Program Phase Detection based on Critical Basic Block Transitions

Paruj Ratanaworabhan[1] and Martin Burtscher[2]
[1] Computer Systems Laboratory, Cornell University
[2] Center for Distributed and Grid Computing, The University of Texas at Austin
paruj@csl.cornell.edu, burtscher@ices.utexas.edu

Abstract

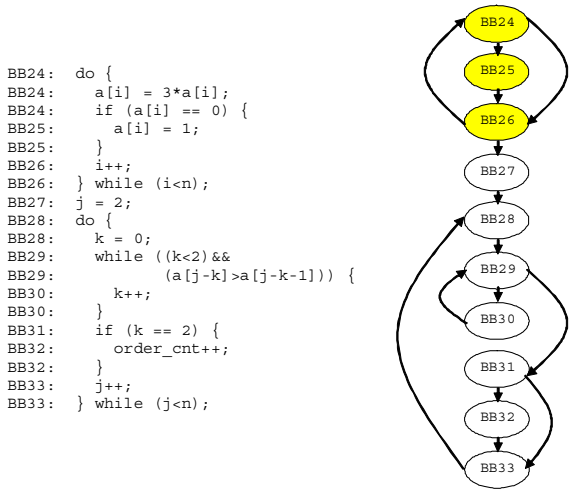
Many programs go through phases as they execute. Knowing where these phases begin and end can be beneficial. For example, adaptive architectures can exploit such information to lower their power consumption without much loss in performance. Architectural simulations can benefit from phase information by simulating only a small interval of each program phase, which significantly reduces the simulation time while still yielding results that are representative of complete simulations. This paper presents a lightweight profile-based phase detection technique that marks each phase change boundary in the program's binary at the basic block level with a critical basic block transition (CBBT). It is independent of execution windows and does not explicitly employ the notion of threshold to make a phase change decision. We evaluate the effectiveness of CBBTs for reconfiguring the L1 data cache size and for guiding architectural simulations. Our CBBT method is as effective at dynamically reducing the L1 data cache size as idealized cache reconfiguration schemes are. Using CBBTs to statically determine simulation intervals yields as low a CPI error as the well-known SimPoint method does. In addition, experimental results indicate the CBBTs' effectiveness in both the self-trained and cross-trained inputs, demonstrating the CBBTs' stability across different program inputs.

1. Introduction

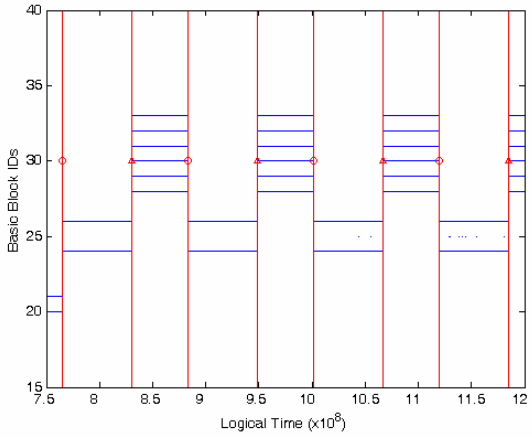
Most programs go through distinct phases during the course of their execution [16]. For illustration purposes, consider the control flow of the code snippet in Figure 1a, which is assumed to process a large array of integers whose elements are uniformly distributed. The first loop scales each element and treats zeros separately. The second loop counts the number of any three consecutives array elements that appear in ascending order. Both loops reside in an outer loop (not shown). The basic block execution profile of this code is given in Figure 1b whose x-axis depicts the logical time in number of committed instructions and the y-axis represents the basic block IDs.

Figure 2 shows the branch misprediction rate of a bimodal [20] and a hybrid branch predictor [13] on this sample code. The x-axis depicts the logical time in number of committed instructions. The y-axis represents the misprediction rate.

The branch misprediction profile shows that the misprediction rates divide the program execution into two distinct phases that repeat. A branch misprediction rate of close to 0% is achieved in the first big phase (Figure 2a and 2b) whereas the second big phase suffers from about a 25% and 8% misprediction rate for the bimodal and the hybrid branch predictor, respectively.



(a)



(b)

Figure 1: Sample code (a) and its basic block execution profile (b)

With the growing interest in optimizing power and performance in adaptive architectures [2], identifying program phases is gaining importance. For instance, if we have two branch prediction units, e.g., a simple and a complex predictor like the Alpha 21264 [8], we may decide, based on the branch misprediction profile, to disable or even turn off the more complicated predictor to save power in the first big phase, realizing that it cannot be used to increase the prediction accuracy in this phase. However, in the second phase, we clearly want to turn it back on because there it greatly reduces the branch misprediction rate.

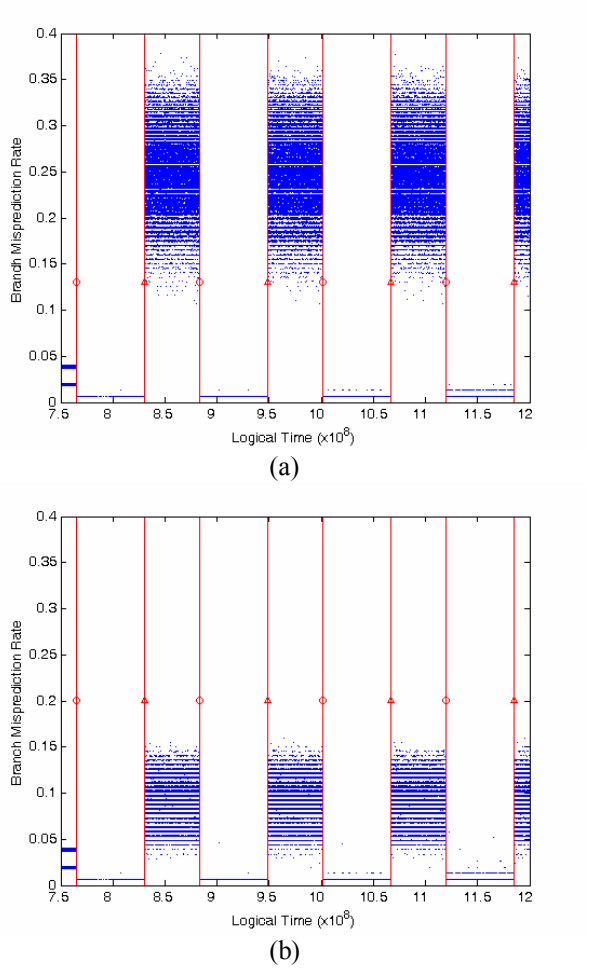


Figure 2: Branch misprediction rate for a bimodal predictor (a) and a hybrid predictor (b)

Architectural simulations can also take advantage of phase behavior. Instead of simulating an entire program run, a small interval from each phase can be chosen and simulated. If the small interval is representative of the longer phase, we can save simulation time without sacrificing accuracy.

