In the 11th International Symposium on High Performance Computer Architecture, February 2005

Transition Phase Classification and Prediction

Jeremy Lau   Stefan Schoenmackers   Brad Calder

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

Abstract

Most programs are repetitive, where similar behavior can be seen at different execution times. Proposed on-line systems automatically group these similar intervals of execution into phases, where the intervals in a phase have homogeneous behavior and similar resource requirements. These systems are driven by algorithms that dynamically classify intervals of execution into phases and predict phase changes.

In this paper, we examine several improvements to dynamic phase classification and prediction. The first improvement is to appropriately deal with phase transitions. This modification identifies phase transitions for what they are, instead of classifying them into a new phase, which increases phase prediction accuracy. We also describe an adaptive system that dynamically adjusts classification thresholds and splits phases with poor homogeneity. This modification increase the homogeneity of the hardware metrics across the intervals in each phase. We improve phase prediction accuracy by applying confidence to phase prediction, and we develop architectures that can accurately predict the outcome of the next phase change, and the length of the next phase.

1 Introduction

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program). During one part of execution a program may be completely memory bound while in another it may always be stalling on branch mispredictions. Due to this time-varying behavior of programs, average statistics gathered about a program may not give an accurate summary of a program's actual behavior. This realization has ramifications for many architecture and compiler techniques including thread scheduling on multi-threaded machines, feedback-directed optimizations, power management, and architecture simulation. To take advantage of the time-varying behavior of programs, we must first develop techniques to automatically and efficiently discover similarities and changes in program behavior over time. We refer to the time varying behavior of programs as "phase behavior."

Phase behavior can be detected by examining a program's working set [5]. More recent research accurately classifies and predicts phases in program execution [1, 7, 8, 23, 24, 25, 21, 10, 14]. Dynamic phase behavior can be exploited to save energy by dynamically reconfiguring caches and processor width [1, 25, 8, 7], to guide compiler optimizations [20, 2], and to assign processes to cores in a heterogeneous multi-core architecture [19]. All of these techniques take advantage of program phase behavior.

To identify phases, we divide a program's execution into non-overlapping intervals. An *interval* is a contiguous portion of execution (a slice in time) of a program. 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 reappear many times as a program executes. *Phase classification* partitions a set of intervals into phases with similar behavior. *Phase prediction* predicts the phase classification of the next interval of execution.

The approach of Sherwood et al [25] focused on identifying phase behavior by tracking executed code. They found a strong correlation between the code profile and the hardware metrics of each interval. They then proposed an architecture for dynamic phase classification and prediction based on code signatures [25]. The architecture in [25] consisted of an *Accumulator Table* to track the code signature of the current phase, a *Past Signature Table* to remember prior code signatures, and a *Phase Prediction Table* to predict the phase that the next phase signature will be classified into. Our work focuses on improving this phase classification and prediction architecture.

In this work we focus on dynamically classifying a program's execution into phases, predicting the phase of the next interval, predicting the outcome of the next phase change, and predicting the length of the next phase. In this paper, we focus on a granularity (instructions per interval) of 10 million instructions, which is at the level of context switching. We focus on this large size because we are interested in applying this technique to phase-based task scheduling optimizations in the future. These optimizations include phase-aware symbiotic task co-scheduling on SMT machines [27] and prediction of remaining execution time for DVS task scheduling in hard real-time systems [18, 26].

The contributions of this paper are:

- **Transition Phase** - An important aspect of dynamic phase classification not dealt with in prior work [25] is how to handle phase transitions. Programs rarely exhibit clean transitions from one phase to the next - instead they often spend some time exhibiting unique behavior between stable phases. These phase transition intervals do not last for very long, and they do not occur very often, which makes it difficult to optimize for their behavior. We group all phase transition intervals into a single *Transition Phase*. This results in the identification of fewer phases, which improves signature table utilization, and increases phase prediction accuracy.

- **Adaptive Classification** - The prior dynamic phase classifier used a single static similarity threshold for all programs to determine if the current interval was similar to a previous interval. We found that one threshold does not fit all programs, and we provide a dynamic approach for adjusting the similarity threshold. We adjust the thresholds on a per-phase basis to split phases with poor homogeneity into multiple smaller phases to improve the homogeneity of each phase.

- **Confidence for Next Phase Prediction** - We examine the benefits of adding confidence counters to the phase prediction approach described in prior work [25], and we show that a phase predictor with confidence counters can achieve 80% accuracy with 70% coverage.

- **Phase Change Prediction** - Prior work focused only on predicting the phase of the next interval of execution. When focusing only on predicting the outcome of the next phase change, we found that only 20% of phase changes were predicted correctly with the predictor described in [25]. We examine improved predictors that can correctly predict the outcome of the next phase change 40 to 60% of the time.

- **Phase Length Prediction** - An expensive optimization or reconfiguration should only be applied if we can amortize its cost over a significant amount of execution. We present an architecture to predict the approximate length of the next phase.

2 Related Work

Several researchers have recently examined program phase behavior. Balasubramonian et al [1] used hardware counters to collect miss rates, CPI and branch frequency information for every 100K instructions. They use the miss rate and the total number of branches executed for each interval to evaluate the stability of the program's behavior. They performed phase-aware dynamic cache reconfiguration to save energy without sacrificing performance. In addition, they examined an Accounting Cache to track statistics for many different cache configurations for each interval to identify working set changes [9].

Dhodapkar and Smith [7, 8, 6] found a relationship between phases and instruction working sets, and they found that phase changes occur when the working set changes. They developed hardware techniques to detect phase changes due to working set changes. With these techniques, multi-configuration units were re-configured in response to phase changes. They used their working set analysis to save energy through instruction cache, data cache and branch predictor re-configuration [7, 8].

Sherwood et al [23, 24], proposed that phase behavior in programs can be automatically identified by profiling the code executed. They used techniques from machine learning to classify the execution of the program into phases (clusters). They found that intervals of execution grouped into the same phase had similar behavior across all architectural metrics examined. They extended this approach to perform hardware phase classification and prediction [25], which we improve upon in this paper.

Deusterwald et al [10] used hardware performance metrics for a given set of intervals along with the instruction mix to predict the performance metrics' next value. This is similar to phase prediction, but instead of predicting a phase ID for the next interval, the value of a hardware metric value is predicted. We focused on extending the phase ID prediction of [25], because it can be used to predict several metrics at once. In addition, the phase ID is based on the code executed, and not on hardware metrics, so it will remain constant across hardware reconfiguration optimizations, as long as the instruction sequence is not changed.

