Comparing Program Phase Detection Techniques

Ashutosh S. Dhodapkar and James E. Smith
Dept. of Electrical and Computer Engineering,
University of Wisconsin – Madison
{dhodapka, jes}@ece.wisc.edu

Abstract

Detecting program phase changes accurately is an important aspect of dynamically adaptable systems. Three dynamic program phase detection techniques are compared – using instruction working sets, basic block vectors (BBV), and conditional branch counts. Because program phases are difficult to define, we compare the techniques using a variety of metrics.

BBV techniques perform better than the other techniques providing higher sensitivity and more stable phases. However, the instruction working set technique yields 30% longer phases than the BBV method, although there is less stability within phases. On average, the methods agree on phase changes 85% of the time. Of the 15% of time they disagree, the BBV method is more efficient at detecting performance changes. The conditional branch counter technique provides good sensitivity, but is less effective at detecting major phase changes. Nevertheless, the branch counter technique correlates 83% of the time with the BBV based technique. As an auxiliary result, we show that techniques based on procedure granularities do not perform as well as those based on instruction or basic block granularities. This is mainly due to their inability to detect changes within procedures.

1. Introduction

General-purpose microprocessor design has traditionally focused on optimizing microarchitectural parameters (e.g. issue window size, cache sizes, etc.) at design time. The goal is to provide good performance on average, over a wide variety of workloads. This can, however, lead to sub-optimal performance and power dissipation for certain programs or specific execution phases of a program.

With the ever-increasing need for performance and growing importance of power efficiency, architects have proposed multi-configuration hardware that dynamically adapts to changing program requirements in order to achieve better power/performance characteristics [1]-[12]. Similarly, on the software side, dynamic code optimization [13][14] is gaining importance with the wide spread acceptance of run-time environments such as Java [15] and .NET [16].

In the presence of dynamically configurable hardware and software, the ability to initiate reconfiguration at the right time is essential. Because programs go through phases of execution wherein their performance is relatively stable [17][18], phase boundaries are a natural choice for performing reconfiguration (or at least determining if reconfiguration will be beneficial). Detecting phase changes accurately is thus an important aspect of dynamically adaptable systems. Furthermore, in systems where overheads associated with reconfiguration decisions are significant, program phase identification may enable reuse of configuration information for recurring phases, thereby improving performance [11][12][19].

There have been several proposals to implicitly [13] or explicitly detect program phase changes [2][10][11][12][19][20][21]. Recently, several researchers have proposed hardware techniques aimed specifically at detecting phase changes, identifying phases and predicting phases [2][10][11][12][19]. In this work, we focus on three of these proposed schemes – the first based on *working set signatures* [10][11], the second based on *basic block vectors* [19], and the third based on a *conditional branch counter* [2]. Because phases are not well-defined, a significant aspect of the paper is the definition of appropriate metrics for comparing these techniques.

The next section presents an overview of previously proposed techniques for detecting program phase changes. Section 3 discusses some of the issues related to the definition of program phases. It also describes various metrics used for comparing the different program phase change detection techniques (called *phase detection* techniques henceforth). Section 4 presents a comparison of the techniques with unbounded hardware resources. Section 5 compares practical hardware implementations based on working set signatures and accumulator tables. Section 6 concludes the paper.

2. Background

Balasubramonian et al. [2] use a *conditional branch counter* to detect program phase changes. The counter keeps track of the number of *dynamic* conditional branches executed over a fixed execution interval (measured in terms of the dynamic instruction count). Phase changes are detected when the difference in branch counts of consecutive intervals exceeds a threshold. Their scheme does not use a fixed threshold. Rather, the detection algorithm dynamically varies the threshold throughout the execution of the program.

In previous work [11], we defined a program phase to be the instruction working set of the program i.e. the set of instructions touched in a fixed interval of time. Program phase changes are detected by comparing consecutive instruction working sets using a similarity metric called the *relative working set distance*. Because complete working sets can be too large to efficiently represent and compare in hardware, we propose the use of lossy-compressed representations of working sets called *working set signatures* [10[11]. Signatures are compared using a metric called the *relative signature distance*. Phase changes are detected when the relative signature distance between consecutive intervals exceeds a preset (fixed) threshold. We show that signatures as small as 32 bytes in size can be used to resolve program phases in most benchmarks studied.

Sherwood et al. [19][20][21] propose the use of basic block vectors (BBVs) to detect program phase changes. BBVs keep track of execution frequencies of basic blocks touched in a particular execution interval. Phase changes are detected when the *Manhattan distance* between consecutive BBVs exceeds a preset threshold. Because entire BBVs cannot be stored in hardware, BBVs are approximated by hashing into an *accumulator table* containing a few large counters [19]. Their results indicate that as few as 32 24-bit counters are sufficient to represent BBVs.

Huang et al. [12] use subroutines to identify program phases. They propose the use of a hardware based call stack to identify program subroutines. The call stack tracks time spent in each subroutine, taking into consideration nesting of subroutines. If the time spent in a subroutine is greater than a preset threshold, it is identified as a major phase. The call stack used in their evaluation is 32 entries deep and each entry is 9 bytes wide.

HP Dynamo [13], a run-time dynamic optimization system, detects phase changes in order to flush stale translations from the cache. Dynamo stores optimized traces of the program, called fragments, in a fragment cache. In steady state most of the instructions are fetched from this fragment cache. A sharp increase in fragment formation rate indicates a phase change and is used to trigger flushing of stale fragments from the cache, making room for new ones. Unlike the previous schemes, Dynamo detects phase changes by observing an *implementation dependent* system characteristic. In general, such schemes can not be easily applied to configurable systems where the system characteristic being observed is a function of the configuration.

