# Dynamic Code Region (DCR) based Program Phase Tracking and Prediction for Dynamic Optimizations

Jinpyo Kim[1], Sreekumar V. Kodakara[2] Wei-Chung Hsu[1], David J. Lilja[2], and Pen-Chung Yew[1]

[1] Department of Computer Science and Engineering, {jinpyo,hsu,yew}@cs.umn.edu,
[2] Department of Electrical and Computer Engineering, {sreek,lilja}@ece.umn.edu,
University of Minnesota-Twin Cities, Minneapolis, MN 55455, USA

**Abstract.** Detecting and predicting a program's execution phases are crucial to dynamic optimizations and dynamically adaptable systems. This paper shows that a phase can be associated with dynamic code regions embedded in loops and procedures which are primary targets of compiler optimizations. This paper proposes a new phase tracking hardware, especially for dynamic optimizations, that effectively identifies and accurately predicts program phases by exploiting program control flow information. Our proposed phase tracking hardware uses a simple stack and a phase signature table to track the change of phase signature between dynamic code regions.

Several design parameters of our proposed phase tracking hardware are evaluated on 10 SPEC CPU2000 benchmarks. Our proposed phase tracking hardware effectively identifies a phase at a given granularity. It correctly predicts the next program phase for 84.9% of times with a comparable small performance variance within the same phase. A longer phase length and higher phase prediction accuracy together with a reasonably small performance variance are essential to build more efficient dynamic profiling and optimization systems.

## 1 Introduction

Understanding and predicting a program's execution phase is crucial to dynamic optimizations and dynamically adaptable systems. Accurate classification of program behavior creates many optimization opportunities for adaptive reconfigurable microarchitectures, dynamic optimization systems, efficient power management, and accelerated architecture simulation [1–4, 7, 9, 14, 22, 23].

Dynamically adaptable systems [1, 3, 10, 26] have exploited phase behavior of programs in order to adaptively reconfigure microarchitecture such as the cache size. A dynamic optimization system optimizes the program binary at runtime using code transformations to increase program execution efficiency. Dynamic binary translation also falls in this category. In such systems, program phase behavior has been exploited for dynamic profiling and code cache management

[9, 14–16, 19, 20]. For example, the performance of code cache management relies on how well the change in instruction working set is tracked by the system.

Dynamic optimization systems continuously track the program phase change either by sampling the performance counters or by instrumenting the code. The sampling overhead usually dominates in the sampling based profiling. While using a low sampling rate can avoid high profiling overhead, it may result in an unstable system where reproducibility is compromised and also may miss optimization opportunities. Using program phase detection and prediction may more efficiently control the profiling overhead by adjusting sampling rate adaptively or by applying burst instrumentation. For example, if the program execution is in a stable phase, profiling overhead can be minimized. (e.g., by lowering the sampling rate), while a new phase would require burst profiling.

A phase detection technique developed for dynamically adaptation systems is less applicable for a dynamic optimization system. This is because collecting performance characteristics at regular time intervals with arbitrary boundaries in an instruction stream is not as useful as gathering performance profiles of instruction streams that are aligned with program control structures. We introduce the concept of Dynamic Code Region (DCR), and use it to model program execution phase. A DCR is a node and all its child nodes in the extended calling context tree (ECCT) of the program. ECCT as an extension of the calling context tree (CCT) proposed by Ammon et al [25] with the addition of loop nodes. A DCR corresponds to a sub tree in ECCT. DCR and ECCT will be explained in later sections.

In previous work [1–4, 6], a phase is defined as a set of intervals within a program's execution that have similar behavior and performance characteristics, regardless of their temporal adjacency. The execution of a program was divided into equally sized non-overlapping intervals. An interval is a contiguous portion (i.e., a time slice) of the execution of a program. In this paper, we introduce dynamic intervals that are variable-length continuous intervals aligned with dynamic code regions and exhibit distinct program phase behavior.

In traditional compiler analysis, interval analysis is used to identify regions in the control flow graph [21]. We define dynamic interval as instruction stream aligned with code regions discovered during runtime. Dynamic intervals as a program phase can be easily identified by tracking dynamic code regions. We track higher-level control structures such as loops and the procedure calls during the execution of the program. By tracking higher-level code structures, we were able to effectively detect the change in dynamic code region, and hence, the phase changes in a program execution. This is because, intuitively, programs exhibit different phase behaviors as the result of control transfer through procedures, nested loop structures and recursive functions. In [12], it was reported that tracking loops and procedures yields comparable phase tracking accuracy to the Basic Block Vector (BBV) method [3, 6], which supports our observation.

