Structures for Phase Classification

Jeremy Lau Stefan Schoenmackers Brad Calder

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

Abstract

Most programs are repetitive, where similar behavior can be seen at different execution times. Proposed algorithms automatically group these similar intervals of execution into phases, where all the intervals in a phase have homogeneous behavior and similar resource requirements.

In this paper we examine different program structures for capturing phase behavior. The goal is to compare the size and accuracy of these structures for performing phase classification. We focus on profiling the frequency of program level structures that are independent from underlying architecture performance metrics. This allows the phase classification to be used across different hardware designs that support the same instruction set (ISA). We compare using basic blocks, loop branches, procedures, opcodes, register usage, and memory address information for guiding phase classification. We compare these different structures in terms of their ability to create homogeneous phases, and evaluate the accuracy of using these structures to pick simulation points for SimPoint.

1 Introduction

The behavior of a program is not random - as programs execute, they exhibit cyclic behavior. Recent research [1, 4, 5, 11, 12, 13, 10, 7], has shown that it is possible to accurately identify and predict these phases in program execution.

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. All the results in this paper use a fixed interval size of 10 million instructions. A *phase* is a set of intervals within a program's execution that have 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 in this paper are specific to the input used to run the program.

Our prior work [11, 12, 10] showed that it is possible to accurately perform phase classification by only examining the code executed. In this paper, we compare the use of many program level structures to guide phase classification. The goal is to compare the size and accuracy of these structures for performing phase classification. We explore the trade-offs of detecting phase behavior by profiling basic blocks, loop branches, procedures, the instruction mix, register usage, and memory address information. We compare and contrast the effectiveness of each program structure for phase classification and to guide the picking of simulation points for SimPoint.

2 Methodology and Metrics

We performed our analysis for the SPEC 2000 programs `ammp`, `bzip`, `galgel`, `gcc`, `gzip`, `mcf`, and `perl`. All programs were run with reference inputs, and `bzip`, `gcc`, `gzip`, and `perl` were run with multiple inputs. When calculating averages for the results, programs with multiple inputs are first averaged, so each program has only one representative result in the overall average. All programs were executed from start to completion using SimpleScalar [2] to gather the performance at 10 million intervals for the complete execution of the program. We log and reset the statistics every 10 million instructions. The baseline microarchitecture model is detailed in Table 1. We chose the above programs since they were the most interesting and challenging for phase classification from our prior studies. We collect all of the frequency vector profiles using ATOM [14].

2.1 Metrics for Evaluating Phase Classification

Since phases are intervals with similar program behavior, one way to measure the effectiveness of phase classification is to examine the similarity of program metrics within each phase. We focus on overall performance in terms of Cycles Per Instruction (CPI) within each phase. After classifying a program's intervals into phases, we examine each phase and calculate the average CPI of all intervals in the phase. We then calculate the standard deviation in CPI for each phase, and we divide the standard deviation by the average to get the *Coefficient of Variation* (CoV). CoV measures standard deviation as a percentage of the average.

We use the CoV to compare different phase classification algorithms. Better phase classifications will exhibit lower CoV. If all of the intervals in the same phase have exactly the same CPI, then the CoV will be zero. We calculate an overall CoV metric for a phase classification by taking the CoV of each phase, weighting it by the percentage of execution that the phase accounts for, and then summing up the weighted CoVs. This results in an overall metric we can use to compare different phase classification algorithms for a given program.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 4 operations per cycle, 64 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

It represents the average percentage of deviation that a phase classification exhibits.

3 Phase Classification and SimPoint

The focus of this research is to investigate the use of different program information to guide automated phase classification. In this section we summarize the SimPoint phase classification approach used.

3.1 Profiling Granularity

To identify phases, we first need to decide how frequently we will monitor the program's behavior. We divide the execution of each program into contiguous non-overlapping *intervals*, each of length 10 million instructions. Our prior work [13, 10] showed that there is repetitive and interesting phase behavior seen at a granularity of 10 million instructions, which is at the same time scale as operating system time slices. We gather profile information every 10 million instructions executed, and afterwords we run the SimPoint phase classification algorithm on this data.

3.2 Data Structures Used to Capture Phase Behavior

The following data structures have been proposed for collecting profiling information for each interval to guide phase classification.

- Working Set Size - For each interval of execution, one would keep track of the total working set size of the information being profiled. For data, this may mean keeping track of the total number of unique words or pages referenced at least once within each interval. Similarly, for code one could keep track of the number of unique basic blocks executed at least once for a given interval.
- Working Set Bit Vectors - For each interval of execution, a bit vector keeps track of whether a given item has been encountered or not during that interval. Bit vectors are more expressive than working set size, since bit vectors can distinguish intervals with the same working set size but where different items (e.g., code or data addresses) are accessed.
- Frequency Vectors - For each interval of execution, a vector records the frequency in which profile items are encountered. This is similar to working set bit vectors, but instead of just keeping track of whether the item was referenced or not, we keep track of the number of times each item was referenced. Frequency profiles are more expressive than bit vectors, because they indicate which parts of the working set are being used more than others. For example, frequency vectors can be used to differentiate two intervals which execute the same parts of code, but happen to exercise the code differently (e.g., they emphasize different paths through a loop).

Dhodapkar and Smith [5, 4] proposed the use of *bit vectors* to track the code's working set for phase classification. Sherwood et.al. [11, 12] proposed the use of *frequency vector* profiles to perform phase classification. Frequency vectors track the proportions in which code was executed during each interval, while bit vectors track which parts of code are executed (the working set), without the use of relative frequencies of execution.

Dhodapkar and Smith [6] recently conducted a study where they compare basic block frequency vectors, to bit vectors of procedures, branches, and instructions with the goal of detecting phase changes in programs. Their study focused on *phase change detection*, while we focus on *phase classification*, which are two very different problems. Phase classification is the problem of grouping together all of the program's execution intervals that have similar behavior, regardless of temporal adjacency. Phase change detection is the problem of identifying phase changes in temporally adjacent execution intervals. In [6], they found working set bit vectors to perform as well as frequency vectors for phase change detection.