3. Comparison Metrics

The existence of program phases has intuitive appeal -- but in practice phases are not easily determined, or even easily defined. Program phase behavior is a result of control passing through procedures and nested loop structures Consequently, program phases have a fractal-like self-similar behavior i.e. high-level (long duration) phases are composed of several lower-level (shorter duration) phases, and in the limit each instruction is a separate phase. In essence, there are no absolute phases and phase behavior can only be observed with respect to a certain *granularity*. Hind et al. arrive at a similar conclusion based on formal analysis of phases and the phase detection problem [22]. Hence, phase detection methods do not literally detect phase changes; rather, they detect changes in program behavior that are assumed to result from phase changes. Nevertheless, we use the term “phase detection techniques” when describing them.

Phase detection techniques typically divide program execution into fixed-length sampling intervals (measured in terms of dynamic instructions executed). Program-related information is collected over the sampling interval and compared with similar information collected over the previous interval. If the two differ by more than a *difference threshold* (Δ_{th}), a phase change (or transition) is indicated. In general, the type of program information collected is *implementation independent*, i.e. it is a function of the dynamic instruction stream, not of performance.

A sequence of two or more intervals containing no indicated phase changes is defined as a *stable region*. A maximal length stable region is defined to be a *stable phase*. Sequences of one or more intervals that do not belong to stable phases are called *unstable regions*. In other words, all the individual intervals belonging to an unstable region are separated by phase changes.

Comparison of phase detection techniques is complicated because, as stated above, there are no absolute phases, and thus there is no *golden standard* that techniques can be compared against. Consequently, we compare phase detection techniques using a variety of metrics that have some practical appeal. Because the goal of this work is to compare hardware based phase detection techniques, we use metrics that are mostly relevant to dynamic optimization and tuning algorithms for adapting multi-configuration hardware.

3.1. Sensitivity and False Positives

Program phase detection techniques have been used in power/performance optimization algorithms [2][10][11] [12][19] and to reduce simulation time of benchmarks by identifying sections of code whose performance is representative of the entire benchmark [20][21]. Thus, one of the desirable properties in a phase detection mechanism is the ability to detect a phase change that results in a significant performance change.

We quantify this property with a metric called *sensitivity* [23]. Sensitivity is defined as the fraction of intervals with significant performance changes (with respect to the preceding interval), which were also indicated to be phase changes by the phase detection mechanism. It should be noted that "significant performance change" is a relative term. In this work, a performance change of 2% is considered significant unless stated otherwise. Consider an example program execution, which consists of 1000 intervals, of which 100 intervals show a significant performance change with respect to the preceding interval. If the phase detection mechanism indicates a phase change in 75 of these 100 intervals, the sensitivity is 75%. If it indicates a change for all 100 intervals, then the sensitivity is 100%. Note that if a phase change were to be indicated for each of the 1000 intervals, the sensitivity would still be 100% because all the significant performance changes were in fact detected.

The instance just given indicates that we must also consider the flip side to sensitivity: the fraction of *false positives* [23]. The fraction of false positives is the fraction of intervals where the performance shows no significant change but the phase detection technique indicates a phase change. Continuing with the previous example – of the 900 intervals where no significant performance change occurs, if the phase detection scheme indicates a change in 90 intervals, then the fraction of false positives is 10%. In the extreme case where all intervals are indicated as phase changes, there are 100% false positives.

High sensitivity is desirable in tuning algorithms because it exposes more tuning opportunities leading to better power/performance characteristics. On the other hand, a large fraction of false positives can cause unnecessary reconfigurations which can lead to significant performance loss and increase in power. Thus, a good phase detection technique should have high sensitivity and a small fraction of false positives.

3.2. Stability and Average Phase Length

Attempting to tune and reconfigure in unstable regions can lead to unpredictable, non-optimal results. Consequently, algorithms such as the ones proposed in [10][11] do not perform tuning while in unstable regions. Tuning algorithms can thus benefit if a large part of program execution is spent in stable phases. We quantify this with a metric called *stability*, which is defined as the fraction of intervals that belong to stable phases.

Most tuning algorithms use trial-and-error mechanisms to arrive at the optimal configuration. That is, they simply try a series of different configurations and determine the best one. These algorithms require several intervals at the beginning of a stable phase to complete the tuning process (the algorithm presented in [12] is an exception). If phases are short (i.e. small number of intervals), tuning never completes. Also, short phases make it difficult to amortize reconfiguration overheads associated with the tuning process. Thus, the *average phase length* is an important metric – defined as the number of intervals that are part of stable phases, divided by the total number of stable phases.

It should be noted that two programs with the same stability can have different average phase lengths. For example, if a program runs for 1000 intervals divided into two length 500 stable phases, the stability is 100% and the average phase length is 500 intervals. However, if the phase changes at the end of every other interval, the stability is still 100% but the average phase length is two. These metrics should therefore be used in conjunction with each other.

3.3. Performance Variance

Because most tuning algorithms are based on the assumption that performance is uniform within a phase, the *performance variance* within a phase can be used as a metric. A good phase detection method should be able to resolve phases with a relatively small variance in performance, compared to the variance across the whole program. A small variance is an indicator that the phase detection mechanism is detecting phase boundaries correctly.

3.4. Correlation

Correlation between phase detection techniques can be useful for comparing their relative ability to detect phase changes. We define correlation between two phase detection techniques as the fraction of intervals for which they agree on the presence or absence of a phase change. If the techniques are highly correlated, then the technique with the simplest implementation is preferable. In the absence of high correlation, the choice of techniques must be based on one or more of the metrics defined above and other advantages associated with the technique and where it is being applied.

4. Performance with Unbounded Resources

Before comparing hardware based phase detection techniques, we evaluate their limits by comparing techniques based on unbounded working sets, BBVs, and

conditional branch counters. In addition to instruction working set based techniques [10][11], we evaluate branch and procedure working set based techniques.

We equalize the granularity of these techniques by choosing a common sampling interval of 10 million instructions. In the course of this research, we tried other sampling intervals, and did not find any qualitative difference in our conclusions.