In this paper, we propose a dynamic code region (DCR) based phase tracking hardware for dynamic optimization systems. We track the code signature of procedure calls and loops using a special hardware stack, and compare against

previously seen code signatures to identify dynamic code regions. We show that the detected dynamic code regions correlate well with the observed phases in the program execution.

The primary contributions of our paper are:

- We showed that dynamic intervals that correspond to dynamic code regions that are aligned with the boundaries of procedure calls and loops can accurately represent program behavior.
- We proposed a new phase tracking hardware that consists of a simple stack and a phase signature table. Comparing with existing proposed schemes, this structure can detects a small number and longer phases. Using this structure can also give more accurate prediction of the next execution phase.

The rest of this paper is organized as follows. Section 2 described related work and the motivation of our work. Section 3 describes why dynamic code regions can be used for tracking change of coarse grained phase changes. Section 4 describes our proposed phase classification and prediction hardware. Section 5 evaluates our proposed scheme. Section 6 discusses the results and compares them to other schemes. Finally, Section 7 summarizes the work.

## 2 Background

**Phase detection and prediction** In previous work [1–5, 9], researchers have studied phase behavior to dynamically reconfigure microarchitecture and re-optimize binaries. In order to detect the change of program behavior, metrics representing program runtime characteristics were collected. If the difference of metrics, or code signature, between two intervals exceeds a given threshold, phase change is detected. The stability of a phase can be determined by using performance metrics (such as CPI, cache misses, and branch misprediction) [4, 9], similarity of code execution profiles (such as instruction working set, basic block vector) [1–3] and data access locality (such as data reuse distance) [5] and indirect metrics (such as Entropy) [7].

Our work uses calling context as a signature to distinguish distinct phases in an extended calling context tree (ECCT). Similar extension of CCT was used for locating reconfiguration points to reduce CPU power consumption [26, 27], where the calling context was analyzed on the instrumented profiles. Our phase tracking hardware could find similar reconfiguration points discovered in the analysis of instrumented profiles because we also track similar program calling contexts. This is useful for phase aware power management in embedded processors. M. Huang et al [10] proposes to track calling context by using a hardware stack for microarchitecture adaptation in order to reduce processor power consumption. W. Liu and M. Huang [29] propose to exploit program repetition to accelerate detailed microarchitecture simulation by examining procedure calls and loops in the simulated instruction streams. M. Hind et al. [24] identified two major parameters (granularity and similarity) that capture the essence of phase shift detection problems.

**Exploiting phase behavior for dynamic optimization system** In dynamic optimization systems [14, 19, 20, 22], it is important to maximize the amount of time spent in the code cache because trace regeneration overhead is relatively high and may offset performance gains from optimized traces [15]. Dynamo [14] used preemptive flushing policy for code cache management, which detected a program phase change and flushed the entire code cache. This policy performs more effective than a policy that simply flushes the entire code cache when it is full. Accurate phase change detection would enable more efficient code cache management. ADORE [9, 22] used sampled PC centroid to track instruction working set and coarse-grain phase changes.

**Phase aware profiling** Nagpurkar et al [18] proposed a flexible hardware-software scheme for efficient remote profiling on networked embedded device. It relies on the extraction of meta information from executing programs in the form of phases, and then use this information to guide intelligent online sampling and to manage the communication of those samples. They used BBV based hardware phase tracker which was proposed in [3] and enhanced in [6].

## 3 Dynamic Code Region based Program Phase Tracking

### 3.1 Tracking Dynamic Code Region as a Phase

In this paper, we propose phase tracking hardware that only tracks functions and loops in the program. The hardware consists of a stack and a phase history table. The idea of using a hardware stack is based on the observation that any path from the root node to a node representing a dynamic code region in *Extended Calling Context Tree (ECCT)* can be represented as a stack. To illustrate this observation, we use an example program and its corresponding ECCT in Figure 1. In this example, we assume that each of loop0, loop1 and loop3 executes for a long period of time, and represents dynamic code regions that are potential targets for optimizations. The sequence of function calls which leads to loop1 is main() → func1() → loop1. Thus, if we maintain a runtime stack of the called functions and executed loops while loop1 is executing, we would have main(), func1() and loop1 on it. Similarly, as shown in Figure 1, the content of the stack for code region 2 would be main() and loop0, while for code region 3 it would be main(), func3() and loop3. They uniquely identify the calling context of a code region, and thus could be used as a signature that identifies the code region. For example, the code region loop3 could be identified by the signature main() → func3() on the stack. Code regions in Figure 1 can be formed during runtime. This is why it is called *Dynamic Code Region (DCR)*. Stable DCR is a sub tree which has a stable calling context in ECCT during a monitoring interval, such as one million instructions.