Huang et al [13] examine tracking procedure calls via a call stack, which can be used to dynamically identify phase changes. Isci and Martonosi [14, 15] dynamically identify power phase behavior with power vectors. Shen et al [22] use Wavelets and Sequitur to build a hierarchy of phase information to represent a program's behavior patterns based on data reuse traces.

Hind et al [12] provide a framework for defining and reasoning about program phase classification by focusing on how to appropriately define granularity and similarity to perform phase analysis.

3 Methodology

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`, `perl` were run with multiple inputs. We chose the above programs since they were the most interesting and challenging for phase classification from our prior studies. From our prior work, `gcc`, `perl` and `galgel` are the most difficult benchmarks for our code-based phase classification approaches out of all of the SPEC benchmarks. `bzip` and `gzip` have complex hierarchical phase patterns and phase behavior. `mcf` is a pointer-based application with a large number of cache misses. We collect all of our profiles with SimpleScalar [3].

We abbreviate the input names as follows: `bzip2/graphic` is `bzip2/g`, `bzip2/program` is `bzip2/p`, `gcc/166` is `gcc/1`, `gcc/scilab` is `gcc/s`, `gzip/graphic` is `gzip/g`, `gzip/program` is `gzip/p`, `perl/diffmail` is `perl/d`, and `perl/splitmail` is `perl/s`.

To generate our baseline fixed length interval results, all programs were executed from start to completion using SimpleScalar. We measure performance and perform phase classification using an interval size of 10 million instructions. The baseline micro-architecture model is detailed in Table 1. In this paper, we focus on intervals of 10M instructions due to space constraints. Similar code-based phase classification techniques [21] work very well at 1M and 100M interval sizes. The minimum number of instructions per interval that can provide enough information for code-based phase classification is an interesting open question, but answering this question is beyond the scope of this paper.

3.1 Metrics for Evaluating Phase Classification

Phases are groups of intervals with similar program behavior, so we measure the effectiveness of our phase classifications by examining 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: $CoV = stddev/mean$.

We use the CoV to compare different phase classifications. 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 use to compare different phase classifications for a given program. It measures the average percentage of deviation within each phases in a phase classification.

4 Phase Classification

Dynamic phase classification can be performed with little to no impact on the design of the processor core. Phase classification divides a program's execution into a set of phases, where each phase represents a unique behavior pattern. This means that an optimization adapted and applied to a single segment of execution from one phase will apply equally well to other segments of execution from the same phase.

4.1 Prior Phase Classification Architecture

The goal of [25] was to create a simple, easily implementable (in hardware or software), run-time phase detection scheme. The challenges of designing an on-line scheme to capture phases are: a) the on-line scheme must discover a program's phase behavior as the program executes, b) only a small fixed amount of storage is available, and c) the computation involved must be low.

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.

An overview of the phase classification architecture is shown in Figure 1. We summarize the approach in [25]:

1. **Track the Code** - The first step is to create a signature of the code currently being executed. The program counter (PC) of every committed branch and the number of instructions (I) committed between the current branch and the last branch are recorded. These two values are placed into a queue, shown on the left side of Figure 1.

2. **Projection and Accumulator Table Update** - An array of N saturating counters (accumulators) holds a signature for the current interval. The PC at the head of the queue is hashed into one of the N counters, and the selected counter is incremented by I. I is used to track the proportion of code executed. For the results in [25], 32 counters were sufficient to detect phase behavior. These first two steps need to be performed at the speed of the processor for each branch committed, but they require only a counter, a hash, and an accumulator update, all of which can be pipelined.

3. **Classification** - After each interval of execution, we classify the signature of the previous interval. To perform phase classification, we must determine if the signature for the previous interval is similar to past signatures. The *Signature Table* in Figure 1 stores signatures seen in the past, along with a unique Phase ID for each signature. To determine if we have seen a similar signature in the past, we search the signature table. If there is an entry in the table that is within a *threshold similarity distance* relative to the current signature in the accumulators, then the current signature is similar to other intervals that were classified into that signature table entry. The Manhattan distance is used to calculate this similarity distance.

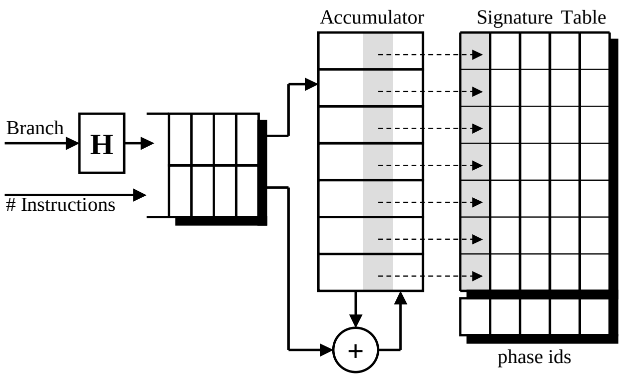


Figure 1: The baseline phase classification architecture. Each branch PC is captured along with the number of instructions executed since the last branch. A hash of the branch PC determines the accumulator entry that will be incremented by the number of instructions executed. After each profiling interval, the values in the accumulators form a signature of the code executed in the last interval. This signature is classified by comparing it against signatures seen in the past

> When a match occurs, the matching signature in the table is replaced with the current signature, and the phase ID for the matched signature table entry is returned. If no match is found, then the signature is marked as a representative of a new phase, and it is inserted into the signature table and allocated a new phase ID. In all cases, a phase ID is returned.
>
> Multiple signatures in the table may satisfy the similarity threshold. When these situations occur, we must select a phase for the current signature. In this paper, we choose the phase that best matches the current signature. In other words, we choose the phase whose signature is most similar to the current signature. The prior approach in [25] classified the current interval into the first phase that satisfied the similarity threshold. We found that choosing the phase with the most similar signature improves the homogeneity of our phase classifications. For all of the results in this paper, when multiple signatures satisfy the similarity threshold, we choose the most similar past signature.

The above phase classification algorithm can be implemented in hardware or software to dynamically identify program phase behavior. With appropriate support, phase behavior can be exploited when the same phase ID occurs again.

4.2 Forming the Signature

Each entry in the accumulator table is 24 bits, so it will never overflow with 10 million instruction intervals. It is not necessary to maintain all 24 bits for signatures in the signature table. In [25], Sherwood et al stored 8 bits from each accumulator in the signature table, resulting in a 256 bit signature. We statically selected bits 14 to 21 from each 24 bit counter when copying the current signature into the table. These bits were selected through design exploration. The choice of bits depends on the number of accumulators used and the current interval length. We describe a method to automatically determine which bits to copy from each accumulator, given the number of accumulators and the number of instructions per interval.

