SimPoint: Picking Representative Samples to Guide Simulation

Brad Calder Timothy Sherwood Greg Hamerly Erez Perelman

0.1 Introduction

Understanding the cycle level behavior of a processor during the execution of an application is crucial to modern computer architecture research. To gain this understanding, researchers typically employ detailed simulators that model each and every cycle. Unfortunately, this level of detail comes at the cost of speed, and simulating the full execution of an industry standard benchmark can take weeks or months to complete, even on the fastest of simulators. Exacerbating this problem further is the need of architecture researchers to simulate each benchmark over a variety of different architectural configurations and design options, to find the set of features that provides the appropriate tradeoff between performance, complexity, area, and power. The same program binary, with the exact same input, may be run hundreds or thousands of times to examine how, for example, the effectiveness of a given architecture changes with its cache size. Researchers need techniques which can reduce the number of machine-months required to estimate the impact of an architectural modification without introducing an unacceptable amount of error or excessive simulator complexity.

Executing programs have behaviors that change over time in ways that are not random, but rather are often structured as sequences of a small number of reoccurring behaviors, which we call phases. This structured behavior is a great benefit to simulation. It allows very fast and accurate sampling by identifying each of the repetitive behaviors and then taking only a single sample of each repeating behavior to represent that behavior. All of these representative samples together represent the complete execution of the program. This is the underlying philosophy of the tool called SimPoint [16, 17, 14, 2, 9, 8]. SimPoint intelligently chooses a very small set of samples called *Simulation Points* that, when simulated and weighed appropriately, provide an accurate picture of the complete execution of the program. Simulating only these carefully chosen simulation points can save hours of simulation time over statistically random sampling, while still providing the accuracy needed to make reliable decisions based on the outcome of the cycle level simulation. This chapter shows that repetitive phase behavior can be found in programs and describes how SimPoint automatically finds these phases and picks simulation points.

0.2 Defining Phase Behavior

Since phases are a way of describing the reoccurring behavior of a program executing over time, let us begin the analysis of phases with a demonstration of the time-varying behavior [15] of two different programs from SPEC 2000, `gcc` and `gzip`. To characterize the behavior of these programs we have simulated their complete execution from start to finish. Each program executes many billions of instructions, and gathering these results took several machine-months of simulation time. The behavior of each program is shown in Figures 1 and 4. Each top figure shows how the CPI changes for these two programs over time. Each point on the graph represents the average value for CPI taken over a window of 10 million executed instructions (which we call an interval). These graphs show that the average behavior does not sufficiently characterize the behavior of the programs.

Note that not only do the behaviors of the programs change over time, they change on the largest of time scales and even here we can find repeating behaviors. The programs may have stable behavior for billions of instructions and then change suddenly. In addition to performance, we have found for the SPEC 95 and 2000 programs that the behavior of all of

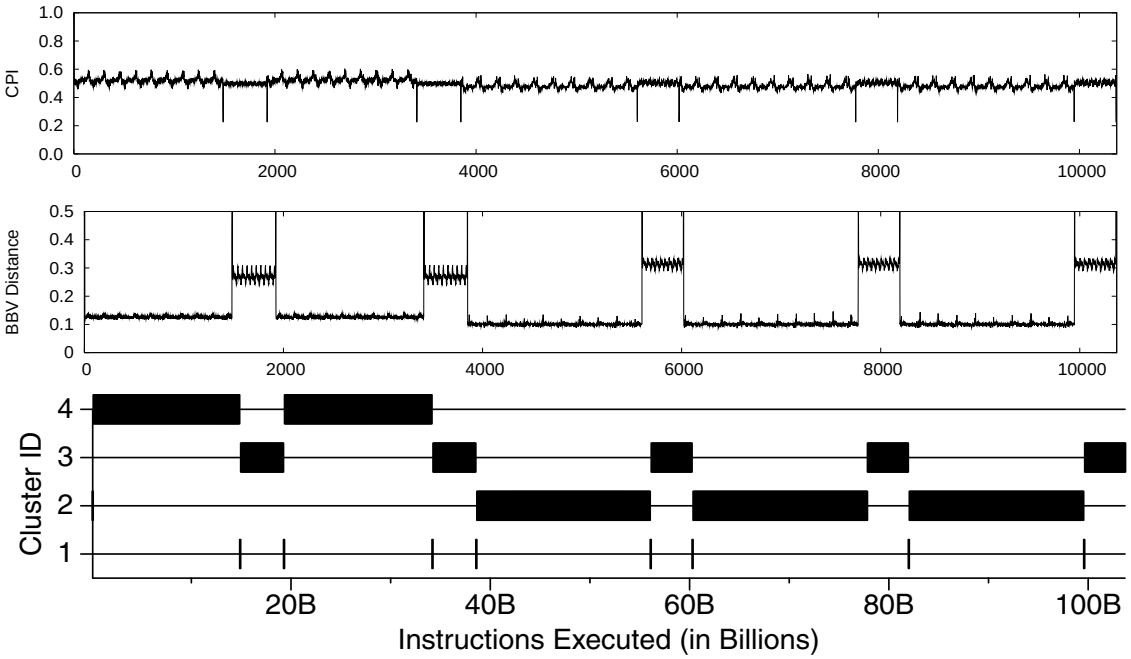

Figure 1: (top graph)Time varying graphs for CPI from each interval of execution for `gzip-graphic` at 10 million interval size. The x-axis represents the execution of the program over time. The results are non-accumulative.