The phase signature table extracts information from the stack and stores the stack signatures. It also assigns a phase ID for each signature. The details of the fields in the table and its function are presented in section 4.

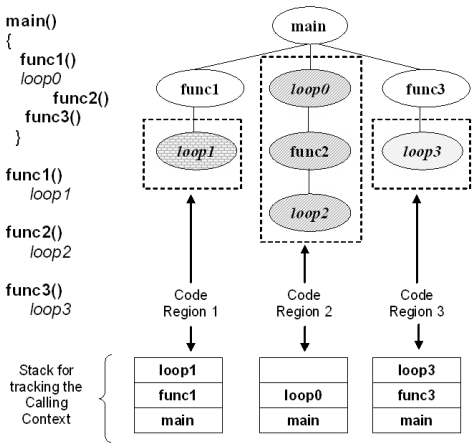



**Fig. 1.** An example code and its corresponding ECCT representation. Three dynamic code regions are identified in the program and are marked by different shades in the tree.

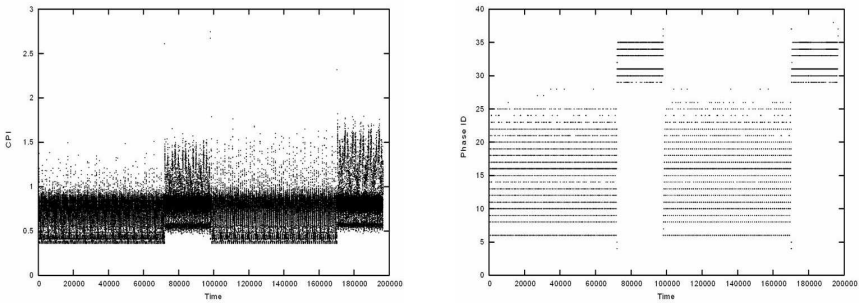
### 3.2 Correlation between Dynamic Code Regions and Program Performance Behaviors

In Figure 2(a), CPI calculated for every 1-million-instruction interval for bzip2 is plotted. We then used the notion of DCR also for every one million instructions and assigned a distinct phase ID for each DCR. Such ID's are then plotted against the time shown in Figure 2(b). Comparing the CPI graph in (a) with the phase ID graph in (b), it can be seen that the CPI variation in the program has a strong correlation with changes in DCR's. This shows that DCR's in a program could reflect program performance behavior and tracks the boundaries of behavior changes. Although BBV also shows the similar correlation, DCR gives code regions that aligned with procedures and loops, which exhibits a higher accuracy in phase tracking and also make it easier for optimization.

There are several horizontal lines in Figure 2(b). It shows that a small number of DCR's are being repeatedly executed during the period. Most DCR's seen in this program are loops. More specifically, phase ID 6 is a loop in `loadAndRLLSource`, phase ID 10 is a loop in `SortIt`, phase ID 17 is a loop in `generateMTFvalues`, and phase ID 31 is a loop in `getAndMoveTofrontDecode`.

### 3.3 Phase Detection for Dynamic Optimization Systems

Dynamic optimization systems prefer longer phases. If the phase detector is overly sensitive, it may trigger profiling and optimization operations too often to cause performance degradation. Phase prediction accuracy is essential to avoid bringing unrelated traces into the code cache. The code cache management system would also require information about the phase boundaries to precisely identity the code region for the phase. The information about the code structure




(a) CPI change over the time (b) Phase Change over the time

**Fig. 2.** Visualizing phase change in bzip2. (a) Change of average CPI. (b) Tracking Phase changes. The Y-axis is phase ID.

can be used by the profiler and the optimizer to identify the code region for operation. Finally, it is generally beneficial to have a small number of phases as long as we can capture most important program behavior; this is because a small number of phases allow the phase detector to identify longer phases and to predict the next phase more accurately. It should be noted that dynamic optimization systems can trade a little variability within each phase for longer phase length and higher predictability, as these factors determines the overheads of the system.

