Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 05-021

Dynamic Code Region-based Program Phase Classification and Transition Prediction

Jinpyo Kim, Sreekumar V. Kodakara, Wei-chung Hsu, David J. Lilja,
and Pen-chung Yew

May 23, 2005

Dynamic Code Region-based Program Phase Classification and Transition Prediction

Jinpyo Kim[*], Sreekumar V. Kodakara[+], Wei-Chung Hsu[*], David J. Lilja[+], Pen-Chung Yew[*]

[*]Department of Computer Science & Engineering
University of Minnesota-Twin Cities
{jinpyo, hsu, yew}@cs.umn.edu

[+]Department of Electrical and Computer Engineering
University of Minnesota-Twin Cities
{sreek, lilja}@ece.umn.edu

Abstract

Detecting and predicting a program's execution phases is crucial to dynamically adaptable systems and dynamic optimizations. Program execution phases have a strong connection to program control structures, in particular, loops and procedure calls. Intuitively, a phase can be associated with some dynamic code regions that are embedded in loops and procedures.

This paper proposes off-line and on-line analysis techniques that could effectively identify and predict program phases by exploiting program control flow information. For off-line analyses, we introduce a dynamic interval analysis method that converts the complete program execution into an annotated tree with statistical information attached to each dynamic code region. It can efficiently identify dynamic code regions associated with program execution phases at different granularities. For on-line analyses, we propose new phase tracking hardware which can effectively classify program phases and predict next execution phases.

We have applied our dynamic interval analysis method on 10 SPEC CPU2000 benchmarks. We demonstrate that the change in program behavior has strong correlation with control transfer between dynamic code regions. We found that a small number of dynamic code regions can represent the whole program execution with high code coverage. Our proposed on-line phase tracking hardware feature can effectively identify a stable phase at a given granularity and very accurately predict the next execution phase.

1 Introduction

Understanding and predicting a program's execution phase is crucial to dynamically adaptable systems and dynamic optimizations. Accurate classification of program behavior opens up many optimization opportunities for adaptive reconfigurable microarchitectures, dynamic optimization systems, power management, and accelerated architecture simulation [1, 2, 3, 4, 7, 9, 14]. In dynamic optimization systems, program phase behavior is exploited for better code cache management and for dynamic profile-guided optimizations [14, 18].

A *phase* is defined as a set of intervals within a program's execution that have similar behavior and performance characteristics, regardless of temporal adjacency [6]. *Phase classification* partitions a set of intervals into phases with similar behavior. *Phase prediction* predicts the phase classification of the next interval of execution.

In previous work [1, 2, 3, 4, 6], the execution of a program was divided into equal sized non-overlapping intervals. An *interval* is a contiguous portion (i.e., a time slice) of execution of a program. Metrics representing program runtime characteristics were calculated for every interval. If the *difference* in the metrics between two intervals exceeds a given threshold, a phase change is assumed. The metrics used in these techniques are generally independent of the microarchitecture implementation, and can be related to code execution profiles using basic blocks.

In this work, we identify a phase using higher-level control structures such as function calls and loops. We track the loops and procedure calls during the execution of the program, and determine the dynamic code regions in the execution that are responsible for different phases. In our analysis, we found that by tracking higher-level code structures, we were able to effectively track the phase changes in a program execution. This is because, intuitively, programs exhibit phase behaviors as a result of control transfer through procedures, nested loop structures and recursive functions. In [6], it was found that tracking loops and procedures gave comparable phase tracking accuracy to the Basic Block Vector (BBV) method [3,6]. This also supports our observation.

Furthermore, in our phase classification scheme, the detected phases are aligned with procedure and loop boundaries. This would be very useful for evaluating compiler optimizations for new architectures/micro-architectures. For example, SimPoint [3] provides the representative intervals for fast simulation using the number of instructions executed as the starting point of the selected intervals. However, the SimPoint method is not suitable for evaluating compiler optimizations where the number of instruction executed would change as a result of new optimization transformations. Using phases aligned with procedures and/or loops would allow the compiler to effectively select representative code regions to simulate in order to quickly determine the effectiveness of optimizations for new architecture/micro-architectures.

In this paper, we propose an off-line software analysis technique and an online hardware structure for performing phase classification. In the off-line analysis technique, we build an Extended Dynamic Call Graph (EDCG) of the program by tracking loops and function calls, and perform Dynamic Interval Analysis (DIA) on the graph to detect dynamic code regions that correspond to phases in the program execution. In the on-line technique, we track the code signature of procedure calls and loops using a special hardware stack, and compare against previously seen code signatures to detect 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 introduce the Extended Dynamic Call Graph (EDCG) to summarize the runtime behavior of the program.**
- **We introduce an off-line analysis technique called Dynamic Interval Analysis to automatically detect dynamic code regions in the EDCG.**
- **We propose a phase classification hardware structure that closely approximates the dynamic interval analysis at runtime.**

The rest of this paper is organized as follows. Section 2 describes a framework that builds the Extended Dynamic Call Graph (EDCG) and analysis techniques that can detect dynamic code regions showing repetitive program behavior. Section 3 describes our proposed phase classification and prediction hardware. Section 4 presents our evaluation methodology, a description of the benchmark evaluated, and the metrics used for evaluation. Section 5 presents the results for our phase classification techniques and compares these results to the BBV [3] scheme. Section 6 presents related work on phase classification and prediction. Finally, Section 7 summarizes this work.

