Phase based adaptive branch predictor: Seeing the forest for the trees

Karthik Jayaraman Vivek Shrivastava Brian Pellin Martin Hock
Department of Computer Sciences, University of Wisconsin-Madsion
{karthikj, viveks, bpellin, mhock}@cs.wisc.edu

Mikko H. Lipasti
Electrical and Computer Engineering, University of Wisconsin-Madison
mikko@engr.wisc.edu

ABSTRACT

Understanding and exploiting the dynamic behavior of programs is key to improving performance beyond what is possible using static techniques. We present an implementation of a phase tracking algorithm and show that it can be successfully applied to existing SPEC2000 benchmarks, many of which show a consistent phase behavior.

We then show the application of phase information to optimize branch prediction. Our results indicate a reduction of up to 44.35% in branch predicition misses, using simpler predictors based on per phase information. We also get an improvement of up to 4.70% in IPC for our target benchmarks.

1. INTRODUCTION

Previous research has shown that understanding and exploiting the dynamic behavior of programs is key to improving performance beyond what is possible using static techniques. A program's control flow is one such aspect of its dynamic behavior that has been extensively used to optimize program execution, typically by identifying the various *phases* in which a program executes.

> A *phase* is defined as a period of execution during which a given measured program metric is relatively stable.

Phases can both be tracked (to detect what phase the execution is in) and predicted (to determine when another phase is about to be entered). The phase behavior seen in any program metric is directly a function of the way the code is being executed. If this behavior can be captured accurately during run time, then it can guide many optimizations specific to the phase defining metric. For example, if our program metric is based on memory references, then we can resize our cache size across various phases. If we are able to efficiently determine the phases and apply the aforementioned optimization, we can improve cache control and wire delay as only a part of cache is active as per the requirements of that particular phase. The most important attribute of a phase is that it often captures many different metrics.

In this paper we analyze the phase tracking and phase prediction schemes. We also look into the underlying predictor models used by such phase prediction schemes and try to assess the implications of varied predictor models.

Our most important contribution is the design and implementation of *phase based adaptive branch predictor*. Using the ability to both determine phases and predict the next phase, we use that information to control reconfigurable hardware. Specifically, we determine, at runtime, the best branch predictor to use for each detected phase. Our results show a reduction of up to 44.35% in branch prediction misses which can lead to reduction in power consumed by the processor in executing mis-speculated instructions and flushing the pipeline thereafter. Further, with the increase in the number of pipeline stages in the modern processors, the impact of reduction in branch prediction misses on power consumption will become more profound.

The remainder of this paper is organized as follows. In Section 2 we discuss the implementation of our phase tracker. Section 4 covers our methodology for applying phase information to applications. Section 5 discusses the applications of branch predictors. Our experimental results are presented in Section 6, and Section 7 discusses the feasibility of implementing our techniques in hardware. Section 8 surveys other work in phase tracking and dynamic hardware configuration. Finally, we draw conclusions in Section 9.

2. PHASE TRACKING

In this section we discuss how we track executing programs, and classify sections of the program into phases. Our phase tracking is an implementation of Sherwood et al's phase tracker [15].

There are two stages to phase tracking. The first stage collects profiling data as instructions execute. The second stage takes a large scale view at the program. After one profiling period, the data from the first stage is taken and classified by the second stage.

2.1 Phase Profiling

When a branch instruction executes, its PC and the number of instructions since the last branch are tracked. This captures information about the basic blocks, as well as the weight of the block in terms of the number of instructions executed. This approach is different from the phase capture technique used by Dhodapkar and Smith [7], where, in place of basic block information, they keep track of lines that have been touched in the instruction cache.

When a branch is detected, its PC is hashed into one of the buckets in the Accumulator. The value in that bucket is then incremented by the number of instructions since the last branch. Each entry in the accumulator is a 24-bit counter. In

all of our experiments it was rare to see any entry saturate during the ten million instruction window.

As with any hashing scheme, there are two important issues. One is the number of buckets to include in the accumulator. On one hand, if too few buckets are used aliases can occur causing two different phases to have similar footprints and on the other hand, too many buckets are a waste of memory. Sherwood et al [15] have shown that 32 buckets are sufficient for most workloads.