4.1. Basic Definitions

The instruction working set is defined as the set of instructions touched over the sampling interval. Similarly, branch/procedure working sets are defined as the set of branches/procedures touched over the sampling interval. In previous work [10][11], we defined a similarity metric called the relative working set distance, to compare working sets. The relative working set distance between intervals *i* and *i-1* is defined as

$$\Delta_{i,i-1} = \frac{\left\|W_i \cup W_{i-1}\right\| - \left\|W_i \cap W_{i-1}\right\|}{\left\|W \cup W_{i-1}\right\|}$$

where W_i and W_{i-1} are working sets collected over intervals *i* and *i-1*. The *Norm* of the set is the number of elements in the set i.e. the cardinality of the set. Since the relative working set distance is a normalized metric, the maximum possible working set difference is 100%.

Sherwood et al. [20] define a BBV to be a set of counters, each of which counts the number of times a static basic block is entered in a given execution interval. In later work [19], they approximate the BBV with an array of counters, where each counter tracks the number of instructions executed by a basic block in a given execution interval. In this study, we use the latter definition for a BBV as it relates more closely to the hardware implementation. The BBV difference between intervals *i* and *i-1* is given by the Manhattan distance

$$\Delta_{i,i-1} = \sum_{j=0}^{\infty} \left| count_{i,j} - count_{i-1,j} \right|$$

where each distinct value of *j* represents a unique basic block.

Phase changes are defined with respect to a difference threshold (Δ_{th}) i.e. a phase change is indicated when $\Delta_{i,i-1} > \Delta_{th}$. In order to compare the techniques, we normalize the BBV and branch count differences to 100%. This is done by dividing the differences by the maximum possible difference, which is *2N* for BBVs and *N* for branch counts, where *N* is the number of instructions in the sampling interval.

4.2. Comparison

We evaluate phase detection techniques, using a modified version of *sim-outorder*, an out-of-order simulator provided as part of the SimpleScalar toolset [24]. The microarchitecture parameters used for performance measurements are shown in Table 1. The results presented are averaged over all SPEC 2000 benchmarks [25] with the exception of *sixtrack* and *facerec*. The latter two could not be run due to shortcomings of the simulation environment. Reference inputs have been used for each benchmark and due to time and resource constraints, each benchmark was run to completion or 15 billion instructions. As mentioned before, a sampling interval of 10 million instructions was used.

Table 1. Microarchitecture Parameters

Processor core	4-wide fetch/decode/issue/commit; 64-entry RUU, 32-entry LSQ; 4 integer ALUs, 1 integer multiplier; 4 FP ALUs, 1 FP multiplier;
Branch Prediction	4K entry *gshare*, 10-bit global history; 2K entry, 2-way BTB; 32 entry RAS
Memory subsystem	I and D-cache: 32KB, 2-way, 64 byte line, latency 1 cycle; unified L2-cache: 512KB 4-way, 128 byte line, latency 6 cycles; memory: width 16-byte, latency 100 cycles

4.2.1. Sensitivity and False Positives

Sensitivity and false positives are typically at odds with each other and are a strong function of the difference threshold. We use *Receiver Operating Characteristic* (ROC) analysis to arrive at difference thresholds for comparing the different techniques. ROC analysis is a widely used technique for analyzing medical tests, which have similar sensitivity and false positive tradeoffs [26]. The ROC curve is a plot of the sensitivity versus false positives for various difference thresholds. In general, the best technique is the one which achieves maximum sensitivity for a given number of false positives.

Fig. 1 shows ROC curves for the different phase detection techniques. These curves are based on the assumption that a significant CPI (cycles per instruction) change is one of more than 2%, i.e. the sensitivity is computed as the fraction of intervals where a CPI change of *more than* 2% is indicated as a phase change. False positives are computed as the fraction of intervals where the CPI changes by *less than* 2%, but a phase change is indicated.

Both sensitivity and false positives increase with decreasing difference thresholds because as the threshold is reduced, even minor fluctuations in CPI are noticed. In order to compare the different techniques, we choose difference thresholds corresponding to the knees of the curves. The reasoning is that beyond the knee, a small increase in sensitivity comes at the expense of a large number of false positives. Moreover, we also try to equal-

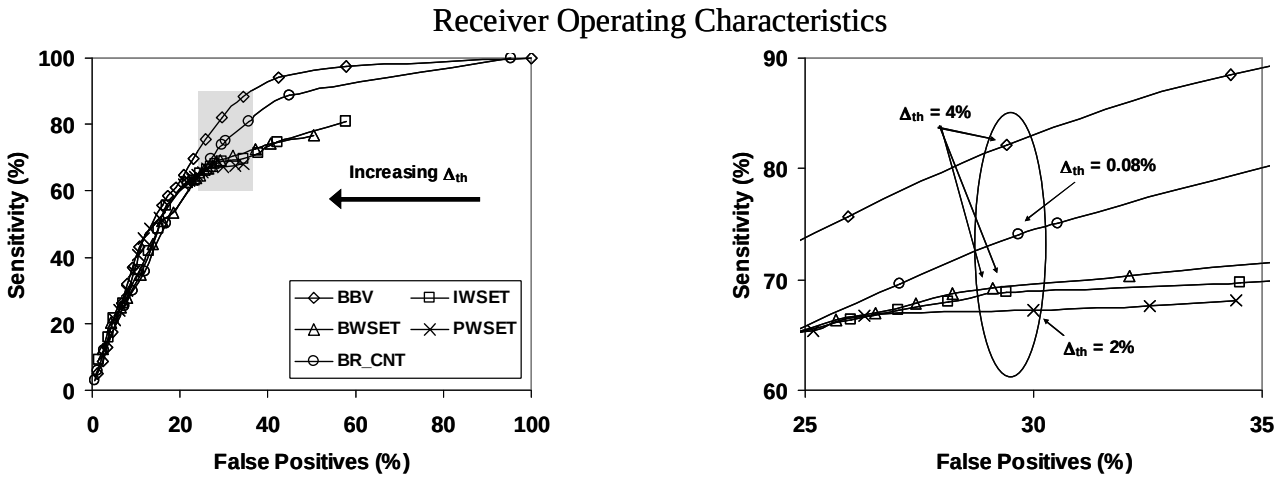

Figure 1. ROC curves for the various phase detection techniques are shown. CPI changes of more than 2% are considered significant. Difference thresholds (Δ_{th}) increase from right to left. IWSET, BWSET and PWSET represent instruction, branch and procedure working set techniques respectively. BR_CNT represents the conditional branch counter technique. The figure on the right shows a magnified view of the shaded part of the left figure. The circled points are chosen for comparison.

ize the false positives to make clear comparisons among methods. We arrive at difference thresholds of 4% for the BBV, instruction, and branch working set techniques; 2% for the procedure working set technique and 0.08% for the conditional branch counter based technique. This choice of thresholds leads to about 30% false positives for each technique.