For our approach, we dynamically choose which bits to copy from the accumulators for each interval of execution. We first compute the average counter value by keeping track of the total amount each counter has been incremented by, and dividing the total count by the number of counters. This division can be performed quickly in hardware if the number of counters is a power of two. Next, we look at the binary representation of the average counter value, and we determine the number of bits needed to represent the average. Finally, we take the number of bits needed and keep two more, so our counters can represent values 2 to 4 times larger than the average. We find that we very rarely see counter values larger than twice the average counter value. If we do find a value larger than twice the average (we check if a more significant bit is set above the bits that we select), we set all of the selected bits to one. This results in the signature table entry containing the maximum possible value when the actual value is too large for us to represent.

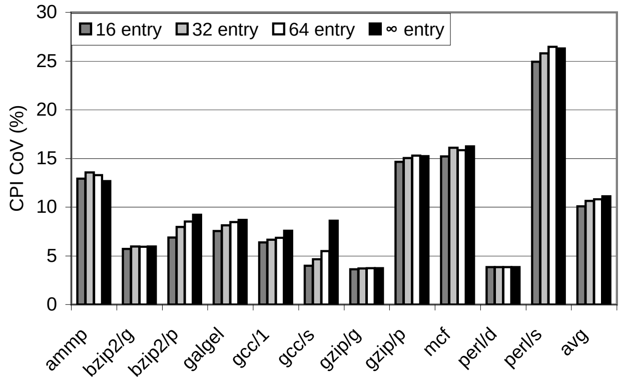
When we compare signatures, we always use the two bits over the average counter value, and for the results in this paper, we use 4 additional less significant bits, for a total of 6 bits per counter/dimension. We found that using fewer than 6 bits per counter produced poor classifications, and using more than 8 bits did not significantly improve results. These findings are consistent with those found in [25].

4.3 Baseline Phase Classification Architecture

We now examine the performance of a baseline phase classification architecture based on [25]. In [25], 32 counters were kept for each signature, and the past Signature Table held an infinite number of entries. Figure 2 shows results when a fixed table size is used with LRU replacement. The left graphs show the CoV of CPI (standard deviation divided by average as described in Section 3), and the right graph shows the number of phases. For these configurations, a similarity threshold of 12.5% is used, meaning a signature must differ from a past signature by less than 12.5% to be classified into the past signature's phase.

We find that a finite number of table entries results in a significant increase in the number of phase IDs generated. The increase in the number of phase IDs generated occurs because signatures are lost due to replacement.

Figure 3 shows the effect of varying the number of accumulator counters used for each signature. The left graph shows CoV of CPI, and the right graph shows the number of phase IDs produced. These results show that 8 counters



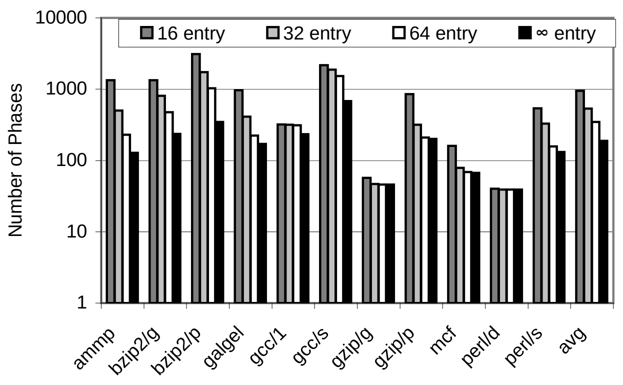
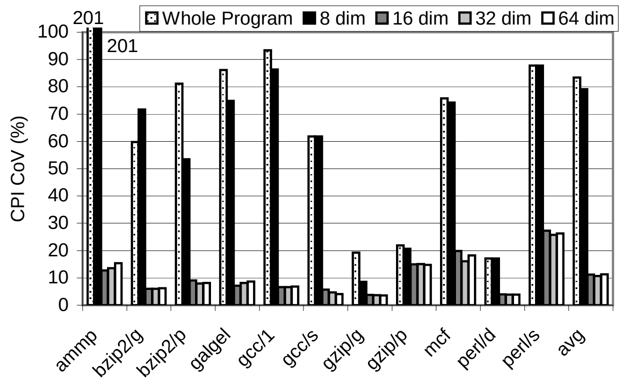


Figure 2: Average per-phase CPI CoV (standard deviation divided by average) and number of phases detected for different numbers of Signature Table entries. The number of phases detected decreases dramatically as we increase the number of entries in the signature table. CPI CoV increases slightly as we add more table entries, due to the reduced number of phases.



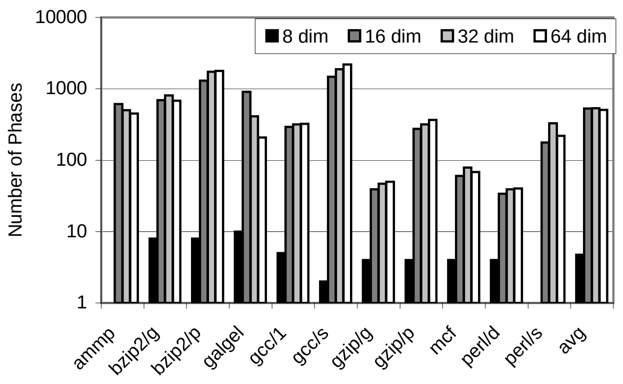


Figure 3: CPI CoV and number of phases detected for different numbers of signature counters. Each signature counter represents a dimension of the projected vector. 8 counters is clearly insufficient for phase classification.

do not provide sufficient resolution for phase classification. These results are consistent with those found in [24], which showed that at least 15 dimensions were needed for best results. In [25] we used 32 counters for each signature, but for the rest of this paper we use 16 counters per signature. We use a 32 entry past signature table with LRU replacement.

The Whole Program results in Figure 3 show the CoV of CPI over all of the intervals in the program. The CoV of CPI over all the intervals in a program is 80% on average. By classifying the intervals into phases according to their code signatures, we can reduce the per-phase CoV of CPI to around 10%.

4.4 Stable and Transition Phases

An important aspect of dynamic phase classification not dealt with in prior work [25] is how to handle phase transitions. Using fixed length intervals, programs rarely exhibit “clean” transitions from one phase to the next - instead they often spend some time exhibiting unique behavior between stable phases. These phase transition intervals do not last for very long, and may not occur very often, which makes it difficult to optimize for their behavior. It is important to identify these infrequently occurring behaviors as transitions because doing so reduces the number of unique Phase IDs generated, which reduces pressure on the signature table and improves prediction accuracy. We group all phase transition intervals into a single *Transition Phase*.