2 Dynamic Code Region-based Program Phase Identification

Understanding whole program runtime behavior is very attractive to researchers studying computer architectures, compiler optimizations, and autonomic system management since it provides a bird's eye view of the whole program execution. In this section, we introduce an off-line technique which converts whole program execution into an annotated tree with statistical information such as cumulated number of instructions executed, cache miss events, miss-speculation failures and so on, to determine repetitive dynamic code regions and their associated performance characteristics at different granularities.

2.1 Extended Dynamic Call Graph

A Call Graph CG= (N, E) is a multigraph, where N is the set of its nodes, and E is the set of edges. Each node in N represents a procedure call. Each edge in E is a triplet (N1, CS, N2) that represents a procedure call to procedure N2 from *call site* CS in procedure N1. A Dynamic Call Graph (DCG) is a call graph that contains only the edges that are observed at runtime. In this respect, DCG may be considered as a subgraph of the static call graph [17]. On the other hand, if a procedure is called from different call paths during execution, there will be multiple nodes representing each of the dynamic call instances in the DCG.

In this study, in order to detect repeated program behavior, we include loop nodes in the DCG. Also, to track context sensitivity, we add one constraint to the graph definition, in that each node can have only one entry point. This constraint requires the graph to be a tree that captures procedure calling contexts and loop nesting structures. We call this *Extended Dynamic Call Graph (EDCG).* Figure 1 shows a simple program along with its EDCG representation.

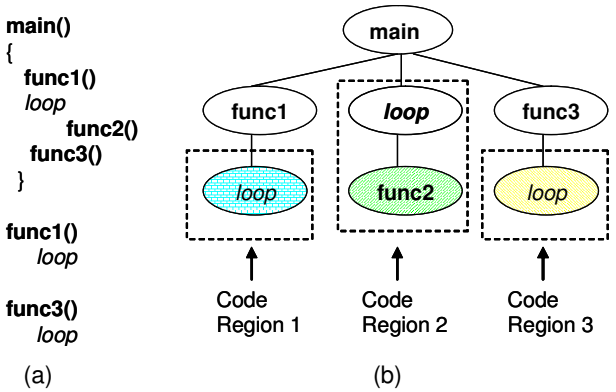


Figure 1. *An example code and its corresponding EDCG representation. Three dynamic code regions are identified in the program and are marked by different shades in the tree.*

Consider the program shown in Figure 1. The function main in the program has two function calls func1 and func3 and a loop with func2 embedded in it. Each of func1 and func3 has a loop in its body. The EDCG of this program is shown in Figure 1(b). In Figure 1(b), there are three potential Dynamic Code Regions which could represent repeated program behavior. A formal definition of Dynamic Code Region is given in section 2.2.

2.2 Dynamic Code Region (DCR)

In region-based compilation framework, a region is defined as a set of connected nodes and edges [21]. Each node in the region represents either a nested region or a basic block. A directed edge is an edge connecting two nodes and represents the control flow from the source node to the target node. Relations among regions are organized hierarchically as a tree structure. This tree is called *region tree*. The EDCG holds a similar region nesting property except the nodes in EDCG are loops and procedure calls, instead of regions.

The nodes in EDCG shown in section 2.1 hold the following properties:

- Each node in the tree except the root node has only one parent. The root node has no parent.
- Each node can have any number of child nodes.

The first property implies that each node in the tree can be used to represent all its child nodes. Thus, each node in the tree represents the code region comprising itself and all the nodes under it. For example, in Figure 1(b), the node main represents the code region comprising of the whole program. Node func2 represents the code region comprising func2 and the loop in the body of func2. Since we are interested in identifying regions in the program which show repeated program behavior, the nodes of interest in the EDCG are loops and recursive function calls.

Since a tree can have many such nodes, we need an algorithm to identify these important nodes within the EDCG. *Dynamic Interval Analysis* identifies dynamic code regions in the EDCG that corresponds to repeated program behavior.

2.3 Dynamic Interval Analysis

Static Interval Analysis is used in both control and data flow analyses [22]. In control-flow analysis, static interval analysis is used to divide the flow graph into regions of various sorts (e.g. if-then-else and while-do) and consolidate each region into an abstract node. The abstract nodes and basic blocks in static interval analysis form a control tree [22]. EDCG introduced in section 2.2 has similar properties to the control tree since parent nodes subsume child nodes.

In Dynamic Interval Analysis (DIA), we search the EDCG to identify regions in the tree which correspond to repeated behavior during runtime. The algorithm of DIA is shown in Figure 2. DIA can be thought of as a breadth-first search of the EDCG with some constraints on frontier expansion. Recall from Section 2.2 that every node in the EDCG represents a dynamic code region consisting of all nodes under it. Statistics regarding the dynamic code region is stored in each node of the EDCG. Specifically, cumulative number of instructions retired in the dynamic code region and the number of times this dynamic code region was visited during the execution of the program are stored in each node. The mean number of instructions during each invocation of the dynamic code region can be calculated. This is the average size of the dynamic code region. One constraint on the breadth first search is that the average size of the dynamic code region should be greater than or equal to a user defined minimum average size. By changing the minimum average size, we can get dynamic code regions of different granularities.

The second, but a weak constraint on the breadth first search is the coverage of the dynamic code region. One property of EDCG is that the sum of the cumulative number of instructions of all nodes at a distance k from the root node is less than the sum of the cumulative number of instructions of all nodes at distance k+1 from the root node. This implies that, as we go down the tree, the total number of instructions covered by the dynamic code regions decreases. We call this the coverage of the dynamic code regions. The second constraint on the breadth-first search is that the coverage of the dynamic code regions in the children of the current node should be greater than or equal to the user specified minimum coverage value.