Figure 2: (middle graph) Time varying graph showing the distance to the target vector from each interval of execution in `gzip-graphic` for an interval size of 10 million instructions. To produce the target vector, we create a basic block vector treating the whole program as one interval. The target vector is a signature of the program's overall behavior.

Figure 3: (bottom graph) Shows which intervals during the program's execution are partitioned into the different phases as determined by the SimPoint phase classification algorithm. The full run of execution is partitioned into a set of 4 phases.

the architecture metrics (branch prediction, cache misses, etc...) tend to change in unison, although not necessarily in the same direction [15, 17]. This change in unison is due to an underlying change in the program's execution, which can have drastic changes across a variety of architectural metrics. The underlying methodology used in this chapter is the ability to automatically identify these underlying program changes without relying on architectural metrics to group the program's execution into phases. To ground our discussions in a common vocabulary, the following is a list of definitions that are used in this chapter to describe program phase behavior and it's automated classification.

- Interval - A section of continuous execution (a slice in time) of a program. For the results in this chapter all intervals are chosen to be the same size, as measured in the number of instructions committed within an interval (either 1, 10, or 100 million instructions [14]). All intervals are assumed to be non-overlapping, so to perform our analysis we break a program's execution up into contiguous non-overlapping fixed length intervals.
- Similarity - Similarity defines how close the behavior of two intervals are to one another as measured across some set of metrics. Well formed phases should have intervals with

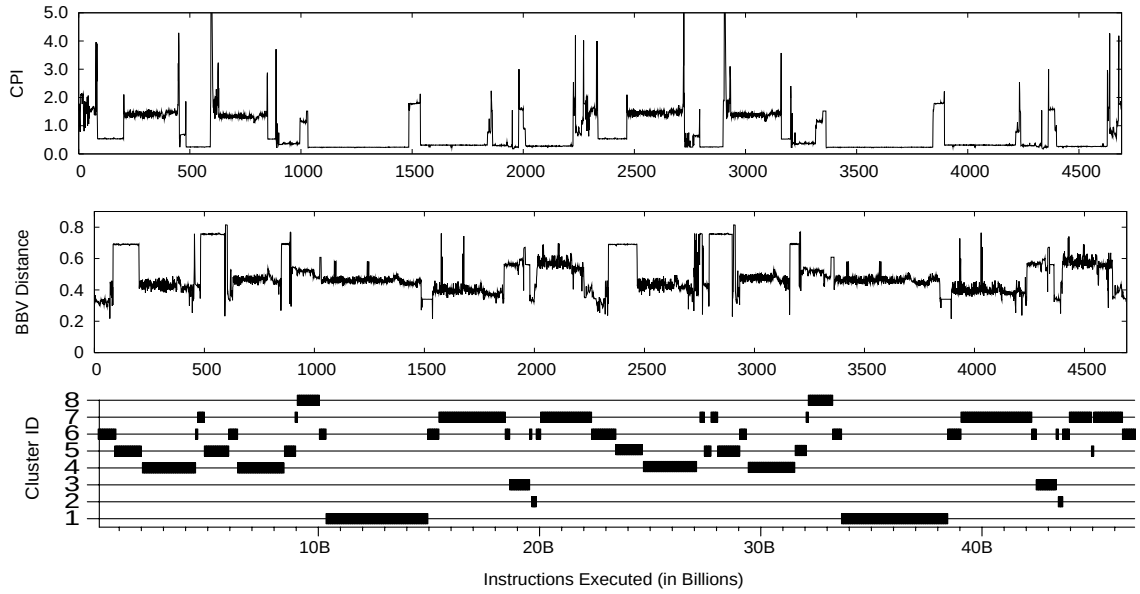

Figure 4: (top graph)Time varying graphs for CPI from each interval of execution for `gcc-166` at 10 million interval size. The x-axis represents the execution of the program over time. The results are non-accumulative.

Figure 5: (middle graph) Time varying graph showing the distance to the target vector from each interval of execution in `gcc-166` for an interval size of 10 million instructions. To produce the target vector, we create a basic block vector treating the whole program as one interval. The target vector is a signature of the program's overall behavior.

Figure 6: (bottom graph) Shows which intervals during the program's execution are partitioned into the different phases as determined by the SimPoint phase classification algorithm. The full run of execution is partitioned into a set of 8 phases.

similar behavior across various architecture metrics (e.g. IPC, cache misses, branch misprediction).

- Phase - A set of intervals within a program's execution that have all behavior similar to one another, regardless of temporal adjacency. In this way a phase can consist of intervals that re-occur multiple times (repeat) through the execution of the program (as can be seen in `gzip` and `gcc`).

- Phase Classification - Phase classification breaks up a program/input's set of intervals on into phases with similar behavior. This phase behavior is for a specific program binary running a specific input (a binary/input pair).

0.3 The Strong Correlation Between Code and Performance

As we mentioned in the prior section, for an automated phase analysis technique to be applicable to architecture design space exploration, we must be able to directly identify the

underlying changes taking place in the executing program. This section is a description of techniques which have been shown effective at accomplishing this.