Another important issue is the choice of hash function. Sherwood suggested using a random projection[14] based hash function. In this function bits are randomly selected from the PC and used to form the hash. We found however, that we got as good if not better results from a simpler function. We used the function:

$$hash = PC \mod \textit{Number of Buckets}$$

There are three important fallouts of using such an accumulator structure. Firstly, since only thirty-two 24-bit counters are required, the hardware cost is relatively cheap. Secondly, the cost to update this structure is fairly low. Though it has to be accessed on every branch instruction, such accesses can be performed in parallel with other operations like branch prediction. Thirdly, this structure is largely architecture independent and hence can be applied to any architecture with a clear notion of branches.

2.2 Accumulator to Phase Footprint

After a profiling period has elapsed, the data in the accumulator into a *phase footprint*. Since a fuzzy form of equivalence is defined between the phase footprints, the full precision of a 24-bit value for each accumulator is not required. Hence a footprint vector is formed from the accumulator entries by reducing the precision using the following equation:

$$(bucket[i] \times \text{Number of Buckets}) / (size\ of\ profiling\ period)$$

The above equation is implemented using the left and the right shift operators for multiplication and division respectively. For our parameters this reduces each bucket to size 6-bits, for a total of 24 bytes to store a footprint.

2.3 Classifying Footprints

A *Past Footprint Table* is used to keep track of past phases. The table has a finite number of entries, and for reasons discussed in [15]
20 entries is sufficient for most programs. Each unique phase footprint is stored in the table. This helps us determine if a new phase is unique or not. Each time we see a new phase, a new entry is allocated in the past footprint table.

Since phase footprints are not precise indicators of uniqueness, a fuzzy equality is defined between phases. To compare two phases, the Manhattan distance[1] between the two footprints is computed. If this distance is within a certain threshold, then the phases are considered to be equal.

Though this operation is slightly expensive, this cost can be justified by the low frequency at which it occurs.

2.4 The Whole Process

The first part of phase tracking is accumulating runtime data. For every branch instruction, the number of instructions since the last branch are stored in the accumulator.

After 10 million instructions the data in the accumulator is converted into a phase footprint. The footprint is then compared with every phase in the past footprint table. If the phase matches no entry in the table, then the footprint is inserted into the table as a new phase.

Figure 1 shows an example of phased behavior in a program. The top entry represents the data our accumulator has collected at the end of each 10 million instruction period. The shade of point in y axis is scaled based on the accumulator count. One can see that repetitions in the phase data we collect corresponds with the repeated behavior in other performance metrics. This gives some intuition that we are collecting the correct information, and this will be quantified further when we discuss our applications.

3. PHASE PREDICTION

Phase tracking allows us to determine what phase of execution just happened. However, in order to reconfigure hardware to target instructions that are currently executing, we need to estimate with high accuracy the phase we are currently in. We compared three different phase predictors: last seen, RLE Markov, and perceptron.

3.1 Last seen

The simplest of phase predictors simply predicts that the next phase is the same as the last one.

3.2 RLE Markov

We have implemented the RLE Markov predictor of [15]. We have modified their hash function slightly. We combine a 27 bit saturating counter representing the number of times a phase is seen combined with a 5 bit counter representing what phase it was. The paper [15] indicates that 20 phases suffice for 80% of instructions.

3.3 Perceptron

We have also implemented a perceptron predictor. We were guided initially by the perceptron branch predictor of [11]. Unlike a branch predictor, which must only differentiate between taken and not taken branches, we have more than two possible outcomes: we must differentiate between several different phases.

3.4 Results

We found that after performing two key adjustments to the phase tracker, last seen proved to be a very accurate predictor, making the added complexity of RLE Markov or perceptrons unnecessary. For example, on gcc, last seen was 96% accurate, compared to 94% accurate for RLE Markov. So we used the Last seen predictor with the following modifications. First, we increased the minimum Manhattan distance necessary to be considered a different phase. This reduced the number of phases drastically. Second, we performed an averaging step when a footprint is classified as a given phase. Instead of keeping the first footprint as defining a phase, we allow a phase to drift toward its most natural center. We found that a weight of 1/16 of the new footprint combined with 15/16 of the original footprint worked well.

4. PHASE BASED OPTIMIZATIONS

> Generality is a special case needed. Special Cases are more generally needed.
> – *Kent M. Pitman*

[1] The sum of element-wise differences.