In this work we examined all three of these techniques for automated phase classification, but only report results for frequency vectors due to space considerations. Working set bit vectors and working set size are sufficient for identifying phase changes, but they did not provide sufficient resolution for our off-line phase classification, when compared to frequency vectors.

3.2.1 Basic Block Frequency Vectors

Our prior approach used the Basic Block Vector (or BBV) [11] as a metric designed 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. More formally, a *Basic Block Vector* (BBV) 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 example, if the 50th basic block is executed 15 times, then bbv[50] = 15. For the weighted results in Section 4, 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.

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. We use the basic block vectors as fingerprints 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 BBVs of two intervals, we can evaluate the similarity of the two intervals. If the BBVs are similar, then the two intervals spend about the same amount of time in roughly the same code, and therefore the performance of those two intervals should be similar.

In this study, frequency vectors are a generalization of basic block vectors, where we track the relative frequencies of events. For example, our loop vectors track the relative frequencies of execution of loop branches. We collect frequency vectors for many program structures, such as procedures, opcodes, register usage, instruction mix, and memory access patterns.

3.3 Using Clustering for Phase Classification

Frequency vectors provide a compact and representative summary of the 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.

To exploit phase behavior, it is useful to have an automated way of extracting phase information from programs. To break the complete execution of the program into smaller groups (phases) that have similar frequency vectors, algorithms from Machine Learning (clustering) have been shown to be very effective [12]. Because the frequency vectors relate 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 such that points within each group are similar to one another (by some metric, often distance), and points in different groups are different from one another. A well known clustering algorithm is k-means [8], and this can be used to accurately break up program behavior into phases. Random Linear Projection [3], which reduces the dimensionality of the input data without disturbing the underlying similarity information, can be used to speed up the execution of k-means. One serious drawback of the k-means algorithm is that it requires a value for k as input, the number of clusters to look for. 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. The following steps summarize the phase clustering algorithm at a high level. We refer the interested reader to [12] 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 15 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.
3. Run the k-means clustering algorithm on the reduced dimensional data with values of k from 1 to M, where M 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. In this step, the k-means algorithm compares the similarity of all intervals, grouping them into phases. 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) [9] 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 larger 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 using the BIC is scored using the BIC formulation given in [9].
5. The final step is to choose the clustering with the smallest k, such that its BIC score is at least X% 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. In this paper, we use the above algorithm with various types of frequency vectors to evaluate the use of different program structures to guide phase classification. We set N (the number of instructions per interval) to 10 million, M (the maximum value of k) to 100, we try 7 different random seeds for each value of k, and we set X (the BIC score threshold, relative to the maximum score) to 90%.

3.4 Using Phase Classification to Guide Simulation

In modern computer architecture research, it is crucial to understand the cycle level behavior of a processor running an application. To gain this insight, detailed cycle level simulators are typically employed. Unfortunately, this level of detail comes at the cost of simulation time, and simulating the full execution of an industry standard benchmark on even the

fastest simulator can take weeks or months. Long simulation times mean that it is only feasible to simulate a small portion of the program, so it is very important that the section simulated is an accurate representation of the program's overall behavior. The off-line phase classification described above provides an accurate and efficient solution to this problem.

After the intervals of execution are classified into phases for each program/input, a single representative from each phase can be selected, and we can estimate the behavior of the remaining intervals by performing detailed simulation *only* on the representative, and extrapolating. To choose a representative, we pick the interval that is closest to the center, or centroid, of each cluster. This selected interval for each phase is called a *Simulation Point* for that phase [10, 12].

We perform detailed simulation on the selected simulation points for each phase. The performance results for each simulation point are then weighted by the size of (number of intervals in) the cluster it represents. Combining these weighted results from each of the simulation points gives an accurate representation of the complete execution of the program/input pair, and significantly reduces simulation time. The above approach is distributed as part of the SimPoint [10, 12] tool.

The goal of our paper is to identify alternative structures that can accurately and succinctly capture phase information. This can improve the efficiency of both SimPoint and phase classification techniques by reducing the amount of information that has to be collected and processed. Therefore, when comparing the different structures for phase classification, we will also evaluate their accuracy for finding simulation points to guide SimPoint simulation.

4 Code Phase Classification

In this section we consider tracking code and ISA-based (instruction mix and register usage) structures for phase classification.

4.1 Control Flow Structures

Our prior work on phase classification is based on basic block frequency vectors. We therefore start by examining the accuracy of phase classifications based on loops and procedures, and compare this to basic blocks.

For tracking procedures, we create a frequency vector with one entry for each static procedure in the program. When tracking loop branch frequency vectors, we create a vector with a dimension (entry) for every intra-procedural backward branch. Since we are performing our analysis at the binary level, we found this to be an adequate approximation for identifying loop branches.

Table 2 shows the number of static basic blocks, loops and procedures found for each of our benchmarks. The first number shows the static frequency vector size for each structure, and the number in parenthesis shows the percentage of those static entries that were encountered on average in each interval of execution. The results show that there were 10 times more basic blocks than loop branches, and twice as many loop branches as procedures. In terms of what was seen in the execution of each interval, on average only 543 basic blocks were executed, 36 loop branches, and 30 procedures.

	basic block max (avg)	loop max (avg)	procedure max (avg)
ammp	15129 (0.5%)	1206 (0.9%)	558 (0.7%)
bzip2	7920 (0.8%)	745 (1.7%)	372 (1.0%)
galgel	44898 (0.2%)	3937 (0.2%)	1234 (0.2%)
gcc	102261 (1.9%)	5383 (2.4%)	2437 (3.4%)
gzip	8769 (2.6%)	806 (2.5%)	384 (3.8%)
mcf	8234 (0.9%)	580 (2.0%)	325 (1.6%)
perl	46310 (2.9%)	2501 (2.3%)	1479 (6.5%)
average	33360 (1.6%)	2165 (1.6%)	970 (3.1%)

Table 2: Maximum vector length, and the average number of non-zero vector elements. Averages are expressed as a percentage of the maximum.