0.3.1 Using an Architecture Independent Metric for Phase Classification

To find phase information, any effective technique requires a notion of how similar two parts of the execution in a program are to one another. In creating this *similarity* metric it is advantageous not to rely on statistics such as cache miss rates or performance, since this would tie the phases to those statistics. If that was done, then the phases would need to be re-analyzed every time there is a change to some architecture parameter (either statically if the size of the cache changed, or dynamically if some policy is changed adaptively). This is not acceptable, since our goal is to find a set of samples that can be used across an architecture design space exploration. To address this, we need a metric that is *independent* of any particular hardware based statistic, yet it must still relate to the fundamental changes in behavior shown in Figures 1 and 4.

An effective way to design such a metric is to base it on the behavior of a program in terms of the code that is executed over time. There is a very strong correlation between the set of paths in a program that are executed and the time-varying architectural behavior observed. The intuition behind this is that the code being executed determines the behavior of the program. With this idea it is possible to find the phases in programs using *only* a metric related to how the code is being exercised (i.e. both what code is touched and how often). It is important to understand that this approach can find the same phase behavior shown in Figures 1 and 4 by examining only the frequency with which the code parts (e.g., basic blocks) are executed over time.

0.3.2 Basic Block Vector

The Basic Block Vector (or BBV) [16] is a structure designed to concisely capture information about how a program is changing behavior over time. A basic block is a section of code that is executed from start to finish with one entry and one exit. The metric for comparing two time intervals in a program is based on the differences in the frequency that each basic block is executed during those two intervals. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block distributions provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides a code signature for that interval of execution, and shows where the application is spending its time in the code. The basic idea is that knowing the basic block distribution for two different intervals gives two separate signatures which we can then compare to find out how similar the intervals are to one another. If the signatures are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

More formally, a *Basic Block Vector* is a one dimensional array, with one element in the array for each static basic block in the program. Each interval in an executed program gets one BBV, and at the beginning of each interval we start with a BBV containing all zeros.

During each interval, we count the number of times each basic block in the program has been entered (just during that interval), and record that number into the vector (weighed by the number of instructions in the basic block). Therefore, each element in the array is the count of how many times the corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. For example, if the 50th basic block has one instruction and is executed 15 times in an interval, then bbv[50] = 15 for that interval. The BBV is then normalized to 1 by dividing each element by the sum of all the elements in the vector.

We recently examined frequency vector structures other than basic block vectors for the purpose of phase classification. We have looked at frequency vectors for data, loops, procedures, register usage, instruction mix, and memory behavior [9]. We found that using register usage vectors, which simply counts for a given interval the number of times each register is defined and used, provides similar accuracy to using basic block vectors. In addition, tracking only loop and procedure branch execution frequencies performed almost as well as using the full basic block information. We also found, for SPEC 2000 programs, that creating data vectors or combined code and data vectors did not improve classification over just using code [9].

0.3.3 Basic Block Vector Difference

In order to find patterns in the program we must first have some way of comparing the similarity of two Basic Block Vectors. The operation needed takes as input two Basic Block Vectors, and outputs a single number corresponding to how similar they are.

We use BBVs to compare the intervals 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 executed during that interval [16]. We use the basic block vectors as signatures for each interval of execution: each vector tells us what portions of code are executed, and how frequently those portions of code are executed. By comparing the BBVs of two intervals, we can evaluate the similarity of the two intervals. If two intervals have similar BBVs, then the two intervals spend about the same amount of time in roughly the same code, and therefore we expect the performance of those two intervals to be similar.

There are several ways of comparing two vectors to one another, such as taking the dot product or finding the Euclidean or Manhattan distance.

The Euclidean distance, which has been shown to be effective for off-line phase analysis [17, 14], can be found by treating each vector as a single point in a D-dimensional space, and finding the straight-line distance between the two points. More formally, the Euclidean distance of two vectors a and b in D-dimensional space is given by:

$$EuclideanDist(a, b) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

The Manhattan distance on the other hand is the distance between two points if the only paths followed are parallel to the axes, and is more efficient for on-the-fly phase analysis [18, 10]. In two dimensions, this is analogous to the distance traveled by a car in a city through

a grid of city streets. This has the advantage that it always gives equal weight to each dimension. The Manhattan distance is computed by summing the absolute value of the element-wise subtraction of two vectors. For vectors a and b in D-dimensional space, the distance is:

$$ManhattanDist(a, b) = \sum_{i=1}^{D} |a_i - b_i|$$

0.3.4 Showing the Correlation Between Code Signatures and Performance

A detailed study showing that there is a strong correlation between code and performance can be found in [8]. The graphs in Figures 4 and 5 give one representation of this by showing the time-varying CPI and BBV distance graphs for `gcc-166` right next to each other. The time-varying CPI graph plots the CPI for each interval executed (at 10M interval size) showing how the program's CPI varies over time. Similarly, the BBV distance graph plots for each interval the Manhattan distance of the BBV (code signature) for that interval from the whole program target vector. The whole program target vector is the BBV if the whole program is viewed as a single interval. The same information is also provided for `gzip` in Figures 1 and 2. The time-varying graphs show that changes in CPI have corresponding changes in code signatures, which is one indication of strong phase behavior for these applications. These results show that the BBV can accurately track the changes in CPI for both `gcc` and `gzip`. It is easy to see that over time the CPI changes accurately mirror changes visible in the BBV distance graph.