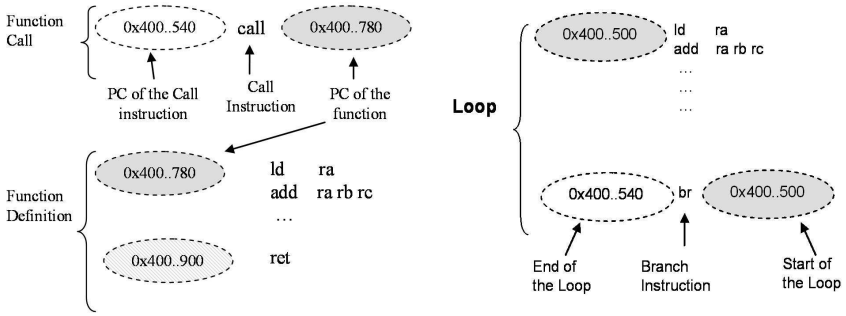
## 4 DCR-based Phase Tracking and Prediction Hardware

We have discussed why DCR can be used to track program execution phases. In this section, we propose a relatively simple hardware structure to track DCR during program execution.

### 4.1 Identifying function calls and loops in the hardware

Function calls and their returns are identified by call and ret instructions in the binary. Most modern architectures have included call/ret instructions defined. On detecting a call instruction (see Figure 3(a)), the PC of the call instruction and the target address of the called function are pushed onto the hardware stack. On detecting a return instruction, they are popped out of the stack. A special case to be considered when detecting function calls and return is recursion. In section 4.3 we describe a technique to deal with recursions.

Loops can be detected using backward branches. A branch which jumps to an address that is lower than the PC of the branch instruction is a backward branch. The target of the branch is the start of the loop and the PC of the branch instruction is the end of the loop. This is illustrated in Figure 3(b). These two addresses represent the boundaries of a loop. Code re-positioning transformations can introduce backward branches that are not loop branches. Such branches may




(a) function call (b) loop

**Fig. 3.** Assembly code of a function call (a) and loop (b). The target address of the branch instruction is the start of the loop and the PC address of the branch instruction is the end of the loop.

temporarily be put on the stack and then get removed quickly. On identifying a loop, the two addresses marking the loop boundaries are pushed onto the stack. To detect a loop, we only need to detect the first iteration of the loop. In order to prevent pushing these two addresses onto the stack multiple times in the subsequent iterations, the following check is performed. On detecting a backward branch, the top of the stack is checked to see if it is a loop. If so, the addresses stored at the top of the stack are compared to that of the detected loop. If the addresses match, we have detected an iteration of a loop which is already on the stack. A loop exit occurs when the program branches out to an address outside the loop boundaries. On a loop exit, the loop node is popped out of the stack.

### 4.2 Hardware Description

The schematic diagram of the hardware is shown in Figure 4. The central part of the phase detector is a hardware stack and the signature table. The configurable parameters in the hardware are the number of entries in the stack and the number of entries in the phase signature table.

Each entry in the hardware stack consists of four fields. The first two fields hold different information for functions and loops. In the case of a function, the first and second fields are used to store the PC address of the call instruction and the PC of the called function respectively. The PC address of the call instruction is used in handling recursions. In the case of a loop, the first two fields are used to store the start and end address of the loop. The third field is a one bit value, called the stable stack bit. This bit is used to track the signature of the dynamic code region. At the start of every interval, the stable stack bit for all entries in the stack which holds a function or a loop is set to '1'. The stable stack bit remains zero for any entry that is pushed into or popped out of the stack during the interval. At the end of the interval, those entries in the bottom of the stack whose stable stack bit is still '1' are entries which were not popped

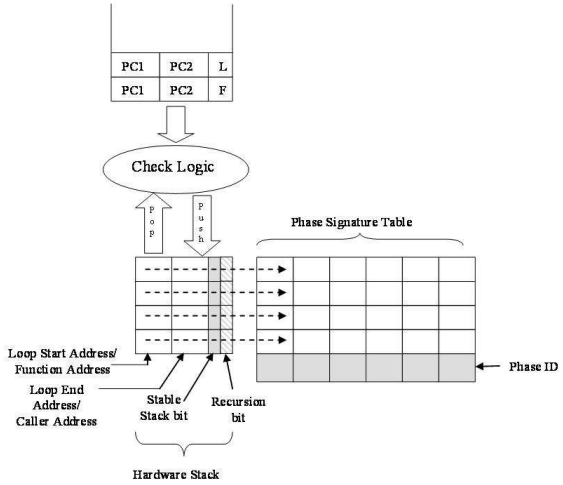



**Fig. 4.** Schematic diagram of the hardware phase detector

during the current interval. These set of entries in the bottom of the stack form the signature to the region in the code to which the execution was restricted to, in the current interval. At the end of the interval, this signature is compared against all signatures stored in the phase signature table. The phase signature table holds the stack signature seen in the past and its associated phase ID. On a match, the phase ID corresponding to that entry in the signature table is returned as the current phase ID. If a match is not detected, a new entry is created in the signature table with the current signature and a new phase ID is assigned to it. If there are no free entries available in the signature table, the least recently used entry is evicted from the signature table to create space for the new entry. The fourth field is a one bit value called the recursion bit and is used when handling recursions. The use of this bit is explained in section 4.3.