We examine two types of code vectors – weighted by the number of instructions executed and unweighted. In the original basic block vectors [11], each entry represented the execution count of the basic block multiplied by the number of instructions in each basic block. These are *weighted* vectors. For unweighted vectors, each vector entry only counts the number of times the structure was encountered, without considering the number of instructions executed in the structure. All results called Basic Block, Loops and Procedures use the vectors weighted by the instruction frequencies. Unweighted results will have `UW` in front of Basic Block, Loops and Procedures. We perform this comparison, since it is easier for a phase tracker to count the number of occurrences of a basic block, loop, or procedure than it is to count the number of instructions executed within each. If we can retain phase consistency and accuracy using only the unweighted information, then this can simplify on-line (dynamic) phase identifiers and trackers.

The weighted structures represent:

- Basic Blocks - Each frequency vector entry indicates the number of instructions executed in each basic block for the execution interval.
- Loops - Loop vectors record the number of instructions executed inside each loop for each execution interval. For nested loops, an instruction's execution is only associated with the current loop nesting level. A loop vector entry only records the number of instructions executed at its loop nesting level.
- Procedures - Procedure vectors record the number of instructions executed *inside* (not hierarchical) each procedure for the execution interval.

The unweighted structures represent:

- Unweighted Basic Blocks - Each frequency vector entry indicates the number of times that basic block was executed for the execution interval.
- Unweighted Loops - The frequency vectors record the number of times each loop branch was executed in an execution interval.

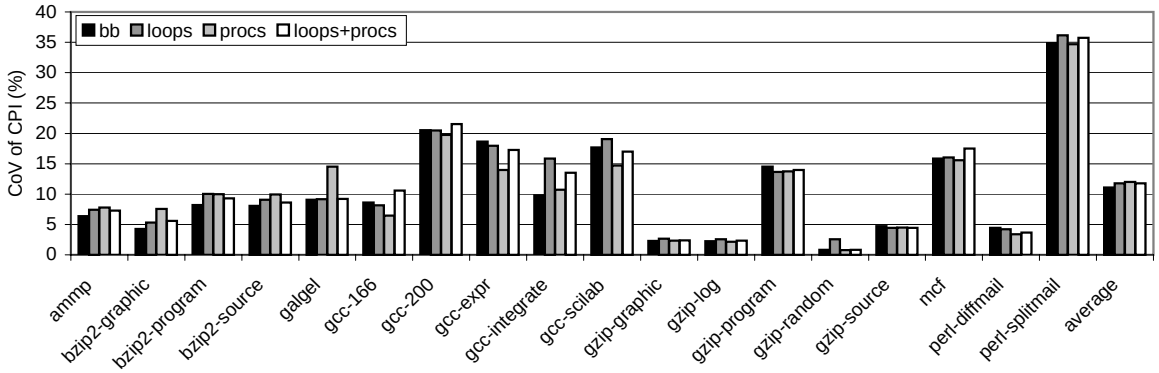

Figure 1: Average CPI Coefficient of Variation (CoV) per phase for code-based vectors (lower is better). CoV measures standard deviation as a percentage of the average. We average CPI CoV's across all phases in each program, producing a metric that measures the the homogeneity of phases discovered in each program.

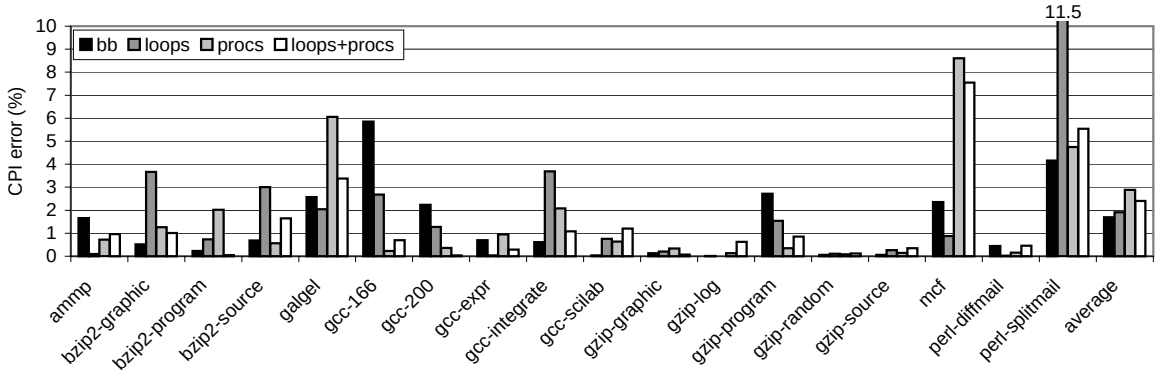

Figure 2: Percent error in CPI estimation, when code-based vectors are used to guide the selection of simulation points.