These plots show that code signatures have a strong correlation to the changes in CPI even for complex programs like `gcc`. The results for `gzip` show that the phase behavior can be found even if the intervals' CPIs have small variance. This brings up an important point about picking samples for simulation based on code vectors versus CPI or some other hardware metric. Assume we have two intervals with *different code signatures* but they have very *similar CPIs* because both of their working sets fit completely in the cache. During a design space exploration search, as the cache size changes, the two interval CPIs may differ drastically since one of them no longer fits into the cache. This is why it is important to perform the phase analysis by comparing the code signatures independent of the underlying architecture, and not based upon CPI thresholds. We have found that the BBV code signatures correctly identify this difference, which cannot be seen by looking at just the CPI. If the purpose of a study is to perform design space exploration it is important to be able to pick samples that will be representative of the program's execution no matter the underlying architecture configuration. See [8], for a complete discussion and analysis on the strong correlation between code and performance.

0.4 Automatically Finding Phase Behavior

Frequency vectors (BBVs, Vectors based on the execution of loops and procedures, or some other behavior discussed in [9]) provide a compact and representative summary of a program's behavior for each interval of execution. By examining the similarity between them, it is clear that there are high-level patterns in each program's execution. In this section we describe the algorithms used to automatically detect these patterns.

0.4.1 Using Clustering for Phase Classification

It is extremely useful to have an automated way of extracting phase information from programs. Clustering algorithms from the field of machine learning have been shown to be very effective [17] at breaking the complete execution of a program into phases that have similar frequency vectors. Because the frequency vectors correlate to the overall performance of the program, grouping intervals based on their frequency vectors produces phases that are similar not only in the distribution of program structures used, but also in every other architecture metric measured, including overall performance.

The goal of clustering is to divide a set of points into groups, or clusters, such that points within each cluster are similar to one another (by some metric, usually distance), and points in different clusters are different from one another. The k-means algorithm [11] is an efficient and well-known clustering algorithm, which we use to quickly and accurately split program behavior into phases. We use random linear projection [5] to reduce the dimension of the input vectors while preserving the underlying similarity information, which speeds up the execution of k-means. One drawback of the k-means algorithm is that it requires the number of clusters k as an input to the algorithm, but we do not know beforehand what value is appropriate. To address this, we run the algorithm for several values of k, and then use a goodness score to guide our final choice for k.

Taking this to the extreme, if every interval of execution is given its very own cluster, then every cluster will have perfect homogeneous behavior. Our goal is to choose a clustering with a *minimum number of clusters* where each cluster has a certain level of homogeneous behavior.

The following steps summarize the phase clustering algorithm at a high level. We refer the interested reader to [17] for a more detailed description of each step.

1. Profile the program by dividing the program's execution into contiguous intervals of size N (e.g., 1 million, 10 million, or 100 million instructions). For each interval, collect a frequency vector tracking the program's use of some program structure (basic blocks, loops, register usage, etc.). This generates a frequency vector for every interval. Each frequency vector is normalized so that the sum of all the elements equals 1.
2. Reduce the dimensionality of the frequency vector data to D dimensions using random linear projection. The advantage of performing clustering on projected data is that it speeds up the k-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original vectors, while preserving the essential similarity information.
3. Run the k-means clustering algorithm on the reduced dimensional data with values of k from 1 to K, where K is the maximum number of phases that can be detected. Each run of k-means produces a clustering, which is a partition of the data into k different phases/clusters. Each run of k-means begins with a random initialization step, which requires a random seed.
4. To compare and evaluate the different clusters formed for different k, we use the *Bayesian Information Criterion* (BIC) [13] as a measure of the "goodness of fit" of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the higher the BIC score, the higher the probability that the clustering is a good fit to the data. For each

clustering ($k = 1 \ldots M$), the fitness of the clustering is scored using the BIC formulation given in [13].

5. The final step is to choose the clustering with the smallest k, such that its BIC score is at least $B\%$ as good as the best score. The clustering k chosen is the final grouping of intervals into phases.

The above algorithm groups intervals into phases. We use the Euclidean distance between vectors as our similarity metric. This algorithm has several important parameters (N, D, K, B, and more) which must be tuned to create accurate and representative simulation points using SimPoint. We discuss these parameters in more detail later in this chapter.

0.4.2 Clusters and Phase Behavior

Figures 3 and 6 show the result of running the clustering algorithm on `gzip` and `gcc` using an interval size of 100 million and setting the maximum number of phases (K) to 10. The x-axis corresponds to the execution of the program in billions of instructions, and each interval is tagged to be in one of the clusters (labeled on the y-axis).

For `gzip`, the full run of the execution is partitioned into a set of 4 clusters. Looking at Figure 2 for comparison, the cluster behavior captured by the off-line algorithm lines up quite closely with the behavior of the program. Clusters 2 and 4 represent the large sections of execution which are similar to one another. Cluster 3 captures the smaller phase that lies in between these larger phases. Cluster 1 represents the phase transitions between the three dominant phases. The cluster 1 intervals are grouped into the same phase because they execute a similar combination of code, which happens to be part of code behavior in either cluster 2 or 4 and part of code executed in cluster 3. These transition points in cluster 1 also correspond to the same intervals that have large cache miss rate spikes seen in the time-varying graphs of Figure 1.

