The Strong Correlation Between Code Signatures and Performance

Jeremy Lau[†] Jack Sampson[†] Erez Perelman[†] Greg Hamerly[‡] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Computer Science, Baylor University
{jl,jsampson,eperelma,calder}@cs.ucsd.edu,{greg_hamerly}@baylor.edu

Abstract

A recent study [1] examined the use of sampled hardware counters to create sampled code signatures*. This approach is attractive because sampled code signatures can be quickly gathered for any application. The conclusion of their study was that there exists a fuzzy correlation between sampled code signatures and performance predictability. The paper raises the question of how much information is lost in the sampling process, and our paper focuses on examining this issue.*

We first focus on showing that there exists a strong correlation between code signatures and performance. We then examine the relationship between sampled and full code signatures, and how these affect performance predictability. Our results confirm that there is a fuzzy correlation found in recent work for the SPEC programs with sampled code signatures, but that a strong correlation exists with full code signatures. In addition, we propose converting the sampled instruction counts, used in the prior work, into sampled code signatures representing loop and procedure execution frequencies. These sampled loop and procedure code signatures allow phase analysis to more accurately and easily find patterns, and they correlate better with performance.

1 Introduction

In recent years, several studies have shown that there is a correlation between code and performance predictability, concentrating mainly on the SPEC2000 programs [2, 6, 7, 8, 10, 13, 17, 18, 16, 3, 11, 14]. This analysis is based upon the fact that as a program executes its behavior is structured into repeating behaviors called phases. A phase is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. Phase analysis shows that both the code traversed and the underlying hardware performance tend to change at the same time. Therefore, one can capture a code signature for a phase, and then use the structure of the code to predict performance.

Recently, Annavaram et al. [1] examined the use of code signatures obtained through periodic sampling to predict performance for database applications and SPEC2000. The main difference between [1] and prior work is the type of code signatures used to perform the analysis. In [1], they propose using a tool called VTune to sample the hardware counters to create what we call *sampled code signatures*. The program counter is sampled once every N instructions executed, and N ranges from 100,000 instructions to one million instructions. Code signatures are formed for every interval of 10 million or 100 million consecutive instructions. They call these code signatures *Extended Instruction Pointer Vectors* (EIPVs). The advantage of this approach is that it can quickly create code signatures for any application running on the machine, and no binary instrumentation is required. The disadvantage is that the sampled code signatures do not cover all of the code executed, which may be necessary for predicting performance, and sampling introduces additional variabilities in instructions executed and CPI due to other processes running on the machine, including VTune itself.

Prior work determined that Basic Block Vectors (BBV) [8, 11] are one of the most accurate techniques for creating code signatures. A BBV for an interval represents how many times each basic block in the binary was executed during that interval. We call this a full code signature, since it accounts for every instruction - effectively, one sample is collected for every basic block. The advantage of this approach is that the code signatures accurately represent the complete execution path, weighted by each basic block's execution frequency. We gather BBVs through binary instrumentation or functional simulation. Compared to program counter sampling, the BBV approach requires more profiling overhead to gather code signatures, but the result is a lossless profile with no variability.

In [1], the authors concluded that there was a weak correlation between code signatures and performance predictability for some database applications. One possible explanation for this result is that the behavior of a database is highly data dependent; the queries issued determine what code gets executed, and the performance of each query is highly dependent on the contents of the query itself and the structure of the queried data. For these reasons, it is understandable that databases exhibit weaker correlations between code and performance.

However, the paper [1] also claims that many of the SPEC2000 benchmarks exhibit weak correlations between code signatures and performance. This claim contradicts the results found in other recent work, such as [8, 13, 16, 11]. One possible explanation for this contradiction is the use of sampled code signatures instead of full code signatures, but this issue was not addressed in [1]. The focus of our paper is to answer the questions raised in [1]: Do full code signatures exhibit a strong or weak correlation with performance, and what

is the relationship between sampled and full code signatures?

Our paper makes the following contributions:

- We present new results showing that there is a strong correlation between code and performance predictability for the SPEC2000 programs using full code signatures (basic block vectors [17]). We show this strong correlation through direct examination of the code signatures, and through off-line phase analysis.
- We show that there is a fuzzy relationship between EIPVs and BBVs. This is shown by (a) comparing the dimensionality of EIPVs and BBVs, (b) receiver operating characteristics, (c) coefficients of variation of CPI, and (d) SimPoint error rates.
- Finally we show that EIPVs can be significantly improved by mapping each EIP to its corresponding loop or procedure.

2 Full Code Signatures and Performance Predictability

The focus of this section is to examine the correlation between code signatures and performance for the full SPEC 2000 benchmark suite for off-line phase analysis. Most of this analysis is the foundation for our prior work [18, 16, 3, 11], but it has not been published before.

We first provide a brief summary of phase behavior and basic block vectors. We then summarize prior work that shows relationships between code signatures and performance. This section concludes with results quantifying the correlation between code and performance.

2.1 Phase Behavior

Programs exhibit large scale repeating behavior; we call this *phase behavior* [18]. To identify phases, we break a program's execution into contiguous non-overlapping intervals. An *interval* is a continuous portion of execution (a slice in time) of a program. For our studies we have used interval sizes of 1 million, 10 million and 100 million instructions [16]. A *phase* is a set of intervals within a program's execution with similar behavior, regardless of temporal adjacency. This means that a phase may appear many times as a program executes. *Phase classification* partitions a set of intervals into phases with similar behavior. The phases that we discover are specific to the input used to run the program.

The key observation for phase recognition is that any architectural metric is a function of the paths a program takes through its code. We can identify phase behavior and classify it by examining the proportions in which different regions of code are executed over time. Accurately capturing phase behavior by only examining program or ISA-level metrics, independent of the underlying architectural details and performance, allows us to partition a program's execution into architecture independent phases. This means that it is possible to use phase information for the same binary and input when performing a design space search, and to guide many optimizations and policy decisions across different architecture configurations.

2.2 Full Code Signatures – Basic Block Vectors

The first step of phase analysis is to collect the frequency distribution of executed code to create signatures that represent the program's behavior at different times in its execution. We perform clustering analysis on these code signatures to group similar parts of the program's execution into clusters based on the similarity of the signatures with SimPoint [18]. Each cluster is a phase.

Our approach uses the Basic Block Vector (BBV) [17] to represent the code signature in order to capture information about changes in a program's behavior over time. A basic block is a single-entry, single-exit section of code with no internal control flow. A *Basic Block Vector* is a one dimensional array, where each element in the array corresponds to one static basic block in the program. We start with a BBV containing all zeroes at the beginning of each interval. During each interval, we count the number of times each basic block in the program has been entered, and we record the count in the BBV for that interval. For example, if the 50th basic block is executed 15 times in an interval, then bbv[50] = 15 for that interval. In addition, we multiply each count by the number of instructions in the basic block, so basic blocks containing more instructions will have more weight in the BBV. Finally, at the end of each interval, we normalize the basic block vector by dividing each element by the sum of all the elements in the vector.

The behavior of the program at a given time is directly related to the code executed during that interval [17]. We perform clustering on BBVs, because each vector contains the frequency distribution of code executed in each interval. By comparing BBVs of two intervals, we can evaluate the similarity of two intervals. If the distance between the two BBVs is small (close to 0), 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. Code signatures grouped into the same cluster exhibit similar CPI, numbers of branch mispredictions, numbers of cache misses, etc.