Dynamic Interval Analysis Algorithm

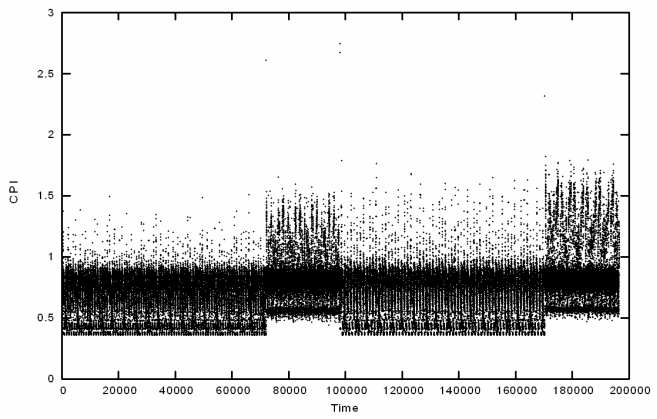
```
w <- root of EDCG;
// root node of Extended Dynamic Call Graph
th <- threshold;
// user defined minimum code region size (100M, 10M, 1M, 100K )
cov_constraints<- coverage;
// coverage constraints (99%, 95%, 90%, 0%)
global_cum_insts <- cumulative insts counts of root node;
// root coverage (100%)
cur_cum_insts <- cumulative insts counts of current frontiers,
                        intial node is root;
frontiers <- Φ;                 // frontiers are children of given node w
DCR_list <- Φ;
DIA (w, cur_cum_insts) {
   frontiers <- children of node w;
   if (frontiers is null) {
         put node w into DCR_list;  return;
}
   sum_insts <- 0;   drill_down <- 0;
   rest_cum_insts <- cur_cum_insts – cumul. insts counts of node w;
   for (each frontiers in node w) {
         sum_cov  +=  cumul. insts counts of node frontiers;
   }
   cur_cum_insts <- rest_cum_insts + sum_cov;
   cur_cov <- cur_cum_insts / global_cum_insts * 100.0;
   if (cur_cov < cov_constraints) put node w into DCR_list;
   else {
for (each frontiers in node w) {
                if (mean num of insts of frontier >  th) {
                      DIA(frontiers, cur_cum_insts);
                      drill_down  += 1;
                }
         }  /* for */
         if (drill_down is zero)
                put node w into DCR_list;
   } /* else */
} /* DIA */
```

Figure 2. *Dynamic Interval Analysis Algorithm*

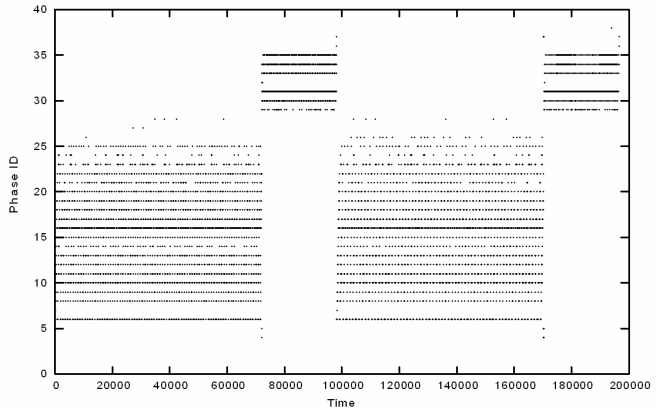
2.4 Correlation of Dynamic Code Region and Program Phase Behavior

In Figure 3(a), CPI calculated for every 1 Million instruction interval for bzip2 is plotted. We detected the DCR of bzip2 for every one million instruction interval and assigned a distinct phase ID for each DCR. This ID is plotted against time plotted that in Figure 3(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 strong correlation with changes in DCR. This implies that DCR in a program precisely reflects program performance behavior and tracks the boundary of behavior changes. We can observe several horizontal lines in Figure 3 (b). These lines mean that small number of DCR's is being repeatedly executed during the execution of bzip2. Most DCR 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.



(a) CPI change over the time



(b) Phase Change over the time

Figure 3. *Visualizing phase change in bzip2. (a) Change of average CPI during program execution. Each point in the graph is the average CPI observed over 1 million instruction interval. (b) Tracking Phase changes over the time (1 million instruction interval) in bzip2 using dynamic code region. The Y-axis is phase ID.*

3 Hardware Support for Phase Classification and Transition Prediction

The previous section described the algorithm of Dynamic Interval Analysis for program phase detection. In order for this analysis to be useful for runtime phase detection, the overhead of the analysis must be kept low. The complete Dynamic Interval Analysis as described above could be too expensive to be used within the framework of a runtime optimizer. In this section, we describe a modification and a hardware support to make runtime phase detection using DIA feasible.

The key observation which leads to this simplification is that the subtree of a node in EDCG that corresponds to a phase in the program execution lies on top of a stack of function calls and loops that leads to it from the start of the program. To illustrate this observation, we use the example program and its corresponding tree representation in Figure 1. In this example, phase 1 is caused by the loop in function func1. The sequence of function calls which lead to this loop is main and func1. Thus if we maintain a runtime stack of the called functions and loops, we would have main and func1 in it for code region 1. This content of the stack would be the signature identifying the code region defined by the execution of the loop in func1. Similarly for code region 2, the content of the stack would be main and loop, while for code region 3 it would be main and func3. These are unique signatures for the different phases in this program and can be easily identified. In the hardware approximation of Dynamic Interval Analysis this signature is captured in a hardware stack. At the end of an interval, we compare this signature to the one seen earlier to detect stable phases and phase changes.