Figure 6 shows how `gcc` is partitioned into 8 different clusters. In comparing this Figure to Figure 4 and 5, we see that even the more complicated behavior of `gcc` is captured correctly by SimPoint. The dominant behaviors in the time-varying CPI and vector distance graphs can be seen grouped together in the dominant phases 1, 4 and 7.

0.5 Choosing Simulation Points from the Phase Classification

After the phase classification algorithm described in the previous section has done its job, intervals with similar code usage will be grouped together into the same phase, or cluster. Then from each phase, we choose one representative interval that will be simulated in detail to represent the behavior of the whole phase. Therefore, by simulating *only* one representative interval per phase, we can extrapolate and capture the behavior of the entire program.

To choose a representative, SimPoint picks the interval that is closest to the center of each cluster. The center is the average of all the intervals in the cluster, and is called the *centroid*. This is analogous to the balance point of all the points that are in that cluster,

if all points had the same mass. It can also be viewed as the interval which behaves most like the average behavior of the entire phase. Most likely there is no interval that exactly matches the centroid, so the interval closest to the center is chosen. The selected interval is called a *Simulation Point* for that phase [14, 17]. Detailed simulation is then performed on the set of simulation points.

SimPoint also gives a weight for each simulation point. Each weight is a fraction; it is the total number of instructions counting all of the intervals in the cluster, from which the simulation point was taken, divided by the number of instructions in the program. With the weights and the detailed simulation results of each simulation point, we compute a weighted average for the architecture metric of interest (CPI, miss rate, etc). This weighted average of the simulation points gives an accurate representation of the complete execution of the program/input pair.

0.6 Using the Simulation Points

After the SimPoint algorithm has chosen a set of simulation points and their respective weights, they can be used to accurately estimate the full execution of a program. The next step is to simulate in detail the interval for each simulation point, to collect the desired performance statistics.

0.6.1 Simulation Point Representation

SimPoint provides the simulation points in two forms:

Simulation Point Interval Number – The interval number for each simulation point is given. The interval numbers are relative to the start of execution, not to the previous simulation point. To get the start of a simulation point, subtract 1 from the interval number, and multiply by the interval size. For example, interval number 15 with an interval size of 10 million instruction means that the simulation point starts at instruction 140 million (i.e. (15-1)*10M) from the start of execution. Detailed simulation of this simulation point would occur from instruction 140 million until just before 150 million.

Start PC with Execution Count – SimPoint also provides for each simulation point the program counter for the first instruction executed for the interval and the number of times that instruction needs to be executed before starting simulation. For example, if the PC is 0x12000340 with an execution count of 1000, then detailed simulation starts the 1000th time that PC is seen during execution, and simulation occurs for the length of the profile interval.

It is highly recommended that you use the simulation point PCs for performing your simulations. There are two reasons for this. The first reason deals with making sure you calculate the instructions during fast-forwarding exactly the same as when the simulation points were gathered. The second reason is that there can be slight variations in execution count between different runs of the same binary/input due to environment variables or operating system variations when running on a cluster of machines. Both of these are discussed in more detail later in this chapter.

0.6.2 Getting to the Starting Sample Image

After choosing the form of simulation points to use, each simulation point is then simulated. Two standard approaches for doing this are to use either fast-forwarding or checkpointing.

Fast-Forwarding – Sort the simulation points in chronological order. Fast-forward to the start of the first simulation point. Simulate at the desired detail for the size of the interval. Repeat these steps, fast-forwarding from one point to the next combined with detailed simulation, until all simulation intervals have been collected.

Checkpointing Starting Sample Image – One advantage of SimPoint is that the state of a program can be checkpointed (e.g., using SimpleScalar's checkpoint facility) right before the start of each simulation point. This checkpointing allows parallel simulation of all of the simulation points at once.

Reduced Checkpoints – Checkpointing is used to obtain the startup image size of the sample to be simulated. A technique proposed in [1] examines only storing the memory words accessed in the simulation point to create a reduced checkpoint. This results in two orders of magnitude less storage then full checkpointing, and significantly faster simulation.

0.6.3 Warmup

Using small interval sizes for your simulation points requires having an approach for warming up the architecture state (e.g., the caches, TLBs, and branch predictor). The following are some standard approaches for dealing with warmup.

No Warmup – If a large enough interval size is used (e.g., larger than 100 million instructions), no warmup may be necessary for many programs. This is the approach used by Intel's PinPoint for simulation [12]. They simulate intervals of size 500 million instructions so they do not have to worry about any warmup issues. They chose to go the SimPoint route with large interval sizes because of the complexity of integrating statistical simulation and warmup into their detailed cycle accurate simulator.

Assume Hit (Remove Cold Structure Misses) – All of the large architecture structures (e.g., cache, branch predictors) make use of a warmup bit that indicates when the first time an entry (e.g., cache block) in that structure is used. If it is the first time, the access is assumed to be a hit or a correct prediction, since most programs have low miss rates. One can also use a miss rate percentage (e.g., 10%) for these cold structure misses, randomly assuming some percentage of the cold start accesses are misses. This a very simple method that provides fairly accurate warmup state, since the miss rates for these structures are usually fairly low [19, 7].