Characterizing phase behavior usually involves the notion of phase metrics, windows of execution, and thresholds. Two execution windows are said to belong to the same phase if the phase metric of the two windows does not differ by more than a preset threshold. Otherwise, they are said to be in different phases. Several phase metrics have been proposed to capture program phase behavior [4]. In the above example, the metric is the branch misprediction accuracy. Shen et al. [15] base the phase behavior on the reuse distance. Balasubramonian et al. [2] utilize information from hardware counters. Dhodapkar and Smith [5] employ working set signatures, and Sherwood et al. [16] use Basic Block Vectors (BBVs).

This paper presents the Miss-Triggered Phase Detection (**MTPD**) algorithm. MTPD is a profile-based technique that does not explicitly use any phase metric, measurement window, or threshold to detect program phase changes. Instead, it employs simple program heuristics that pertain to the program's working set at the basic block level, and uses this information to discover the critical basic block transitions (CBBTs). CBBTs mark phase transition points in the program's binary and are used to delineate the program phases. A CBBT can be thought of as a program's phase marker [9, 15] that requires two reference points, a previous and a next BB, to signal a phase change.

To motivate MTPD and CBBTs, consider the code in Figure 1 once more. The BB working set of the first loop is {BB24, BB25, BB26}. The rarely executed BB25 is hardly visible in Figure 2a. For the second loop, the BB working set is {BB27, BB28, BB29, BB30, BB31, BB32, BB33}. In this example, we see a direct correlation between the branch prediction accuracy and the set of BBs being executed. Looking at the source code, we see that the first loop contains two easily predictable conditional branches, a loop end branch and a rarely taken branch that checks for a zero array element. The second loop contains one easily predictable loop-end branch and two difficult to predict conditional branches, one associated with the condition in the inner while loop and the other with the if condition to update *order_cnt*. These two branches are, however, not completely unpredictable. The if branch behavior is dependent upon the inner while branch. Whenever the inner while branch falls through and enters the inner loop twice, this if branch will also fall through. As for the inner while branch, its behavior is—to a certain extent—dependent upon itself. If it falls through twice, the next time it will be taken as the k<2 condition is met. A hybrid branch predictor should be able to capture this behavior with relatively high accuracy, as the branch misprediction profile in Figure 2b shows.

For this code, the transition from BB26 to BB27 (marked in Figure 2 with up triangles) represents the critical transition. Whenever this transition occurs, it flags a shift in the program behavior, in this case, as expressed by the branch prediction accuracy. We call this transition from BB26 to BB27 a critical basic block transition (**CBBT**). The other CBBT (not shown in the sample code and marked with circles in Figure 2) is BB23 to BB24, which represents the transition from the outer loop to the two inner loops, i.e., the two loops in the sample code.

The goal of this work is to automatically identify CBBTs and to use them to discover program phase changes. Our approach distinguishes itself from Lau et al.'s [9] and Shen et al.'s [15], two other profile-based phase detection schemes that generate similar phase markers in the following respects.

1. In contrast to Lau et al.'s technique, which considers only loops and procedures, MTPD operates at a finer granularity; it narrows each phase change down to individual basic blocks. This allows MTPD to more precisely identify the location where a phase change occurs (in the source code and in the executable). We will show in Section 2.2 that there are cases where operating at this fine granularity is necessary to discern important phase behavior. Even though Shen et al.'s technique considers every instruction as a potential phase change point, when producing the final markings, it constructs a phase hierarchy through grammar compression that often increases the phase granularity beyond a single basic block.

2. Contrary to Lau et al.'s and Shen et al.'s techniques, which use only a single reference point for a phase change

marker (a code boundary such as a loop or function header), CBBTs mark phase transition not at code boundaries but at transitions from one boundary to another. This tends to make the phase marking very stable across program inputs. The CBBTs obtained with one input often faithfully track changes in the phase length and the number of phase repetitions that result from different inputs (see Section 2.3).

3. Our approach breaks ties with the use of execution windows—of fixed or variable lengths—and explicit thresholds. In addition, it does not employ a specific phase metric. Even though our notion of basic block signature (Section 2) seems to resemble Dhodapkar and Smith's working set signature [5], there is a key difference between the two. To detect a phase change, the working set signature scheme uses a fixed window measurement and a set threshold, whereas the BB signature scheme has no notion of either. BB signatures are explained in more detail in the following section. Being largely independent of a phase metric, execution window, and threshold makes our technique less susceptible to poor parameter selection and helps to minimize the overfitting problem common to profile-based approaches where performance is good with the self-trained inputs but poor with cross-trained inputs.

The remainder of this paper is organized as follows. Section 2 describes the MTPD algorithm in detail. Section 3 presents the evaluation of a phase detection scheme using CBBTs and investigates the use of CBBTs for dynamic cache reconfiguration and picking architectural simulation points. Section 4 discusses related work. Section 5 concludes the paper with a summary.

2. Miss-Triggered Phase Detection

The Miss-Triggered Phase Detection (MTPD) algorithm identifies a program's phase change points at the basic block level, i.e., in the program's binary, and works as follows. First, the algorithm profiles an application on at least one input to generate a stream of BB identifiers (IDs). MTPD conceptually maintains an infinite-size cache of basic block IDs and monitors the misses that occur in this cache. As the program under investigation transitions from one phase to another for the first time, it is likely to start executing a new working set of BBs, which will cause compulsory misses in the BB ID cache. MTPD scrutinizes the BB transitions that cause such misses using heuristics that are based on typical program behavior during a phase transition and identifies CBBTs based on these heuristics. The following subsections explain the operation of MTPD in more detail.

2.1 Finding the CBBTs

Step 1: Create a cache with infinite capacity and prepare the BB execution traces.

In this initialization step, we create a data structure to represent the ideal cache for the BB IDs. The most appropriate structure seems to be a chained hash table as it allows for efficient searching while faithfully mimicking infinite capacity (as long as there is enough memory). On the benchmarks we evaluated, a hash table with 50,000 entries results in virtually no collisions. We generated the BB execution traces using ATOM [21]. ATOM assigns a unique ID to each BB, and the BB traces simply consist of the sequence of IDs of the executed basic blocks. The BB traces derived from the execution of the SPEC CPU2000 programs with the train inputs range from 1 GB (*mgrid*) to about 10 GB (*bzip2*) in size.

Step 2: Sequentially read in BB IDs from a trace or stream and record the compulsory cache misses.

The BB IDs can be obtained from any source. For programs that generate very large BB execution traces, streaming in BB information may be the most appropriate approach. In any case, MTPD checks whether the just-read-in BB ID is already in the cache (i.e., has been seen before) and records a miss if it is not. As an example, let us look at Figures 1 and 2 again. Prior to logical time $8.3 \cdot 10^8$, the cache includes BBs 24, 25, and 26. However, BBs 27 through 33 are not in the cache. At time $8.3 \cdot 10^8$, the algorithm sees BB 27 for the first time and thus records a miss.