3.1 Identifying function calls and loops in the hardware

Function calls and their returns are identified by call and ret instructions in the program. Most modern architectures have call/ret instructions defined. On detecting a call instruction, the target PC of the instruction is pushed on the stack. A call to a function whose PC address is already present in the stack implies a recursive call. If a recursion is detected, the function is not pushed on the stack. On detecting a return instruction, the function is popped out of the stack. Since a return instruction can be present anywhere in the body of a function, even inside loops, multiple pop operations on the stack might be needed before the entry corresponding to the function can be removed.

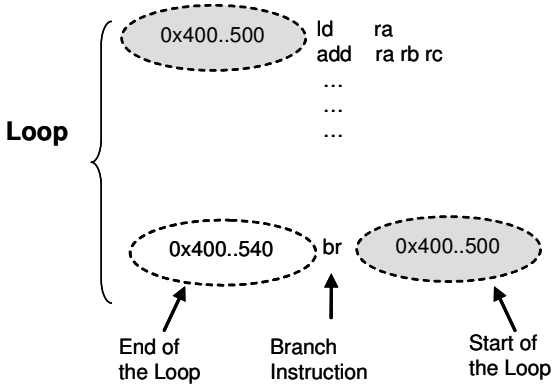


Figure 4. *Assembly code of a loop. The target address of the branch instruction is the start of the loop and the PC address of the branch instruction is the end of the loop.*

Loops can be detected using backward branches. A branch which jumps to an address that is less than the PC of the branch itself is considered as the boundary of a loop. Code re-positioning transformations may introduce backward branches that are not loop branches. Such branches may temporarily stay on the stack and get removed quickly. The starting address of the loop is the target of the branch and the ending address of the loop is the PC of the backward branch. This is illustrated in Figure 4. On detecting a loop, the two addresses marking the loop boundary are pushed on the stack. In essence, to detect a loop we only need to detect the first iteration of the loop. In order to prevent pushing subsequent iterations of the loop onto the stack, 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 is compared to that of the detected loop. If it is the same, we have detected the loop and the newly detected loop boundary is not pushed onto the stack. A loop exit is detected when the program exits the loop boundary. On a loop exit, the loop is popped from the stack. The algorithm for phase detection is summarized in Figure 5.

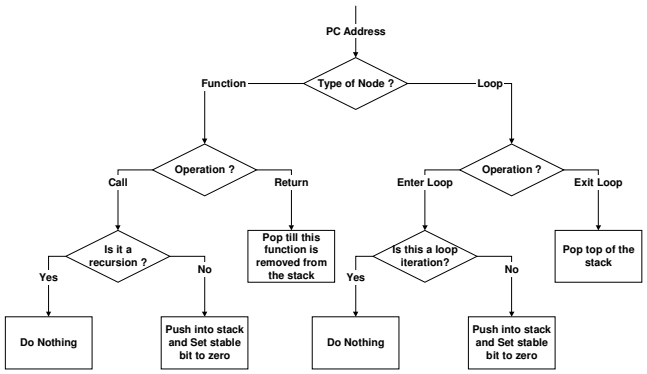


Figure 5. *Conditions checked in the phase detection hardware*

3.2 Hardware Description

The schematic diagram of the hardware is shown in Figure 6. The central part of the program phase detector is a hardware stack and the signature table. Each entry in the hardware stack consists of three fields. The first field is to store the PC address of the function or the start of the loop. The second field is used to store the ending address of the loop. In the case of a function, this field is not used. The third field is a one bit value, called the stable stack bit. At the start of an interval, the stable stack bit for every active entry in the stack is set to '1'. Whenever an entry is popped, all fields of the entry, including the stable stack bit, is reset to '0'. At the end of the interval, those entries in the bottom of the stack whose bits are still set to '1' form the signature to a region to which the execution was restricted to in the current interval. At the end of an interval, the stable stack bottom is compared against all entries in the phase signature table for a match. If there is a match, the phase ID corresponding to that entry in the signature table it returned as the current phase ID. If a match is not detected, a new entry is created in the signature table with a new phase ID. If there is no free entry in the signature table, the entry which was least recently used is replaced with this new entry and new phase ID is created for this entry. The check logic showed in Figure 6 implements the algorithm presented in Figure 5.

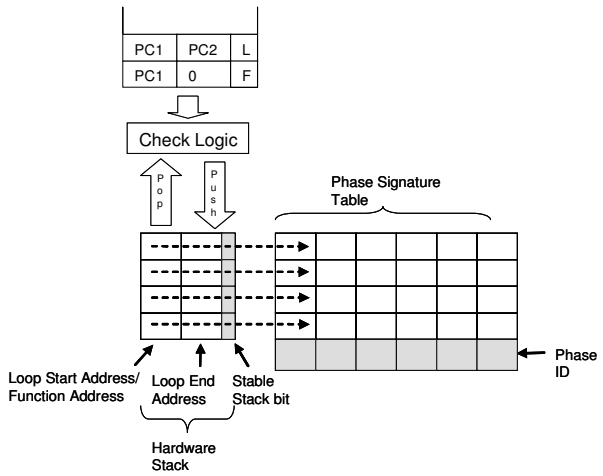


Figure 6. *Schematic diagram of the hardware phase detector*

4 Evaluation

4.1 Evaluation Methodology

Pin Tool and pfmon [11, 13] were used in our study. Pin is a dynamic instrumentation framework developed at Intel for Itanium processors [13]. Pin provides a simple, but flexible API for transparently inserting instrumentation routines at runtime. Instrumentation routines were used for building Extended Dynamic Call Graph (EDCG) and identifying Dynamic Code Region (DCR). The phase prediction hardware was simulated within dynamic instrumentation routine while executing the program with Pin tool.