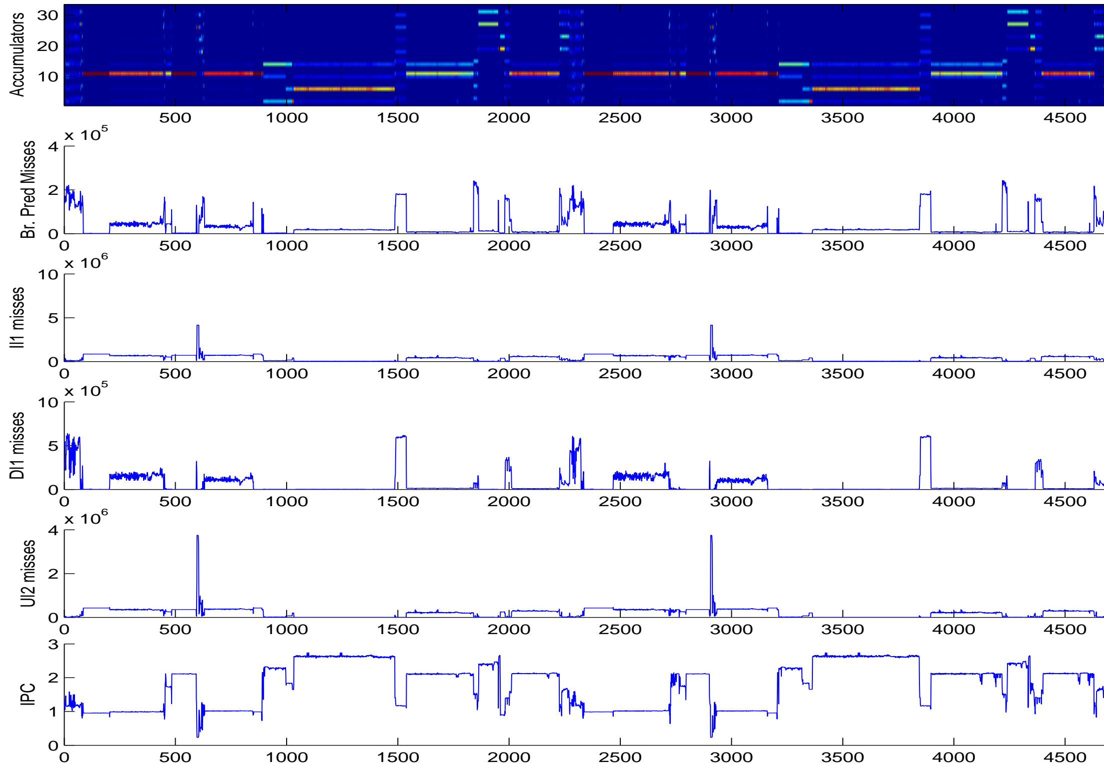


Figure 1: Phased Behavior of gcc ***This graph shows the accumulator data that our phase tracker collects, as well as several performance metrics collected in SimpleScalar. Each point of the x-axis is at a scale of 10 million instructions.***

This quote succinctly summarizes the motivation for our phase based optimizations presented in this section. Microprocessors are designed to provide good average performance over a variety of workloads. This can lead to inefficiencies in both power and performance for individual program and during individual phases within the same program [7]. However, microarchitectures with multi configuration units(e.g. caches, predictors, instruction windows) are able to adapt dynamically to program behavior. This is typically done by **tuning** when a phase change is detected or predicted-i.e. sequencing through a series of trial configurations and selecting the best.

Due to continuous progress in microarchitecture and chip technology, the trade-offs involving performance, power, and complexity become increasingly difficult and warrant increasingly sophisticated optimization methods. One promising optimization approach is to configure microarchitecture features dynamically to adapt to changing program characteristics ([7],[1],[4],[2],[12],[17],[6], [10]). As the program runs, it passes through phases of execution where its performance characteristics and its hardware requirements may vary [14],[18]. Performance and/or power consumption can be optimized *on-the-fly* if significant phase changes can be detected and dynamic microarchitecture configurations can be invoked in response to the phase changes.

In most of the proposed implementations, configurable units are designed to have a number of fixed configurations e.g. four different cache sizes or a configurable cache whose portions can be turned off dynamically. Then, the runtime configuration algorithm selects from one of the multiple available configurations.