The transition phase is represented with phase ID zero. Each entry in the signature table is augmented with a small saturating usage counter called the *Min Counter* that counts the number of intervals that have been classified into each phase. If the current signature does not match any signature table entries, the new signature is added to the signature table, the Min Counter is set to zero, and the signature is assigned phase ID zero (transition phase). If the current signature matches an existing signature table entry, the Min Counter for that entry is incremented. When the Min Counter goes above a threshold value, the signature will be assigned a real phase ID. With this approach, both infrequently appearing behavior and the first few intervals from each stable phase will be assigned to the transition phase. Stable phases are long, so misclassifying the first few intervals from each stable phase is a small price to pay for the benefits of transition phase classification.

Figure 4 shows four graphs that evaluate the utility of the transition phase. Results are shown with a 12.5% and 25% similarity threshold and a transition min counter threshold of 4 and 8. A value of 4 means that each signature must appear 4 times before it is considered stable, otherwise those intervals are put into the transition phase. The baseline result is a configuration with a 12.5% similarity threshold and a min count threshold of 0. The first graph (top left) shows that the CPI

CoV is not significantly affected by the addition of the transition phase. The transition phase is not included in the CPI CoV calculations. The bottom-left graph shows the percentage of intervals classified into the transition phase.

The second graph (top right) shows that tens of phase IDs are generated (instead of hundreds) with a transition phase. A min count threshold of 8 results in 30% of the program's intervals classified into the transition phase for programs like `gcc/scilab` due to somewhat frequent transitions between relatively short stable phases. On average the transition phase accounts for about 6% of the program's execution for a similarity threshold of 25% and min count of 8. The last graph (bottom right) shows the number of mispredictions produced by a last-value phase ID predictor for the baseline configurations with and without the transition min counter. Last-value phase prediction is discussed in more detail in Section 5.2.1. By placing infrequently occurring phase IDs into a transition phase, we reduce mispredictions by 6% compared to our baseline. The remaining results in this paper use a transition min count threshold of 8.

The percent of execution (intervals) that are classified as phase transitions affects the amount of optimization that can potentially be applied to a program. Therefore, it is important to maintain a low CoV, accurately classify intervals into stable phases and transitions, and maintain high predictability. Out of the configurations in Figure 4, the 25% similarity threshold provides a good balance. It has a higher CoV than the 12.5% similarity threshold configuration, but it has a lower phase ID misprediction rate and significantly fewer intervals classified into phase transitions. Another reason why we prefer the 25% similarity threshold configuration is because the resulting CPI CoV and number of phases produced are comparable to the results of the offline phase classification algorithm used in SimPoint [24].

4.5 Average Stable and Transition Run Lengths

As defined in [6], the phase length is the number of contiguous intervals that are classified into a single phase. For dynamic optimizations, it is important to have long stable phases, and short transitions. However, longer stable phases and shorter transition phases generally lead to higher CoVs. In the extreme, a classifier may classify all intervals into a single phase. That phase will be as long as possible, but it will also have a high CoV. Alternatively, a classifier may classify all intervals into separate phases. In that case, the CoV would be zero, but with a maximum phase length of 1, optimizations could not learn from past program behavior. Any classifier must make tradeoffs between phase length and intra-phase variability; the best tradeoff depends on how the phase information will be used. Some applications of phase behavior can tolerate more intra-phase behavior variability than others.

Figure 5 shows the average stable phase length and average transition phase length, with error bars indicating the standard deviation in phase length. For all programs except gcc, the average stable phase length is higher and has a greater variability than the transition phase lengths. This is ideal, since it indicates that the classifier is finding long stable phases. `perl/diffmail` and `gzip/graphic` have exceptionally high average lengths, because they are both relatively short, and the classifier only identifies a few phases, most of which are long and stable.

4.6 Performance Feedback to Dynamically Adjust Thresholds

The phase classifier described in [25] explored the design space and chose a static similarity threshold to use for all programs for classifying signatures into phases. We found that a single similarity threshold does not work well for all programs, and we suggest a dynamic approach for adjusting the similarity threshold. For example, if we compare the CPI CoV results for 12.5% threshold and 25% threshold in the top left graph of Figure 4, we see that we can produce more homogeneous clusterings for some programs by using a lower threshold, such as `mcf`, while other programs do not benefit, such as `gzip/graphic`.

The goal is to adapt the similarity threshold to balance the tradeoff between homogeneity (CoV CPI), number of phases, predictability and time spent in transitions. The previous section assumed an overall static similarity threshold used for all phases. In this design, we keep a similarity threshold *for each signature* in the Signature Table. This similarity threshold is initialized to a static value, but can be tightened based on the homogeneity of the intervals classified into that phase.

To determine when to adjust the similarity threshold, we monitor the CPI of the intervals that are put into the phase. When a new phase ID is created, we store a running average of the CPI with the phase ID. If the signature currently being classified into a phase has a significant CPI deviation, as determined by a *performance deviation threshold*, from the average CPI for the phase, then the similarity threshold is lowered (cut in half). When this occurs, the average CPI and statistics associated with that phase ID are cleared. It is important to note that we still perform phase classification based only on code signature similarity. The performance feedback metric (CPI) is only used to determine when a given phase ID threshold should be tightened. If an application performs an optimization that may affect the program's CPI, the feedback-based phase classifier will have all of its CPI data flushed during reconfiguration.

Figure 5 shows the results of the dynamic similarity threshold approach: dynamic thresholds result in lower CPI CoV with small increases in the number of phases detected and transition time, compared to the static threshold approach. Results are shown for a performance deviation threshold of 50%, 25% and 12.5%. Using a 25% deviation threshold means that when a signatures CPI differs by more than 25% from the average CPI for the best-matching signature table entry, the similarity threshold is halved for that signature table entry.