pfmon was used to collect performance counter data from Itanium-2 processors. pfmon is a tool which reads performance counters of the Itanium processor [13]. We use CPI as the overall performance metric to analyze the stability of the detected phases. CPI for each interval of instructions is obtained using periodic sampling of the performance counters using pfmon. Because real machine measurements on Itanium are used, there can be random noise in the collected data. To reduce the noise in the data, the data collection was repeated 3 times and the average of the 3 runs were used for all measurements.

CPI metric was measured for every interval of 1 Million instructions. All the data reported in this paper was collected from 900 Mhz Itanium-2 processors with 1.5 M L3 cache running on Redhat Linux Operating System version 2.4.18-e37.

4.2 Metrics

The metrics used in our study are *the number of distinct phases*, *Coefficient of Variance* (CoV) of CPI, and *correct next phase prediction ratio*. The number of distinct phases corresponds to the number of dynamic code regions detected in the program. A metric which quantifies the variability of program performance behavior is *Coefficient of Variation* and is given by

$$CoV = Standard\ Deviation / Mean$$

CoV provides a relative measure of the dispersion of data when compared to the mean. A smaller *CoV* for the performance metrics within each phase implies that the phase is more stable.

We present a weighted average of *CoV* on different phases detected in each program. The formula for calculating the weighted average of the *CoV* for each benchmark is given by

$$\sum(n_i \times CoV_i) / \sum n_i$$

where,

n_i is the number of intervals in the phase *i*

CoV_i is the *CoV* of performance metric of phase *i*

We use weighted average of the *CoV* to give more weight to the *CoV* of phases that have more number of intervals (ie., longer execution times) and hence better represents the *CoV* observed in the program.

4.3 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]. Three integer benchmarks were evaluated with 2 different input sets to illustrate the effects of different input sets on the performance of the phase detection hardware. Reference input sets were used for all benchmarks. A total of 13 benchmark and input set combinations were evaluated. All the benchmarks were compiled using gcc (version 3.4) at O3 optimization level. A summary of the benchmarks and input set combination evaluated is shown in Table 1.

No	Benchmark	Input Set
1	gzip_1 gzip_2	input.source 60 input.log 60
2	vpr	net.in arch.in place.in route.out –nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8 net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
3	gcc	166.i –o 166.s
4	mcf	inp.in
5	crafty	crafty.in
6	eon	chair.control.kajiya chair.camera chair.surfaces chair.kajiya.ppm ppm pixels_out.kajiya
7	perlbmk	-I./lib splitmail.pl 850 5 19 18 1500
8	bzip2_1 bzip2_2	input.program 58 input.source 58
9	mesa	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm
10	ammp	ammp.in

Table 1. *Summary of the benchmark and input combinations used in our evaluation*

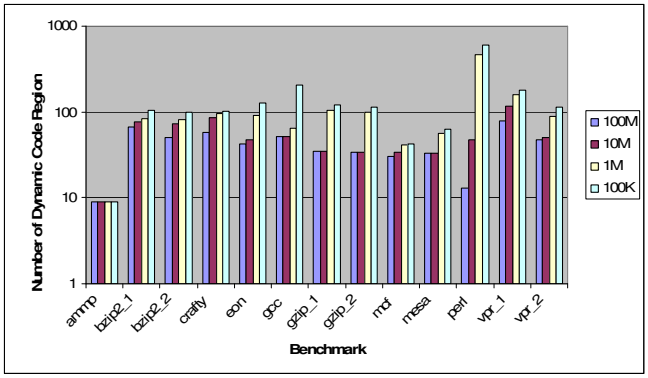
5 Experimental Results

In this section we present the results of our on-line and off-line phase classification schemes. Section 5.1 shows the results for our off-line scheme. In Section 5.2 and 5.3, we present the results of our hardware scheme and the comparison of our hardware scheme to the basic block vector (BBV) scheme described in [3,6]. Pin tool was created to get the basic block vectors. 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 metric which we use for comparison is the total number of phases detected, the CoV of the CPI within each phase and the next phase prediction accuracy.

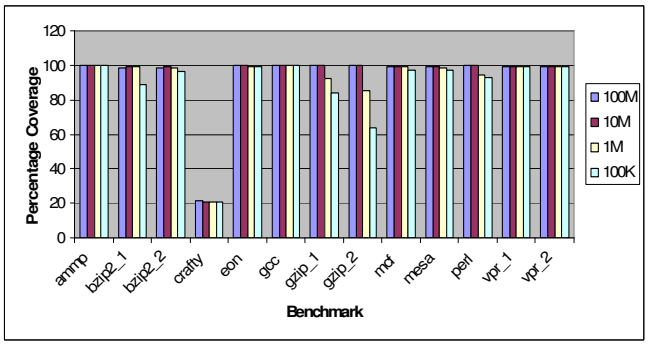
5.1 Results for Dynamic Interval Analysis