It is evident that BBVs perform the best – with a sensitivity of 82%, followed by conditional branch counter (74%) and working set techniques (70%). The working set techniques do not perform well because they do not keep track of instruction (or branch/procedure) execution frequencies. Consequently, the maximum sensitivity achievable ($\Delta th = 0$) by the instruction working set technique is limited to 81%.

Amongst the working set methods, the procedure based method shows slightly lower sensitivity than the other two. This is expected because it fails to detect phase changes within procedures. Results show that the procedure based working set method achieves a maximum sensitivity of only 68% compared to 81% achieved by the instruction working set method. This is a fundamental problem with procedure based phase detection methods.

Redefining a "significant CPI change" leads to qualitatively similar results for BBV and working set based techniques. Fig. 2 shows ROC curves assuming that a CPI change of 10% (rather than 2%) is significant. The curves are similar to those in Fig. 1 except that each technique achieves higher sensitivity for a given number of false positives and the difference in sensitivity between the BBV and working set methods decreases. This is to be expected because a 10% change in CPI is more easily detected compared to a 2% change.

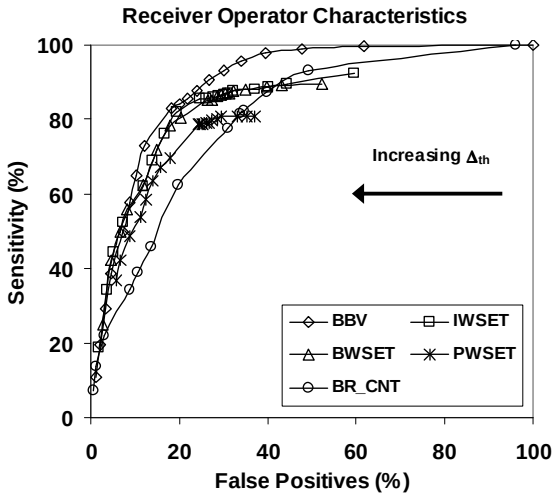

Figure 2. ROC curves for the various phase detection techniques are shown. CPI changes of more than 10% are considered significant. (See Fig. 1 caption for more details)

A more interesting result is that in this case the working set based techniques work better than the conditional branch counter based technique, if the fraction of false positives is limited to less than 40%. This means that the working set based techniques are more efficient at detecting *major* phase changes, a result in agreement with our previous results [11].

4.2.2. Stability and Average Phase Length

Fig. 3 shows, for BBV and working set methods, how stability and average phase length vary with respect to the difference threshold. Fig. 4 shows the same for the conditional branch counter method. Clearly, the stability of each method increases with the difference threshold because fewer changes are detected due to reduced sensitivity (see Fig. 1). For the difference thresholds chosen for comparison in the previous section (circled in the figure) the working set based methods achieve slightly greater stability (64%) compared to the BBV (62%) and conditional branch counter (63%) based schemes.

The average phase length roughly increases with the difference threshold because small perturbations in program behavior are not indicated as phase changes. For the chosen difference thresholds, instruction and branch working set techniques lead to 30% longer phases on average compared to BBVs and 38% longer phases on average compared to the conditional branch counter technique. Given that the stability shown by each of these techniques is similar, using BBVs or branch counts leads to a larger number of *shorter* phases. This may not be desirable for tuning algorithms with large performance overheads associated with reconfiguration.

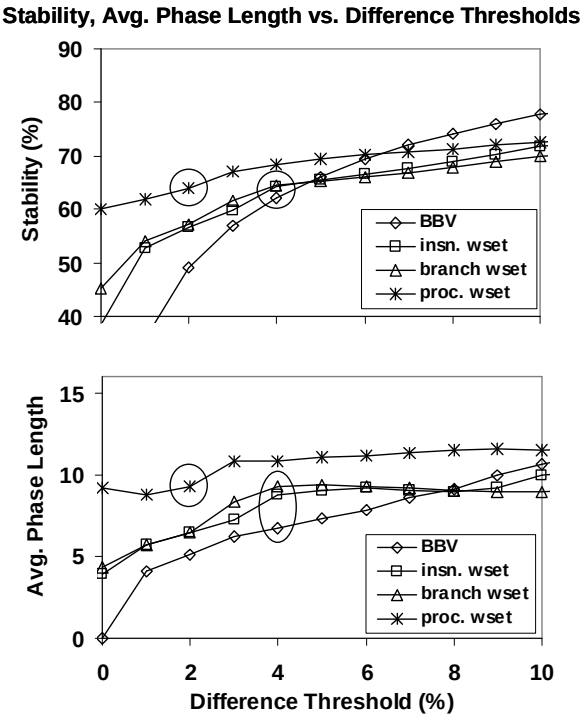

Figure 3. Variation of stability and average phase length with respect to the difference threshold. The phase length is shown in terms of number of intervals. The circled points correspond to the difference thresholds used for comparison.