The approach taken here is primarily directed towards development of *dynamic configuration algorithms* to improve performance (power, IPC). The goal is to use the underlying phase information provided by the phase predictor to select the best configuration for the next phase. So basically a change in phase acts a trigger for the configuration algorithm to tune the configuration for the new phase.

4.1 Dynamically Configurable Hardware

A number of proposals have been made for adaptive hardware mechanisms targeted at performance/power optimization. A few examples of configurable hardware is as follows:

Configurable Caches and TLBs - sizes and associativity are adjusted in response to program referencing behavior [17],[2].

Allocation of memory hierarchy resources -cache memory resources are divided among levels in the cache hierarchy [4] or configured for other uses, e.g instruction reuse.

Configurable Branch Predictor -the length of the history for a single predictors can be varied [12] or different predictors can be selected for different phases.

Configurable Instruction Windows -sections of the issue window are disabled when there is low instruction level parallelism [6].

Configurable Pipelines -portions of clustered microarchitectures can be disabled, or a pipeline can vary between inorder, out-of-order, and pipeline gating [8].

Of course, these various methods can be combined together in one system. Balakrisan et al [3] present the design of a completely adaptive microarchitecture. Huang et al [9], describe a general framework and algorithms that are intended to deal with processors containing several configurable units.

4.2 Dynamic Reconfiguration Algorithm using Phase information

Methods for controlling multi configuration hardware generally involve a form of feedback where some performance characteristics (e.g. (IPC) or miss rate) is measured for a fixed interval (also called a "window", "period", "step", etc.) and reconfiguration decisions are based on current and past measurements.

The *phase predictor* provides the trigger by providing the next phase information to the algorithm. If there has been a phase change, the *dynamic reconfiguration algorithm* starts a *tuning* sequence, where different configurations are tried and the best configuration is chosen for the newly started phase. The reconfiguration algorithm is run only once per phase, and the best configuration is stored in a special data structure. If the same phase reoccurs, the best configuration for the reoccurring phase can be identified by indexing into the *statistics table* without undergoing the *tuning* sequence again. This is a good optimization as program behavior remains same within a phase and saves the overhead of tuning sequence for the recurring phases.

5. PHASE-BASED ADAPTIVE BRANCH PREDICTORS

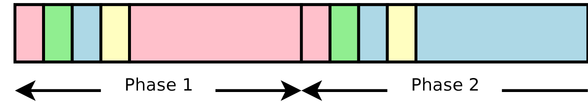
Previous work in adaptive branch predictors has been in combining branch predictors [13]. In this case, there is a meta-level predictor, which chooses from among a set of predictors(typically 2 in number), one which has the highest accuracy at any point of time. This choice is made at every branch instruction. Here we apply phase-based optimizations to branch predictor selection.

Phase-based Branch predictors. The performance of branch predictors is not uniform across the entire execution of the program. In this particular application, we try to exploit the phase behavior of programs to choose an appropriate branch predictor for one entire phase. We try to choose a branch predictor that is best for a phase by using the fact that program execution is uniform throughout a phase. Once a branch predictor is chosen for a phase, it would be used for the rest of the phase and any other recurrences of the phase. This is achieved in the following way:

- A set of branch predictors, with sizes smaller or comparable to that of a baseline two level predictor is chosen.
- At the beginning of each new phase, the branch predictors from the selected set are used, one by one for a profiling period.
- The number of branch predictor misses and the miss rate are stored for each predictor profiled.
- After all the predictors are profiled, the best predictor is chosen based on the following criterion:

  Choose the predictor with the least miss rate as the best. If two predictors differ in miss rates by less than 1% then chose the simpler one.

So the number of instructions executed before a stable predictor is chosen is $profiling\ period \times number\ of\ predictors$. The following figure summarizes the technique used:



The rationale behind this rule is that if the number of misses that a predictor has is lesser, then its accuracy is higher. Moreover since the accuracy is higher, the number of misspeculated instructions are lesser, and consequently the power consumption is also reduced. But, in case the predictors have close miss rates, then the simpler predictor is chosen, keeping in mind lower power consumption.

Once a branch predictor is chosen for a phase, it is associated which that phase. So, if that phase re-occurs, the profiling need not be repeated. The information about when the phase is going to change is provided by the phase predictor. Though there may be more misses during the profiling periods, these are out-balanced by the reduction in the number of misses for the rest of the phase.