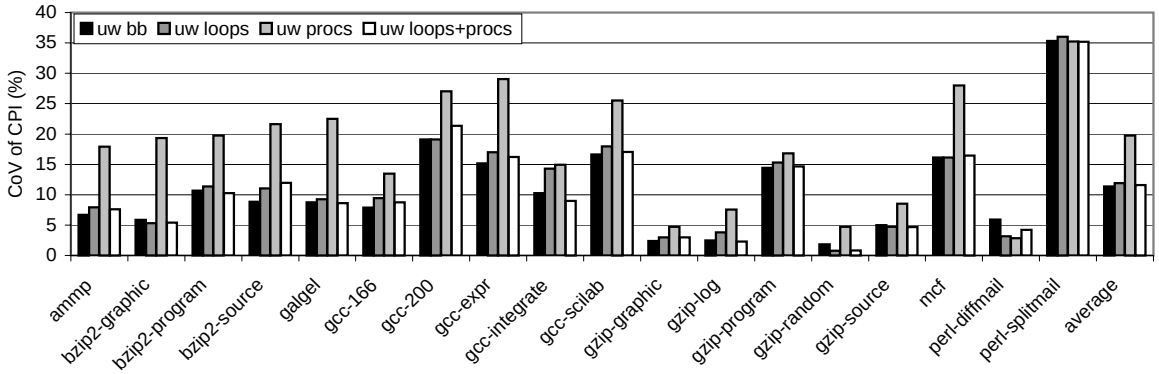

Figure 3: Average CPI Coefficient of Variation for unweighted code-based vectors

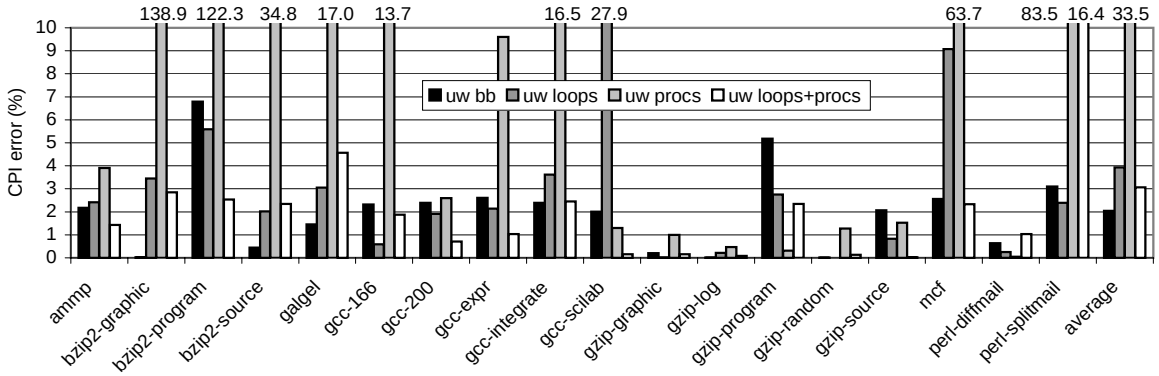

Figure 4: Percent SimPoint CPI error for unweighted code-based vectors.

- Unweighted Procedures - These frequency vectors record the number of times each procedure was invoked (called) in an execution interval.

Figure 1 shows the CoV of CPI for weighted basic blocks, loops, procedures, and the combination of loops with procedures. For Loops+Procs, each execution interval has the loop vector and procedure vector concatenated, and this higher dimension vector is sent to SimPoint to perform phase classification. The CoV CPI results show that in some instances, tracking procedures alone works well for phase classification, whereas in others (e.g., `galgel`) tracking procedures performs worse (identifies less homogeneous). We find that loop-intensive benchmarks such as `galgel`, perform significantly worse when only tracking procedures, since they can't determine the intra-procedural control flow. It can only determine that the program is spending a significant amount of time somewhere in this procedure. For `galgel` in particular, a significant number of intervals fall entirely within a single procedure. Overall, we find that the use of loop vectors results in a similar CoV of CPI as basic block vectors. Tracking procedures alone works well for some applications, and tracking procedures in addition to loops provides slightly better performance.

Figure 2 shows the error in CPI estimation when we use these code structures to choose simulation points. One representative interval containing 10 million instructions is chosen from each phase (see Section 3.4). The results show that, for most benchmarks, tracking only Procedures or only Loops results in approximately the same CPI errors, whereas combining the two provides slightly better results (i.e. you get the best of each method).

Figure 3 shows the CoV in CPI for unweighted code vectors. Comparing these results and the weighted results, we see that removing the weights does not significantly affect CoV of CPI for basic block vectors and loop vectors. This result has promising implications for profilers and on-line phase analysis applications, because it is easier to track unweighted vectors. Procedure vectors, on the other hand, perform quite a bit worse when the weighting is removed.

Figure 4 shows the error in estimated CPI for unweighted code vectors. These results show that weights can be useful for guiding the selection of simulation points. The average error in estimated CPI doubles for basic block vectors, and triples for loop vectors. Even so, unweighted Loops+Procs has an average error rate of 3%.

4.2 Profiling Instruction Mix

An alternative to tracking code constructs is to track the instruction mix. To this end, we consider tracking the following two types of information:

- Memory Instructions - We maintain a vector entry for every static load and store instruction in the program. Each frequency vector entry keeps track of the number of times each load or store were executed.

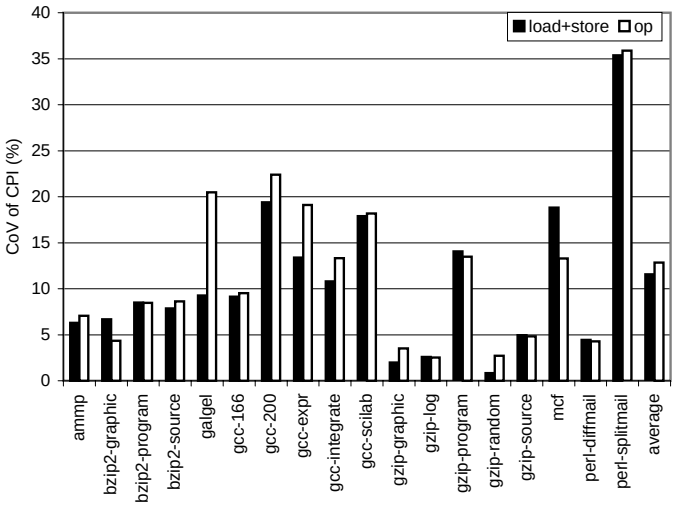

Figure 5: Average CPI Coefficient of Variation for instruction-based vectors.

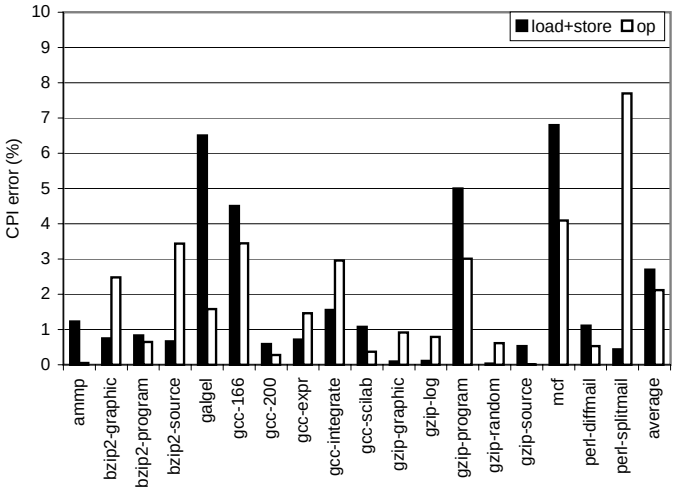

Figure 6: Percent SimPoint CPI error for instruction-based vectors.

- Opcodes (Instruction Mix) - The frequency vector contains 64 entries, where each entry represents a unique opcode in the Alpha ISA. Thus, the vector length is fixed for all programs. The frequency vector for each interval represents the number of times each opcode was executed.