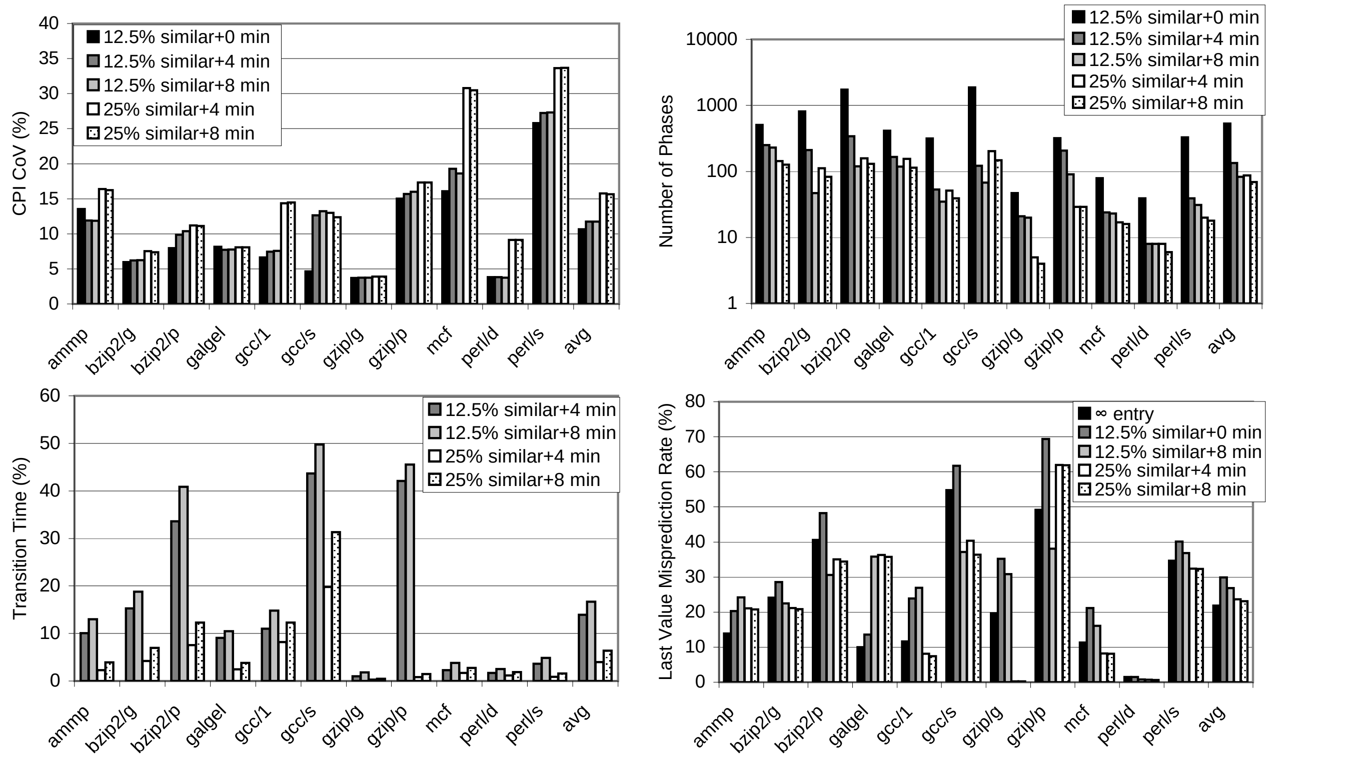


Figure 4: CPI CoV, number of phases detected, normalized transition time, and last value misprediction rate for a similarity threshold of 25% and 12.5%, and min count threshold of 4 and 8. "25% similar" means a signature can be no more than 25% different from a past signature to be classified into that phase. "4 min" count means 4 intervals must be classified into a phase before it is assigned a real phase ID. Intervals classified into a phase with no phase ID are classified into the transition phase. We compare a classifier with infinite signature table entries, our baseline configuration (32 entries, 12.5% similarity threshold), and a classifier that uses a transition phase thresholds of 4 and 8.

When we use the 25% deviation threshold, we see our approach does not significantly affect programs that do not benefit from a lower similarity threshold such as `gzip/graphic` and `galgel`: the results are quite similar to the 25% static similarity threshold results. On the other hand, programs that do benefit from a lower similarity threshold, such as `mcf` and `perl/splitmail`, result in a lower CoV of CPI, without the creation of significantly many more phases.

5 Next Phase Prediction

The prior section described our phase tracking architecture, and how it classifies intervals into phases. In this section, we focus on phase prediction. For many applications, it may be beneficial to predict future phase changes. By predicting future phase changes, the system can reconfigure for the code it will execute, rather than reconfiguring in response to changes in program behavior. By dynamically classifying the program into phases, we can not only predict when a phase change will occur, but also predict what the next phase will be.

Many phase prediction techniques have been proposed ranging from simple last value predictors to more accurate Markov models [25, 10]. This prior work focused on *Next Phase Prediction*. Next phase prediction predicts the Phase ID for the next interval of execution. In this section, we re-evaluate next phase prediction with our new phase classifier, and we examine the accuracy and coverage with confidence counters.

For this study we used a predictor architecture that allows us to examine a few different types of predictors. Each predictor we examined falls into one of two categories. The *Last Value Predictor* always predicts that the phase of the next interval will be the same as the phase of the previous interval. It is used both as a baseline, and as the default prediction when we do not have a prediction from a better predictor.

We also examine a variety of *Phase Change Predictors*, which predict the outcome of the next phase change. We experiment with a variety of phase change predictors, including Markov predictors and Run Length Encoding predictors. These predictors all store phase change information in tables, which we call the "phase change table." The contents of the table vary depending on the type of predictor used, but each predictor uses its table to predict the outcome of the next phase change.

5.1 Adding Confidence

In this paper, we use confidence information for phase prediction. As with most predicted data, it is important to know how certain we are that the prediction is correct. Confidence

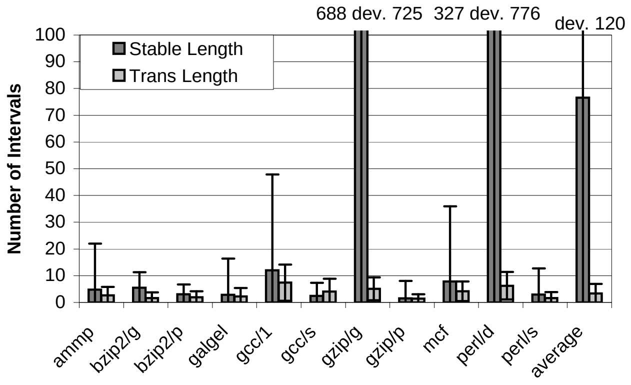


Figure 5: Average phase lengths, in intervals of 10M instructions, for stable and transition phases. The error bars indicate the standard deviation in phase lengths.