6. EXPERIMENTATION

Here we present results of adapting the branch predictor to phase-specific behavior. *SimpleScalar v3.0d* [5] was modified to incorporate phase prediction and reconfiguration algorithms. We evaluated our scheme using SPEC2000 benchmarks.

6.1 Phase-based branch predictor

We have used a profiling period of 10 million instructions. We have tested our multiple branch predictor scheme using the following set of candidate predictors:

2 Level [1:1024:8] 2Level predictor [19] with 1 shift register in the first level and 1024 counters in the second level. It uses a shift register of width 8 bits. This is the default 2Level predictor configuration for SimpleScalar and the baseline predictor for our experiments.

Bimodal [1024] BiModal predictor [16] with 1024 counters.

2 Level [8:512:8] 2Level predictor with 8 shift register in the first level and 512 counters in the second level. It uses a shift register of width 8 bits.

2 Level [1:512:8] 2Level predictor with 1 shift register in the first level and 512 counters in the second level. It uses a shift register of width 8 bits.

We choose **2 Level [1:1024:8]** as the baseline predictor for our experiments as it has the maximum complexity (in terms of size) amongst all the candidate predictors. Please note that this is important because if our reconfiguration algorithm chooses any other candidate predictor (apart from the baseline) for any particular phase of the experimental workload, we can deduce that a simpler branch predictor outperforms a more complex one and hence shows the efficacy of phase behavior in reconfiguration.

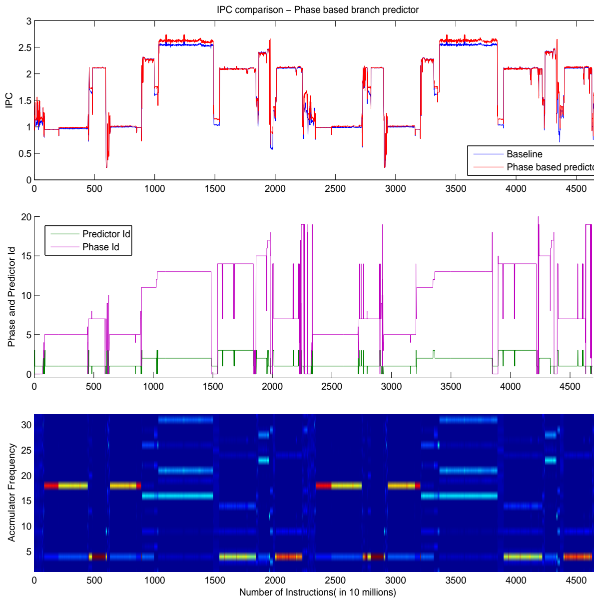


Figure 2: IPC comparison for gcc. ***This graph provides a comparison of IPC in the phase-based branch predictor scheme with the IPC in the baseline case for the* gcc *benchmark. The second part of the graph shows the phases predicted by the phase predictor and the corresponding branch predictor chosen for that phase. The third part of the graph shows the scaled values of the accumulators used in the phase tracking scheme.***

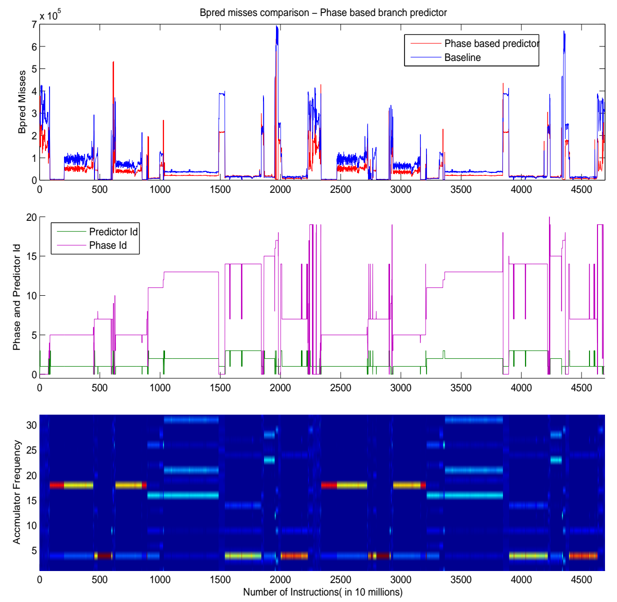


Figure 3: Branch predictor misses comparison for gcc. ***This graph provides a comparison of branch predictor misses in the phase-based branch predictor scheme with those in the baseline case for the* gcc *benchmark. The second part of the graph shows the phases predicted by the phase predictor and the corresponding branch predictor chosen for that phase. The third part of the graph shows the scaled values of the accumulators used in the phase tracking scheme.***