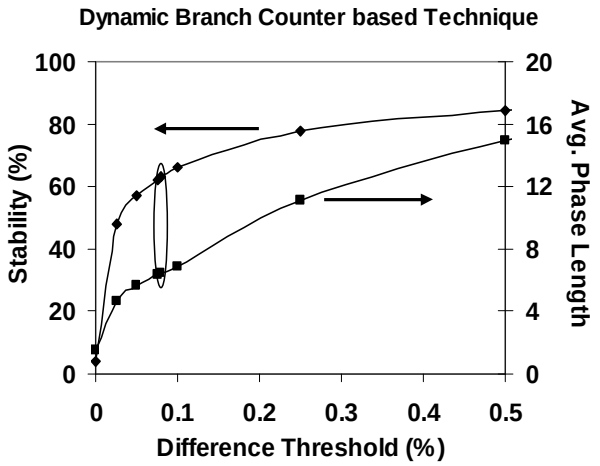

Figure 4. Stability (primary axis) and average phase length (secondary axis) with respect to difference threshold, for the conditional branch counter technique. The circled points correspond to difference thresholds used for comparison.

4.2.3. Performance Variance

Although large average phase lengths are desirable, the performance stability within a phase is also important. Fig. 5 shows the percent coefficient of variance (standard deviation/average) in CPI within stable phases, averaged over all benchmarks. The difference thresholds used were the ones arrived at in Sec. 4.2.1. Each of the techniques achieved less than 2% variance in CPI within stable phases as compared to a 116% CPI variance across all intervals (i.e. including unstable regions).

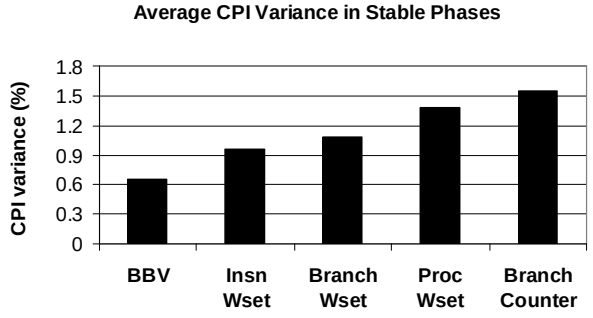

Figure 5. Average CPI variance within stable phases. Difference thresholds used were 4% for BBV, instruction and branch working sets, 2% for procedure working set, and 0.08% for the conditional branch counter technique.

Using BBVs leads to least performance variance (0.65%) within phases. Instruction and branch working set techniques achieve just less than 1% variance. Using procedure working sets leads to 1.4% variance which

further establishes that procedure based techniques do not work as well as instruction and branch based techniques. Interestingly, the conditional branch counter technique performs the worst with a CPI variance of 1.6%, although its sensitivity is higher than any of the working set techniques. This means that the technique detects more relatively small phase changes and misses some of the larger phase changes compared to the working set techniques.

4.3. Correlation

Because BBVs perform better than the other techniques on most metrics, we compute correlation between the BBV technique and each of the other techniques. The difference thresholds used were the ones arrived at in Sec. 4.2.1. The instruction and branch working set schemes show 85% correlation, while the procedure working set and conditional branch counter based schemes show 80% and 83% correlation respectively. Since each of the techniques agrees with the BBV technique more than 80% of the time, an important question is – do they agree on most of the *major* phase changes?

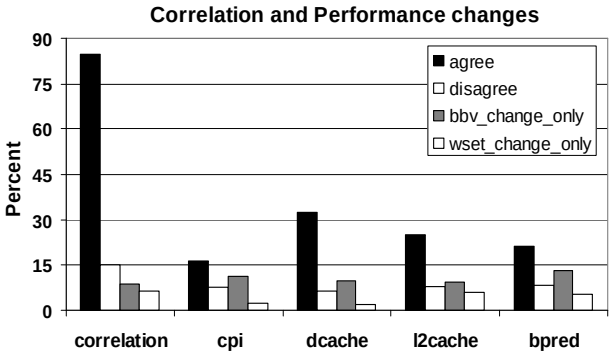

Figure 6. A comparison of BBV and instruction working set schemes. The first set of four bars (correlation) shows the fraction of time that the two techniques agree, disagree, only BBV detects a change and only working set detects a change. The remaining four sets show the relative change in CPI; data and L2 cache miss rates and branch misprediction rates for each of the four types of events described above.

To answer this question, we compare the BBV and instruction working set techniques in more detail (Fig. 6). The first set of four bars in Fig. 6 show respectively, the percent of time the two techniques agree on the presence or absence of a phase change (*agree*), the percent of time they disagree (*disagree*), the percent of time the BBV technique indicates a change but the instruction working set technique does not (*bbv_change_only*) and the percent of time the instruction working set technique indicatess a change but the BBV technique does not (*wset_change_only*). The next four sets of bars show the average relative change in performance metrics (CPI; data and L2 cache miss rates; branch misprediction rate) for each of these four events (*agree*, *disagree*, *bbv_change_only*, *west_change_only*). We do not include instruction cache miss rates because being close to zero they cause very large relative changes that cannot be well represented on this graph. Note that for the performance metrics, the *agree* case consists *only* of events where both techniques indicate a phase change, i.e. it does not contain cases where both agree that there is no phase change.

As seen from the figure, the BBV and the instruction working set technique agree about 85% of the time. Of the 15% of the time they disagree, they are split roughly equally. Focusing on the first two bars in each set of performance metrics, it is evident that the relative performance change seen when both the methods agree on a change is much higher than the relative change seen when they disagree. This means that *most of the major phase changes are detected by both methods*.

In cases where the two techniques disagree, the relative change in performance seen when only the BBV indicates a phase change (third bar in each set) is higher than the relative change seen when only the working set method indicates a change (fourth bar in each set). This means that the BBV based technique is better at detecting changes in performance. This is expected because the BBV inherently contains much more information compared to the instruction working set. However, it should be noted that this happens less than 9% of the time on average.