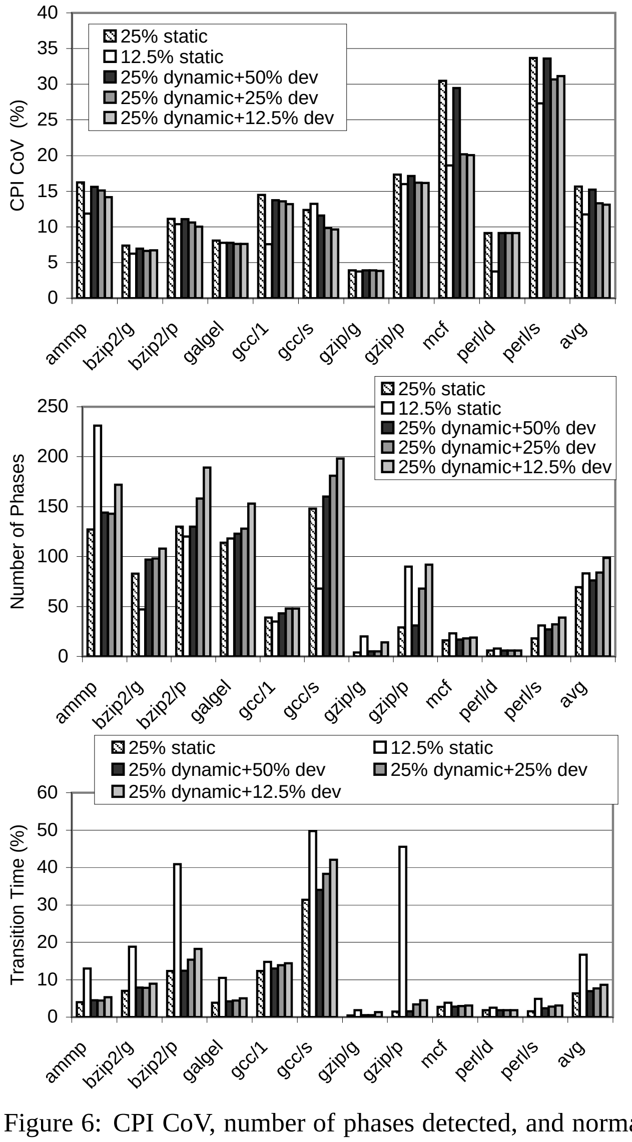


Figure 6: CPI CoV, number of phases detected, and normalized transition time for a variety of phase splitting configurations. “25% static” means a static 25% similarity threshold is used for phase classification. No phase splitting occurs in static configurations. “25% dynamic+50% dev” means a dynamic threshold is used. The per signature thresholds start at 25%, and tighten whenever we classify an interval whose cycle count differs from the average for the phase by 50%.

counters [16, 11] are a common technique to evaluate the confidence of predictions is the use *confidence counters*. A confidence counter is an N bit saturating counter. Each time a correct prediction is generated, the counter is incremented by certain amount, and each time an incorrect prediction is generated, the counter is decremented by another amount. When generating predictions, if the current value of the confidence counter is above a threshold (typically close to the maximum), then it is confident, otherwise, it is not confident. The idea is to verify predictions a few times before trusting them. Confidence counters reduce the number of incorrect predictions due to events that are difficult to predict, and build confidence in correct predictions.

For the designs in this section, two sets of confidence counters are used. One set is associated with the phase change predictor, and the other set is associated with the last value predictor. For the phase change predictors, each entry in the phase change table has its own confidence counter. When an entry is first inserted into the table, its confidence counter is reset. Every subsequent time that entry is used for prediction, we increment or decrement the counter depending on whether the prediction was correct or incorrect.

For the last value predictor, we maintain a confidence counter for each phase. Whenever a new entry is added to the phase ID signature table, we reset the associated confidence counter. Each time a correct last-value prediction occurs, we increment the counter for that phase, and every time an incorrect last-value prediction occurs, we decrement the counter for that phase. This has the benefit of advancing stable phases to a confident status and demoting unstable phases to an unconfident status. This is desirable, since predicting last value will do well in stable phases, and poorly in rapidly changing ones.

Since incorrectly predicting a phase change is generally worse than failing to detect a phase change (as predicting a phase change may prematurely start an expensive optimization), we only use confident phase change table results when performing next interval phase prediction. If they are not confident, we use the last value prediction.

For all of the results in this section, we use a phase classifier as described in the previous section with 6 bits per accumulator, 16 accumulators, 32 signature table entries, 25% similarity threshold, 8 min counter threshold, and 25% performance deviation threshold. All the phase predictors treat the transition phase like any other phase. The phase change predictor has 32 entries and is 4-way set associative. We experimented with a variety of confidence counter configurations for both types of confidence counters, but due to space constraints we only show one configuration here. We use a 3 bit confidence counter for last value predictions with a confidence threshold of 6 (1 less than fully saturated), and we use a 1 bit counter for the phase change predictions. In both cases we increment and decrement by 1 for correct and incorrect predictions.

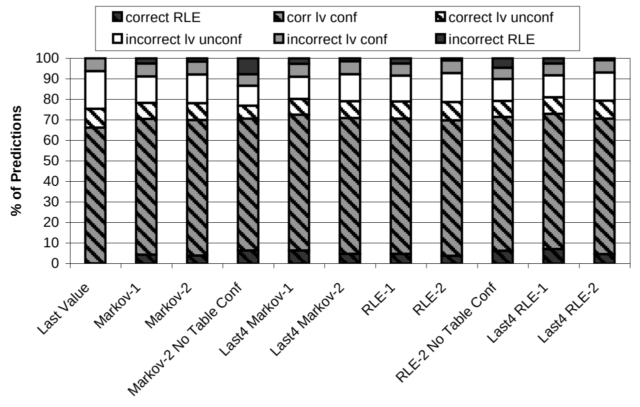


Figure 7: Next Phase Prediction Results. Next Phase Prediction predicts the Phase ID for the next interval of execution.

5.2 Next Phase Prediction

Next phase predictors predict the Phase ID for the next interval of execution, and this is done for every interval of execution. The information used to perform the prediction includes information about the current and past Phase IDs and their lengths. Figure 7 shows all of the results for next interval phase prediction discussed in this section.

5.2.1 Last Value

The simplest predictor we consider is a last-value predictor. As its name implies, a last value predictor always predicts that the value will stay the same. So if the current interval was classified into phase A, the last-value predictor will predict that the next interval will also be classified into phase A. The first bar in Figure 7 shows the accuracy of using last value prediction. Without confidence counters, last value phase prediction achieves 75% accuracy on average with 25% miss rate. With confidence counters, we achieve 67% accuracy with a miss rate of just 7%.

The last value results also show that 25% of the transitions between adjacent intervals results in a phase change. Prior work [25, 10] used Markov models to capture these phase changes, which we revisit here.

5.2.2 History Markov

A classic prediction model that is easily implementable in hardware is the Markov Model. Markov Models have been used in computer architecture to predict everything from prefetch addresses [17] to branches [4] in the past. The basic idea behind a Markov Model is that the next state of the system is related to the last set of states.

We examine phase prediction accuracy with a 32 entry 4-way associative Markov table indexed by a hash of history of the last N *unique* phase IDs. We call these predictors Markov-N. Figure 7 shows prediction results for Markov-1 and Markov-2. These two designs store and predict a single Phase ID - we also consider storing the outcome of the last 4 unique phase changes. We call these predictors Last 4 Markov. These predictors produce a correct prediction if the actual outcome of the phase change is one of the last 4 unique outcomes. This could be an acceptable type of predictor if higher coverage is more important than accuracy. "Markov-2 No Table Conf" shows results for a Markov-2 model without confidence counters for the Markov table.