2.3 Prior Work Relying on the Relationship Between Code and Performance

Dhodapkar and Smith [6, 7, 8] found a relationship between phases and instruction working sets, and found that phase changes tend to occur when the instruction working set changes. They track the instruction working set with bit vectors, with one bit for each basic block; a bit is set when the corresponding basic block is executed. They detect phase changes in hardware by comparing bit vectors. With their approach, multi-configuration units can be re-configured in response to phase changes. They use their working set analysis for instruction cache, data cache and branch predictor reconfigurations to save energy. In [8] they do a detailed comparison of BBVs with their bit vector approach for on-line

I Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Pred	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mem Disam	load/store queue, loads may execute when all prior store addresses are known
Func Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Mem	8k byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Alpha Simulation Model.

phase classification, and found that BBVs are very accurate. Their paper focused on the accuracy of on-line phase classification techniques, whereas this paper focuses on off-line phase classification. We use some of their analysis to examine the correlation between code signatures and performance.

Huang et al. [10] associate changes in program behavior with major subroutine changes. They show that code behavior is quite homogeneous within each major subroutine, and fairly heterogeneous across major subroutines. They show that dynamic reconfiguration on major subroutine boundaries effectively reduces power consumption. They also improve the accuracy of sampled simulation by examining call graphs to identify major subroutines for statistical simulation [13].

Patil et al. [14] use BBVs to characterize program behavior on Itanium systems. With one set of BBVs, they examine the relationship between BBVs and performance on three different implementations of the Itanium architecture. On average, the difference between actual performance and BBV predicted performance was less than 10%. The authors also show that BBVs can predict L3 misses and causes for stalls with reasonable accuracy.

In [17, 18], we proposed that phase behavior in programs can be automatically identified by profiling the code executed. We used techniques from machine learning to classify the execution of the program into phases (clusters). We found that intervals of execution grouped into the same phase exhibit similar behavior across all examined architecture metrics. We extended this approach to perform hardware phase classification and prediction [19, 12]. Our prior work is missing detailed analysis showing the strong correlation between code and performance predictability for our off-line phase analysis approach, which we focus on in this section.

2.4 Methodology

For the analysis in this section we simulated all of the SPEC 2000 benchmarks compiled for the Alpha ISA over multiple inputs. We provide results for 45 program/input combinations. Compiler flags and binaries can be found at `http://www.simplescalar.com/`. For these benchmarks, SimpleScalar `sim-outorder` was used for full detailed simulation of each program and collection of BBV code signatures. At every interval of execution, architecture statistics are printed along with a BBV. The baseline microarchitecture model we simulate is detailed in Table 1. We simulate an aggressive 8-way dynamically scheduled microprocessor with a two level cache. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

2.5 Correlation of Code Signatures with Performance

To show the relationship between code signatures and performance predictability, we use the approach of Dhodapkar et al. [8]. The idea is to evaluate the relationship between significant CPI changes and significant vector changes between every pair of consecutive intervals. Clearly this analysis depends on the definition of "significant," so we define two thresholds: ct, the CPI significance threshold, and vt, the vector significance threshold. These thresholds determine which intervals have significant CPI changes, and which intervals have significant code signature changes.

To determine if CPI_n for an interval is significantly different from the CPI_{n-1} for the previous interval, we determine if $|CPI_n - CPI_{n-1}|/CPI_{n-1} > ct$. If ct is .05, the current CPI is significantly different from the previous CPI if it differs by at least 5%, compared to the CPI of the last interval.

To determine if a vector V_n for an interval is significantly different from the vector V_{n-1} for the previous interval, we determine if $Manhattan\_distance(V_{n-1}, V_n)/2 > vt$. The Manhattan distance is the sum of the absolute values of pairwise differences as described in [18]. All vectors are normalized so their elements sum to 1, so the minimum Manhattan distance is 0, and the maximum Manhattan distance is 2. So if vt is .05, the current code signature significantly differs from the previous signature if it executes at least 5% of its code differently (either completely different code, or the same code in different proportions). For these experiments, we use full basic block vectors - the vectors are not projected to reduce their dimensionality, so each basic block vector contains the full basic block profile for each interval of execution.

Given a CPI significance threshold and a vector significance threshold, we look at three quantities: the number of true positives, the number of false positives, and the number of significant CPI changes. These quantities are described in detail below.

True positives are the fraction of intervals with significant CPI changes that are successfully detected by code signatures. To calculate the true positives, we count the number of intervals where both significant CPI change *and* significant vector difference are detected, and divide by the number of intervals with significant CPI change.

False positives are the fraction of intervals without significant CPI changes for which the code vectors *do* report a significant change. To calculate the false positives, we count the number of intervals where *no* significant CPI change is detected and significant vector difference *is* detected, and divide by the number of intervals with no significant CPI change. Note that a false positive is not necessarily a mistake in this case, since a program can perform two completely different

calculations that exhibit very different microarchitectural behavior, but happen to have very similar CPI. In addition, intervals with different code signatures, but similar CPI can exhibit diverse CPI if we change the architecture, as shown in Section 2.6. What is more important is that the intervals grouped together have similar performance, which we examine in Section 2.7.

In the extreme case, if vt is 0%, then almost every interval transition will appear to be a significant vector change (except when the vectors are exactly the same), so we expect nearly 100% true positives and nearly 100% false positives, since everything will appear different, regardless of the CPI significance threshold. At the other extreme, if vt is 100%, then no significant vector changes will ever appear, and we will have 0% true positives and 0% false positives.

Figure 2 shows the percentage of intervals that are considered significant CPI changes for 10 million interval size. Figure 2 shows that 20% of the interval transitions resulted in a CPI change greater than 10%, and only 5% of the program's interval transitions had a CPI change greater than 50%. Dhodapkar and Smith [8] used thresholds of 2% and 10% for ct, whereas we examine a much larger range of ct. We are interested in identifying large CPI changes in addition to smaller CPI changes, because applications such as SimPoint are more interested in large-scale program behavior.

Figure 1 shows the *Receiver Operating Characteristic* (ROC) [15] curves at a variety of CPI significance thresholds and vector significance thresholds. There is one data series for each CPI significance threshold, and each data point represents one vector significance threshold from $\{5\%, 10\%, 15\%, \ldots 95\%\}$ from right to left. Each point is the average over 45 benchmark and input pairs from SPEC2000. Results are shown for four CPI significance thresholds (ct) of 10%, 30%, 50% and 90%, when using a 10 million (10m) interval size.

Figure 1 shows the trade-off between true and false positives for different vector significance thresholds. With a CPI significance threshold (ct) of 50% and a BBV significance threshold (vt) of 35%, BBVs accurately track 90% (y-axis) of the interval transitions with at over 50% change in CPI. In addition, BBVs falsely detected a significant code change in 13% (x-axis) of the intervals with less than 50% change in CPI. This shows that it is possible to predict CPI changes with high accuracy (high true positives and low false positives) across a wide range of significance thresholds just by looking for changes in code signatures. The graph also shows that it is easier to predict larger CPI changes than smaller CPI changes.

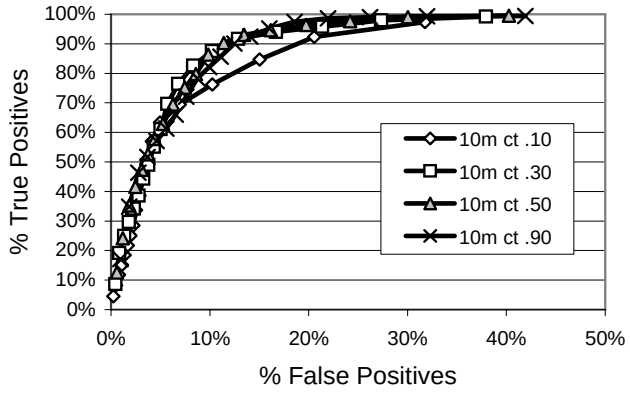

Figure 1: Receiver operating characteristic curves for different CPI significance thresholds and BBV significance thresholds. Each point represents a vector significance threshold decreasing from left to right. Results are shown for a 10 million interval size for four different CPI significance thresholds. "10m ct .10" means 10m granularity, and 10% CPI significance threshold.