6.2 Results

We have tested our multiple branch predictor scheme on *gcc, mpr, vcf* integer benchmarks. We have evaluated the effect of our scheme on *Instructions Per Cycle (IPC) and Branch Prediction misses.* Figure 2 and 3 depict the IPC and branch prediction misses for our phase based *multiple branch predictor scheme* for the *gcc* benchmark. The gcc benchmark was run for *46 billion* instructions (till completion). The bottommost image in Figures 2,3 depicts the accumulator frequency for the gcc benchmark and clearly outlines the repeating phase behavior of gcc over long execution times. The middle portion of Figures 2,3 show the phase information as predicted by our phase predictor and also the corresponding branch predictor used in that phase.

The topmost graph in Figure 2 compares the *IPC* measurements for our scheme (in **red**) as compared to the baseline *2 Level [1:1024:8]* predictor (in **blue**)[2]. As evident from the graphs, our scheme consistently performs equal or better than the base scheme and switches between various predictors depending on the phase information. For example, in the window between 10billion and 15billion instructions, our scheme chooses a 2 Level predictor with a configuration of [8:512:8], that has **half** the number of counters (512) as compared to the baseline scheme (1024) but still achieves **3.2%** more IPC than the base case.

The topmost graph in Figure 3 compares the *branch prediction misses* for our scheme (in **red**) as compared to the baseline *2 Level [1:1024:8]* predictor (in **blue**). As the graph shows, our scheme clearly outperforms the baseline scheme by dynamically choosing simpler predictors based on phase information provided by the phase predictor (as shown in the middle graph). Our scheme achieves **44.35%** less branch prediction misses than the baseline scheme. Similarly, Figures 4 and 5 provide a comparison of IPC and branch prediction misses for **mcf** benchmark.

The total number of instructions for mcf benchmark was *35billion*. As the Figure 5 shows, our scheme is able to achieve comprehensive reductions in branch prediction misses by alternating between a BiModal predictor with 1024 counters and 2 Level predictor with 512 counters and 8 first level shift registers. Again as in gcc, our scheme achieves **43.48%** less branch prediction misses as compared to the larger baseline predictor.

Figures 6 and 7 provide similar comparisons for the **vpr** benchmark. The total number of instructions executed were 3.14billion. As clear from the figures, vpr has only two phases and our scheme chooses BiModal with 1024 entries over the baseline predictor and achieves **28.98%** lower miss rate as compared to the baseline. So as this benchmark illustrates, our scheme is useful even in scenarios where there is not much change of phases.

So the above results clearly establish the efficacy of phase based multiple branch predictors in reducing branch prediction misses. Figure 8 summarizes the IPC improvements of our scheme over the baseline scheme. We perform better than

[2] Color codes have been used since it is otherwise difficult to show the improvement and the overall picture at the same time.

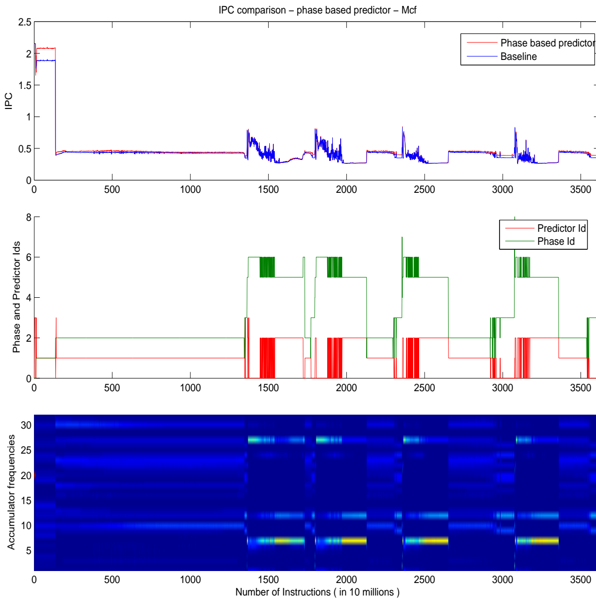


Figure 4: IPC comparison for mcf. ***This graph provides a comparison of IPC in the phase-based branch predictor scheme with the IPC in the baseline case for the* mcf *benchmark. The second part of the graph shows the phases predicted by the phase predictor and the corresponding branch predictor chosen for that phase. The third part of the graph shows the scaled values of the accumulators used in the phase tracking scheme.***

