instruction and data cache miss rates are shown on the same graph. Since all of these different results are shown on the same graph, each graph has two Y-axis.

For each graph, the left and right Y-axis are labeled with the metrics that use that axis. For most of the graphs, percent RUU occupancy, and value and address miss rates use the left Y-axis. Similarly, I-Cache miss rate, branch miss rate, and IPC usually use the right Y-axis. The D-Cache miss rate is shown on either axis depending upon the program and axis scale in order to allow interesting trends to be seen.

The X-axis is in terms of 100 million *committed* instructions. We ran all of the programs to completion, and found them to either (1) converge to a constant behavior until the last few 100 million instructions, or (2) have a repeatable cyclic behavior until the end of their execution. Because of this, and to save space, we only show enough of the program to demonstrate the cycles we found. For `hydro`, `tomcat`, and `bzip`, 5 billion instructions is enough to clearly demonstrate the cyclic nature of the programs. `Vortex` has cycles of a much larger scale, on the order of 150 billion instructions, and `wave` has cycles on the order of 7 billion instructions. `Vpr` has mild cyclic tendencies, but the pattern is not as concrete as for other programs.

5 Choosing Where to Simulate and Error Analysis

SimpleScalar [1], one of the fastest simulators, executes on the order of 1000 times slower than hardware. SimpleScalar emulates the execution of a program and allows the simulation to execute down speculative paths of execution. This is critical for accurately modeling speculative execution and recovery techniques for many of the latest architecture features being studied in the field. Most researchers use a cycle level simulator similar to SimpleScalar, executing only

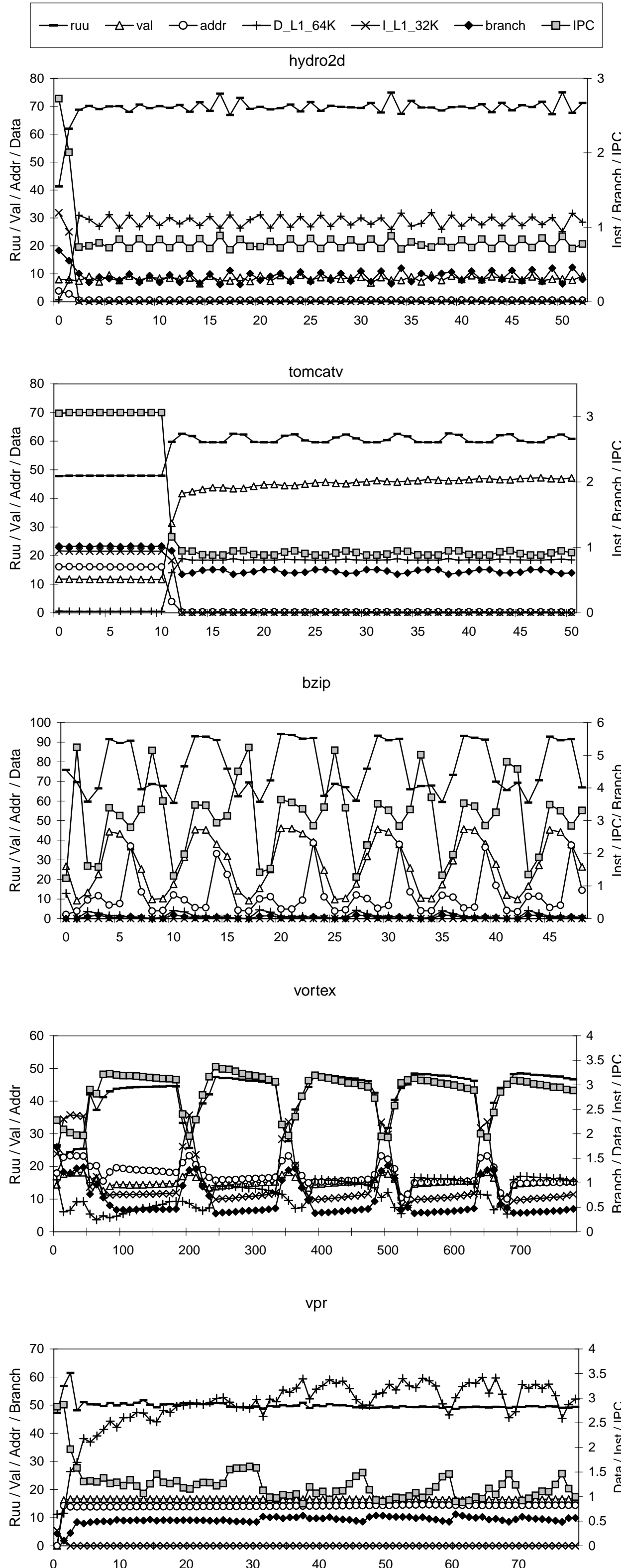

Figure 8: Time varying behavior for the programs `hydro2d`, `tomcatv`, `bzip`, `vortex`, and `vpr`. Each unit of the X-axis represents 100 million committed instructions.

a small fraction of the program. A few hundred million instructions may be typically executed, starting from a predetermined point.