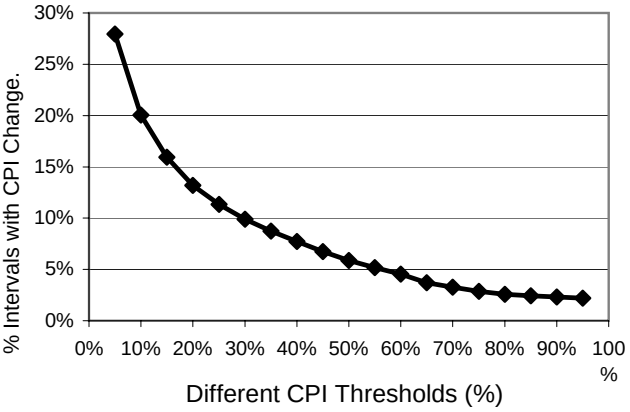

Figure 2: For each CPI significance threshold, we show the percentage of intervals with significant CPI change. Results are shown for a 10m interval size.

2.6 Phase Behavior in Programs with Small and Large CPI Variance

Annavaram et al. [1] classify benchmarks into four quadrants: (Q-I) for benchmarks with low CPI variance and weak phase behavior, (Q-II) for those with low CPI variance and strong phase behavior, (Q-III) for those with high CPI variance and weak phase behavior, and (Q-IV) for those with high CPI variance and strong phase behavior. For our use of phase analysis, it is not clear to us how meaningful it is to classify benchmarks based upon CPI variance, since the CPI variance changes depending on the architecture configuration examined and the interval size used. To illustrate this, Figure 3 shows the range of CPI variances across 18 different memory hierarchy configurations and interval sizes of 1, 10, 100, and 1000 million instructions. It is clear that `gcc-166` has a wide range of CPI variances which depend on machine configuration, and the CPI variance of `art-110` changes depending upon the interval size used.

In addition, [1] classifies `gzip` and `art` into Q-I, and `gcc` into Q-III, stating that they have weak phase behavior. The reasons we believe they did not find phase behavior are (a) their use of sampled code signatures, which we examine

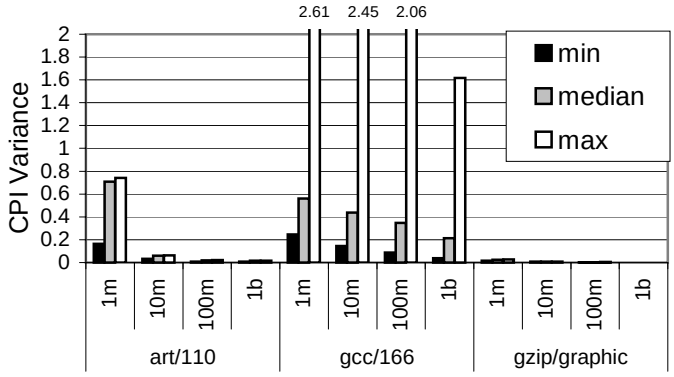

Figure 3: CPI Variance at different granularities, with different memory hierarchy configurations for 3 programs. Each program was run at the specified granularity on 18 different memory hierarchy configurations to completion in SimpleScalar. The architecture configuration results with minimum, median, and maximum CPI variance are shown for each program.

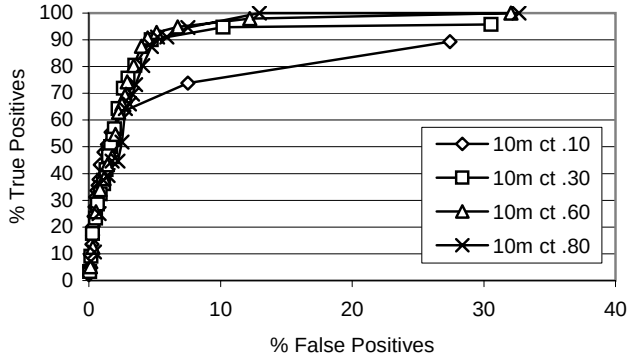

Figure 4: gcc-166 - Receiver operating characteristic curves for gcc-166 on the memory hierarchy configuration with maximum CPI variance from our 18 memory hierarchy configurations.

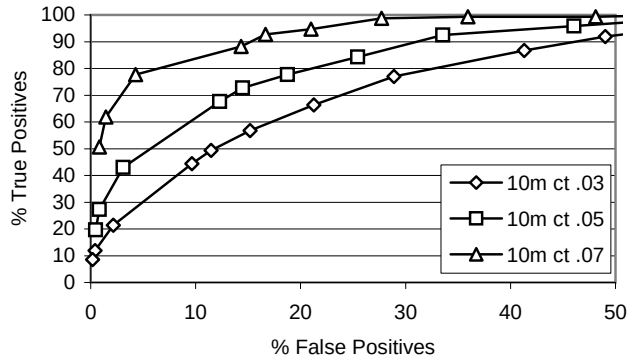

Figure 5: gzip-graphic - Receiver operating characteristic curves for gzip-graphic on the memory hierarchy configuration with minimum CPI variance from our 18 memory hierarchy configurations.

in more detail in Section 3, (b) if the overall CPI variance of a program is low, their approach will not detect many phases because they build their clusters based on CPI variance - in [1] they state that if CPI variance is already low, it does not make sense for them to split the execution into more clusters, (c) before performing their clustering, they filter out infrequently used dimensions - this can distort code signatures if the filtered out data was not evenly distributed across many intervals. In comparison, our analysis uses full code signatures for classification, without examining CPI data, and we use random projection [5] to reduce dimensionality.

We have found that it is important to project randomly instead of filtering out infrequently used dimensions, because dimensions that account for a small percentage of overall execution can be very important indicators of program behavior if they occur with high temporal locality. For example, if a program executes ten million instructions in `printf`, but the program runs for ten billion instructions, `printf` accounts for just 0.1% of the program's execution, but if the program only calls `printf` at the end of execution to display results, then those ten million instructions are highly indicative of the program's behavior - they should not be thrown away. If this information is discarded, it will be difficult to correctly classify the behavior of the program when it calls `printf`.

To illustrate some issues with classifying applications to specific quadrants, Figure 4 shows the ROC graph for `gcc-166` and Figure 6 shows `gcc-166`'s time varying CPI and BBV distance graphs. The time varying graph shows time in units of 10 million instructions executed on the x-axis, and CPI or BBV distance on the y-axis. The CPI time-varying graph shows how the `gcc`'s CPI changes over time. Similarly, the BBV distance graph plots the Manhattan distance from each vector to the whole program target vector. The whole program target vector is the BBV if the whole program is viewed as a single interval: it represents the program's overall basic block profile. The BBV distance to the target vector shows how much a program's code profile for a 10m slice of execution differs from its overall code profile. The same information is also provided for `gzip` in Figures 5 and 7. The time-varying graphs visibly show that changes in CPI have corresponding changes in code signatures, which indicates strong phase behavior for these applications.