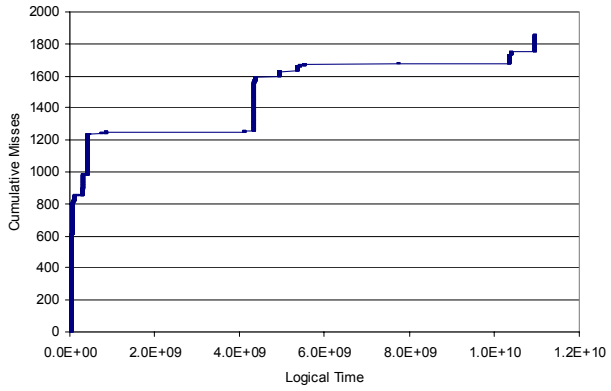
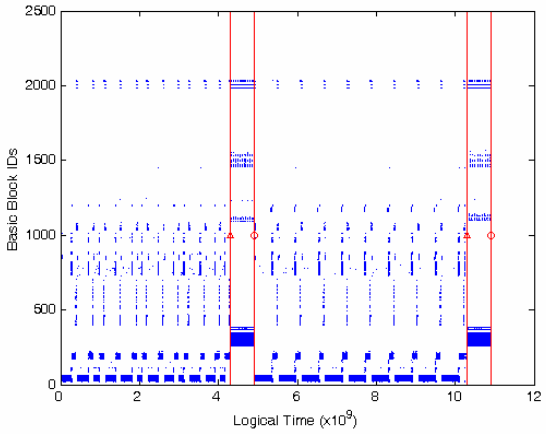


Figure 3: Cumulative number of compulsory BB misses in *bzip2*

Step 3: Record the basic block transitions that are followed by a series of compulsory misses.

When looking closely at the profile of compulsory misses in the ideal cache, e.g., that of *bzip2* executed with the train input shown in Figure 3, we see that misses often occur in bursts. Intuitively, this matches the expected behavior of program execution. For instance, a program might execute a given working set for a while and then transition to another working set, where it again stays for a while. After that, it may transition back to a previously visited working set or move on to a new working set. Based on such behavior, we derived the following simple heuristic: BB transitions that signal a true phase change are typically followed by a series of closely spaced BB misses. This is expected as the program is transitioning to a new set of BBs (i.e., a new working set). In the example from Section 1, misses for BBs 28 through 33 will almost immediately follow the transition from BB26 to BB27, which is indeed a critical transition that signals a phase change.



(a)

```
void compressStream ( FILE *stream, FILE *zStream )
{

// some initialization code

   while (True) {

      blockNo++;
      initialiseCRC ();
      loadAndRLEsource ( stream );
      ERROR_IF_NOT_ZERO ( ferror(stream) );
      if (last == -1) break;

******************** CBBT ********************

      blockCRC = getFinalCRC ();
      combinedCRC = (combinedCRC << 1) | (combinedCRC >> 31);
      combinedCRC ^= blockCRC;

// more code
```

(b)

Figure 4: *bzip2*'s phase behavior at the coarsest level; CBBT phase marking in BB profile (a) and source code (b)

Step 4: Form the BB transition signatures.

After a new BB transition is encountered, we form a BB signature for that transition by grouping together the BBs that miss in close temporal proximity in the ideal cache following the transition. At this point, we have a record of this new transition and its signature, the latter of which is representative of the BB working set after this transition. In the illustrative example, the transition is from BB26 to BB27 and its signature is {BB28, BB29, BB30, BB31, BB32, BB33}. Note that the length of the signature can vary depending on the BB working set size.

Step 5: Identify the CBBTs.

In the last step, we identify CBBTs based on BB transitions and their associated signatures. There are 2 cases to consider.

The first case deals with recorded BB transitions that only occur once in the BB stream. Such transitions may signal a phase change to or from a non-recurring phase. They often reveal interesting large-scale program behavior as the next section illustrates. Basically, a non-recurring transition must satisfy the following conditions to be regarded as a CBBT: 1) It must have an associated signature, i.e., form a signature of length greater than zero, 2) the sum of frequencies of occurrence of all BBs in the signature must be greater than the phase granularity of interest, and 3) a non-recurring CBBT must be separated in logical time from the previous non-recurring CBBT by at least the phase granularity magnitude.

The second case deals with recorded BB transitions that occur multiple times. They may indicate that the program is transitioning back to a previously seen phase. Here, we compare the stream of unique BBs that are encountered after the transition with its previously stored signature. If the set of encountered BBs is a subset of the stored signature, we consider the transition stable and flag it as a CBBT.

Each recorded CBBT also contains logical timestamp information of its first and last occurrence (Time_First_CBBT and Time_Last_CBBT) and its frequency of occurrence (Frequency_CBBT). With these parameters, we can approximate the phase granularity of a given CBBT by the following formula:

$$\text{Phase Granularity of CBBT}_i \approx (\text{Time\_Last\_CBBT}_i - \text{Time\_First\_CBBT}_i) / (\text{Frequency\_CBBT}_i - 1)$$

This information allows the user to select how fine-grained a phase behavior to detect.

Occasionally, recurring transitions have instances where some rare control flow conditions introduce BBs that are not in the original signature. To account for this situation, we consider two signatures to match if at least 90% of their BBs are the same, which increases the robustness of the algorithm. The C source code for MTPD is available on-line at:

http://www.csl.cornell.edu/~paruj/cbbt.html

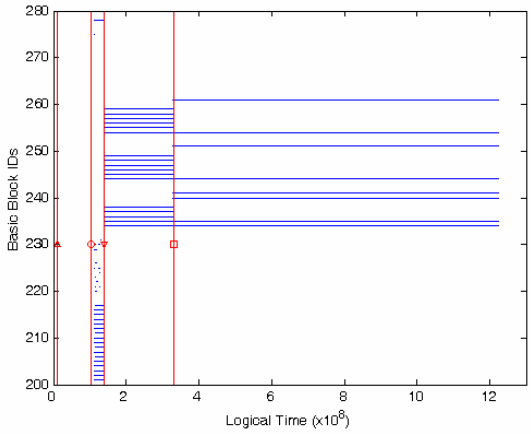
Once the CBBTs are discovered, the application code can be instrumented at the CBBTs using a binary rewriting tool such as ATOM or ALTO [14]. If the source code is available, it could even be augmented with ISA-independent phase-change markings (see Section 2.2 below). A CBBT involves two basic blocks, representing the critical transition, that are consecutively executed. In our working example, the consecutive execution of BB26 and BB27 marks a phase change.

2.2 CBBT Source-Code Association

In this section, we show on two examples how the CBBTs discovered by the MTPD algorithm map to source code.

bzip2 example

At the coarsest granularity, *bzip2* has two distinct phases as shown in Figure 4a. In the source code we see that this transition, indeed, signals a major change as the program execution switches from compression to decompression (marked with up triangles in Figure 4a) and vice versa. Mapping this critical transition back to the source code identifies the code section shown in Figure 4b. The while (True) loop follows some initialization code in the *compressStream* function. The program executes this loop many times to compress the input data. Then, shortly after logical time $4 \cdot 10^9$, a CBBT is executed.