Figures 5 and 6 show the CoV of CPI and the error in SimPoint estimated CPI when performing phase classification based on load/store instructions and the instruction mix.

The results show that tracking either loads and stores or opcodes performs comparably to tracking basic block vectors in terms of CoV of CPI. When we consider the use of these vectors for SimPoint CPI estimation, we find that tracking loads and stores or opcodes perform slightly worse compared to tracking basic blocks (on average). In particular, the maximum errors are slightly higher.

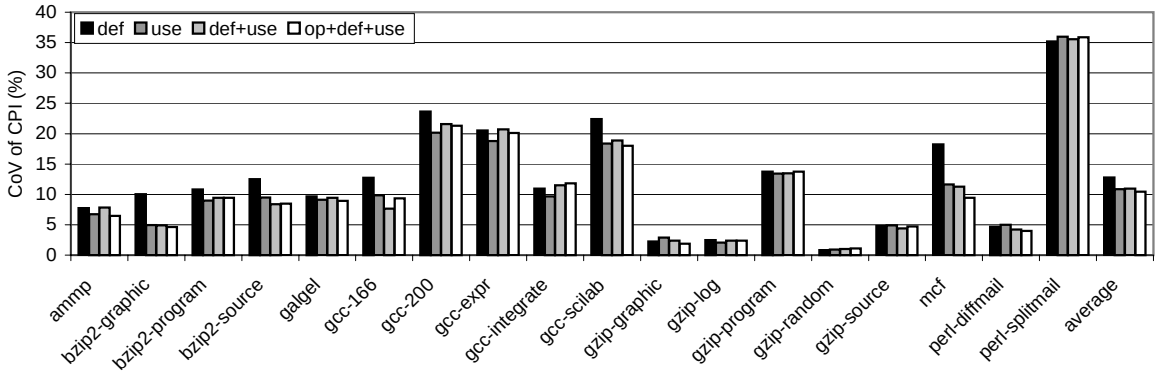

Figure 7: Average CPI Coefficient of Variation for register-based vectors.

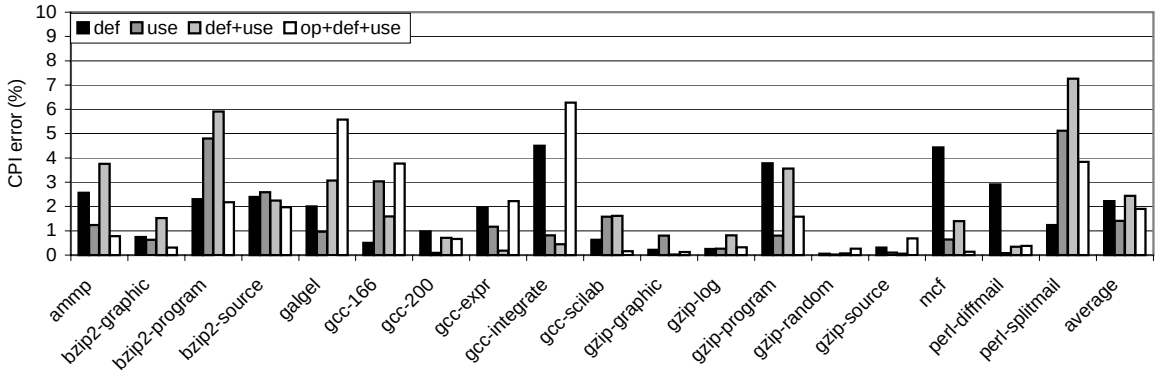

Figure 8: Percent SimPoint CPI error for register-based vectors.

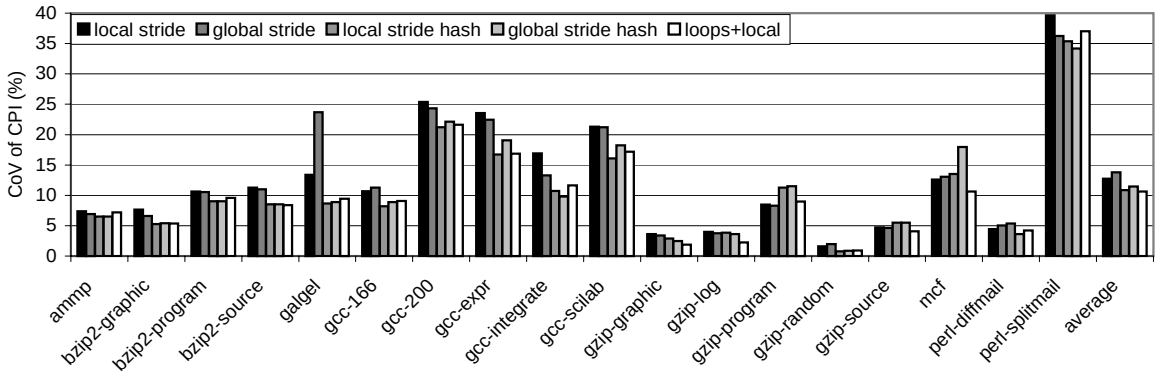

Figure 9: Average CPI Coefficient of Variation for memory-based vectors.

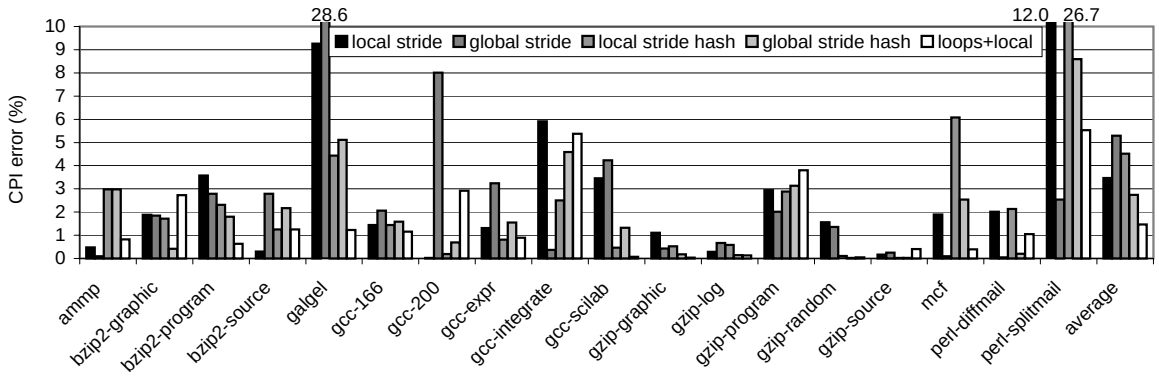

Figure 10: Percent SimPoint CPI error for memory-based vectors.