Figure 4 shows that BBVs for `gcc-166` track CPI changes of 30% or more with 90% accuracy and only 5% false positives. This shows that code signatures have a *strong* correlation to CPI changes. This can also be seen in Figure 6 where you can see over time that the CPI changes are mirrored with a high accuracy by changes seen in the BBV distance graph.

In Figure 1 we used CPI thresholds of 10% to 80%, since those found the dominating phase behavior if one had to pick a standard set of thresholds. Figure 7 shows the CPI and BBV distance time varying graphs for `gzip-graphic` on the ar-

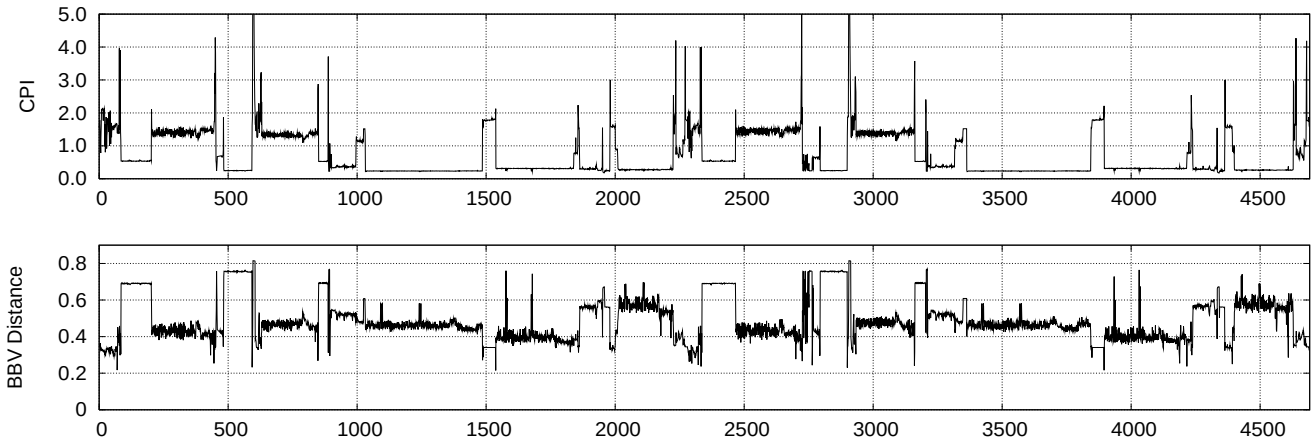

Figure 6: Time varying graphs for CPI and distance of the BBV to the target vector for each interval of execution in gcc-166 at 10m granularity. To produce the target vector, we sum all the BBVs, and normalize the counts so they add up to 1. The target vector is a profile of the program's overall behavior. The x-axis shows execution time, in tens of millions of instructions.

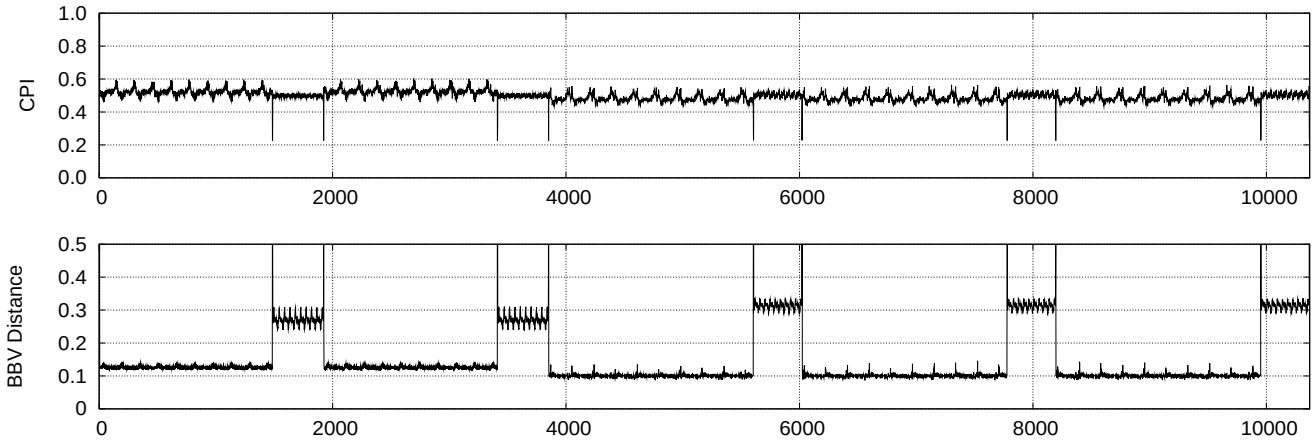

Figure 7: Time varying graph of CPI and BBV distance to the target vector from each interval of execution for gzip-graphic at 10m interval size.

chitecture configuration with *lowest CPI variance* out of our 18 configurations. This graph shows that phase behavior can be found by examining very small CPI changes. This is why it is important to perform the phase analysis by clustering code signatures in an architecture-independent manner, instead of relying on CPI thresholds. To show this, Figure 5 plots the ROC curves for CPI thresholds of 3%, 5% and 7% for `gzip` in the same low CPI variance configuration. The results show that even though `gzip` has very small CPI changes (less than 0.01 CPI variance), the CPI changes are still strongly correlated to BBV changes.

The results in this section show a strong correlation between code signatures and performance, which contradicts the quadrant classification for `gzip` and `gcc` in [1]. It also emphasizes the importance of performing classification based on code signatures, which are independent of the underlying architecture. Because the CPI variance can change depending on the architecture, it is easier to find phase behavior in programs with low CPI variance by examining code signatures.

2.7 Clustering Performance

This section has shown that there is a strong relationship between CPI changes and code signature changes. We are also interested in the homogeneity of phases after clustering similar code signatures. We measure homogeneity by examining the CPI data for each phase.

SimPoint [18] is a tool that uses the k-means algorithm from machine learning to group code signatures into clusters based on signature similarity. The single most representative code signature from each group is selected for detailed simulation, and the results of each detailed simulation are extrapolated to estimate the program's overall behavior.

We present two metrics for our SimPoint results: coefficient of variation (CoV) of CPI, and estimated CPI error. The coefficient of variation is the standard deviation divided by the average. In other words, it is the standard deviation expressed as a fraction of the average. To calculate the whole program CoV of CPI, we take the CPI data for each interval, calculate the standard deviation and the average, and divide.

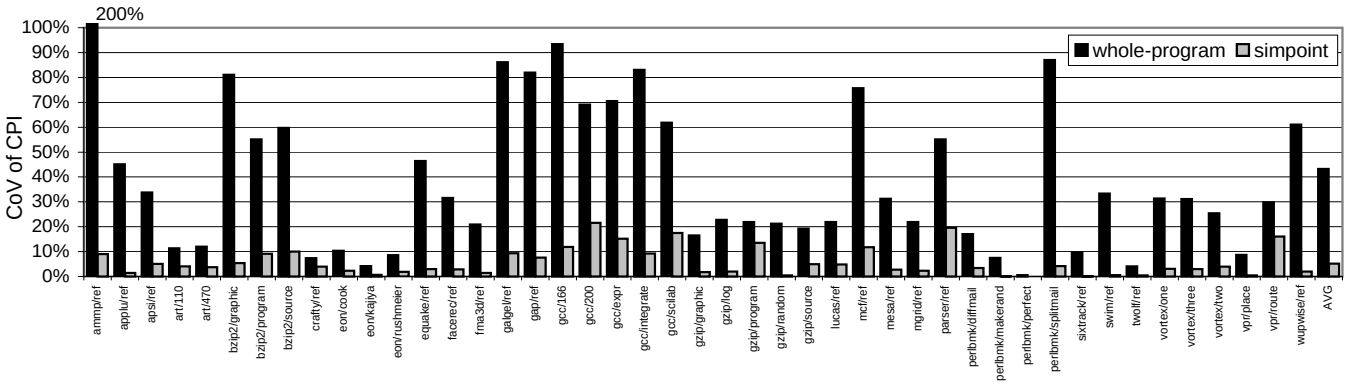

Figure 8: Coefficient of variation for CPI. CoV = stddev/avg. To calculate the SimPoint CoV, we compute the CoV for each phase, then compute the weighted average CoV across all phases. Each per-phase CoV is weighted by the number of intervals in its phase.