(a)

```
double phi2(t)
double t;
{
  double value;

  if (t <= Exc.t0) {

    value = 2.0 * PI / Exc.t0 / Exc.t0 * sin(2.0 * PI * t / Exc.t0);
    return value;

  }

************** CBBT **************

  else
    return 0.0;
}
```

(b)

Figure 5: *equake*'s phase behavior at the coarsest level; CBBT phase marking in BB profile (a) and source code (b)

The same critical transition occurs again shortly after logical time $10 \cdot 10^9$. This CBBT corresponds to the fall-through path of the if (last == -1) statement to the break statement. Once program execution reaches this point, it terminates the while loop and enters decompression mode. The CBBT boundary is also marked in Figure 4b.

equake example

Figure 5a shows *equake*'s phase boundaries at the coarsest granularity. At this level, there exists no recurring phase behavior; the program keeps transitioning to new working sets. The last phase transition (marked with a square) exhibits an interesting CBBT behavior. The critical transition is from BB254 to BB261.

Figure 5b shows the source code of the procedure *phi2*. This procedure compiles into ten basic blocks whose IDs range from 253 to 262. BB254 is a commonly executed BB prior to and after the CBBT for the last phase transition; it represents the if (t <= Exc.t0) condition. Prior to encountering the critical transition from BB254 to BB261, the if (t <= Exc.t0) condition is always met. The program takes the "then" path and returns the calculated *value*. BB261, which represents the "else" block, is not yet in the program working set. At the phase transition, the program branches to the "else" path and returns 0.0. As we can see from Figure 5a, this "else" path (i.e., the jump from BB254 to BB261) becomes the regular path after encountering this CBBT. Note that phase detection schemes that operate at the loop or procedure level would not have caught this last phase transition in *equake* because it occurs inside an if statement.

2.3 Self-Trained versus Cross-Trained CBBTs

To determine whether CBBTs mark phase boundaries in an input-independent way, we investigate how well CBBTs perform when applied to program runs with the train inputs (self-trained) and the reference inputs (cross-trained). Note that we obtain the CBBTs from profiles with the train inputs in both cases. If CBBTs indeed mark inherently critical program transitions, they should perform well on the self- and cross-trained inputs.

This section provides a qualitative assessment of CBBT markings. A more quantitative evaluation will follow in the next section. We find that the CBBTs discovered with the train inputs do, in fact, work well on the reference inputs. Figure 6 shows the CBBT markings for *mcf* and *gzip*. For clarity, only large-scale phase markings (at a granularity of a billion executed instructions) are shown.

Clearly, the CBBT markings adapt well to changes in the phase length and the number of phase recurrences due to different program inputs. In case of *mcf*, a 5-cycle phase behavior with the self-trained input is correctly partitioned into a 9-cycle phase behavior with the cross-trained input. Note that each unique CBBT is marked with a different symbol. For *mcf*, the up triangles represent the CBBT transitions into a phase where the two functions *primal_bea_mpp* and *refresh_potential* are frequently executed whereas the circles represent the CBBT transitions into a phase where the function *price_out_impl* is frequently executed. For *gzip*, the first two phase cycles toggle between *deflate_fast* (down triangle) and *inflate_dynamic* (up triangle) phases, and the next three cycles alternate between *deflate* (circle) and *inflate_dynamic* (up triangle) phases.

3. Evaluation

This section studies how the phase boundaries marked by CBBTs can be used to detect phase changes and evaluates CBBTs' effectiveness. Section 3.1 describes the benchmark programs. Section 3.2 discusses phase detection using BB worksets (BBWS) and BB vectors (BBV). Applying CBBT phase detection for dynamic cache configuration and for picking representative architectural simulation points is the subject of Sections 3.3 and 3.4, respectively.

3.1 Benchmarks

We evaluate CBBT-based phase detection on ten programs from the SPEC CPU2000 benchmark suite [4]. They are the four floating-point programs *art*, *equake*, *applu*, and *mgrid* and

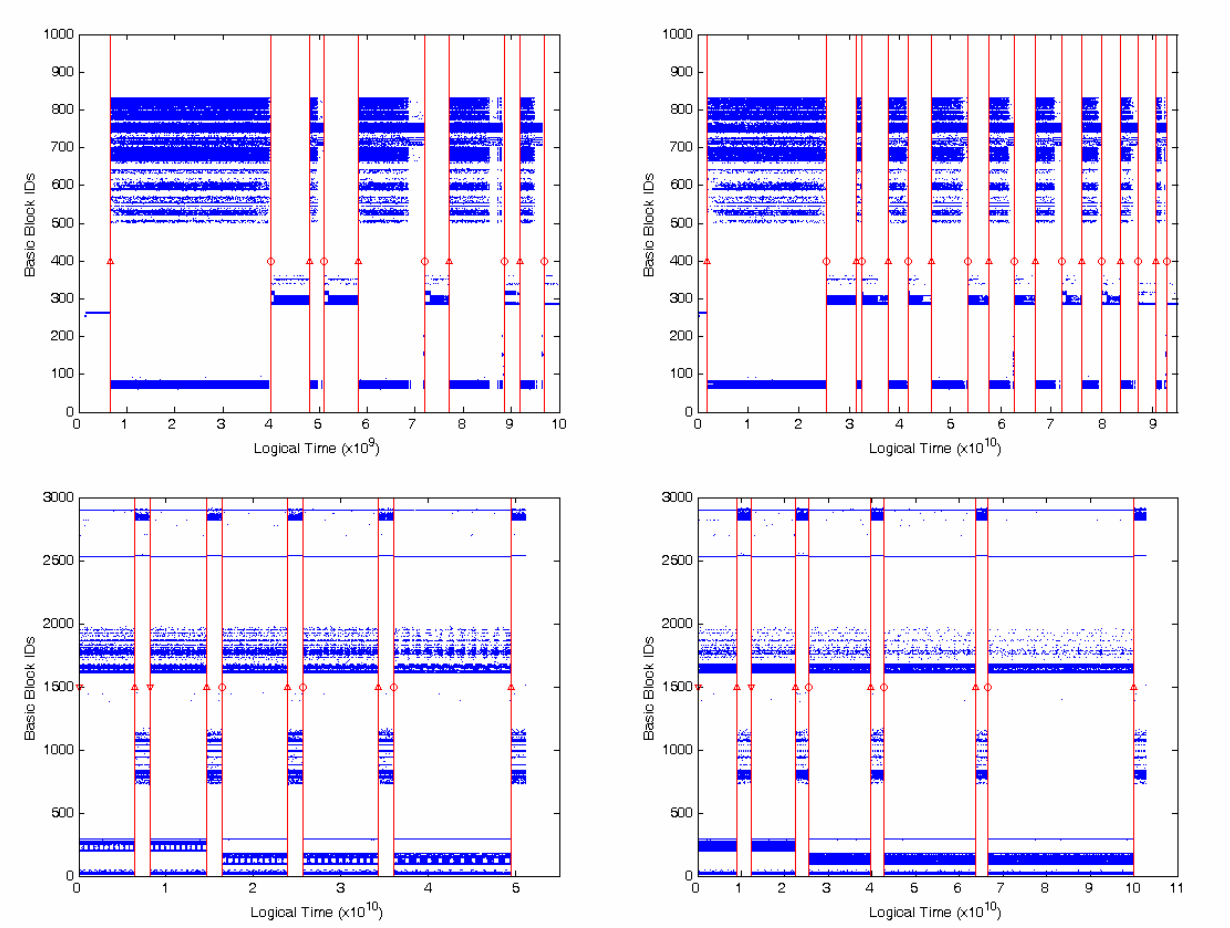


Figure 6: *mcf*'s BB profile with CBBT phase markings (upper panels). *gzip*'s BB profile with CBBT phase markings (lower panels). Self-trained CBBT markings (left panels) and cross-trained CBBT markings (right panels)