In Table 2 we see the baseline behavior of the six programs for the architectural metrics discussed in section 4 for the complete run of the application. In addition to this we have included the initialization phase and the period duration (in 100 of millions of instructions) as determined by using the BBDA analysis discussed in section 3. We have found that to get the most representative sample of a program, at least one full period must be simulated.

5.1 Simulation Points

To evaluate the accuracy of the period length found using BBDA, we now compare the behavior of simulating for a single period to that of simulating the program's complete execution. We choose a preferred period as the simulation starting point by building a BB difference graph for each program with a BBV duration equal to the period length shown in Table 2. We then take the minimum point in this new BB difference graph as the the preferred period to simulate.

Table 3 compares the performance of several different metrics for the preferred period simulated and compares this with the baseline metrics shown in Table 2. The column labeled start is where the simulation was started from, and the simulation was ran for one complete period with the length shown in Table 2. For each of these experiments, cold start effects were eliminated by warming up the simulator with the full execution of the program to that point. The metrics examined are the same as examined in Section 4. In addition to this, associated with each metric is an error. The error is the percent difference between the metric measured over the preferred period we simulated versus the complete execution of the program.

The IPC values for the periods match very closely with the execution of the program as a whole. For all the programs there was less than a 5% difference between the IPC of the preferred simulation period and the full program execution. Most of the other metrics match up very closely as well. We show '–' for instruction cache error results for most of the programs, since the instruction cache miss rates were too low (below 0.05%) to represent any meaningful error.

The results for `vpr` show that we were able to capture its IPC within 4.3% when simulating 200 million instructions (one period) starting 74.6 billion instructions into the program. Even though there are different basic blocks executing in different proportions across the run of the application, the chosen sample is still very close to the execution as a whole.

5.2 Limited Simulation

Due to time constraints a researcher cannot typically simulate the whole program, but instead can simulate only for a few hundred million instructions, which is usually smaller

name	init	period	bpred	ruu	IPC	d miss	i miss	val miss	addr miss
bzip	2	9	4.2%	75.8%	2.681	1.7%	0.000%	25.1%	13.3%
hydro	5	17	0.4%	68.7%	0.793	14.6%	0.022%	8.3%	0.6%
tomcat	13	5	0.8%	59.6%	0.955	9.7%	0.043%	46.2%	1.0%
vortex	40	144	0.6%	43.4%	2.726	0.9%	0.979%	15.2%	16.4%
vpr	4	2	9.3%	49.8%	1.143	3.0%	0.001%	16.6%	14.2%
wave	68	70	0.6%	62.2%	2.596	7.4%	0.000%	38.1%	7.9%

Table 2: The first two columns show the length of the initialization phase and the size of the period in hundreds of millions of instructions. The average branch misprediction rate, ruu occupancy, instructions per cycle, data cache miss rate, instruction cache miss rate, value misprediction rate, and address miss prediction rate are also shown for the full run to completion.

than the period. To determine where to simulate given this constraint, we build a BB difference graph for each program with a BB vector duration of N, where N is the number of instructions in 100 of millions the user is willing to simulate. We then take the minimum point of that graph to represent the ideal simulation point.

Table 4 shows the effect of using only a limited amount of simulation time. Here we limit the amount of simulation time to only 300 million committed instructions, starting at the instruction, in 100 of millions, shown in the first column of Table 4. We can see that the error rate has gone up over that in Table 3. However, due to the fact that we have carefully selected our starting point with the algorithms presented in Section 3, the results we get are within acceptable bounds. The worst case IPC difference is 6%

The one program that does not do well with the smaller run size is `bzip`. For `bzip` the address miss rate and the value miss rate are off by around 80%. As our periodic results show, 900 million simulated instructions are needed to capture the small period in `bzip`, and simulating for 300 million instructions was simply too small to capture the behavior of the loop.