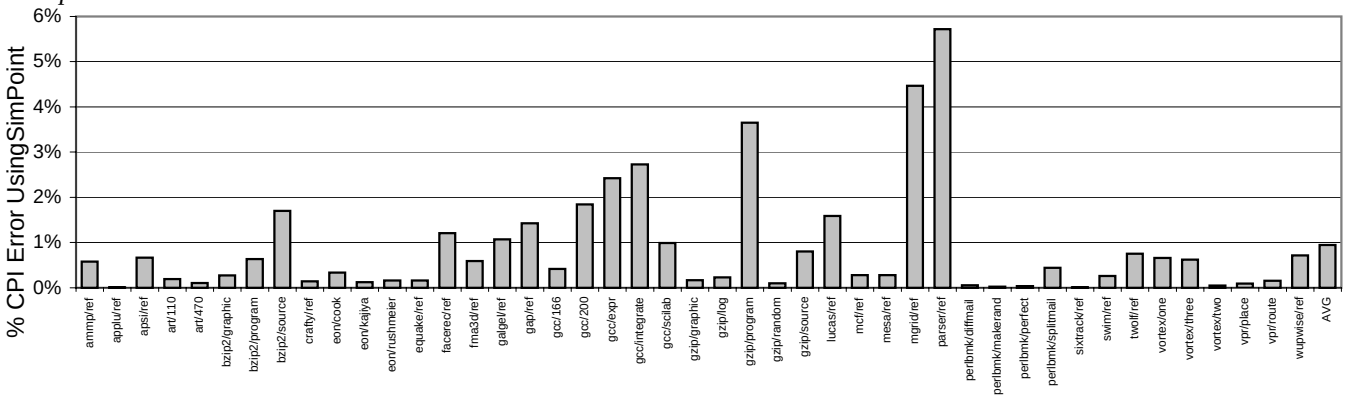

Figure 9: Estimated CPI error using SimPoint at 10m interval size.

To calculate the SimPoint CoV of CPI, we perform the CoV of CPI calculation for each cluster (phase), and then compute the weighted average of the per-cluster CoVs of CPI. Each per-cluster CoV is weighted by the number of intervals in its cluster.

Figure 8 shows the whole program and SimPoint CoV of CPI. The SimPoint CoVs are significantly lower than the whole program CoVs, which shows that clustering the intervals based only on code signatures successfully groups the intervals into phases with low intra-phase CoV of CPI. This shows a strong correlation between code and performance, since our clusters are built by examining only code similarities, yet each cluster contains intervals with very similar CPIs.

Figure 9 shows the amount of error in SimPoint estimated CPI. If the error is 5%, that means the program CPI we estimate is $\pm$5% different from the CPI of a complete simulation of the program. These results also indicate a strong correlation between code and performance, because the representatives for each cluster chosen by SimPoint (by examining only code signatures) have CPIs that are highly representative of their cluster's average CPI.

I Cache	64k 2-way set-associative, 64 byte blocks, 3 cycle latency
D Cache	64k 2-way set-associative, 64 byte blocks, 3 cycle latency
L2 Cache	2Meg 16-way set-associative, 64 byte blocks, 20 cycle latency
Memory	275 cycle round trip access
Branch Pred	hybrid - 16K Meta, 8K entry Bimodal, 8K 8-bit history L2. 16KB BTB and 16 entry RAS
O-O-O Issue	out-of-order issue of up to 4 instructions/cycle, 32 entry ROB
Mem Disam	32 entry load/store queue, loads may execute when all prior store addresses are known
Func Units	4-integer ALU, 4-load/store units, 3-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV

Table 2: Baseline x86 Simulation Model.

3 Sampled Code Signatures

To track program behavior, we create code signatures by profiling basic blocks, loops, and procedures. All of these approaches account for every instruction executed [11]. Alternative approaches, such as performance counter sampling, avoid the overheads of full code profiling. The performance counter approach uses processor performance counters to gather code samples at a rate set relative to the frequency of some chosen processor event, such as instruction commit.

3.1 Sampling with VTune

The recent paper by Annavaram et al. [1] used VTune to gather performance counter data to guide phase analysis. In

this section we examine using the same mechanism to create sampled vectors for phase analysis and compare the sampled vectors to full non-sampled vectors. To gather our results, we used the Remote Data Collector for VTune 7.2 with driver kit 3.2. We gathered instruction retirement events on a 3.06 GHz Intel Xeon processor with 512KB cache running RedHat Linux kernel 2.6.1. We performed runs sampling every 100 thousand instructions, over a set of SPEC CPU2000 benchmarks. The binaries were compiled with gcc 3.2.2, optimized with the -O3 flag and statically linked. For purpose of comparison, we gather detailed simulation results and full BBV traces for the same benchmarks with SimpleScalar-x86(v4.0) [4]. Not all of the SPEC programs currently run on SimpleScalar-x86, and we provide results for all of the program/input pairs that ran to completion. The baseline micro-architectural model we simulate is detailed in Table 2.

VTune produces a sample trace for each program run. Sample data is obtained via VTune's `sf5dump` utility, which post-processes a trace to produce an output file where every sample has an instruction address (Extended Instruction Pointer, or EIP), cycle count, process ID, CPU ID, thread ID, and process name. The output from `sf5dump` was passed through a filtering and coalescing tool, which removes samples that belong to processes other than the one we were profiling and samples with kernel-space EIPs. Filtering of kernel samples removed at most 0.5% of the samples for any benchmark examined. We found that removing the kernel samples made it easier to identify user program phase behavior.

After the filtering step, the post-processing tool forms vectors of EIPs (EIPVs) by accumulating consecutive sequences of the EIPs that passed the filtering requirements until they account for the desired instruction length for the vector. We used vectors of 10 million instructions. Timing information for each vector was gathered by summing the differences in cycle counts between each sample in the vector and the sample, filtered or otherwise, immediately preceding it.

This method slightly differs from the filtering scheme used in [1]. In that work, infrequently used EIPs of the program are filtered out as a means to reduce dimensionality prior to vector formation. This technique may distort the program code signatures in two ways: (a) removing the infrequent EIP dimensions can overlook significant program behavior, and (b) filtering out EIPs prior to vectorization will result in a incomplete instruction count of the program since the missing EIPs will not be represented in the vectors.

One concern with VTune's current sampling approach is that it uses a uniform sampling rate. Uniform sampling is vulnerable to synchronization problems with repetitive portions of a program's execution. In such cases a repeating region may be sampled at the same point multiple times, and the representation of the region will lack robustness. If the region is executed again later, a different EIP may be repeatedly sampled and it will not carry any similarity to the previous sampling from the same region.