the six integer programs *bzip2, gap, gcc, gzip, mcf* and *vortex*. Four integer benchmarks have high phase complexity [11] (*gap, gcc, mcf* and *vortex*) and the remaining two have medium complexity (*gzip* and *bzip2*). The four floating-point programs are quite regular and can be classified as having low complexity. In general, floating-point programs do not have as much phase complexity as their integer counterparts, and the four selected programs seem to be representative of all the floating-point programs in the suite. Each program was compiled with the DEC-Alpha C compiler version 6.5 with the "-O3 -arch ev68" optimization flags. The integer programs further include feedback optimization. Our reference machine is an 833 MHz Alpha 21264B running Tru64 UNIX 5.1. Except for *gzip* and *bzip2*, for which we use two additional input sets (named graphic and program), all benchmarks are evaluated with two inputs, namely the SPEC provided train and reference inputs. The train inputs are used for self-trained phase detection; the reference inputs, as well as the two additional inputs for *gzip* and *bzip2*, are used for cross-trained phase detection.

3.2 CBBT-based Phase Detection

This section provides a quantitative assessment of the CBBTs' effectiveness in detecting phase change. We use two microarchitecture independent characteristics, namely BB worksets (BBWSs) and BB vectors (BBVs), as phase characteristics. BBWSs capture the unique BB IDs touched by a program executing for a given period of time. BBVs are similar but include the frequency with which each BB is touched. We use normalized BBVs where each entry is divided by the total frequency of all BBs in the vector. The vector dimension is kept constant and its size is determined by the program/input combination that touches the maximum number of distinct BBs (in our case this is *gcc/train*). Note that we do not suffer from the "curse of dimensionality" as we do not use these vectors for further classification or statistical inference. The microarchitecture independent BBVs and BBWSs have been shown to strongly correlate with microarchitecture dependent characteristics such as the IPC and the L1 cache miss rate [4].

Next, we need to decide the desired level of phase granularity. As different CBBTs correlate with different phase granularities per the formula given in Section 2.1 (Step 5), they allow phase behavior to be detected at various levels. This study evaluates phase behavior at a granularity that corresponds to 10M executed instructions.

The CBBT phase detector works as follows. It associates a unique phase characteristic (i.e., a BBWS or a BBV) with each CBBT. Every time this CBBT is encountered during execution, a program phase change is signaled, and the phase the

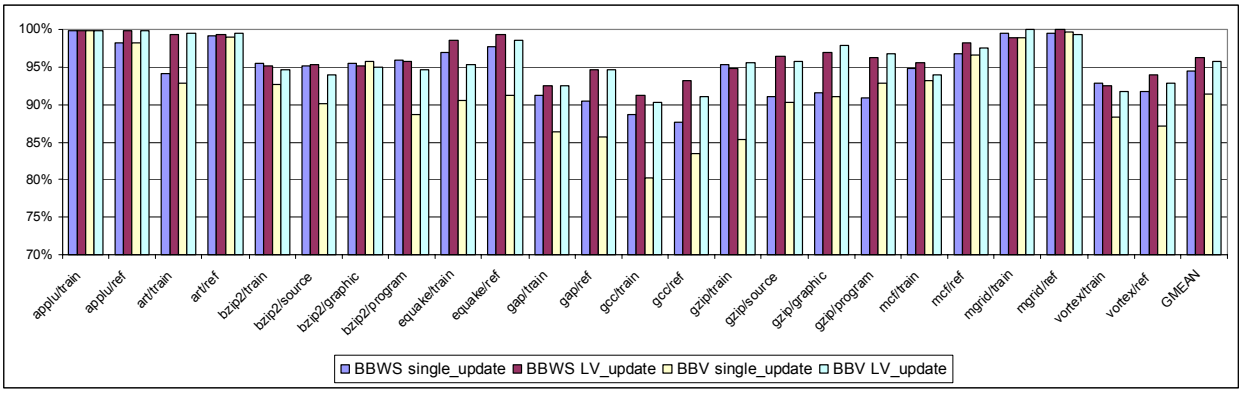


Figure 7: BB workset and BBV similarities as measured in percentage of their normalized forms

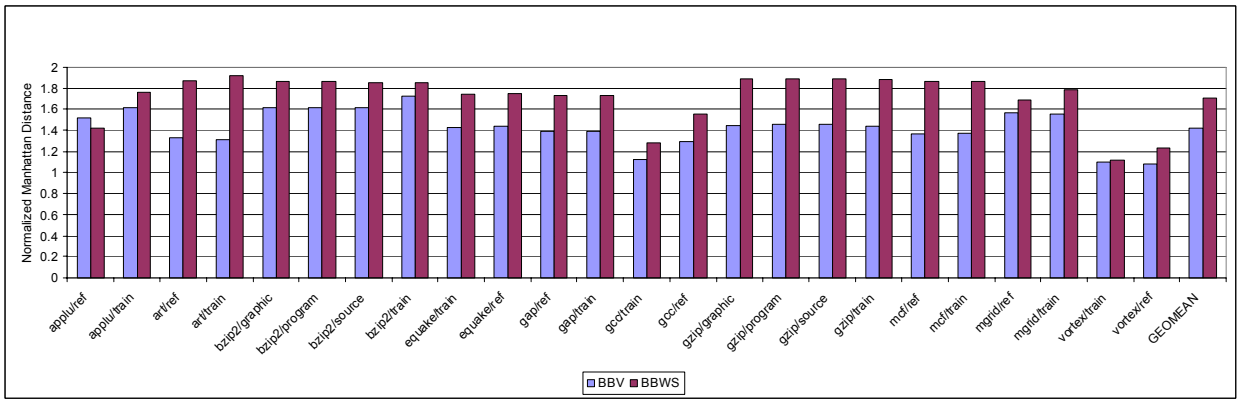


Figure 8: Average Manhattan distance between two CBBT phases. When calculating this value, we compare each CBBT phase to every other CBBT phase. The number of comparison is ${}^{n}C_2$ where n is the number of CBBT phases in a given program

CBBT transitions to is predicted to have the characteristics that are associated with the current CBBT. When a CBBT is encountered for the first time, the phase detector makes no prediction. Instead, it collects the phase characteristics to be associated with this CBBT. The phase characteristics are gathered from the time this CBBT is encountered until the next CBBT occurs, which signals the end of the phase associated with the current CBBT.

When updating the characteristics associated with a CBBT, we use two policies—single update and last-value update. In the single update policy, the characteristics associated with the CBBT the first time it is seen is used to predict the characteristics of the phase this CBBT initiates every time it is encountered. In the last-value update policy, the characteristics associated with the CBBT that starts a phase are always updated at the end of the phase.