In this section we analyze the results of dynamic interval analysis when applied to SPEC 2000 benchmarks. Figure 7 (a) shows the number of dynamic code region detected by DIA for different benchmark programs and different granularities of the dynamic code region. Four different granularities namely 100K, 1M, 10M and 100M is evaluated for each benchmark. The y-axis is the number of dynamic regions detected by the algorithm and is plotted in log scale. The constraint on the coverage was set to zero. This means that the algorithm will find dynamic regions in the EDCG of the specified granularity, without being terminated early due to coverage constraint. From Figure 7(a) except for ammp, the number of dynamic code regions detected for all programs increase as we decrease the size of the granularity. This is a natural consequence of the algorithm. Also, except perl and vpr_1, for all other programs the number of dynamic code regions is less than 100 for a granularity of 1 Million instructions or more. We found that among the detected DCR's, the number of DCR's which were executed frequently was much less than 100 and varied with the

benchmark. Figure 7(b) gives the percentage coverage of the detected dynamic code regions on termination of the algorithm for different benchmarks and different granularities. Except for crafty, the coverage is 80% or more for granularities 1 Million or more for all benchmarks. These observations imply that dynamic profiling and dynamic optimization systems can cover most of the execution of the program by monitoring a small number of dynamic code regions.



(a)



(b)

Figure 7. *(a) Number of dynamic code region detected by Dynamic Interval Analysis for different granularities. (b) Percentage coverage of the detected code region for different granularities*

5.2 Results for Phase Detection Hardware

Figure 8 shows the number of phases detected using different configurations of our phase detection hardware across different benchmark programs. The x-axis shows the benchmark programs and the y-axis shows the number of phases detected. Y-axis is plotted in log scale. The last bar is the average number of phases detected across all programs. For each benchmark program, there are 4 bars which correspond to 16/16, 32/64, 64/64 and infinite hardware resources. 16/16 means that there are 16 entries in the hardware stack and 16 entries in the signature table. Similarly 32/64 and 64/64 means 32 and 64 entries in the hardware stack and 64 entries in the signature table, respectively. *Infinite* corresponds to infinite number of entries in both hardware stack and signature table. It should be noted that except in the case of 16/16, the number of phases detected for all programs is very close to or exactly the same as that of using the infinite hardware.

Figure 9 shows the variation of the weighted average of percent Covariance (CoV) on CPI for different benchmarks and hardware configurations. The y-axis shows the weighted average of CoV and the x-axis shows different benchmark programs. The last 4 bars show the average of the CoV for different hardware configurations. It should be noted that the CoV for the infinite hardware is similar to 32/64 or 64/64 configurations. On average the CoV for the phases detected by our hardware is 15%.

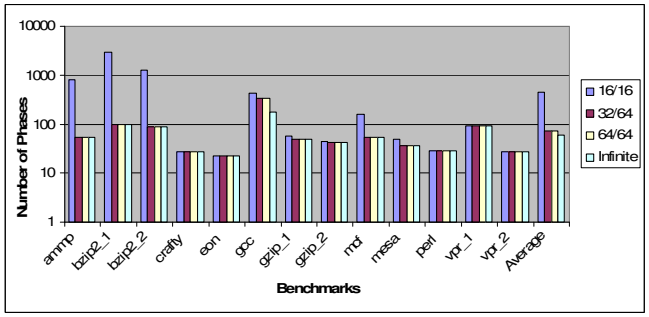


Figure 8 *Number of phases detected for different sizes of the phase detection hardware.*

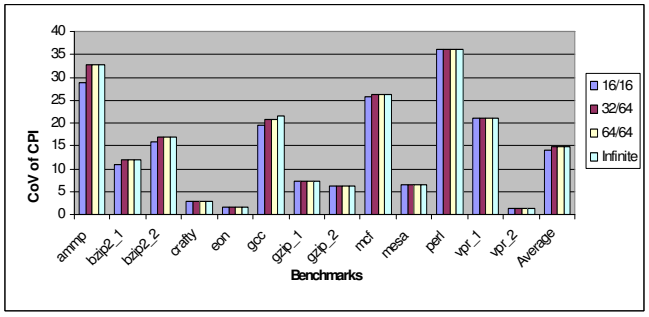


Figure 9 *Weighted average of the CoV of CPI for different configurations of the phase detection hardware*

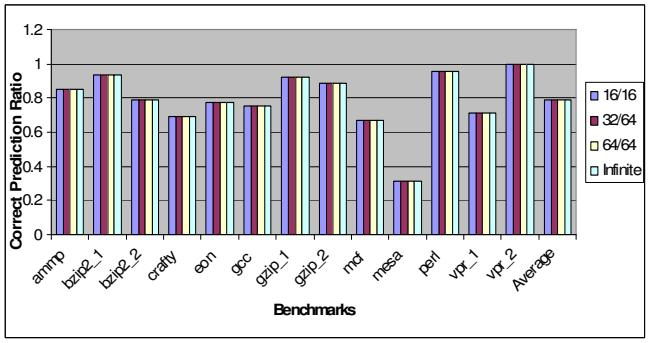


Figure 10 *The performance of last-phase predictor in detecting the phase of the next interval.*

Figure 10 shows the performance of a simple last-phase predictor for different benchmark programs. A last-phase predictor is one which predicts the next phase to be the same

as the current one. Thus a last-phase predictor predicts stable phase behavior. On an average, a simple last-phase predictor correctly predicts the next phase ID 80% of the time.

Figure 11 shows the performance of a 256-entry direct-mapped Markov Predictor with Run Length Encoding [3]. The Markov predictor on an average correctly predicts next phase ID 84% of the time.