3.2 Sampled Code Signatures

An EIPV is a record of which instructions were captured during periodic sampling of the instruction stream over a given interval of execution. In this sense, its structure is similar to that of a BBV, but the sampled nature of its construction leads to the entries having different meanings; the BBV entries are records of all basic blocks that have executed during an interval, thereby accounting for all executed instructions, whereas EIPV entries represent which instruction addresses were sampled during that interval of execution.

There are two main issues with using EIPVs. The first is that sampling 1 EIP out of every 100,000 EIPs does not fully characterize a program's behavior, especially for programs with large code bases, such as `gcc`. The other issue is that the clustering approach we used and the one used in [1] are based upon grouping vectors (intervals) together based on their similarity across the vector dimensions. When sampling EIPs, two EIPs in the same basic block can be sampled different numbers of times. Instructions in the same basic block are always executed the same number of times, due to the single-entry single-exit nature of basic blocks. Because sampled EIPs in the same basic block can have different sample counts, EIPVs may appear farther apart than they should be.

To address these two issues we propose mapping EIP sample data to larger code structures. To accomplish this, we use Pin [14] to find the range of EIPs that correspond to the start and end of each loop and procedure in each program. We examine results for two sampled code signatures based on mapping EIPs to only procedures and mapping EIPs to loops and procedures. For the procedure only sampled code signatures, each vector has one dimension for every executed procedure in the program. The EIPs for an interval are mapped to their corresponding procedure dimension, which is the procedure they are from. Therefore, the final normalized sampled `vtune-proc` vector represents the percent of sampled execution that occurred in each procedure during that interval.

We also examine a sampled code signature mapping the EIPs to loops and procedures. This is called `vtune-loop` in the results below. For this code signature sample, each dimension represents either a loop or procedure in the program's execution. The loops and procedures are treated as static nesting levels in the binary, and the EIP is mapped to the most deeply nested structure that has its start and end PC range cover the PC of the EIP sample. In other words, each EIP maps to the most deeply nested loop/procedure that contains it. In [11] we showed that creating full code signatures with vectors of loops and procedures performed better than only using procedures, because some programs spend most of their time in a small number of procedures, and loops provide more information about a program's control flow. In addition, we found the loop-and-procedure vectors perform just as well as the full BBVs [11].

3.3 Dimensionality of Full and Sampled Code Signatures

VTune is an attractive data gathering tool for EIPVs because the imposed overhead is small enough that the impact on the

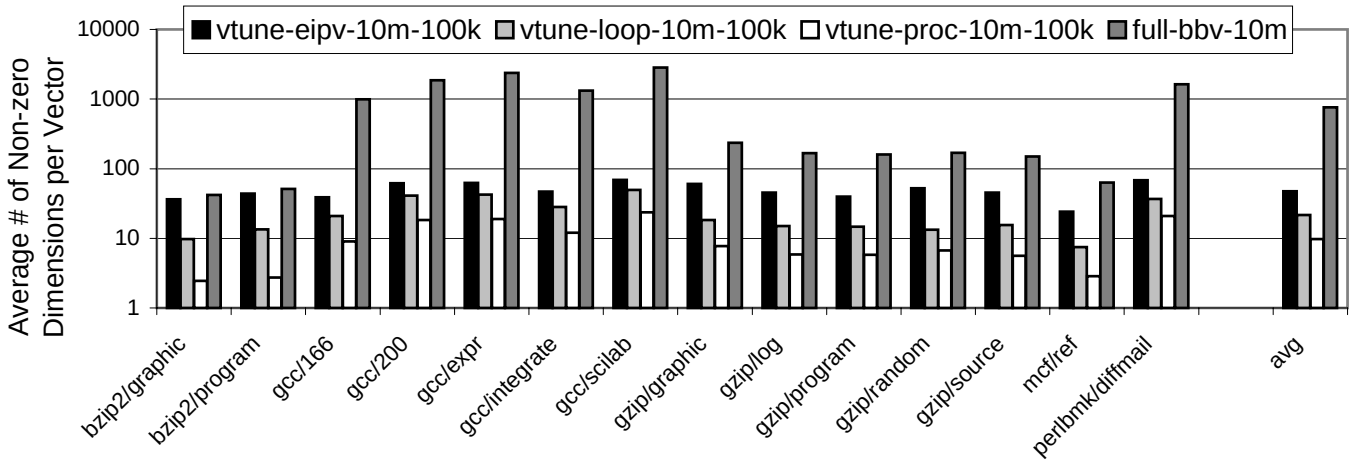

Figure 10: Average number of unique non-zero dimensions per vector per interval of execution for 10m instructions per interval. The y-axis is a logarithmic scale. "vtune-eipv-10m-100k" are EIPVs collected at 10m granularity with VTune sampling every 100k instructions. "vtune-loop-10m-100k" results from mapping each EIP to its corresponding loop or procedure. "vtune-proc-10m-100k" results from mapping each EIP to its corresponding procedure. "full-bbv-10m" are full BBVs collected from SimpleScalar x86. All subsequent graphs use this naming scheme.

program under analysis is low for sample sizes of every 100 thousand instructions to every 1 million instructions. The average increase in execution time when running VTune with sampling at every 100K instructions was 5.6% and at every 1 million instructions was 1.5%. Increasing the sampling frequency will incur more overhead and eventually bias the observed performance of the application significantly.

The acceptable sampling rate places a limitation on the EIPV: it will always provide a sparse representation of program execution. This limitation may not be a handicap for programs with small working sets that can be represented with a few samples. Complex programs, however, such as `gcc` and `perlbmk`, generally have larger working sets of instructions. These complex programs often execute a substantial number of distinct instructions within an interval, and sparsely sampling such an interval will not effectively represent it.

Figure 10 shows the average number of non-zero dimensions per vector for several different vector types, sampling rates, and granularities. Results are provided for EIPVs (vtune-eipv), sampled loops plus procedures based on the EIPVs (vtune-loop), sampled procedures based on the EIPVs (vtune-proc), and the BBVs (full-bbv) from SimpleScalar. We *dynamically* count the number of non-zero dimensions per vector, which is the number of unique EIPs, loops, or procedures sampled for the VTune results. For the SimpleScalar x86 results, the number of non-zero dimensions per vector counts the average number of unique basic blocks executed in each interval. This metric provides a rough measurement of code complexity found in each interval for the different vector types. The comparison between the number of basic blocks from the sampled EIPVs and from the full BBVs is a comparison of the degree of code coverage provided by sampling. As seen in Figure 10, a comparison between sampled EIPVs and full BBVs for `gcc` shows that the actual number of executed basic blocks per interval is more than ten times larger than that reported by the sampled EIPVs, which is in terms of instructions and the number of dimensions in terms of instructions should be significantly larger than basic blocks. Even for less complex programs, the differences in dimensionality are still notable.

We also wanted to examine the ranking of the dimensions between the sampled data and full profiling to see how well they matched. To do this we generated two profiles for each code construct examined. We gathered a profile of the complete program's execution using the sampled data with VTune and a full count profile using Pin [14]. We then plot in Figure 11 how well these two profiles match in terms of percent of executed instructions seen in their accumulative profiles.

For the eip results, we sort the full instruction count profile by each instruction's execution frequency. Starting with the most frequently executed instruction of the full profile, we accumulate the instruction's execution counts until we reach 5% of the total number of dynamic instructions executed. This percentage is represented on the x-axis in Figure 11. We take these same static instructions from the accumulative EIP profile, and accumulate their dynamic instruction count and plot it on the y-axis. The process is then repeated for other x-axis values. The same is done for loops. Figure 11 therefore shows the difference in dynamic instruction count between the sampled and full instruction and loop profiles. The results show that the relative ranking of EIPs is off by up to 5% from the full profile, whereas when looking at the sampled data in terms of loops plus procedures it is almost an exact match with `perfect`. This shows that creating sampled vtune-loop vectors provides a representation closer to the full profile than using EIPVs.