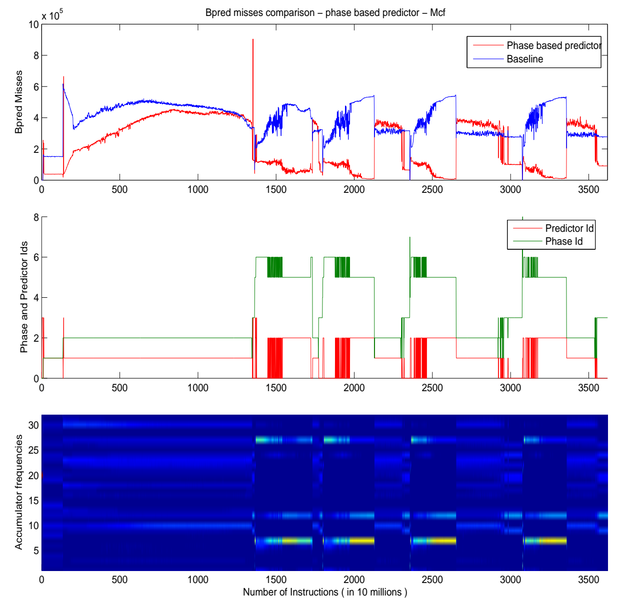


Figure 5: Branch predictor misses comparison for mcf. ***This graph provides a comparison of branch predictor misses in the phase-based branch predictor scheme with those in the baseline case for the* mcf *benchmark. The second part of the graph shows the phases predicted by the phase predictor and the corresponding branch predictor chosen for that phase. The third part of the graph shows the scaled values of the accumulators used in the phase tracking scheme.***

the baseline on all the tested benchmarks and the gain varies between (2.24%- 4.70%). The reduction in branch prediction misses achieved by our scheme is shown in Figure 9. As evident from the figure, we achieve comprehensive reductions in the branch prediction misses (28.98%-44.35%), which can definitely lead to power savings as discussed in Section 7.

7. DISCUSSION

The results from multiple branch predictor scheme show that we can comprehensively reduce the branch prediction misses by dynamically choosing the branch predictor configuration on the basis of phase information. Branch prediction misses can be directly correlated with the number of misspeculated instructions.

Our scheme also empirically illustrates that choosing a simple branch predictor is a good strategy for many phases. We used a baseline 2 Level predictor with 8 entries in the first level and 1024 entries in the second level and show that choosing a predictor of half the size of the baseline predictor (with just 512 entries in the second level), gives us considerable reductions in branch prediction misses for some phases.

Implementing our scheme in hardware is also feasible with minimal effort. We propose a scheme where the baseline predictor with the largest configuration can be instantiated in the processor and our dynamic reconfiguration algorithm can selectively switch on/off portions of the baseline predictor to convert it into simpler predictors depending on the phase requirements. We believe that our scheme provides a neat strategy for power savings in a modern out-of-order processor.

Power Savings. Modern processor have a large number of pipeline stages. Whenever a branch is mis-predicted, a large number of instructions are fetched from the wrong path, which then have to be flushed. So the penalty associated with fetching and later flushing these instructions is high in terms of power and hence makes the reduction in branch prediction misses an important factor for saving power in a modern processor. As the branch prediction misses reduce, the number of mis-speculated instructions in the pipeline will also reduce, thereby reducing the power consumption.

We also achieve marginal gains in IPC for our tested benchmarks. So we believe our scheme can provide good power savings without any loss in performance (IPC).

8. RELATED WORK

The phase tracking work by Sherwood et al [15] has formed the starting point of much of our work.

Dhodapkar and Smith [7] implemented a phase tracking scheme. In their scheme, phases are identified by which cache lines are touched in an instruction window.