A good CBBT phase detector should be able to accurately foretell the characteristics of the phase that a CBBT initializes. That is, the phase a CBBT transitions to should have similar characteristics to the characteristics associated with this CBBT. Figure 7 shows the quality of the CBBT phase detector on 24 benchmark/input combinations in terms of BBWS and BBV similarities. The similarity of BBVs and BBWSs for a given phase is measured in terms of the Manhattan distance between the two BBVs and BBWSs. Because we use normalized vectors, the Manhattan distance gives the difference in percent. Note that the figure is not zero-based to improve readability. Generally, the last-value update performs well. It outperforms single update in all cases and achieves over 90% similarity with both metrics, indicating that the CBBT-based phase detector accurately predicts the phase characteristics.

Accurate phase prediction is, nevertheless, not the only property that we desire. We must also consider the quality of the detected phases. A good phase detector must sufficiently distinguish one phase from another. Figure 8 shows on average how distinct two detected CBBT phases are based on the Manhattan distance between the two phases. The maximum distinction occurs when the Manhattan distance is 2, i.e., the two phases have no overlapping code execution. With the CBBT phase detector, we find that the Manhattan distance between two different phases is at least 1, meaning that each of the two phases has over 50% non-overlapping code execution and is, therefore, quite distinct from the other.

3.3 Dynamic Cache Reconfiguration

This section investigates how CBBT-based phase detection can be used to dynamically reconfigure the L1 data cache size. This reconfiguration scheme exploits program phase behavior to reduce the cache size by turning off cache ways [1] in phases where a large L1 cache is not necessary. Doing so can result in considerable energy saving without much loss in performance. We evaluate the cache reconfiguration with CBBTs corresponding to a granularity of 10M executed instructions. We follow the implementation setup given by Shen et al. [15].

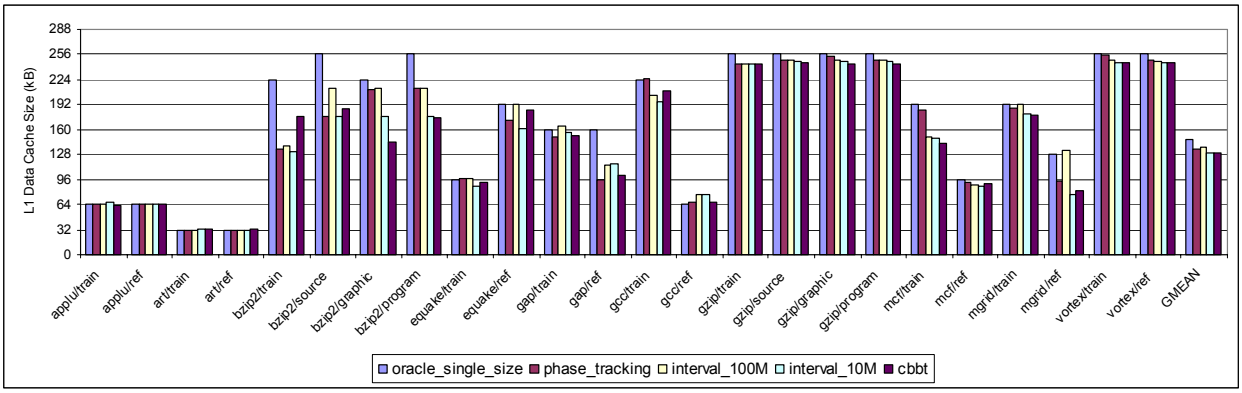


Figure 9: Adaptive cache resizing

There are eight L1 cache sizes that can be selected while the program executes. They range from the smallest size of 32 kB to the largest size of 256 kB in increments of 32 kB. For each cache size, the number of cache lines and the block size stay constant (512 lines and 64 bytes). Increasing (or decreasing) the cache size is achieved by varying the degree of associativity from 1 (direct-mapped) to 8. We use ATOM to model and simulate these cache configurations as well as the dynamic reconfiguration logic. We compare CBBT-based cache reconfiguration with three idealized techniques. For each benchmark/input combination, each technique tries to maintain a miss rate within 5% of the 256 kB cache miss rate while reducing the active cache size as much as possible.

Adaptive cache resizing with CBBTs works as follows. When a CBBT is encountered for the first time, it finds the optimal cache size for the phase it initiates by performing a binary search for the best cache size over the next four 10k instruction intervals of the phase. The miss rate of the 256 kB cache is determined in the first interval. The next interval is tried with half of this size, i.e., 128 kB. If the miss rate increase is below the 5% threshold, 64 kB is tested next. Otherwise 192 kB is tried. This process repeats one more time to detect the best minimal cache size. Once this size has been determined for a newly encountered CBBT, it is associated with that CBBT, and will be applied when this CBBT is encountered again. However, when this CBBT is later encountered and this cache size results in a miss rate difference (increase or decrease) of more than 5% when compared to the miss rate of the previous instance of this CBBT phase, the cache size will be reevaluated following the binary search steps given above. (This is similar to the last-value update policy in the previous section).

The three idealized techniques—single-size oracle, phase tracking, and interval based—work as follows. The single-size oracle uses the best single cache size that, when used throughout the entire program execution, will not result in a miss rate exceeding the 5% bound. Phase tracking implements an idealized version of Sherwood's phase tracker [19], where phase prediction is assumed to be 100% correct. This is a BBV-based technique that uses BBV signatures gathered for every 10M instruction execution interval and a threshold to recognize a phase change. Instead of using a compressed BBV signature, we use the full BBV signature and investigate thresholds of 10%, 50%, and 80%. On average, we did not find that these various thresholds yielded substantially different results. Hence, we picked the threshold that performs best for this study (i.e., 10%, which is the same as in the original phase tracker paper). The ideal interval-based technique is straightforward. The program execution is chopped up into fixed length windows of 10M and 100M committed instructions. An oracle determines the best cache size for each interval.

Figure 9 shows the effective cache size for the 24 benchmark/input combinations. Except for the programs *applu* and *art*, the phase-based cache resizing techniques—phase tracking, fixed interval, and CBBT—reduce the effective cache size below that of the single-size oracle. On average, we find that the realizable CBBT scheme performs as well as the two idealized phase-based schemes. It effectively reduces the cache size by half. Compared to the single-size oracle scheme, the CBBT scheme is able to exploit phase information and thus further reduces the effective cache size from about 150 kB to 128 kB, which is a 15% reduction. It performs almost as well as the 10M fixed interval and slightly better than the idealized phase tracker, which indicates that the phase behavior to be exploited for dynamic cache reconfiguration has about this level of granularity. In general, the CBBT approach performs well because it generally stays synchronized with the miss rate characteristics, whereas an interval-based oracle scheme, e.g., 10M, can be "out of sync", i.e., a 10M interval may straddle both a high and a low miss rate region, so the cache size must be conservatively chosen to accommodate the high miss rate region to stay within the 5% bound.

As a final note, the drawback of using the cache miss rate to evaluate the effectiveness of CBBT phase detection is that it does not explicitly spell out the run-time or the energy consumption. Nevertheless, we opted to use this metric for simplicity and reproducibility. For each program/input combination, we can take measurements throughout the full execution without resorting to sampling. This is in contrast to an evaluation with a more elaborated cache model and using cycle-accurate simulation to measure time and energy. Although the latter approach is theoretically more sound, in practice, it can suffer from measurement inaccuracies because of inadequate modeling, unfaithful simulation, and sampling issues.