3.4 Correlating Code Signatures and CPI Changes

Figure 12 shows the Receiver Operating Characteristic (ROC) curves as described in Section 2.5. These plots show that

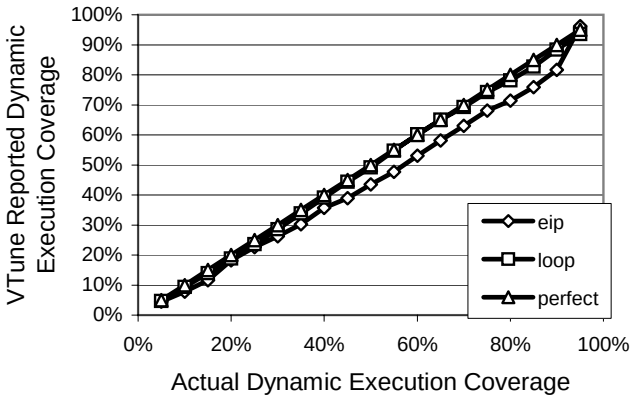

Figure 11: VTune sample coverage in terms of instructions executed. We collect two profiles of program behavior: a full instruction-level profile using Pin, and a sampled instruction-level profile using VTune, sampling every 100K instructions. The X-axis represents the amount of dynamic execution covered by the most frequently executed instructions (eip) or loops from the full profile. The Y-axis shows the amount of dynamic execution covered from the sampled profile using the same set of EIPs or loops that accounted for the top X% from the Pin coverage. Results closer to the "perfect" line are better. Results above the perfect line are over represented, and results below the line are under represented.

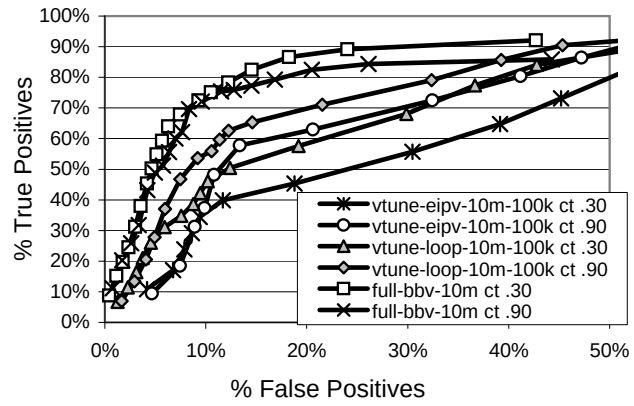

Figure 12: Receiver operating characteristic at 10m interval size. The number after "ct" in the series name is the CPI threshold used to identify a significant change in CPI.

EIPVs can be used to predict CPI changes, but not as accurately as full BBVs. Converting EIPVs to sampled loops plus procedures improves predictability as shown in Figure 12. Larger CPI changes are easier to predict, independent of vector type, which is consistent with previous results. Recall that our ROC curves are clustering-independent: these results were produced just by examining the change in CPI between adjacent intervals of execution and the differences between unprojected vectors from adjacent intervals of execution. The ROC curves show correlations between significant vector differences and significant performance changes for various CPI and vector significance thresholds.

3.5 Clustering Performance

In this section we apply SimPoint [9] to the VTune sampled code signatures and the full BBV signatures. Figure 13 shows the coefficient of variation of CPI after clustering with SimPoint as described in Section 2.7. Figure 14 shows SimPoint CPI estimation accuracy. Results are shown using a 10 million interval size. For the sampled code signatures, samples were taken every 100K instructions. To make a more fair comparison across all of the techniques, we fixed k, which is the number of clusters (phases) formed in SimPoint using the k-means algorithm, to $k = 61$ for each of the techniques. This value is based on the average number of clusters chosen by SimPoint for the full BBV results across all the benchmarks.

The results show that using the full BBV information has a much lower CoV of CPI and estimated CPI error rate, compared to sampled vectors. In particular, `gcc` and `gzip` have high variations due to insufficient data in the EIPV signatures to correctly cluster the intervals into homogeneous phases. Using sampled loops and procedures reduces estimated CPI error and CoV of CPI overall, and significantly helps `gcc` and `perl`. The results also show that mapping EIP samples to procedures (`vtune-proc`) has significantly worse error rates for a few `gcc` inputs, whereas mapping to loops and procedures (`vtune-loop`) consistently performs well.

Figure 15 shows the absolute percent CPI error over the program's execution. This is a different way of looking at CPI error than in Figure 14. Sometimes, SimPoint selects representatives that over- or under-estimate the average CPI for their phase. When these CPI estimates are combined to produce an estimate for the program's overall CPI, an over-estimated CPI can hide ("cancel out") another under-estimated CPI, resulting in a low error rate, even when the representatives are not well chosen. To address this problem, Figure 15 shows the absolute CPI error over the complete execution of the program, without allowing positive and negative per-phase estimated CPI errors to cancel each other out. In other words, Figure 14 shows how well SimPoint estimates a program's overall CPI, and Figure 15 shows how well SimPoint estimates the average CPI of *each phase*.

For example, Figure 14 shows that SimPoint estimates `gcc-200`'s overall CPI with 1% error with EIPVs, but if we look at the per-phase CPI estimates in Figure 15, the error is 25%. In this case, the representatives selected by SimPoint were actually not very representative, even though the overall estimated CPI error is low, because the per phase errors in estimated CPI canceled each other out when calculating an overall CPI estimate. In contrast, `gcc-166` was not so lucky where it had absolute error of 45% when using EIPVs, and these did not cancel each other out as well, and the overall CPI error was 33% as shown in Figure 14.

The results show that full-bbvs tend to have fairly consistent behavior when looking at overall CPI error and absolute

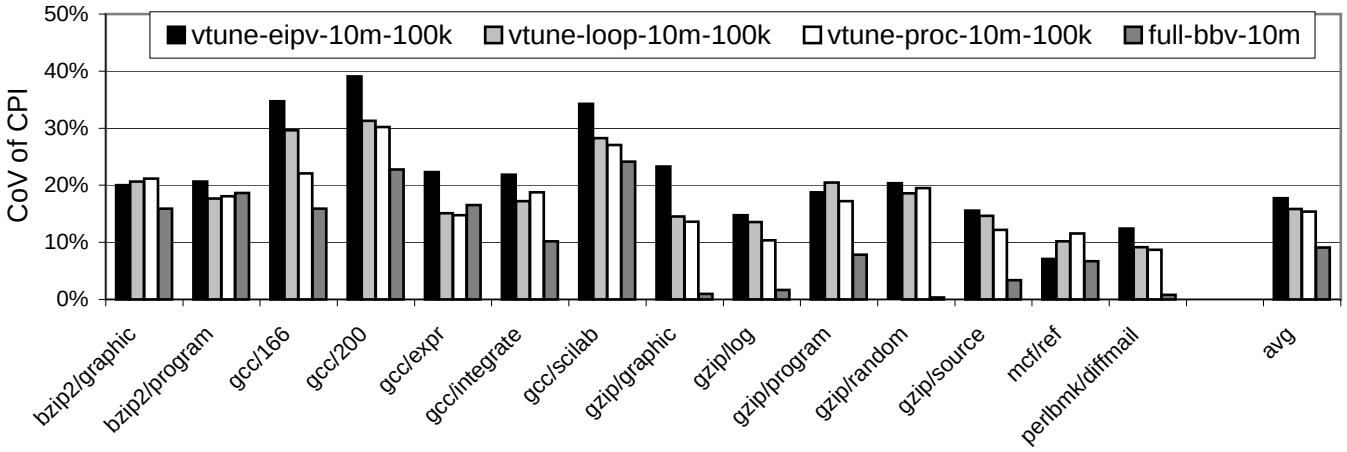

Figure 13: Coefficient of variation of CPI after clustering with 61 clusters at 10m granularity.