We now examine the performance of choosing our simulation point by picking it to be just after the initialization phase. Table 5 shows the same metrics as presented in Table 4 for a section of execution past the initialization stage by one period. The start of simulation is chosen to be the initialization time plus the time for one period. The intuition behind this is to simulate the soonest time past initialization, while still allowing for a full period of simulation to "warm up" the architectural structures such as the cache and branch predictor. In looking at Table 5, we see that using this scheme provides higher errors for important metrics such as IPC, branch prediction and data cache miss rates over using BBDA to find a preferred starting point as shown in Table 4.

6 Related Work

In this section we describe work related to finding simulation points, techniques for using sampling for simulation, and statistical simulation.

6.1 Time Varying Behavior of Programs

We presented in [10] a first attempt at showing the periodic patterns and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy. Skadron et. al [11] also examined creating similar time varying graphs for branch miss rates. They then used these graphs to manually choose where to fast-forward to pick out the simulation periods, similar as to what we proposed in [10].

6.2 Automatically Finding Where to Simulate

Concurrent to the work presented in this paper, Lafage and Seznec proposed an automated approach for choosing representative slices of a program's execution [6].

They propose a technique similar to [10] to gather statistics over the execution of the program to completion. There are two major differences. First, they propose to use metrics that are architecture independent to characterize the behavior of the program. They evaluate two such metrics, one which captures spatial locality and one which captures temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of the both the program's full execution and the execution of samples. The second point they propose is to use clustering and choosing algorithms to find a set of samples which captures the full execution of the program.

Our approach is cooperative in that the metrics and analysis we propose, and the clustering and choosing algorithms developed in [6] could be easily used together, and this is an area of future research.

6.3 Statistical Sampling

Our basic block distribution analysis accurately finds representive periods for simulation, but some of these periods are still too long for conducting detailed simulation studies. Therefore, in section 5 we examine choosing a few hundred million instructions to simulate from these long periods, and

name	start	bpred	err	ruu	err	IPC	err	data	err	inst	err	val	err	addr	err
bzip	150	4.2%	1%	75.4%	0.5%	2.8	5.1%	1.3%	25.8%	0.0%	–	25.4%	1.1%	15.7%	17.9%
hydro	6	0.3%	16%	69.8%	1.7%	0.8	2.5%	14.8%	1.5%	0.0%	–	8.2%	1.8%	0.6%	9.1%
tomcat	12	0.8%	3%	60.5%	1.5%	0.9	1.5%	9.8%	1.1%	0.0%	–	41.1%	12.4%	0.9%	17.1%
vortex	382	0.6%	2%	43.7%	0.8%	2.8	1.9%	0.9%	1.2%	1.0%	2.8%	15.2%	0.1%	16.3%	0.7%
vpr	746	9.0%	3%	49.7%	0.3%	1.2	4.3%	3.1%	6.4%	0.0%	–	16.6%	0.0%	14.4%	1.3%
wave	127	0.6%	9%	60.7%	2.5%	2.5	3.3%	7.7%	4.4%	0.0%	–	40.4%	6.1%	8.5%	7.8%

Table 3: Results for simulating one complete period through the application. The first column shows the starting instruction of the period simulated (in 100s of millions of instructions). The branch missprediction rate, ruu occupancy, instructions per cycle, data cache miss rate, instruction cache miss rate, value misprediction rate, and address miss prediction rate are shown for one complete cycle of the program's execution. Next to each of these columns is the percent difference between that metric for the period and the same metric for the full program execution.

name	start	bpred	err	ruu	err	IPC	err	data	err	inst	err	val	err	addr	err
bzip	1733	4.0%	6%	63.8%	18.8%	2.5	5.9%	1.8%	8.0%	0.0%	–	14.2%	77.2%	7.3%	82.0%
hydro	36	0.3%	12%	69.2%	0.9%	0.8	3.9%	14.8%	1.4%	0.0%	–	8.4%	0.4%	0.6%	9.3%
tomcat	144	0.8%	1%	60.9%	2.2%	1.0	1.9%	9.5%	2.0%	0.1%	–	39.8%	16.2%	1.1%	13.7%
vortex	330	0.6%	3%	41.9%	3.6%	2.8	3.4%	0.7%	16.3%	1.0%	4.0%	15.7%	3.8%	17.7%	7.7%
vpr	746	9.0%	3%	49.7%	0.3%	1.2	4.3%	3.1%	6.4%	0.0%	–	16.6%	0.0%	14.4%	1.3%
wave	1036	0.3%	84%	61.5%	1.2%	2.8	6.0%	7.9%	6.7%	0.0%	–	37.0%	2.8%	6.5%	20.8%