There is one underlying thread in all these discussions. The performance of 32/64 or 64/64 phase detector is very similar to the hardware with infinite resource. This makes the phase detection hardware very cost effective.

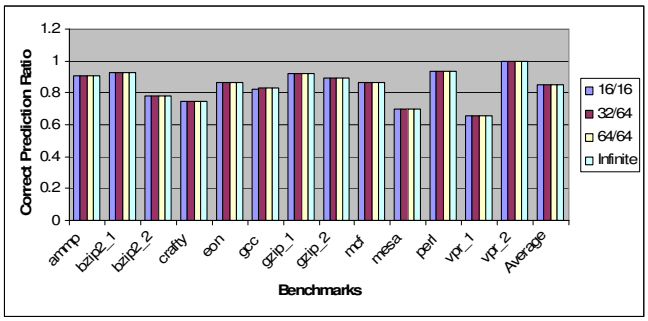


Figure 11 *The performance of 256-entry Markov Predictor in detecting the phase of the next interval.*

5.3 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].

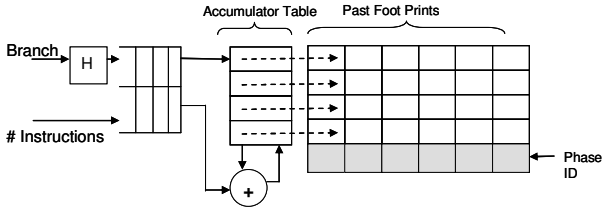


Figure 12 *BBV-based phase tracking hardware*

The BBV-based phase detection hardware [3,6] is shown in Figure 12. 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 vector 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. Also, in the BBV-based method, a phase change is detected by comparing the Manhattan distance of two vectors to a threshold value. In this study we change the threshold value from 100 thousand instructions to 600 thousand instructions so we can see the effect of threshold on the performance of the hardware.

Figure 13 compares the number of phases detected for the BBV technique and our phase detection technique. The y-axis shows the number of distinct phases detected and is plotted in log scale. For the BBV technique, there are 3 bars for each benchmark corresponding to a threshold value of 100K, 400K and 600K 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 and hence fewer number of phases are detected. DCR-based technique detects comparable number of phases to the BBV technique when the threshold value is set at 600K.

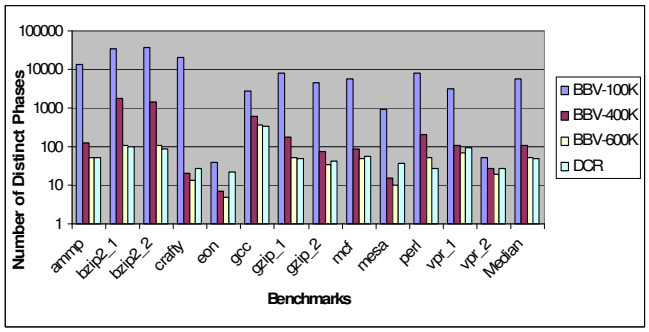


Figure 13 *Comparison of the number of phases detected between BBV- and DCR-based phase detection schemes. A 32 entry accumulator table/hardware stack and 64 entry phase signature table were used. The first 3 columns for each benchmark correspond to a threshold value of 100K, 400K and 600K respectively.*

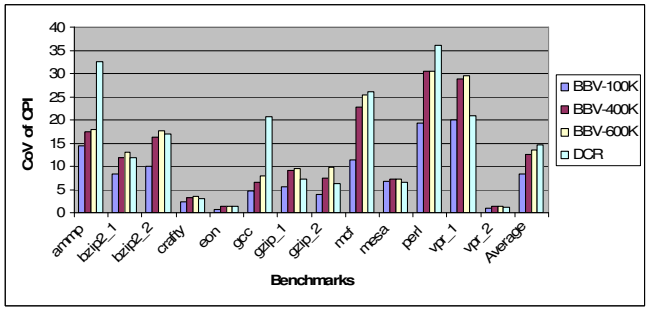


Figure 14 *Comparison of the weighted average of the CoV of CPI between BBV- and DCR-based phase detection schemes. 32-entry accumulator table/hardware stack and 64-entry phase signature tables were used. The first 3 columns for each benchmark are for a threshold value of 100K, 400K and 600K, respectively.*

Figure 14 compares the weighted average of the CoV of CPI for phases detected by the BBV technique and our phase detection technique. The last four bars give the average CoV. In the case of BBV technique, as we increase the threshold value fewer number of distinct phases are detected which increases the variation within each phase. Thus, as we increase the threshold value, the CoV increases. 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 technique with 400K threshold value. For bzip2_2 and eon the CoV is less than equal to the CoV detected by the BBV technique with 600K threshold value and for ammp, gcc, mcf and perl the CoV is

greater than that detected by the BBV technique with 600K threshold value. On average the %CoV for our phase detection hardware is 14.7% while for the BBV based technique they are 12.57% and 13.4 % for a threshold of 400K and 600K respectively.