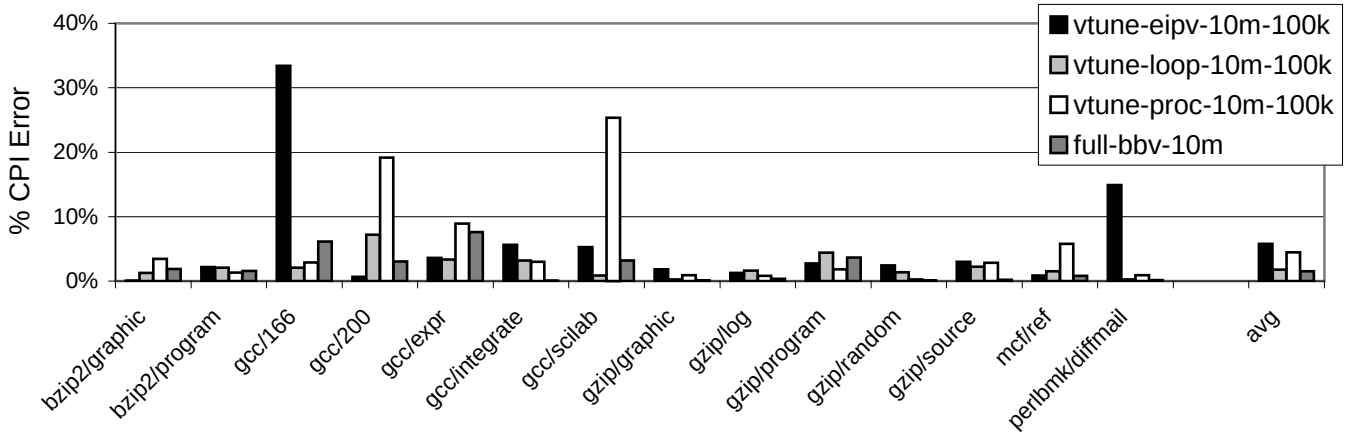

Figure 14: SimPoint estimated CPI error with 61 clusters at 10 million instructions per interval.

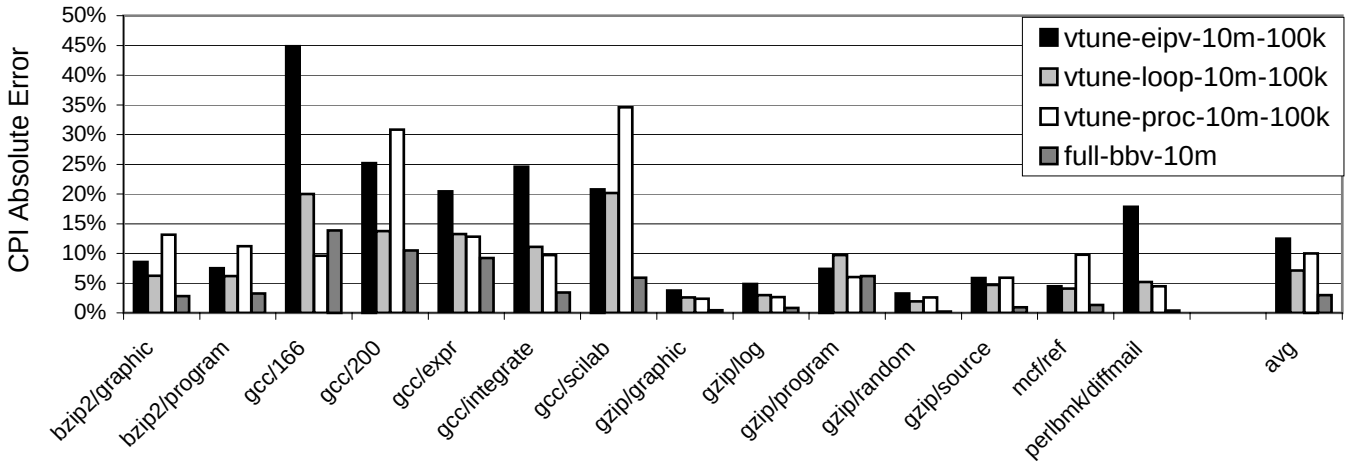

Figure 15: SimPoint absolute error after clustering with 61 clusters at 10 million instructions per interval.

CPI error. Thus, the representatives for each phase selected by SimPoint based on full-bbv data not only represent the whole program well, but also represent each phases well. The results show that the absolute error for full non-sampled basic block vectors is below 15% for all programs, with an average of 2% across all programs examined.

4 Summary

In this paper, we demonstrated that a strong correlation exists between code and performance for SPEC2000 programs. This was shown in three ways:

- The high correlation between CPI changes and code signature changes seen in receiver operating characteristic plots (Figure 1). The correlation seen in the ROC curves is independent of projection and the clustering algorithm used.
- The low intra-phase coefficient of variation of CPI, compared to the high whole-program coefficient of variation of CPI, when intervals are grouped into phases by examining only the code signatures (Figure 8).
- The low error rates when predicting overall CPI with code signature phase analysis (Figure 9).

We also show a weaker correlation between EIPVs and performance. EIPVs show weaker correlations between CPI changes and code signature changes in ROC curves, higher intra-phase CoV of CPI, and higher absolute CPI error rates. The primary reason for this is the loss of information due to sampling. This was shown in the differences in dimensionality, where `gcc` had 10 times the number of unique basic blocks profiled compared to the number of unique EIPs detected.

Finally, we showed that EIPVs can be converted to sampled vectors of loops and procedures by mapping each EIP to its corresponding code construct using a tool like Pin [14], and building new vectors which track loop or procedure distributions instead of basic block distributions. Mapping sampled EIP code signatures to loops improves results because combining samples taken from the same code structure reduces noise introduced by sampling. We found that mapping EIP code signatures to procedures does not always improve results, because some programs do not use very many procedures, which makes it difficult to detect phase behavior at the procedure level. In general, mapping EIPVs to loop plus procedure vectors improves ROC curves, decreases CoV of CPI, and decreases estimated CPI error rates, but not enough to make the results as good as full code profiling.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft.

References

[1] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *International Symposium on Microarchitecture*, December 2004.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkada. Memory hierarchy reconfigurtion for energy and performance in general-purpose processor architectures. In *33th Annual International Symposium on Microarchitecture*, December 2000.

[3] 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.

[4] 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.

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

[6] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[7] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[8] A. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *36th Annual International Symposium on Microarchitecture*, December 2003.

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

[10] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

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

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

[13] W. Liu and M. Huang. EXPERT: Expedited simulation exploiting program behavior repetition. In *International Conference on Supercomputing*, June 2004.

[14] 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.

[15] M. S. Pepe. The statistical evaluation of medical tests for classification and prediction. *Oxford University Press*, 2003.

[16] 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.

[17] 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.

[18] 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. http://www.cs.ucsd.edu/users/calder/simpoint/.

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