4.3 Phase Profiling Registers

We also consider the use of register usage information to guide phase classification.

- Register Definitions - We track the number of times each register number was a destination register in each execution interval. The Alpha ISA has 32 integer registers, and 32 floating point registers. We do not distinguish between the floating point and the integer registers (floating point register 7 and integer register 7 map to the same element in our vector), so our vectors contain 32 entries for all programs.
- Registers Used - The number of times each register number was an operand in each execution interval. This vector also contains only 32 entries, one for each register number.

We also consider other combinations, such as definitions + uses, and opcodes + definitions + uses. When combining vectors, we concatenate the vectors together, so opcodes + definitions + uses vectors have 128 entries (64 + 32 + 32), before feeding them into the SimPoint phase classifier.

Figure 7 shows the CoV of CPI for these vector types. This graph indicates, as expected, that vectors with more information and dimensions can be more effective for phase classification. Figure 8 shows the error in SimPoint estimated CPI for these vector types. For the benchmarks we consider, we find that tracking register definitions is most effective for guiding the selection of simulation points, performing comparably to basic block vectors. Since the number of entries is static for all programs and relatively small, and they result in an accurate phase classification, register vectors is one of the most attractive structures from this study. This may be particularly interesting for a dynamic hardware-based phase classification architecture.

4.4 CoV of Different Architecture Metrics

We now examine the benefit of using phase information to guide sampling. We find that samples taken from a single phase exhibit much less variation in all metrics when compared to samples taken across all intervals. We break the execution of a program into intervals of 10M instructions, and we then calculate the Coefficient of Variation (CoV) over all of these intervals of execution. This represents the variation seen when randomly sampling over the complete execution of the workload.

An issue that arises is the impact of infrequently occurring events. For example, if a program exhibits very few instruction cache misses, a small change in the number of instruction cache misses will appear to be a huge change when we calculate the percentage of change (which is essentially what CoV measures). To reduce the impact of rare events, we do not include into the average results for benchmarks where events do not occur at least once per 1000 instructions executed. We use this "1/1000 filter" on instruction cache misses, data cache misses, second level cache misses, and branch mispredictions.

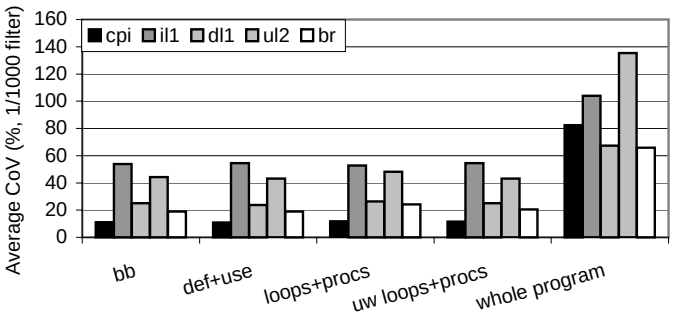

Figure 11: Average Coefficient of variation per phase for CPI, instruction cache misses, data cache misses, 2nd level cache misses, and branch mispredictions. A program contributes to the average only if it averages at least one miss every 1000 instructions executed. The set of bars marked Whole Program show the average CoV for each metric over all the intervals in each program.

Figure 11 shows the average CoV of CPI, instruction cache misses, data cache misses, second level cache misses, and branch mispredictions per phase for a selection of the vector types that we consider. The bars on the far right show the CoV over a run of the whole program. This graph shows that the CoV of many architectural metrics are significantly more stable within the phases we discover than when considering them across the whole program.

5 Data Phase Classification

In this section, we discuss an evaluation of performing phase classification based on memory profiling techniques.

- Local Stride - we build a frequency vector that captures the distribution of strides exhibited by each load or store in the program. So if we see that a load is accessing memory location 5, and that load previously accessed memory location 2, we increment the third $(5-2)$ element in our frequency vector by one.
- Global Stride - similar to local stride, but instead of tracking the stride of each load or store in the program, we keep track of the stride between temporally adjacent memory accesses (loads and stores are tracked separately). So if we see a load that accesses memory location 7, and the last load we saw accessed memory location 3, then we increment the fourth $(7-3)$ element in our frequency vector by one. Global stride vectors are more attractive from a profiling perspective, because the profiler only needs to keep track of two addresses: the last memory addresses loaded and stored. In comparison, local stride vectors require the profiler to keep track of the last memory address accessed by each load and store.
- Local Stride with PC Hash - same as local stride, except that we hash the index into the frequency vector with the lower bits from the PC of the current load or store. By

combining an aspect of the code executed with the memory access pattern, we hope to characterize the code executed *and* the data accessed in each interval.

- Global Stride with PC Hash - same as global stride, except that we hash the index into the frequency vector with bits from the PC of the current load or store.
- Loops with Local Stride - we concatenate the vectors produced by the loop tracker with the vectors produced by the local stride tracker.

It should be noted that these vectors can become *very* large. To reduce the amount of data we have to store, we do not allow memory vectors to grow beyond 200000 elements. All indices are calculated modulo the max vector size. Limiting the maximum vector size results in collisions, but this can be thought of as part of the process of random projection, which occurs before the vectors are clustered.

Figure 9 shows the coefficient of variation in CPI for our memory-based vectors. our most expressive vectors, the local stride with PC hash and local stride with loops, resulted in the lowest CPI CoV. Global stride vectors produce clusterings with about 7% higher CPI CoV then local, so they may be an acceptable alternative to local stride vectors for some applications.