5. Hardware Implementations

The previous section compared the performance of BBV, working set and conditional branch count techniques using unbounded hardware resources. In practice, BBVs and working sets are too large to be efficiently stored and compared in hardware. In previous work [10][11], we proposed a hardware structure called the *working set signature*, which is a compact representation of the working set. Similarly, Sherwood et al. [19] proposed an array of accumulators (counters) called the *accumulator table* to represent BBVs. In this section, we compare the performance of these hardware implementations. The hardware implementation for a conditional branch counter is equivalent to its unbounded implementation for the interval sizes studied, and thus is not discussed.

5.1. Hardware Structures

5.1.1. Working Set Signature

A working set signature is a lossy-compressed representation of the complete working set [10][11]. The signature is formed by sampling the working set i.e. program

counters (PCs) over a fixed interval of instructions (e.g. 10 million) and hashing the samples into an *n*-bit vector using a random hash function (Fig. 7). The signature is reset at the beginning of each interval to remove stale working set information.

Phase changes are detected by comparing consecutive signatures using the relative signature distance defined as

$$\Delta = \frac{\|S_1 \oplus S_2\|}{\|S_1 + S_2\|},$$

i.e. (ones count of exclusive OR of signatures)/(ones count of inclusive OR of signatures). If the relative signature distance is greater than a preset threshold, a phase change is indicated. We propose the use of virtual machine software to compute the relative signature distance [10][11]. However, it can also be done in hardware.

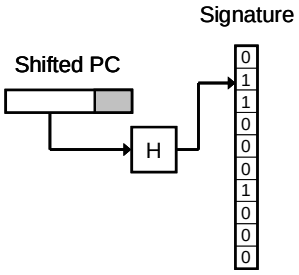

Figure 7. Working set signature mechanism. PCs, shifted by a few bits, are hashed into the signature bit vector and the corresponding entries are set. The hash function used is pseudo-random.

The signature fill-factor depends on the signature size as well as the working set size. Large working sets can saturate small working set signatures. In order to reduce the pressure on instruction working set signatures, the PCs are shifted by a few bits thereby reducing the number of unique elements that are hashed. In this work, we shift PCs by five bits, which is equivalent to sampling blocks of eight instructions. Shifting may not be necessary for branch and procedure working set signatures because only 20 - 25% of committed instructions are branches and a mere 1 - 2% are procedure entry points.

5.1.2. Accumulator Table

The accumulator table (Fig. 8) is an array of counters indexed by hashing branch PCs. Whenever a branch PC is encountered, the corresponding counter is incremented by the number of instructions committed since the last branch. The accumulator table collects samples over a fixed interval of instructions and is reset at the beginning of each interval.

To prevent overflow, each accumulator is made large enough to be able to count up to the number of instructions in the interval. For example, if the interval is 10 million instructions, then each accumulator is 24-bits wide. Phase changes are detected by comparing consecutive arrays using a Manhattan distance metric (see Sec. 4.1). If the distance is greater than a preset threshold, a phase change is indicated.

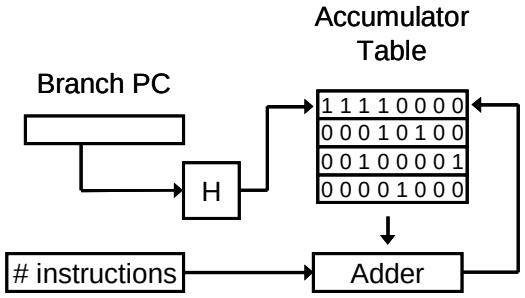

Figure 8. Accumulator table update mechanism. Branch PC is hashed into the table and the corresponding counter is incremented with the number of instructions committed since the last branch. Hash function used is pseudo-random.

5.2. Design Space

In this work, we study three different sizes for each of the hardware based phase detection techniques. The sizes for instruction/branch signatures (4096, 1024, 512 bits) and accumulator tables (1024, 128, 32 entries) are similar to those used in previous work [10][11][19]. Procedure signature sizes were chosen to be 1024, 256 and 64 bits because procedure working sets are much smaller compared to corresponding instruction/branch working sets.

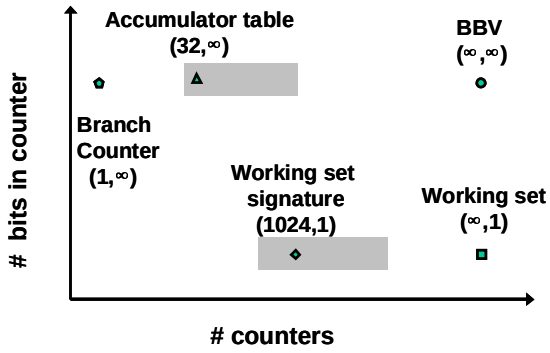

Figure 9. The position of each of the techniques in the design space is shown. The X-axis represents the number of counters used to capture information. The Y-axis shows the number of bits used in each counter.

These hardware techniques, along with the unbounded cases considered in the previous section, span a wide design space as shown in Fig. 9. Each technique can be categorized in terms of the number of counters and the number of bits in each counter. The unbounded BBV

contains maximum information with unbounded counters each with an unbounded number of bits, the accumulator tables have a small number of unbounded counters and finally the conditional branch counter based scheme has one unbounded counter, although it counts only conditional branches. The working set based techniques form a similar spectrum albeit with a larger number of single-bit counters. It should be noted that accumulator table counters have a bounded number of bits, but they are considered unbounded because they are large enough to prevent overflows for a given sampling interval.

5.3. Comparison

Fig. 10 shows the correlation of each of the hardware based techniques with the corresponding unbounded case i.e. instruction working set signatures are compared to complete instruction working sets, accumulator tables to BBVs, etc. The difference threshold used is 4% for the instruction and branch working set signatures, 2% for the procedure working set signature and 4% for the accumulator table. (see Sec. 4.2.1.)

It is evident that the hardware schemes are highly correlated with their corresponding unbounded schemes. As the number of bits/entries is reduced, the correlations drop off mainly due to increased aliasing. However, the smallest size hardware structure still correlates more than 90% of the time (in each case) with the unbounded scheme.