Table 4: The same metrics as presented in Table 3, but for an automatically chosen 300 million instruction simulation point. The error from comparing the sampled metric to the full execution of the program is listed next to each metric.

name	start	bpred	err	ruu	err	IPC	err	data	err	inst	err	val	err	addr	err
bzip	11	4.9%	17%	74.3%	2.0%	2.2	23.2%	2.8%	68.9%	0.0%	–	22.7%	10.8%	8.5%	55.4%
hydro	22	0.3%	12%	69.6%	1.4%	0.8	2.4%	14.8%	1.7%	0.0%	–	8.5%	1.8%	0.6%	8.6%
tomcat	18	0.6%	28%	61.0%	2.4%	0.9	4.6%	10.1%	5.1%	0.0%	–	44.0%	6.1%	0.3%	237%
vortex	184	0.4%	42%	46.4%	6.9%	3.2	17.6%	0.9%	6.1%	0.7%	36%	14.8%	2.3%	16.2%	1.6%
vpr	6	1.1%	740%	58.1%	16.6%	3.0	162%	0.4%	621%	0.0%	–	16.6%	0.2%	13.8%	2.6%
wave	138	0.9%	55%	60.5%	2.8%	2.4	9.1%	7.3%	1.1%	0.0%	–	40.1%	5.5%	7.7%	2.3%

Table 5: The same metrics as presented in Table 3 for a section of 300 million simulated instructions chosen one period after the end of the initialization phase. Next to each of these columns is the percent difference between that metric for the chosen simulation and the same metric for the full program execution.

we showed how close this approach comes to the overall execution of the applications we examined. Another approach is to use sampling simulation inside of a representative period found using BDDA in order to maintain accuracy while reducing simulation time.

Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These techniques take a number of contiguous execution samples, referred to as clusters in [3], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated.

To use sampling one has to address the issue of how to deal with the state of the machine when switching from one cluster to starting the simulation of another cluster. One option for providing meaningful results is to first sample a large number sequential instructions in order to provide results due to the time it takes to warm up the architecture structures (.e.g, caches) as well as taking a large number of samples to be sure to capture the large scale behavior of the program. Conte et.al. [3] proposed another option for the reconciliation of such disjoint sample points, whereby the structures holding state are not reset between clusters. For example, the cache would not be flushed and the BTB would not be reset when switching simulation from one cluster to the next. The hope is that the state of the machine from the end of one cluster is similar to the start of another disjoint cluster.

6.4 Statistical Simulation

Another technique to improve simulation time is to use statistical simulation such as that presented by Oskin et al. [8]. Using statistical simulation, the application is run once and a

synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. This technique could also benefit from Basic Block Distribution Analysis. First, by using BBDA there may not be a need to execute the programs to completion in the first place, a very time consuming step. Second, separate traces could be gathered for different phases, rather than trying to get one phase that represents the average behavior of application as a whole.

7 Summary

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program.

In this paper, we present Basic Block Distribution Analysis as an automated approach for finding where to simulate in order to achieve an accurate estimate of the complete program. The basic block distribution of the program's entire execution can be gathered quickly and efficiently using a basic block or edge profiler, with no need for cycle accurate simulation. The basic block distribution we form from this profile acts as a fingerprint for the whole program's behavior. This fingerprint is then used to automatically find the end of the initialization phase and the period duration for the programs we examine. We then quantify and show that basic block distribution analysis is highly correlated with architectural metrics including IPC, branch miss rate, cache miss rates, value misprediction, address misprediction, and reorder buffer occupancy.

Our results show that if we simulate the application for one complete period that the IPC error rates are 5% or less for the programs we examine. We further show that if we are constrained to only 300 million instructions of simulation time that the most representative instructions are not necessarily found right after the initialization phase, but rather typically straddle the transition from one phase to the next. Using basic block distribution analysis, we show that it is possible to find these small representative sections of the program, which result in an error in IPC of 6% or less.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045, and a grant from Compaq Computer Corporation.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction Level Parallelism*, May 2000.

[3] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[4] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[5] J. Gonzalez and A. Gonzalez. Memory address prediction for data speculation. Technical report, Universitat Politecnica de Catalunya, 1996.

[6] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[7] S. McFarling and J. Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. Association for Computing Machinery, 1986.

[8] M. Oskin, F. T. Chong, and M. Farrens. Hls: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[9] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, 1998.

[10] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[11] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–81, November 1999.

[12] G.S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[13] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.