Figure 10 shows the error in calculated CPI when each type of memory vector is used to guide the choice of simulation points. The combination of loops and local stride produce one of our lowest errors in estimated CPI. Overall, the CPI error results for memory vectors are not as consistent as the CPI CoV results. This is because the estimated CPI error from one phase to another can be additive or may cancel out. If the representative of one phase has lower CPI than the actual average CPI of the phase, and another representative of another phase has a higher CPI than the average of that phase, these errors will be hidden. The converse can also be true in terms of the errors being additive. This is the reason for there not being a high correlation between CPI CoV and SimPoint CPI error.

In addition to the memory-based profiling techniques discussed above, we also experimented with other memory profiling techniques with less success. We briefly describe these less successful techniques here.

- Memory working set size - we kept track of the number of unique words of memory accessed during each interval, with the working set size being a 64-bit unsigned integer. This is the most space-efficient profiling data structure, but also the least informative. This technique simply did not provide enough information for phase classification. In contrast, for the problem of phase change detection, working set size can be a good indicator of when phases change.
- Working set bit vectors - we use a bit vector similar to those used in [6]. For each memory access, the lower m bits of the address are dropped, and the result is hashed into a 32K bit vector. This bit vector indicates memory working set size, as well as the memory chunks accessed. After every interval we classify using the same relative distance metric as the code bit vectors in [6], and group phases using the same smallest relative distance below the threshold from [5].
  Using bit-vectors on memory accesses did not perform nearly as well as using bit-vectors on instruction accesses. A large part of this is due to the higher dimensionality of the memory accesses. Since so many more memory locations are accessed, the bit-vector is subject to aliasing. Increasing the bit-vector's size to 32K bits reduced the aliasing somewhat, but the memory bit-vector was still highly sensitive to noise. Due to the larger vector size necessary and its noise-sensitivity, using memory bit-vectors (for code or data addresses) classified significantly more phases than any of the other methods we examined. While these memory bit-vectors may be adequate for detecting phase changes, they did not perform as well for the goal of phase classification.
- Memory access frequency vectors - each element in the frequency vector is a counter for 256 contiguous bytes of memory. Whenever any word of memory is accessed, we increment the counter corresponding to the 256-byte region that the word lies in. These vectors did not work well for phase classification because two intervals with very different memory access vectors often have similar behavior. This occurred when two intervals with similar behavior walked over different parts of memory. This realization was the motivation for the memory stride vectors described above.

6 Overall Comparison

In this section, we summarize and compare the effectiveness of the various types of vectors discussed in this paper.

Figure 12 shows the average CPI CoV for all vector types, and Figure 13 shows the average error in estimated CPI when we use these vectors to guide the selection of simulation points. It is clear that many of the vector types that we experiment with can be used to produce accurate phase classifications, with low overhead. Figure 14 shows the average number of phases detected by each vector type. We see that all vector types detect 40-60 phases on average. It is interesting to note that register use vectors result in the detection of fewer phases compared to basic block vectors, yet CPI CoV is not significantly affected.

6.1 Comparison Over Multiple Cache Configurations

All the metrics we track are independent of the underlying performance metrics and are not tied to any particular architecture using the same ISA. Therefore, the phase behavior that we discover should appear on different architecture configurations. To evaluate this claim, we compute CoV of CPI and SimPoint error in estimated CPI on 18 processors with very different memory hierarchy configurations. To produce these configurations, we start with the baseline architecture,

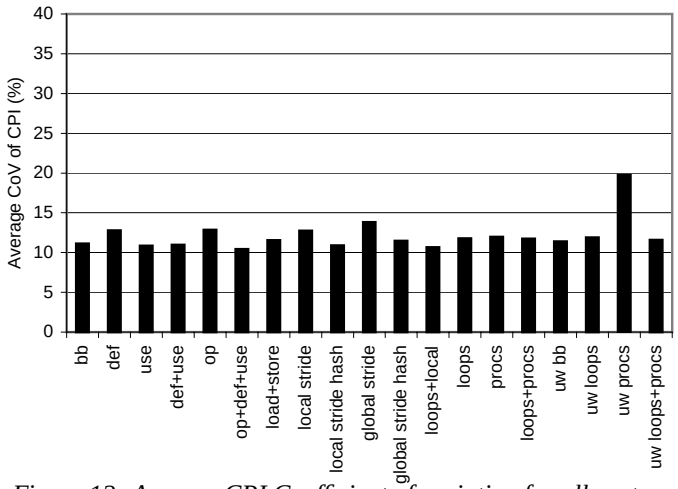

Figure 12: Average CPI Coefficient of variation for all vector types.

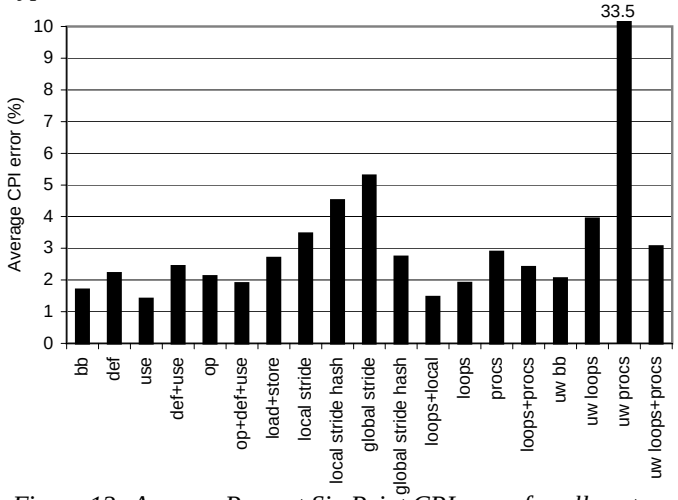

Figure 13: Average Percent SimPoint CPI error for all vector types.