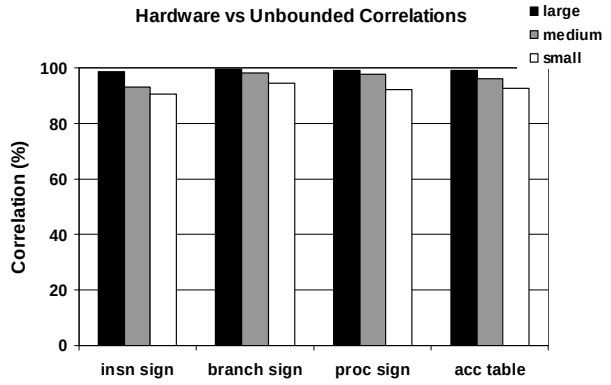

Figure 10. The correlation of each hardware scheme with the corresponding unbounded scheme is shown. The figure shows correlations for three different sizes of instruction and branch working set signatures (4096, 1024, 512), procedure working set signatures (1024, 256, 64) and accumulator tables (1024, 128, 32).

It is worth comparing the accumulator table with an equivalent instruction working set signature. We consider the smallest accumulator table i.e. 32 entries, since it shows reasonably high correlation with BBVs. To prevent overflow for a sampling interval of 10 million instructions, each of the accumulators should be at least 24-bits wide. This translates to a total of 32*24 = 768 bits. This is comparable in area to a 1024-bit signature taking into account the extra sense amplifiers and fast adder(s) required by the accumulator table. Correlation between the 32-entry accumulator table and 1024-bit instruction working set signature showed results similar to those given in Fig. 6 and thus are not repeated. The two hardware techniques correlate 87% of the time and the change in performance metrics when the techniques agree is much higher than the change when they disagree. This means that both techniques agree on major phase changes. This is not surprising given the fact that both these techniques are highly correlated to their unbounded cases and a similar observation was made there.

5.3.1. Conditional Branch Counter

Sherwood et al. [19] evaluated the performance of accumulator tables by using a metric called the *visible phase difference*. The visible phase difference is the ratio of the phase difference (Manhattan distance) observed using the accumulator table, to the phase difference observed using unbounded BBVs. The visible phase difference of the unbounded BBV is 100%.

The accumulator table size for their algorithms was chosen to be 32 entries, because the visible phase difference achieved by using 32 entries is 72%. However, this is not necessarily a good metric to use because phase changes are detected based on a difference threshold and as long as the phase difference is above threshold, it does not matter what the visible phase difference is. As an example, if the difference threshold is 10% and the unbounded BBV shows a phase difference of 90%, it does not matter if the phase difference achieved by the accumulator table is 80% or 25% because a phase change is detected in both cases. This explains why the 32-counter method agrees with the unbounded BBV 93% of the time even when results from [19] show that it achieves a visible phase difference of only 70%.

This means that perhaps an even smaller number of counters can provide reasonable phase detection ability. In fact, the conditional branch counter [2], which is an extreme example with a single counter, works quite well correlating 83% of the time with the unbounded BBV scheme.

5.4. Other Considerations

Because the hardware schemes discussed in the previous section correlate (agree) most of the time, the decision to use a particular technique may be based on other considerations such as hardware complexity and additional attributes that may be useful for tuning algorithms.

5.4.1. Hardware Complexity

The hardware used in the conditional branch counter scheme is clearly the simplest and warrants no further discussion. The working set signature requires a 1-bit wide RAM array with one read/write port. The instruction sampling hardware samples each instruction (at most 4 in a 4-wide superscalar) and hashes it to get the signature bit to be set. One possible optimization is to sample only one instruction every two to four cycles. We have seen that this periodic sampling technique works reasonably well because the signature only tracks the number of static instructions touched and not the number of *times* they were touched. This can simplify the hardware significantly and make it amenable to a slow-transistor implementation, thereby saving power.

The accumulator table uses a 24-bit wide RAM array, with one read and one write port. Separate read and write ports may be needed for throughput reasons. The sampling hardware is more complex than that used in the working set signatures as it has to analyze the retire stream to detect positions of branches and increment counters appropriately. Moreover, since the Manhattan distance is based on instruction counts, dropping samples may not advisable, thus making fast hardware essential. Additionally, the accumulator table also requires a fast 24-bit adder to update the accumulators.

It is clear that the accumulator table is more complex and less power efficient compared with the signature method. However, it should also be noted that neither of these schemes would form an appreciable fraction of hardware in a modern microprocessor, and thus their small contribution to power dissipation/ complexity may not be a concern.

5.4.2. Recurring Phase Identification

The ability to identify recurring phases is a desirable attribute in phase detection techniques. This property can be used in tuning algorithms to reuse previously found optimal configurations for recurring phases [10][11][12][19]. This eliminates a significant fraction of reconfigurations, leading to performance improvements. In our previous work [11], we show that phase-identification based algorithms can reduce reconfigurations by as much 92% over a subset of SPEC 2000 integer benchmarks.

Working set signatures and BBVs have been shown to identify recurring program phases [10][11][19]. Whether conditional branch counters can be used to identify recurring phases remains to be shown.

5.4.3. Estimating Working Set Size

Working set signatures have an added advantage that they can be used to estimate the working set size directly [11]. The working set size k can be estimated from the fill factor f (number of ones) of the signature using the relation

$$k = \frac{\log(1-f)}{\log(1-\frac{1}{n})},$$

where, n is the signature size. In cases where performance of a unit is directly related to the working set size (e.g. instruction and data caches) signatures can be used to determine the optimal configuration without going through a tuning process. This has been shown to reduce the number of reconfigurations by 74% in a particular instruction cache tuning algorithm [11].

However, it should be noted that to make use of this property, the signature should capture the *same* working set that the unit performance is dependent on. For example, instruction working set signatures can be used to configure instruction caches but not data caches.