### 4.3 Handling special cases

**Recursions** In our phase detection technique, all functions that form a cycle in the call graph (i.e., they are recursive calls) are considered as members of the same dynamic code region.

In our hardware, all recursions are detected by checking the content of the stack. A recursion is detected when the address of the function being called is already present in an entry on the stack. This check assumes that an associative lookup of the stack is done during every push operation. Since the number of entries on the stack is small, the associative lookup hardware would be feasible.

To avoid stack overflow during a recursion, no push operation is performed after a recursion is detected. The recursion bit is set to '1' for the entry corresponding to the function marking the boundary of the recursion. Since we no

longer push any entry onto the stack, we cannot pop any entry from the stack on detecting a return instruction, until it is out of the recursion cycle. This can be detected when a return instruction jumps to a function outside of the recursion cycle. All entries in the stack that are functions, which lies below the entry whose recursion bit is set, are outside the recursion cycle. After a recursion is detected, the return address of all subsequent return instructions are checked against these entries. On a match, all entries above the matched entry are flushed, and normal stack operation is resumed.

**Hardware Stack Overflow** Recursion is not the only case in which a stack overflow could occur. If the stack signature of a dynamic code region has more elements than the stack can hold, the stack would overflow. We did not encounter any stack overflow for a 32-entry stack. But if it did occur, it is handled very similar to a recursive call described earlier. On a stack overflow, no further push operation is performed. The address to which the control gets transferred during a return instruction is checked for a match to an address in the stack. If it matches, all entries above this instruction are removed from the stack and normal stack operation is resumed.

## 5 Evaluation

### 5.1 Evaluation Methodology

Pin and pfmon [11, 13] were used in our experiments on evaluating the effectiveness of the phase detector. Pin is a dynamic instrumentation framework developed at Intel for Itanium processors [13]. A Pin instrumentation routines was developed to detect function calls and returns, backward branches for loops and to maintain a runtime stack.

pfmon is a tool which reads performance counters of the Itanium processor [13]. We use CPI as the overall performance metric to analyze the variability within detected phases. We modified pfmon to get CPI for every one million instructions. To minimize random noise in our measurements,the data collection was repeated 3 times and the average of the 3 runs was used for the analysis. These CPI values were then matched with the phase information obtained from the Pin Tool, to get the variability information.

All the data reported in this paper were collected from a 900 Mhz Itanium-2 processor with 1.5 M L3 cache running Redhat Linux Operating System version 2.4.18-e37.

### 5.2 Metrics

The metrics used in our study are the number of distinct phases, average phase length, Coefficient of Variance (CoV) of CPI, and ratio of next phase prediction. The number of distinct phases corresponds to the number of dynamic code regions detected in the program. Average phase length of a benchmark program

gives the average number of contiguous intervals classified into a phase. It is calculated by taking the sum of the number of contiguous intervals classified into a phase divided by the total number of phases detected in the program. Coefficient of Variation quantifies the variability of program performance behavior and is given by

$$CoV = \frac{\sigma}{\mu} \tag{1}$$

CoV provides a relative measure of the dispersion of data when compared to the mean. We present a weighted average of CoV on different phases detected in each program. Weighted average of the CoV gives more weight to the CoV of phases that have more intervals (i.e., longer execution times) in it and hence, better represent the CoV observed in the program.

The ratio of next phase prediction is the number of intervals whose phase ID was correctly predicted by the hardware divided by the total number of intervals in the program.

### 5.3 Benchmarks

Ten benchmarks from the SPEC CPU2000 benchmark suite (8 integer and 2 floating point benchmarks) were evaluated. These benchmarks were selected for this study because they are known to have interesting phase behavior and are challenging for phase classification [3]. Reference input sets were used for all benchmarks. Three integer benchmarks, namely gzip, bzip and vpr, were evaluated with 2 different input sets. A total of 13 benchmarks and input sets combinations were evaluated. All benchmarks were compiled using gcc (version 3.4) at O3 optimization level.