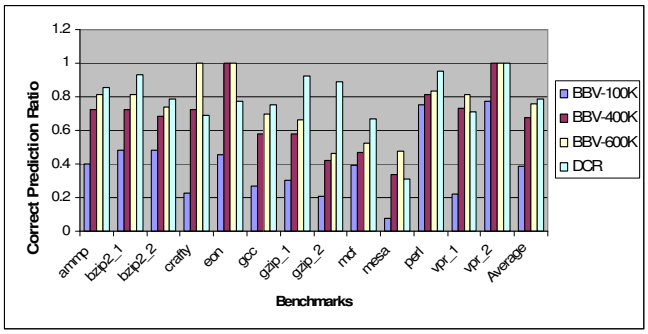


Figure 15 *Comparison of the performance of a simple last-phase predictor between BBV- and DCR-based phase detection. 32 entry accumulator table/hardware stack and 64-entry phase signature table were used. The first 3 columns for each benchmark are for a threshold value of 100K, 400K and 600K, respectively.*

Figure 15 compares the performance of the last-phase predictor for predicting phases detected by the BBV technique and our phase detection technique for different benchmark programs and different threshold values. The y-axis shows the correct prediction ratio. The last four bars show the average across all benchmark programs. For the BBV technique, the accuracy of the next phase prediction increases as we increase the threshold value. This is a consequence of the number of phases detected for different threshold values. As we increase the threshold, fewer number of distinct phases is detected which in turn increases the predictability of phases. For all programs except crafty, eon, mesa and vpr_1 the last phase predictor performs better for the phase ID sequence generated for our phase detection technique. On average using our phase detection technique, the last-phase predictor predicts the correct next phase ID 78.82% of the time. For BBV based technique, the average correct prediction ratio is 38.8%, 67.5% and 75.68% for thresholds 100K, 400K and 600K, respectively.

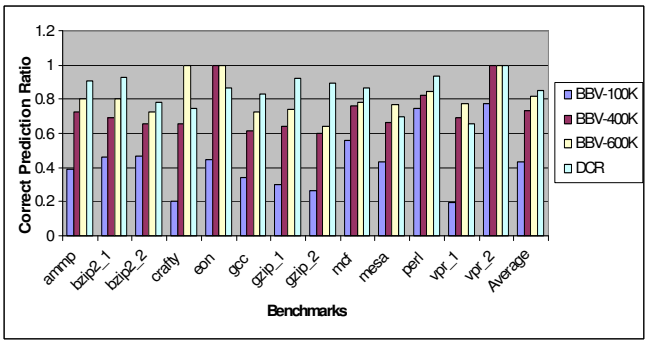


Figure 16 *Comparison between BBV- and DCR-based phase detection hardware on the performance of a 256-entry Markov Predictor in predicting the phase ID of the next interval. A 32-entry accumulator table/hardware stack and 64 entry phase signature table was used. The first 3 columns for each benchmark are for BBV method using threshold values of 100K, 400K and 600K, respectively.*

Figure 16 compares the performance of the 256-entry Markov Predictor using BBV technique and our phase detection technique for different benchmark programs and different threshold values. A similar trend seen in the last-phase predictor can be seen here. Except crafty, eon, mesa and vpr_1, the Markov predictor predicts the next phase ID's better for the phase ID's generated using our phase detector over the phase ID's generated by BBV technique for different threshold values. 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 ratio are 42.9%, 73.3% and 81.5% for thresholds of 100K, 400K and 600K, respectively.

6 Related Work

Phase detection and prediction: In previous work [1, 2, 3, 4, 5, 9], researchers have studied phase behavior to re-optimize binary in response of time-varying program behavior. In order to detect the change of program behavior, metrics representing program runtime characteristics were calculated. If the difference of metrics between two intervals exceeds the given threshold, or code signature between two intervals is different, phase change is detected. The stability of phase can be evaluated 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, 2, 3] and data access locality (such as data reuse distance) [5] and indirect metrics (such as Entropy) [9]. J. Lau et al. [8] examined the use of program structures (such as basic blocks, loops, and procedures) and memory access profile (such as local stride, global stride, and loops with local strides) for phase tracking and compared their approach to the BBV method. Some of the phase detection methods can also be used to identify representative program execution intervals in traces to drastically reduce program simulation time.

Exploiting phase behavior in dynamic optimization system: In Dynamic optimization systems [14, 19, 20, 23], 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, 23] 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].

7 Conclusion and future work

Regions have been used in advance compiler optimizations [21]. Regions are usually identified by detecting loops in the control flow graph. Relationship between regions can be represented by a region tree. We propose to build an Extended Dynamic Call Graph (EDCG) by tracking every executed procedure calls and loops during runtime. Each node in EDCG could be a potential Dynamic Code Region (DCR). We also introduce Dynamic Interval Analysis (DIA) to identify coarse-grained Dynamic Code Regions to represent program execution phases.

We have evaluated the effectiveness of our phase classification and prediction scheme using the PIN/Itanium tool and Pfmon on a set of SPEC benchmark programs with known phase change behaviors. We found that a small number of DCRs can represent the whole program execution with a high code coverage. DCR is relatively stable. This implies that dynamic optimization systems could focus on a small number of DCR for efficient optimizations.

We propose to use DCR for precisely tracking program phase behaviors. Our DCR-based phase tracking hardware could accurately predict the next execution phase. Accurate prediction of the next execution phase would allow the dynamic optimization system to activate or launch optimized code for the next execution phase. Our off-line DIA can effectively detect program execution phases and connect them to program control structure. This would allow the compiler to select the most time consuming code regions to simulate and quickly evaluate the effectiveness of optimizations for new architectures and micro-architectures.

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] Y. Liu, Z. Zhang, and R. Qiao. A Region-Based Compilation Infrastructure, In *7th Workshop on Interaction between Compilers and Computer Architectures,* February 2003.

[22] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufman, 1997.

[23] H. Chen, J. Lu, W.-C Hsu, P.-C Yew. Continuous Adaptive Object-Code Re-optimization Framework, In *9th Asia-Pacific Computer Systems Architecture Conference*, 2004