Stale State – This is a method of not resetting the architecture structures between simulation points, and instead they are used in the state they were in at the end of the prior simulation point we just fast-forwarded from [4].

Calculated Warmup – One can calculate the working set of the most recently accessed data, code and branch addresses before a simulation point. Then start the simulation of architectural components W instructions before the simulation point, where W is large

enough to capture the working set size held by the architecture structures. After these W instructions are simulated, all statistics are reset and detailed simulation starts at that point. The goal of this approach is to bring the working set back into the architecture structures before starting the detailed simulation [3, 6].

Continuously Warm – This approach continuously keeps the state of certain architecture components warm (e.g., caches) even during fast-forwarding [20]. This is feasible if an infrastructure provides fast functional and structure simulation during fast-forwarding. Keeping the cache structures warm will increase the time it takes to perform fast-forwarding, but it is very accurate.

Architecture Structure Checkpoint – An architecture checkpoint is the checkpoint of the potential contents of the major architecture components (caches, branch predictors, etc) at the start of the simulation point [1]. This can be used to significantly reduce warmup time, since warmup consists of just reading the architecture structure checkpoint from the file and using it to initialize the architecture structures.

If you decide to use small interval sizes, Calculated Warmup and Architecture Checkpointing provide the most accurate and efficient warmup, although we have found that for many programs Assume Hit and Stale State are fairly accurate.

0.6.4 Combining the Simulation Point Results

The final step in using SimPoint is to combine the weighted simulation points to arrive at an overall performance estimate for the program's execution. One cannot just use the standard mean for computing the overall miss rate, since we need to apply a weight to each sample.

Each weight represents the proportion of the total execution that belongs to its phase. The overall performance estimate is the weighted average of the set of simulation point estimates. For example, if we have 3 simulation points and their weights are [.22, .33, .45] and their CPIs are (CPI1, CPI2, CPI3), then the weighted average of these points is: $CPI = 0.22 * CPI1 + 0.33 * CPI2 + 0.45 * CPI3$

The weighted average CPI is the estimate of the CPI for the full execution.

0.6.5 Pitfalls to Watch for When Using Simulation Points

There are a few important potential pitfalls worth addressing to ensure accurate use of SimPoint's simulation points.

Calculating Weighted IPC – For IPC (instructions/cycle) we cannot just apply the weights as above. We first would need to convert all the simulated samples to CPI before computing the weighted average as above, and then convert the result back to IPC.

Calculating Weighted Miss Rates – To compute an overall miss rate, first we must calculate both the weighted average of the number of cache accesses, and the weighted average of the number of cache misses. Dividing the second number by the first gives the

cache miss rate. In general, care must be taken when dealing with any ratio because both the numerator and the denominator must be averaged separately and *then* divided.

Accurate Instruction Counts (No-ops) – It is important to count instructions exactly the same for the BBV profiles as for the detailed simulation, otherwise they will diverge. Note that the simulation points on the SimPoint website include only correct path instructions and the instruction counts include no-ops. Therefore, to reach a simulation point in a simulator, *every* committed instruction (including no-ops) must be counted.

System Call Effects – Some users have reported system call effects when running the same simulation points under slightly different OS configurations on a cluster of machines. This can result is slightly more or less instructions being executed to get to the same point in the program's execution, and if the number of instructions executed is used to find the simulation point this may lead to variations in the results. To avoid this, we suggest using the Start PC and Execution Count for each simulation point as described above. Another way to avoid variations in startup is to use checkpointing as described above.

0.6.6 Accuracy of SimPoint

We now show the accuracy of using SimPoint for the complete SPEC 2000 benchmark suite and their reference inputs. Figure 7 shows the simulation accuracy results using SimPoint for the SPEC 2000 programs when compared to the complete execution of the programs. For these results we use an interval size of 100 million and limit the maximum number of simulation points (clusters) to no more than 10 for the off-line algorithm. With the above parameters SimPoint finds 4 phases for `gzip`, and 8 for `gcc`. As described above, one simulation point is chosen for each cluster, so this means that a total of 400 million instructions were simulated for `gzip`. The results show that this results in only a 4% error in performance estimation for `gzip`. Note, if you desire lower error rates, you should use smaller interval sizes and more clusters as shown in [14].

For the non-SimPoint results, we ran a simulation for the same number of instructions as the SimPoint data to provide a fair comparison. The results in Figure 7 show that starting simulation at the start of the program results in a median error of 58% when compared to the full simulation of the program, whereas blindly fast forwarding for 1 billion instructions results in a median 23% IPC error. When using the clustering algorithm to create multiple simulation points we saw a median IPC error of 2%, and an average IPC error of 3%. In comparison to random sampling approaches, we have found that SimPoint is able to achieve similar error rates requiring significantly (5 times) less simulation (fast-forwarding) time [14]. In addition, statistical sampling can be combined with SimPoint to create a phase clustering that has a low per-phase variance [14]. Recently, using phase information has even been applied to create accurate and efficient simulation for multi-program workloads for Simultaneous Multithreading [2].