## 6 Experimental Results

In this section, we present the results of our phase classification and prediction hardware. There are two configurable parameters in our hardware. They are the size of the stack and the size of the phase history table. We evaluated four different configurations for the hardware. The size of the stack and the size of the phase history table were set to 16/16, 32/64, 64/64 and infinity/infinite respectively. We found no significant differences between them in the metrics described above. Due to lack of space, we are unable to show the results here. Interested readers can refer to [30] for the analysis.

In 6.1, we compare the results of our hardware scheme to the basic block vector (BBV) scheme described in [3, 6]. A Pin tool was created to get the BBV. These vectors were analyzed off-line using our implementation of the phase detection mechanism described in [3, 6] to generate the phase ID's for each interval. The metrics which we use for comparison include: the total number of phases detected, average phase length, the next phase prediction accuracy and the CoV of the CPI within each phase.

### 6.1 Comparison with BBV Technique

In this section, we compare the performance of our phase detection hardware with the phase detection hardware based on BBV [3, 6]. The BBV-based phase detection hardware is described in [3]. There are 2 tables in the hardware structure, namely the accumulator table which stores the basic block vector and the signature table which stores the basic block vectors seen in the past. These structures are similar in function to our hardware stack and signature table, respectively. To make a fair comparison, we compare our 32/64 configuration against a BBV-based hardware which has 32 entries in the accumulator table and 64 entries in the signature table.

We compare our phase detector and BBV based detector using four parameters namely, number of phases detected, phase length, predictability of phases, and stability of the performance within each phase. We compare our results with those of the BBV technique for two threshold values namely 10% and 40% of an one-million-instruction interval. The original BBV paper [3] sets the threshold value to be 10% of the interval. It should be noted that the phases detected in the BBV-based technique may not be aligned with the code structures. Aligned phases are desirable for Dynamic Binary Re-Optimization System. We still compare against the BBV method because it is a well accepted method for detecting phases [28].

**Number of phases and phase length** Table 1 compares the number of phases detected for the BBV technique and our phase detection technique. For the BBV technique, there are 2 columns for each benchmark that correspond to a threshold value of 10% and 40% of one million instructions, respectively. In the case of BBV technique, as we increase the threshold value, small differences between the basic block vectors will not cause phase change. Hence, less number of phases is detected as we go from 10% to 40% threshold value. Recall that, in a Dynamic Binary Optimization system, on detecting a new phase, the profiler will start profiling the code, which will cause significant overhead. Hence, for such systems less number of phases with longer per phase length is desirable. We can see that in the original BBV technique with 10% threshold, the number of phases detected is 100 times more than those detected in the DCR based technique. In the BBV technique as we go from 10% to 40% threshold value, the number of phases detected becomes smaller, which is expected. But even at 40%, the number of phases detected in BBV technique is 2x more than those detected by our technique. Table 1 also shows the average phase lengths for the BBV technique and our phase detection technique. The median phase length of our technique is 100 times more than those found in BBV technique with 10% threshold value, and two times more than those found in BBV technique with 40% threshold value. Although in the case of `eon` and `mesa` the phase length for BBV with 40% threshold value is 3 times that of DCR technique, these programs are known to have trivial phase behavior. The larger difference is due to difference in the number of phases detected in our case.

**Table 1.** Comparison of the number of phases and length of phases detected between BBV- and DCR-based phase detection schemes. A 32-entry accumulator table/hardware stack and a 64-entry phase signature table were used.

| | Number of phases | | | Length of phases | | |
|---|---|---|---|---|---|---|
| Benchmarks | BBV-10% | BBV-40% | DCR-32/64 | BBV-10% | BBV-40% | DCR-32/64 |
| ammp | 13424 | 122 | 53 | 58.83 | 6472.27 | 15027.77 |
| bzip2_1 | 35154 | 1796 | 99 | 5.59 | 111.51 | 1984.60 |
| bzip2_2 | 37847 | 1469 | 87 | 4.24 | 108.30 | 1845.51 |
| crafty | 20111 | 20 | 27 | 14.57 | 15073.26 | 10314.54 |
| eon | 38 | 7 | 22 | 6128.94 | 31520.43 | 10029.18 |
| gcc | 2650 | 599 | 337 | 19.70 | 86.84 | 158.15 |
| gzip_1 | 8328 | 182 | 48 | 13.90 | 629.34 | 2450.40 |
| gzip_2 | 4681 | 77 | 42 | 11.11 | 678.23 | 1273.73 |
| mcf | 5507 | 88 | 55 | 19.52 | 1219.19 | 1950.67 |
| mesa | 945 | 15 | 37 | 517.82 | 32899.13 | 13402.89 |
| perl | 8036 | 201 | 28 | 13.76 | 497.59 | 3579.22 |
| vpr_1 | 3136 | 105 | 91 | 47.17 | 1398.50 | 1618.96 |
| vpr_2 | 51 | 27 | 27 | 3715.51 | 7018.19 | 7018.19 |
| median | 5507 | 105 | 48 | 19.52 | 1219.19 | 2450.40 |