5.2.3 Run Length Encoding

By observing the phase classification stream,in [25] we found that two pieces of information are important for prediction: the set of phases leading up to the prediction, and the amount of time spent in those phases. We noted that misprediction rates could be reduced by creating a compressed representation of the program's stable state with a *Run Length Encoding* (RLE) Markov predictor [25]. The idea behind the predictor is to use a run-length encoded version of the Phase ID history to index into a prediction table. An RLE-N predictor uses the most recent N (Phase ID, Run Length) pairs from the run length encoded Phase ID history. The index into the prediction table is a hash of the phase identifier and the number of times the phase identifier has occurred in a row.

The lower order bits of the hash function provide an index into the prediction table. We then compare the tag for that entry with the current RLE history to determine if we have a match. When there is a match, the phase ID stored in the table provides a prediction of the next phase to occur in execution. When there is no match, the prior phase ID is predicted to be the next interval's phase ID.

We only update the predictor table when there is (1) a change in the phase ID, or (2) when there is a table match. We only insert an entry when there is a phase ID change, since we want to predict when the phase is going to change. Execution intervals where the same phase ID occurs several times in a row do not need to be stored in the table, since they will be correctly predicted with the last value prediction generated when a table miss occurs. This reduces table capacity constraints and avoids polluting the table with last value predictions. In addition, if there was a table tag hit on an interval classified into phase ID A, and the table incorrectly predicts a phase change (the next interval is actually classified into phase ID A), we remove the entry from the table, because the last value predictor would have produced a correct prediction in this case.

Figure 7 shows the next interval phase prediction accuracy for RLE-1, and RLE-2 with and without confidence counters. The Last4 RLE-N results shows the accuracy achieved if we store the outcomes of the last 4 unique phase changes, and we count matches with any of the last 4 unique outcomes as correct predictions. The results show that run-length encoding provides only a small improvement over Markov. It is able to accurately predict an additional 4% to 8% of the 25% interval transitions that change to a different phase ID.

Overall, Figure 7 shows that last value prediction works very well for next phase prediction, and more complicated

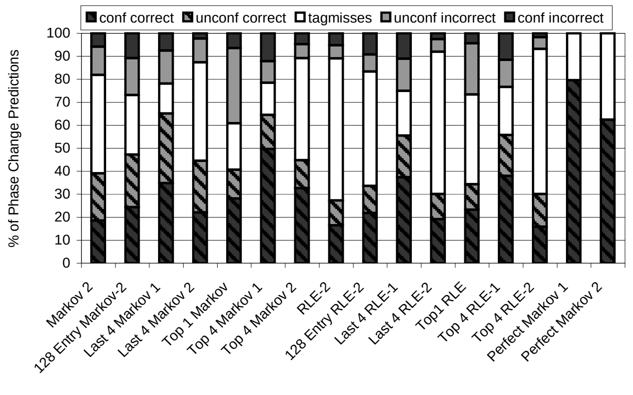


Figure 8: Phase Change Prediction Results. Phase Change Prediction predicts the outcome of the next phase change.

predictors provide marginal benefits considering the added complexity. This is because last value predictors are very good at predicting stable behavior, and most programs exhibit long periods of stable behavior. The more complicated predictors only improve prediction accuracy for phase changes, which we examine in more detail next.

6 Phase Change and Phase Length Prediction

The next interval phase prediction results in the last section showed that it is difficult to predict which intervals will end a stable phase, and that it is difficult to predict the outcome of the phase change. 25% of interval transitions are to a different phase ID, but only 20% of phase changes can be correctly predicted. This prompted us to focus on improving phase change prediction. *Phase Change Prediction* predicts the outcome of the next phase change. We also examine *Phase Length Prediction*, which predicts the length of the next phase.

6.1 Phase Change Prediction

Phase change prediction is the process or predicting what Phase ID will occur after the next phase change. These phase changes may invoke an optimization. In this subsection, we do not predict when the phase change will occur (we will look at that in the next section). The prediction results in the last section focused on predicting the phase of the next interval, while the results in this section focus on predicting the outcome of the next phase change, whenever it may occur.

25% of the interval change result in phase changes. Figure 8 shows the percentage of phase changes that can be correctly predicted with Markov and RLE predictors. The Perfect Markov N results were calculated by marking a phase change as correctly predicted if the phase change was ever seen in the past. This means that the last two columns of Figure 8 shows the percentage of phase changes that are seen more than once. The miss-rates of these perfect predictors show the impact of cold-start effects. The prediction rates for these perfect predictors are an upper bound on the accuracy of any predictor because even a perfect predictor with infinite memory can not correctly predict a phase change it has never seen. Figure 8 shows that a first order perfect Markov model can correctly predict at most 80% of phase changes due to cold-start effects.

For phase change prediction, we examined the same Markov and RLE predictors with 32 entry, 4-way associative tables as described in the previous section. Last 4 shows the coverage if each Markov table entry tracks the outcome of the last 4 unique phase changes, and counts a correct prediction if the outcome of the phase change matches any of the last 4 unique outcomes. The standard Markov and RLE predictors only track the outcome of the last phase change. We also examine a Top-N predictor that tracks the N most frequently occurring outcomes for each phase change. We count a correct prediction if the actual next Phase ID matches the most frequently occurring transition (Top-1) or any of the Top 4 most frequently occurring transitions (Top-4).

Figure 8 shows that 40% to 65% of the phase changes can be accurately predicted without confidence counters. Markov-2 can achieve 40% coverage, compared to the 62% of phase changes that occur more than once, with an 18% misprediction rate. Using the traditional confidence counters results in only a 5% misprediction rate, but coverage decreases to 19%. The biggest potential appears to be from being able to predict the phase change with Top-1, Top-4 or the Last-4 phase change predictors. In particular, Top-4 Markov-1 using confidence was able to achieve 50% accuracy with 11% mispredictions.

6.2 Phase Length Prediction

In addition to predicting which phase ID will come next, it is also useful to know how long the next phase will last for. Since program execution is typically long periods of stability broken up by short transition periods, knowing how long the next phase will execute for will help us avoid potentially expensive reconfigurations for phases that will not execute long enough to benefit from them. When we are about to leave a phase, we predict the length of the next phase (the phase after the phase change).

6.2.1 Run-length Classes

When starting a new phase of execution, we first tried to predict the exact length of the next phase (in terms of intervals), but this was difficult, since there were often small differences in phase lengths. But an application may just want to know the approximate length of the next phase: will it be short, medium, or long? For this study, we group phase run lengths into four sets: 1-15, 16-127, 128-1023, and >= 1024 intervals. This roughly corresponds to phase run lengths of 10-100M instructions, 100M-1B instructions, 1B-10B instructions, and more than 10B instructions.