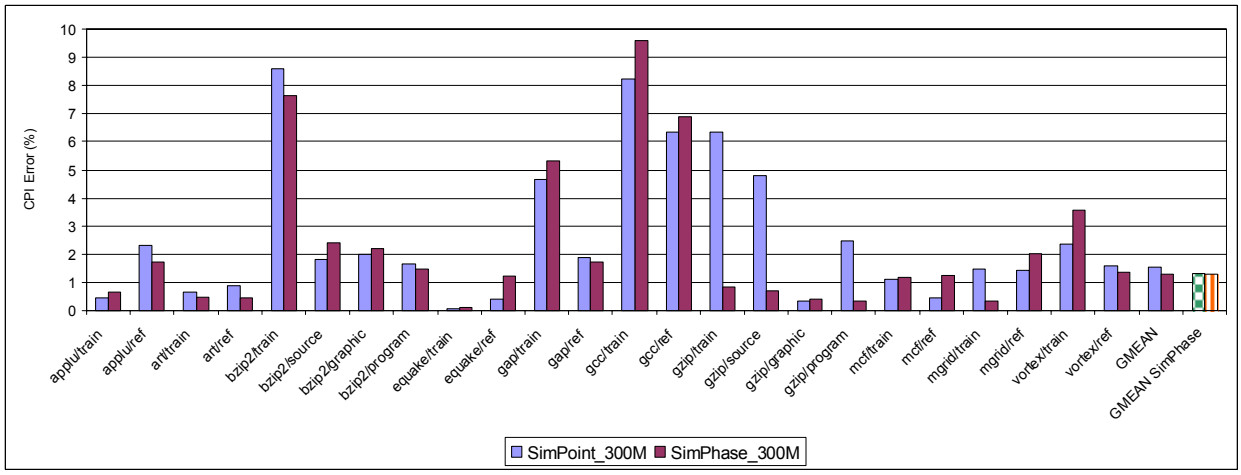


Figure 10: CPI error of SimPhase and SimPoint. The two rightmost bars indicate the GMEAN CPI error for SimPhase with self-trained CBBTs (left bar) and cross-trained CBBTs (right bar)

3.4 SimPhase and SimPoint

This section demonstrates an approach to picking architectural simulation points using a program's CBBTs, which we call SimPhase. We compare SimPhase with SimPoint [18], a popular tool that is widely used for this purpose. We use version 3.2 of SimPoint [7], released in February 2006.

To run SimPoint, a user specifies two parameters, an interval size and the number of simulation points. The latter is specified through the parameter maxK, which is essentially the number of distinct phases to be discovered by SimPoint. maxK times the interval size is the maximum number of instructions to simulate. In this study, we limit the number of simulated instructions to 300 million and use the interval_size/maxK combination of 10M/30. This combination has been shown to provide good accuracy while staying within a reasonable simulation time [6].

SimPoint gathers BBV profiles for each non-overlapping interval throughout the execution of a specific program/input combination. Then, a k-mean clustering algorithm groups similar BBVs from each interval and forms maxK BBV clusters. After that, a representative BBV from each cluster is chosen. The start time of the interval that generates the representative BBV becomes the simulation point for that cluster, i.e., for that phase. The SimPoint algorithm picks a representative BBV that is closet to the centroid of the cluster.

Our SimPhase approach incorporates, in a sense, the reverse process of SimPoint. In SimPhase, "clustering" is performed first. Then, in going from one instance of a "cluster" to another instance of the same "cluster", a similarity metric between the two instances is calculated and the simulation point is picked accordingly. To allow a direct comparison with SimPoint, SimPhase also uses BBVs as a phase metric.

SimPhase starts by generating CBBTs that divide the program execution into distinct regions of code (see Figure 6 for an example of large-scale phase boundary marks). We can think of this code division as performing a form of clustering. A program phase is marked by one CBBT at the start and another CBBT at the end. Once these boundaries are determined for a given program, they are reused across all inputs for this program. A natural place for SimPhase to pick a simulation point is at the midpoint of a phase (recall that SimPoint also tries to pick simulation points at the clusters' centroids).

Once we have obtained the CBBT markings, we rerun the program with the same or a different input to determine simulation points for a given program/input combination. For the first instance of each CBBT, a BBV profile is calculated for this CBBT. The phase extends from this CBBT to the next. A simulation point is selected midway between these two marks. When the same CBBT is encountered again, SimPhase calculates a new BBV profile and compares it to the most recent BBV for this CBBT. If the two BBVs differ by more than a preset threshold, there is, indeed, a significant change in BBVs detected, and, hence, another simulation point is picked. Once this process ends, we divide the number of simulated instructions by the number of simulation points selected to obtain the number of instructions to simulate for each simulation point. This last number is analogous to the interval size in SimPoint. In this study, we cap the number of simulated instructions to the same as in the SimPoint case, i.e., 300M instructions. However, depending on how the clusters are generated, SimPoint may choose to simulate fewer instructions, whereas SimPhase will always simulate the full 300M instructions. The threshold used to gauge BBV similarities for SimPhase is 20%. This relatively low threshold allows more simulation points to be picked for the limit of 300M simulated instructions. Note that the SimPoint parameters chosen for this case also result in a large number of clusters (maxK=30) that will yield up to 30 simulation points to be simulated with 10M instructions each.

Our evaluation platform for SimPoint and SimPhase is SimpleScalar [3], a popular cycle-accurate simulator that models an out-of-order superscalar machine with varying configurations. We use SimpleScalar version 3 with the machine configuration given in Table 1 to measure the CPI errors resulting from the use of both methods compared to full simulation runs. The simulation points from both SimPhase and SimPoint are weighed according to the phases they represent before performing the final CPI calculation.

Figure 10 shows the CPI errors for the 24 benchmark/input combinations. Although there are some variations in a few programs, on average, the error rates for both approaches are

comparable. With the number of simulated instructions limited to 300M, the geometric mean CPI errors for SimPoint and SimPhase are 1.56% and 1.29%, respectively. This should not come as a surprise as both SimPhase and SimPoint use the same metric to represent program phases, i.e., BBVs. So, eventually, given decent means of grouping representative BBVs, the measured CPI errors depend on how strongly an architecture independent characteristic such as a BBV correlates with an architecture dependent characteristic like CPI.

Table 1: Baseline machine for comparing SimPhase and SimPoint

Parameter	Values
Issue width	4-way
Branch predictor	4K combined
ROB entries	32
LSQ entries	16
Int/FP ALUs	2 each
Mult/Div units	1 each
L1 data cache	32 kB, 2-way
L1 hit latency	1 cycle
L2 cache	256 kB, 4-way
L2 hit latency	10 cycles
Memory latency	150

SimPhase offers an interesting feature, however. The phase boundaries marked by the CBBTs using the train inputs can also be used for other inputs. With SimPoint, every time there is a change in program inputs, one needs to perform new k-mean clustering with a new BBV profile. If we revisit the results shown in Figure 10 and compare the SimPhase geometric mean CPI errors (the two rightmost bars in Figure 10) for the runs with the train inputs (self-trained CBBT) to those with the reference inputs (cross-trained CBBT), we find that there is no significant difference between the two average CPI errors, which are 1.31% and 1.28%. In fact, the cross-trained CBBTs seem to do a little better. This is because there are programs, notably *gcc* and *gap*, whose phase behavior is more subtle when run with the train inputs; this phase behavior becomes more discernible when run with the reference inputs.