**Performance Variance within same phase** Figure 5 compares the performance of the 256-entry Markov Predictor using BBV technique and our phase detection technique. Except eon and vpr_1, the Markov predictor predicts the next phase better using our phase detector. On average using our phase detection technique, the Markov predictor predicts the correct next phase ID 84.9% of the time. Using BBV-based technique, the average correct prediction ratios are 42.9% and 73.3% for 10% and 40% respectively.

**Phase Prediction Accuracy** Figure 6 compares the weighted average of the CoV of CPI for phases detected by the BBV technique and by our phase detection technique. The last four bars give the average CoV. From the figure we can see that BBV-10% has the least average CoV value. This is because the number of phases detected by BBV-10% is much higher than the number of phases detected by BBV-40% or our technique. In the case of BBV-10%, the variability of CPI gets divided into many phases, thus reducing the variability observed per phase. On average the %CoV of our phase detection hardware is 14.7% while it is 12.57% for the BBV-40%. Although the average variability of our technique is greater than BBV-40%, the numbers are comparable. In fact for bzip2_1, crafty, gzip_1, gzip_2, mesa, vpr_1 and vpr_2, the CoV of the DCR based technique is less than or equal to the CoV observed for the BBV-40% . For ammp, gcc, mcf and perl the performance variation is higher in the dynamic code regions detected. The higher performance variation within each dynamic code region may be due to change in control flow as in the case of gcc or change in data access patterns as in the case of mcf.

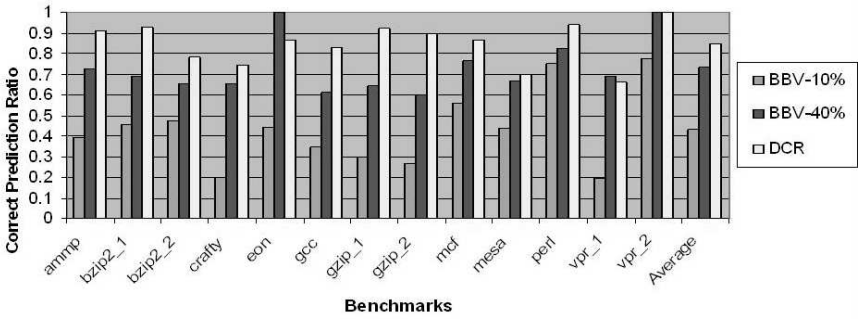



**Fig. 5.** Comparison between BBV- and DCR-based phase detection A 32-entry accumulator table/hardware stack and a 64-entry phase signature table were used. The first 2 columns for each benchmark are for BBV method using threshold values of 10% and 40% of one million instructions respectively.

From the above discussions we can conclude that, our hardware detects less number of phases, has a longer average phase length, has higher phase predictability and is aligned with the code structure, all of which are desirable characteristics of a phase detector for a Dynamic Binary Re-optimization system. The CoV of the phases detected in our technique is higher but comparable to that observed in the BBV technique with 40% threshold value. The phase difference is detected using an absolute comparison of phase signatures, which makes the hardware simpler and the decision easier to make. The 32/64 hardware configuration performs similar to an infinite sized hardware, which makes it cost effective and easier to design.

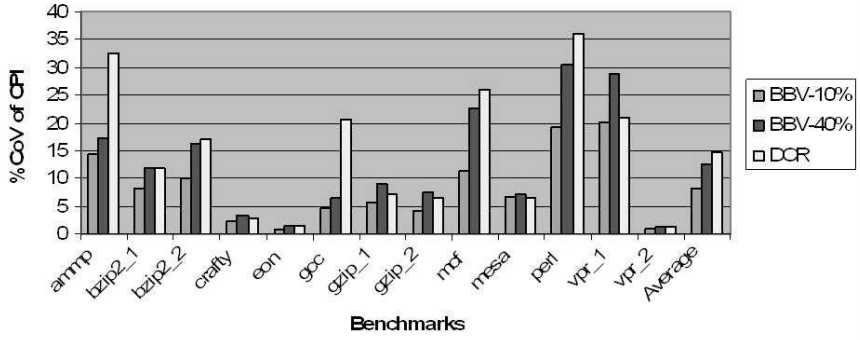



**Fig. 6.** Comparison of the weighted average of the CoV of CPI between BBV- and DCR-based phase detection schemes. A 32-entry accumulator table/hardware stack and a 64-entry phase signature tables were used.