6. Conclusions

The BBV based technique provides better sensitivity and lower performance variation in phases compared to the other techniques. The instruction and branch working set techniques have similar performance on each of the metrics described. These techniques are less sensitive than the BBV technique mainly because working sets contain less information compared to BBVs. However, the instruction working set technique provides slightly higher stability and achieves 30% longer phases on average compared to the BBV technique. This can benefit trial and error based tuning algorithms. On average, the BBV and instruction working set schemes agree on phase changes 85% of the time. Of the 15% time they disagree, the BBV is more efficient at detecting important performance changes. As an auxiliary result, we show that procedure working set based techniques do not perform quite as well as the other working set based methods. This is mainly due to their inability to detect phase changes within procedures.

One of the surprising results of this study is that a simple conditional branch counter scheme performs quite well and agrees with the unbounded BBV scheme 83% of the time. However, it does lead to shorter average phase lengths and higher performance variance within phases compared with the BBV and working set schemes. This indicates that the branch counter based technique fails to detect some of the major phase changes.

Finally, we find that the hardware schemes i.e. working set signatures and the accumulator table approximate their corresponding unbounded cases (working sets and BBV) very closely, correlating more than 90% of the time even for the smallest structures considered. Also, equivalent sized instruction working set signatures and accumulator tables agree on phase changes 87% of the time.

Given the high correlation between these techniques, the choice of technique may be guided by other considerations. While the conditional branch counter is the simplest to implement, signatures and accumulator tables can be used to identify recurring phases – leading to more efficient tuning algorithms. Signatures also provide the added advantage that they can be used to estimate certain working set sizes and immediately configure the corresponding microarchitectural units such as caches.

Finally, in this work, we dealt with a very large design space composed of several variables including sampling intervals, difference thresholds, and hardware sizes. Admittedly, the results therefore represent a very small slice of the design space. On the other hand, in the process of conducting this research we did simulate a large number of variations and found the results to be qualitatively similar to those reported here.

7. Acknowledgements

This work is being supported by an NSF grant CCR-0311361, SRC grant 2000-HJ-782, Intel and IBM.

8. References

[1] D. H. Albonesi, "Dynamic IPC/clock rate optimization," *in Proc. of the 25th Annual Intl. Sym. on Computer Architecture*, Jun. 1998, pp. 282-292.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general purpose architectures," *in Proc. of the* 33rd *Annual Intl. Sym. on Microarchitecture*, Dec. 2000, pp. 245-257.

[3] S. H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep submicron high-performance I-caches," *in Proc. of the 7th Intl. Sym. on High Performance Computer Architecture*, Jan. 2001, pp. 147-157.

[4] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," *in Proc. of the 27th Annual Intl. Sym. on Computer Architecture*, Jun. 2000, pp. 214-224.

[5] T. Juan, S. Sanjeevan, and J. Navarro, "Dynamic history-length fitting: a third level of adaptivity for branch prediction," *in Proc. of the 25th Annual Intl. Sym. on Computer Architecture*, Jun. 1998, pp. 155-166.

[6] D. Folegnani, and A. González, "Energy-effective issue logic," *in Proc. of the 28th Annual Intl. Sym. on Computer Architecture*, Jun. 2001, pp. 230-239.

[7] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose, "Energy efficient co-adaptive instruction fetch and issue," *in Proc. of the 30th Annual Intl. Sym. on Computer Architecture*, Jun. 2003, pp. 147-156.

[8] R. Bahar, and S. Manne, "Power and energy reduction via pipeline balancing," *in Proc. of the 28th Annual Intl. Sym. on Computer Architecture*, Jul. 2001, pp. 218-229.

[9] M. Huang, J. Reneau, S.-M. Yoo, and J. Torrellas, "A framework for dynamic energy efficiency and temperature management," *in Proc. of the 33rd Annual Intl. Sym. on Microarchitecture*, Dec. 2000, pp. 202-213.

[10] J. E. Smith, and A. S. Dhodapkar, "Dynamic microarchitecture adaptation via co-designed virtual machines," *in 2002 Intl. Solid State Circuits Conference, Digest of Technical Papers*, pp. 198-199, Feb. 2002.

[11] A. S. Dhodapkar, and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," *in Proc. of the 29th Annual Intl. Sym. on Computer Architecture*, May 2002, pp. 233-244.

[12] M. Huang, J. Renau, and J. Torrellas, "Positional adaptation of processors: application to energy reduction," *in Proc. of the 30th Annual Intl. Sym. on Computer Architecture*, Jun. 2003, pp. 157-168.

[13] V. Bala, E. Duesterwald, S. Banerjia, "Dynamo: A transparent dynamic optimization system," *in Proc. of the Conf. on Programming Language Design and Implementation*, ACM SIGPLAN, 2000, pp. 1-12.

[14] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, "Adaptive Optimization in the Jalapeno JVM," *in Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[15] *http://java.sun.com*

[16] *http://www.microsoft.com/net*

[17] Timothy Sherwood, and Brad Calder, "Time varying behavior of programs," *UC San Diego Technical Report UCSD-CS99-630*, Aug. 1999.

[18] J. Henning, "SPEC CPU2000 memory footprint," online at *http://www.spec.org/ cpu2000/analysis/memory*

[19] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *in Proc. of the 30th Annual Intl. Sym. on Computer Architecture*, Jun. 2003, pp. 336-347.

[20] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation," *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2001, pp. 3-14.

[21] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically characterizing large scale program behavior," *Proc. of 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 45-57.

[22] M. J. Hind, V. T. Rajan, and P. F. Sweeney, "Phase shift detection: a problem classification," *IBM Research Report RC-22887*, Aug. 2003.

[23] D. Grunwald, A. Klauser, S. Manne and A. Pleszkun, "Confidence estimation for speculation control," *Proc. of the 25th Intl. Sym. on Computer Architecture*, July 1998.

[24] D. Burger and T. Austin, "The SimpleScalar tool set version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[25] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28-35, Jul. 2000.

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