0.6.7 Relative Error During Design Space Exploration

The absolute error of a program/input run on one hardware configuration is not as important as tracking the change in metrics across different architecture configurations. There is a lot

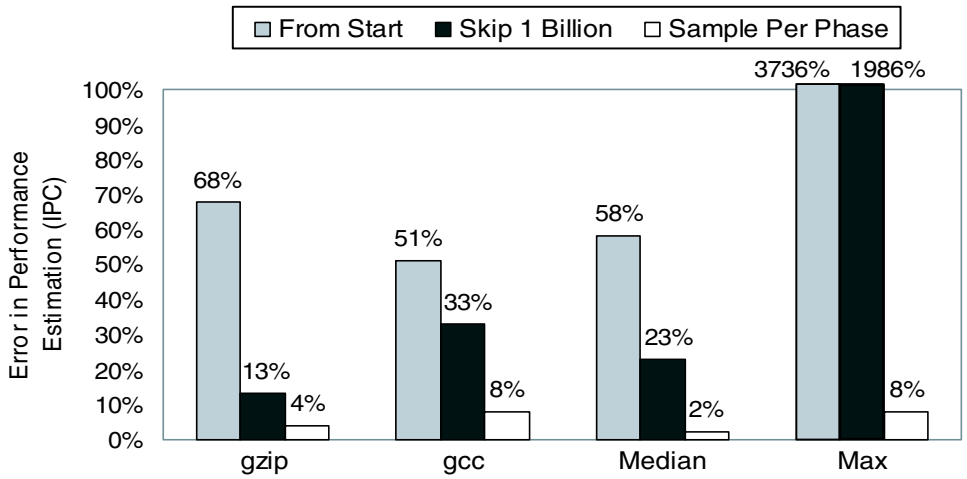

Figure 7: Simulation accuracy for the SPEC 2000 benchmark suite when performing detailed simulation for several hundred million instructions compared to simulating the entire execution of the program. Results are shown for simulating from the start of the program's execution, for fast-forwarding 1 billion instructions before simulating, and for using SimPoint to choose less than ten 100 million intervals to simulate. The median results are for the complete SPEC 2000 benchmarks.

of discussion and research into getting lower error rates. But what often is not discussed is that a low error rate for a single configuration is not as important as achieving the same relative error rates across the design space search and having them all biased in the same direction.

We now examine how SimPoint tracks the relative change in hardware metrics across several different architecture configurations. To examine the independence of the simulation points from the underlying architecture, we used the simulation points for the SimPoint algorithm with a 1 million interval size and max K set to 300. For the program/input runs we examine, we performed full program simulations while varying the memory hierarchy, and for every run we used the same set of simulation points when calculating the SimPoint estimates. We varied the configurations and the latencies of the L1 and L2 caches as described in [14].

Figure 8 shows the results across the 19 different architecture configurations for `gcc-166`. The left y-axis represents the performance in Instructions Per Cycle and the x-axis represents different memory configurations from the baseline architecture. The right y-axis shows the miss rates for the data cache and unified L2 cache, and the L2 miss rate is a local miss rate. For each metric, two lines are shown, one for the true metric from the *complete* detailed simulation for every configuration, and the second for the estimated metric using our simulation points. For each graph, the configurations on the x-axis are sorted by the IPC of the full run.

Figure 8 shows that the simulation points, which are chosen by only looking at code usage, can be used across different architecture configurations to make accurate architecture design trade-off decisions and comparisons. These results show that simulation points track the relative changes in performance metrics between configurations. One interesting observation

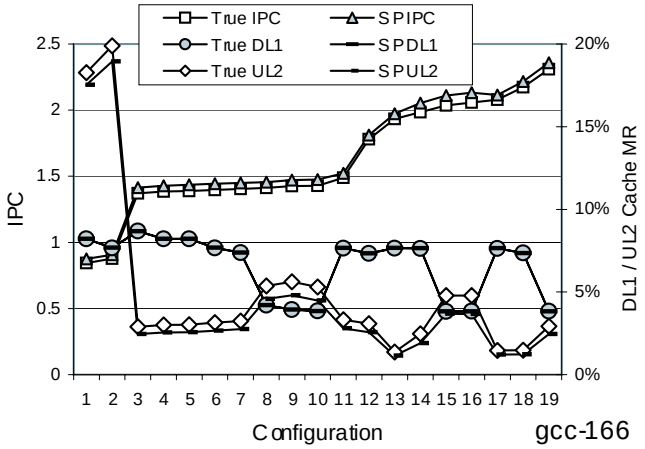

Figure 8: This plot shows the true and estimated IPC and cache miss rates for 19 different architecture configurations for the program `gcc`. The left y-axis is for the IPC and the right y-axis is for the cache miss rates for the L1 data cache and unified L2 cache. Results are shown for the complete execution of the configuration and when using SimPoint.

is that although the simulation results from SimPoint have a bias in the metrics, this bias is consistent and always in the same direction across the different configurations for a given program/input run. This is true for both IPC and cache miss rates. One reason for this bias is that SimPoint chooses the most representative interval from each phase, and intervals that represent phase change boundaries may (if they occur enough) or may not (if they do not occur enough) be represented by a simulation point.

0.7 Discussion About Running SimPoint

The SimPoint toolkit implements the algorithms described in this chapter. There are a variety of parameters which can be tuned when running the tool to create simulation points for new benchmarks, architectures, or inputs. In this section, we describe these parameters and discuss how they may be adjusted to meet your simulation needs.