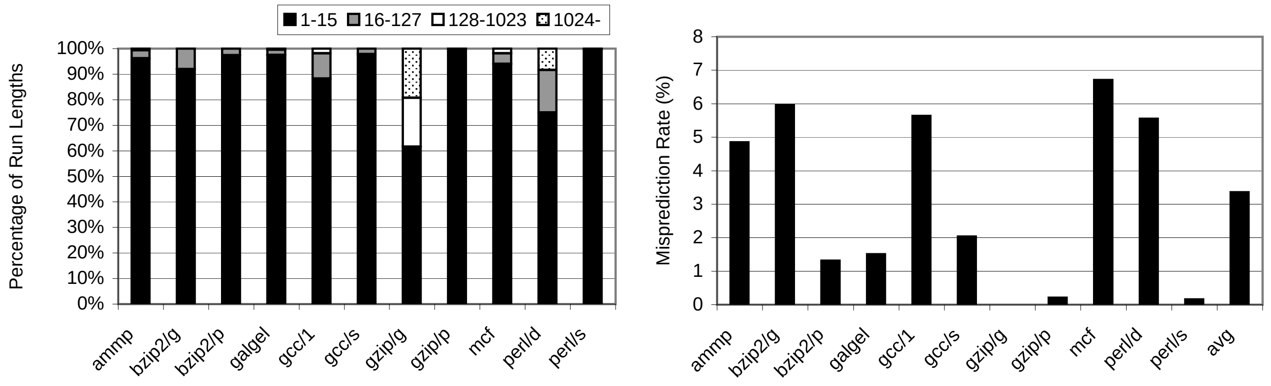


Figure 9: Percentage of Run Lengths shows the number of phases classified into each run length class. The Misprediction Rate is the accuracy of a 32 entry 4-way associative RLE-2 predictor when predicting the next run length class.

We consider a simple hardware predictor based on the architecture described in the prior section which, given phase length history, predicts which of these four classes the length of the next phase will fall into.

Figure 6.2 shows, for each phase change, how often the next phase was classified into each of our four phase length categories. These results are for classifying the run length for all phases, including the transition phase. It is important to notice that most of the programs have at least 90% of their phase changes transitioning into phase length runs of 150 million instructions or less. Therefore, just statically predicting a small phase will perform well for these programs. Even so, we found that a few programs like `gzip` and `perl` have 40% and 25% of their transitions into long stable phases, so we examine the ability to accurately predict whether the current phase is going to be short or long.

6.2.2 Phase Run Length Prediction

To provide phase length prediction we used the RLE-2 predictor described in the prior section, which is 32 entry 4-way associative, but instead of predicting a phase ID, we use it to predict the phase run length group. For this experiment we did not use any confidence counters, because the accuracy was already very high, but we did add a hysteresis counter to each table entry. This counter only assigns a new run length value to the entry if the length class has been seen twice in a row. This was done to filter "noise" in the phase lengths of some of the more complex programs (such as `gcc`). Some phases in these programs were occasionally longer or shorter than normal, and adding this counter ensures that any difference in phase length is representative of stable behavior.

Figure 6.2 also shows length prediction accuracy for all phase changes. The results show that overall the misprediction rate is very low. This isn't surprising with most of the programs having stable runs less than 150 million instructions. Even so, `gzip-graphic` and `perl-diffmail`, have a wide range of stable phase run length behavior, and we can achieve a misprediction rate close to 0% and 5.5% for these programs, respectively.

While most of the phase transitions go to short phases, tracking and predicting the length with the RLE-2 predictor provides additional benefits, because it is useful to know when we are transitioning into a long stable phase. These long stable phases are precisely the intervals targeted by expensive optimizations and reconfigurations.

7 Summary

In this paper we re-examine the run-time phase classification and prediction architecture from [25], and we provide four contributions over this prior approach.

We separated intervals of execution into phase transitions and frequently recurring phases. We used a counter to heuristically identify phase transitions and classify them into a special "transition phase," which lead to increased phase prediction accuracy and improved signature table utilization by reducing the number of unique phases detected.

We adaptively adjusted the phase classification similarity threshold to produce phases with more similar behavior. We found that some benchmarks have different sensitivity to the similarity threshold (maximum allowed difference between signature vectors) with respect to the homogeneity of the performance metrics of the intervals in a given phase. We presented an adaptive phase classification architecture that can dynamically tune the similarity threshold by monitoring the homogeneity in cycle count for each phase. Note that this approach still only uses the code signatures to perform phase classification to maintain a level of independence from the underlying hardware.

We re-examined next phase prediction, and the results showed that it is very difficult to predict the outcome of a phase change. To improve phase prediction, we focused on predicting only the outcome of phase changes. The results showed that more aggressive techniques like remembering the top 1, top 4 or last 4 phase changes are needed to significantly improve phase change prediction. In addition, we showed that confidence counters can be used in phase prediction to improve accuracy at the cost of coverage.

We also examined phase length prediction, where the length of the next phase is predicted. Predicting the exact length of the next phase is difficult, but we found that breaking the lengths into classes of 10-100M instructions, 100M-1B instructions, 1B-10B instructions, and more than 10B instructions produced more accurate predictions, although most phases were in the smallest phase length class (10-100M).

Overall, we found that the quality of phase classifications produced by our online classification algorithm were comparable to those produced by SimPoint, an offline phase classification algorithm. In terms of phase prediction, we found that programs spend 75% of their time in stable phases, which makes it difficult to show significant improvements over a simple predictor that always predicts that the program's behavior is stable. Finally, we show that more advanced techniques are needed to accurately predict phase changes.

Acknowledgments

We would like to thank Tim Sherwood and 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. CCF-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] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, December 2002.

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

[4] I.-C. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, October 1996.

[5] P.J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, March 1972.

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

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

[8] 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.

[9] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M.L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *11th International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

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

[11] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[12] M. Hind, V. Rjan, and P. Sweeney. Phase shift detection: A problem classification. Technical report, IBM, August 2003.

[13] 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.

[14] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Workshop on Workload Characterization*, September 2003.

[15] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *36th International Symposium on Microarchitecture*, December 2003.

[16] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th International Symposium on Microarchitecture*, December 1996.

[17] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[18] W. Kim, D. Shin, H.S. Yun, J. Kim, and S.L. Min. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

[19] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Processor power reduction via single-ISA heterogeneous multi-core architectures. *Computer Architecture Letters*, 2, April 2003.

[20] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[21] 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.

[22] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[23] 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.

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

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

[26] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2), April 2001.

[27] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.