4. Related Work

Sherwood et al. [16] exploit the time varying behavior of programs to reduce simulation time. They propose off-line algorithms to capture program phases. Their first method finds a representative code segment that has a similar BB distribution as the entire program [17]. Their second approach is based on clustering and uses the k-mean algorithm [18]. Both schemes are strictly concerned with off-line phase detection. However, the authors later also propose a hardware-based method for on-line phase detection and prediction [19]. They approximate their off-line algorithms through the on-line discovery of key parameters. For example, they approximate the BB length by the number of executed instructions between branches. Sherwood et al. further propose a phase prediction scheme, which is later enhanced by Lau et al. [10].

Dhodapkar and Smith [5] improve upon the algorithm proposed by Balasubramonian et al. [2] by using working set signatures to detect phase changes on-line and employing this information to guide hardware re-tuning. They weigh the importance of each working set segment equally regardless of frequency. This is in contrast to the hardware-based detector proposed by Sherwood et al., which takes into account the frequency of the executed BBs in a working set.

Shen et al. [15] propose a software-based phase prediction scheme that gathers phase knowledge through off-line profiling and incorporate this information into the program binary. Their off-line profiling requires two traces. As their phase behavior is based on reuse distance [12], they need a reuse distance trace along with a BB execution trace. The information obtained from profiling results in a form of phase markings, which is used to form hierarchical phase markers. Such markers are constructed through grammar compression using SEQUITUR, and are incorporated into the original binary through binary rewriting for phase prediction.

Lau et al. [9] present a profile-based approach to identify code locations that correlate with phase changes. Such locations can serve as software phase markers independent of program inputs. They demonstrate how to obtain the phase markers through the formation of a Hierarchical Call-Loop graph. This graph is basically an abstraction of the full control flow graph where only loops and procedure calls are represented and contains information about each loop and procedure execution.

Lau et al.'s software phase markers may also be used to pick simulation points for a program's binary across different inputs. This feature is comparable to CBBT-based phase boundary markings. Perelman et al.'s work [14] goes further by providing such markers across different ISAs to generate cross binary simulation points. The CBBT approach has the potential to perform such cross ISAs markings as well. As seen in Section 2.2, phase boundaries marked by CBBTs can be directly associated with high-level source code.

5. Summary

This paper presents a light-weight phase detection mechanism that identifies program phase change points via Critical Basic Block Transitions (CBBTs). Such CBBTs are identified using the Miss-Triggered Phase Detection (MTPD) algorithm. When a series of misses to an infinite size basic block ID cache is encountered while traversing a stream of BB IDs, program execution may have reached a phase change point. The MTPD algorithm employs simple program heuristics to determine whether such a point is likely to be an actual phase change and, if so, records the corresponding CBBT. We show how mapping a CBBT back to the source level allows us to discern program phase behavior in the source code. We evaluate CBBTs for guiding dynamic cache resizing and picking representative architectural simulation points, respectively. Our CBBT-based scheme performs well on both accounts. In dynamic cache reconfiguration, it is able to effectively reduce the cache size to 50% of the maximum cache size on our benchmarks without increasing the miss rate by more than 5%, a performance that is comparable to idealized adaptive cache resizing schemes. In picking appropriate architectural simulation points, our CBBT-based approach produces an average

CPI error that is comparable to the widely used SimPoint method. In addition, the CBBT approach is as effective on the self-trained inputs as on the cross-trained input.

6. References

[1] D. H. Albonesi, *Selective Cache Ways: On-Demand Cache Resource Allocation*, *32nd International Symposium on Microarchitecture*, 1999, pp. 248-259.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, *Memory Hierachy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures*, *Proceedings of the 33rd International Symposium on Microarchitecture*, Monterey, California, 2000.

[3] D. Burger and T. Austin, *The SimpleScalar Tool Set, Version 2.0*, University of Wisconsin-Madison Computer Sciences, 1997.

[4] A. S. Dhodapkar and J. E. Smith, *Comparing Program Phase Detection Techniques*, *Proceedings of the International Symposium on Microarchitecture*, 2003.

[5] A. S. Dhodapkar and J. E. Smith, *Managing Multi-Configuration Hardware via Dynamic Working-Set Analysis*, *Proceedings of the International Symposium on Computer Architecture*, Anchorage, Alaska, 2002.

[6] G. Hamerly, E. Perelman, J. Lau and B. Calder, *SimPoint 3.0: Faster and More Flexible Program Analysis,* Journal of Instruction Level Parallelism (2005).

[7] http://www.cse.ucsd.edu/~calder/simpoint/.

[8] R. E. Kessler, *The Alpha 21264 microprocessor*, *IEEE Micro*, 1999, pp. 24-36.

[9] J. Lau, E. Perelman and B. Calder, *Selecting Software Phase Markers with Code Structure Analysis*, *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[10] J. Lau, S. Schoenmackers and B. Calder, *Transition Phase Classification and Prediction*, *11th International Symposium on High Performance Computer Architecture*, 2005.

[11] T. Li and C. B. Cho, *Complexity-based Program Phase Analysis and Classification*, *Proceedings of the 2006 International Conference on Parallel Architectures and Compilation Techniques*, Seattle, Washington, 2006.

[12] R. L. Mattson, J. Gecsei, D. Slutz and I. L. Traiger, *Evaluation Techniques for Storage Hierarchies*, IBM Systems Journal (1970), pp. 78-177.

[13] S. McFarling, *Combining Branch Predictors*, *Tech. Rep. WRL Technical Note TN-36*, Digital Equipment Corporation - Western Research Laboratory, 1993.

[14] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly and B. Calder, *Cross Binary Simulation Points, Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2007.

[15] X. Shen, Y. Zhong and C. Ding, *Locality Phase Prediction*, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, 2004.

[16] T. Sherwood and B. Calder, *Time Varying Behavior of Programs*, *Technical Report UCSD-CS99-630*, UC San Diego, 1999.

[17] 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*, 2001, pp. 3-14.

[18] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, *Automatically Characterizing Large Scale Program Behavior*, *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45-57.

[19] T. Sherwood, S. Sair and B. Calder, *Phase Tracking and Prediction*, *Proceedings of International Symposium on Computer Architecture*, San Diego, California, 2003.

[20] J. E. Smith, *A Study of Branch Prediction Strategies*, *Proc. 8th Int. Sym. on Computer Architecture*, 1981, pp. 135-148.

[21] A. Srivastava and A. Eustace, *ATOM: A System for Building Customized Program Analysis Tools*, *Proceedings of SIGPLAN,* 1994.