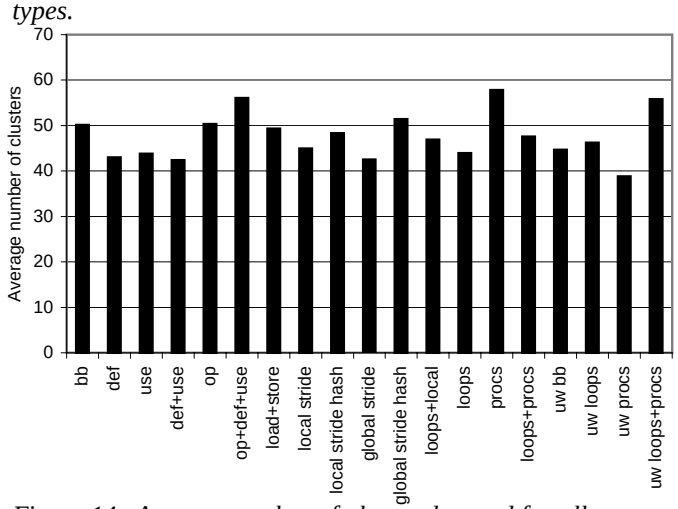

Figure 14: Average number of phases detected for all vector types.

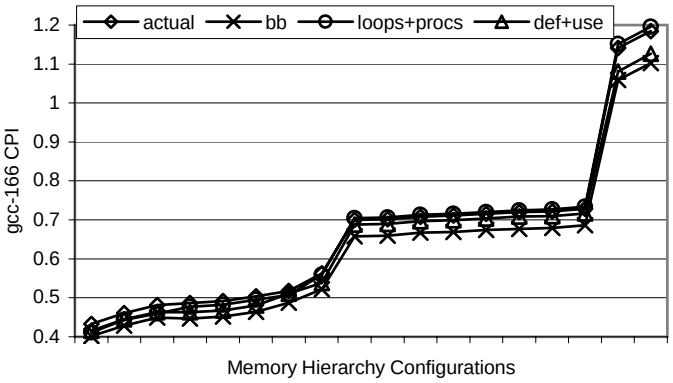

Figure 15: Actual CPI and calculated CPI on 18 different memory hierarchy configurations for gcc-166. Each point on the x-axis represents a different configuration.

and vary the latency, size, and associativity of the L1 and L2 caches, and memory as described in [10].

Figure 15 shows the actual CPI and estimated CPI with various vector types for gcc-166. It is clear that the estimated CPI and the actual CPI are highly correlated for the vector types that we consider, even across different architectures, and even for `gcc`, a benchmark with complex phase behavior. This indicates that the phases discovered can be used to estimate performance across different architecture configurations. The performance within each phase is homogeneous, even on architectures with different memory hierarchies.

Figure 16 shows the CoV of CPI averaged across gcc-166 and gzip-graphic on all architecture configurations, and the error in estimated CPI. Note, these results are only for two programs, so it is difficult to compare these with the results in the previous section over many programs. Ideally we would like to show 18 configurations and 18 benchmark+input pairs, but a huge amount of simulation time would be required. Overall, these results show that tracking alternative program structures (such as register usage, or loops), we can accurately capture program behavior with low overhead, even across different memory hierarchy configurations.

7 Summary

This paper focused on performing phase classification by tracking the use of program structures. All the structures we track are independent from underlying architecture metrics. Therefore, the phases we discover are not tied to any particular architecture configuration. Our prior approach to phase classification used basic block frequency vectors, and the focus of this work was to examine different program and ISA level structures to accurately and concisely perform phase classification.

The following are the main observations we found:

- Weighted loop frequency vectors result in almost as low CoV in CPI as basic blocks vectors. Tracking unweighted

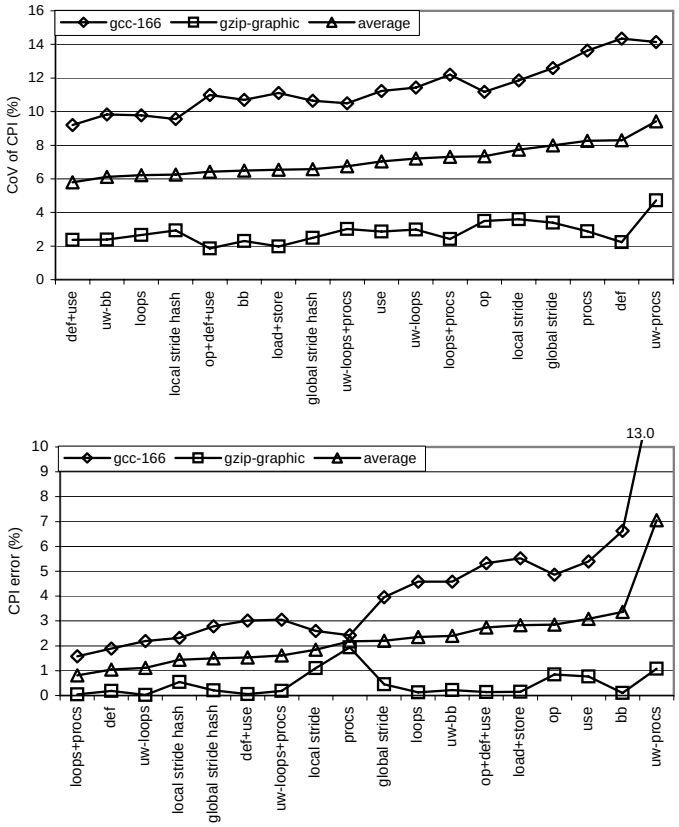

Figure 16: CPI Coefficient of variation and calculated CPI error (%) for all vector types averaged across different 18 configurations.

loop vectors provides slightly less accurate results, but with shorter vectors, as shown in Table 2.

- Tracking register usage results in phase classifications with slightly more accurate results (in terms of CoV of all metrics) than basic block vectors, with even shorter vectors. Register usage vectors are particularly attractive because of their small size: the length of each register use vector is equal to the number of registers in the instruction set. Compared to basic block vectors, the register use vectors are quite efficient.
- Profiling memory strides produces phase classifications with slightly less accuracy than basic block vectors, with the memory vectors are significantly longer than basic block vectors. However, hashing local stride with the PC of the current memory access instruction and combining local stride with the loop vectors produced slightly lower overall CoV.

Overall, we found that the register use vectors and loop vectors are efficient yet very effective alternatives to basic block vectors for phase classification.

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] 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.

[2] 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.

[3] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

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

[5] 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.

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

[7] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *12th International Conference on Parallel Architectures and Compilation Techniques*, October 2003.

[8] 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.

[9] 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. Morgan Kaufmann, San Francisco, CA, 2000.

[10] 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.

[11] 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.

[12] 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.

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

[14] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.