## 7 Conclusion and Future Work

We have evaluated the effectiveness of our DCR-based phase tracking hardware on a set of SPEC benchmark programs with known phase behaviors. We have shown that our hardware exhibits the desirable characteristics of a phase detector for dynamic optimization systems. The hardware is simple and cost effective. The phase sequence detected by our hardware could be accurately predicted using simple prediction techniques. We are currently implementing our phase detection technique within a dynamic optimization framework. We plan to evaluate the effectiveness of our technique for dynamic profile guided optimizations such as instruction cache prefetching and for code cache management.

## References

1. 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.
2. 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.
3. T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In 30th Annual International Symposium on Computer Architecture, June 2003.
4. 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
5. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In International Conference on Architectural Support for Programming Languages and Operating Systems, 2004
6. J. Lau, S. Schoenmackers, and B. Calder. Transition Phase Classification and Prediction, In the 11th International Symposium on High Performance Computer Architecture, February, 2005.
7. M. Sun, J.E. Daly, H. Wang and J.P. Shen. Entropy-based Characterization of Program Phase Behaviors. In the 7th Workshop on Computer Architecture Evaluation using Commercial Workloads, February 2004.
8. M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. The Fuzzy Correlation between Code and Performance Predictability. In the 37th International Symposium on Microarchitecture, December 2004.
9. J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, and P.-C. Yew. The Performance of Data Cache Prefetching in a Dynamic Optimization System. In the 36th International Symposium on Microarchitecture, December 2003.
10. M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In 30th Annual International Symposium on Computer Architecture, June 2003.
11. PIN - A Dynamic Binary Instrumentation Tool. http://rogue.colorado.edu/Pin.
12. J. Lau, S. Schoenmackers, and B. Calder. Structures for Phase Classification. In IEEE International Symposium on Performance Analysis of Systems and Software, March 2004.
13. http://www.hpl.hp.com/research/linux/perfmon.

14. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2000.
15. K. Hazelwood and M.D. Smith. Generational cache management of code traces in dynamic optimization systems. In 36th International Symposium on Microarchitecture, December 2003.
16. K. Hazelwood and James E. Smith. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems. In second Annual IEEE/ACM International Symposium on Code Generation and Optimization, March 2004.
17. M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In third Annual IEEE/ACM International Symposium on Code Generation and Optimization, March 2005.
18. P. Nagpurkar, C. Krintz and T. Sherwood. Phase-Aware Remote Profiling. In the third Annual IEEE/ACM International Symposium on Code Generation and Optimization, March 2005.
19. D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In First Annual International Symposium on Code Generation and Optimization, March 2003.
20. W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization. In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000.
21. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufman, 1997.
22. H. Chen, J. Lu, W.-C Hsu, P.-C Yew. Continuous Adaptive Object-Code Reoptimization Framework, In 9th Asia-Pacific Computer Systems Architecture Conference, 2004
23. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeno JVM, in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, October 2000.
24. M.J. Hind, V.T. Rajan, and P.F. Sweeney. Phase shift detection: a problem classification, in IBM Research Report RC-22887, pp. 45-57.
25. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI), June 1997.
26. G. Magklis, M. L. Scott, G. Semeraro, D. A. Albonesi, and S. Dropsho, Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In Proceedings of the International Symposium on Computer Architecture, June 2003.
27. C.-H Hsu and U. Kermer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2003.
28. A. S. Dhodapkar and J. E. Smith, Comparing program phase detection techniques. In the 36th International Symposium on Microarchitecture, December 2003.
29. W. Liu and M. Huang, EXPERT: expedited simulation exploiting program behavior repetition. In Proceedings of the 18th annual international conference on Supercomputing, June 2004.
30. J. Kim, S.V. Kodakara, W.-C. Hsu, D.J. Lilja and P.-C Yew, Dynamic Code Region-based Program Phase Classification and Transition Prediction, in Uinversity of Minnesota, Computer Science & Engineering Technical Report 05-021, May 2005.