Size of interval – The number of instructions per interval is the granularity of the algorithm. The interval size directly relates to the number of intervals, since the dynamic program length is the number of intervals times the interval size. Larger intervals allow more aggregate profile (basic block vector) representations of the program, while smaller intervals allow for more fine-grained representations. The interval size affects the number of simulation points; with smaller intervals more simulation points are needed than when using larger intervals to represent the same proportion of the program. Perelman [14] showed that using smaller interval sizes (1 million or 10 million) results in more accuracy when using SimPoint. The disadvantage is that with smaller interval sizes warmup becomes more of an issue, whereas with larger interval sizes warmup is not as much of an issue and may be preferred for some simulation environments [12].

Number of intervals – There should be a fair number of intervals for the clustering algorithm to choose from. A good rule of thumb is to make sure to use at least 1,000 intervals in order for the clustering algorithm to be able to find a good partition of the intervals. If there are too few intervals, then decrease the interval size to obtain more intervals for clustering.

K – The maximum number of clusters (K), along with the interval size, represents the maximum amount of simulation time that will be needed when looking to choose simulation points. If SimPoint chooses a number of clusters that is close to the maximum allowed, then it is possible that K is too small. If this is the case and more simulation time is acceptable, it is better to double the K and re-run the SimPoint analysis.

Creating simulation points with SimPoint comes down to recognizing the tradeoff of accuracy for simulation time. If a user wants to place a low limit on the number of clusters to limit simulation time, SimPoint can still provide accurate results, but some intervals with differing behaviors may be clustered together as a result.

Random Seeds – The k-means clustering algorithm starts from a randomized initialization, which requires a random seed. It is well-known that k-means can produce very different results depending on its initialization, so it is good to use many different random seeds for initializing different k-means clusterings, and then allow SimPoint to choose the best clustering. We have found that in practice, using 5 to 7 random seeds works well.

Number of iterations – The k-means algorithm iterates either until it hits a maximum number of iterations or until it reaches a point where no further improvement is possible (whichever is less). In most cases 100 iterations is sufficient for the maximum number, but more may be required, especially if the number of intervals is very large compared to the number of clusters. A very rough rule of thumb is the number of iterations should be set to $\sqrt{N/k}$, where N is the number of intervals and k is the number of clusters.

Number of dimensions – SimPoint uses random linear projection to reduce the dimension of the clustered data, which dramatically reduces computational requirements while retaining the essential similarity information. SimPoint allows the user to define the number of dimensions to project down to. In our experiments we project down to 15 dimensions, as we have found that using it produces the same phases as using the full dimension. We believe this to be adequate for SPEC 2000 applications, but it is possible to test other values by looking at the consistency of the clusters produced when using different dimensions [17].

BIC percent – The BIC gives a measure of the goodness of the clustering of a set of data, and BIC scores can be compared for different clusterings of the same data. However, the BIC score is an approximation of a probability, and often increases as the number of clusters increase. This can lead to often selecting the clustering with the most clusters. Therefore, we look at the range of BIC scores, and select the score which attains some high percentage of this range (e.g. we use 90%). When the BIC rises and then levels off, this method chooses a clustering with the fewest clusters that is near the maximum value. Choosing a lower BIC percent would prefer fewer clusters, but at the risk of less accurate simulation.

0.8 Summary

Understanding the cycle level behavior of a processor running an application is crucial to modern computer architecture research, and gaining this understanding can be done efficiently by judiciously applying detailed cycle level simulation to only a few simulation points. The level of detail provided by cycle level simulation comes at the cost of simulation speed, but by targeting only one or a few carefully chosen samples for each of the small number of behaviors found in real programs, this cost can be reduced to a reasonable level.

The main idea behind SimPoint is the realization that programs typically only exhibit a few unique behaviors which are interleaved with one another through time. By finding these behaviors and then determining the relative importance of each one, we can maintain both a high level picture of the program's execution and at the same time quantify the cycle level interaction between the application and the architecture. The key to being able to find these phases in a efficient and robust manner is the development of a metric that can capture the underlying shifts in a program's execution that result in the changes in observed behavior. In this chapter we have discussed one such method of quantifying executed code similarity, and use it to find program phases through the application of statistical and machine learning methods.

The methods described in this chapter are distributed as part of SimPoint [14, 17]. SimPoint automates the process of picking simulation points using an off-line phase classification algorithm, which significantly reduces the amount of simulation time required. These goals are met by simulating only a handful of *intelligently* picked sections of the full program. When these simulation points are carefully chosen, they provide an accurate picture of the complete execution of a program, which gives a highly accurate estimation of performance. The SimPoint software can be downloaded at:

http://www.cse.ucsd.edu/users/calder/simpoint/

Acknowledgments

This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, a UC MICRO grant, and a grant from Intel and Microsoft.

References

[1] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for uniprocessor and simultaneous multithreading simulation. Technical Report UCSD-CS2004-0803, UC San Diego, November 2004.

[2] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[3] T. M. Conte, M. A. Hirsch, and W. W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, 1998.

[4] 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.

[5] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, 2000.

[6] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation.

In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[7] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[8] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[9] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[10] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, February 2005.

[11] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[12] 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 *International Symposium on Microarchitecture*, December 2004.

[13] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734, 2000.

[14] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

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

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, October 2002.

[18] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[19] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1991.

[20] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, June 2003.