Albonesi et al [3] have presented the design of an adaptive microprocessor based on feedback that leads to considerable power savings on their benchmarks reported. Also, they make use of Smith's [7] phase tracker mechanism to identify phase changes to trigger their adaptations. However we make use of the phase prediction mechanism proposed by Sherwood et al [15].

Huang et al [9] proposed positional adaptations, which uses program structure to identify major program phases. Specifi-

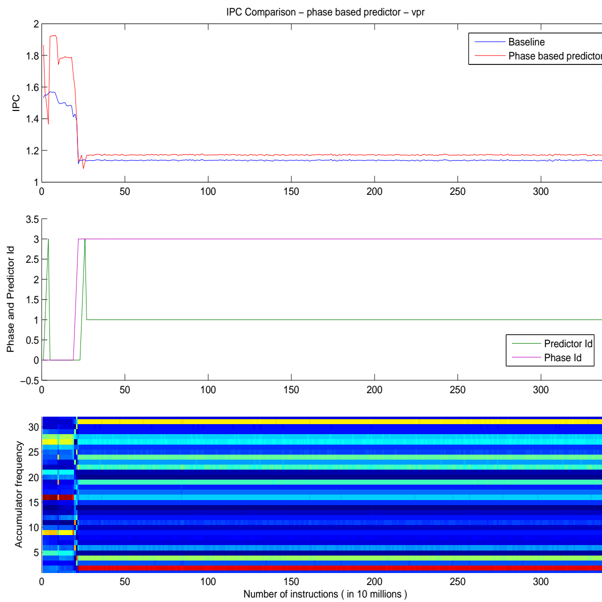


Figure 6: IPC comparison for vpr. ***This graph provides a comparison of IPC in the phase-based branch predictor scheme with the IPC in the baseline case for the* vpr *benchmark. The second part of the graph shows the phases predicted by the phase predictor and the corresponding branch predictor chosen for that phase. The third part of the graph shows the scaled values of the accumulators used in the phase tracking scheme.***

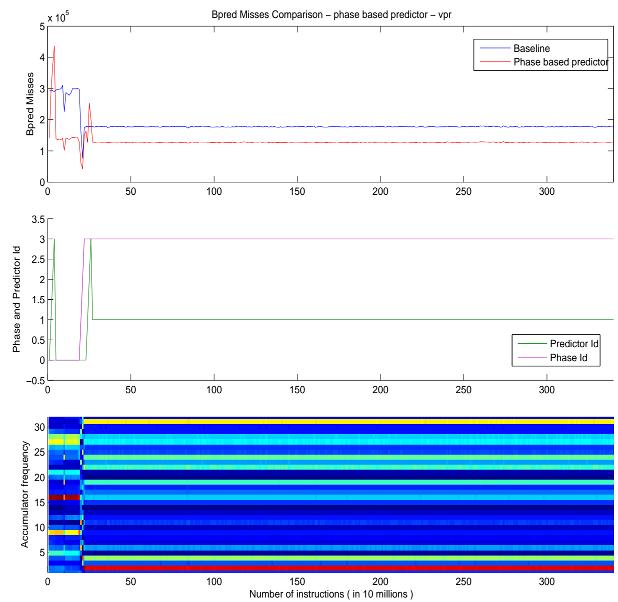


Figure 7: Branch predictor misses comparison for vpr. ***This graph provides a comparison of branch predictor misses in the phase-based branch predictor scheme with those in the baseline case for the* vpr *benchmark. The second part of the graph shows the phases predicted by the phase predictor and the corresponding branch predictor chosen for that phase. The third part of the graph shows the scaled values of the accumulators used in the phase tracking scheme.***

cally, as the "Managing Multiple Low-Power Adaptation Techniques: The Positional Approach" sidebar describes, this approach uses either compile-time or run-time profiling to select an appropriate configuration for long-running subroutines. In the static approach, a profiling run measures the total execution time per invocation of each subroutine. Developers identify phases as subroutines with values for those quantities that exceed preset thresholds, then they instrument the entry and exit points of these subroutines to trigger a reconfiguration decision.

McFarling [13] proposed the idea of combining the advantages of multiple branch predictor by using a selector to choose the best branch predictor for the current branch. The different branch predictors take advantage of different observed patterns in branch behavior. They achieve an accuracy of 98.1% for their benchmarks but increase the complexity of branch prediction mechanism that might become a bottleneck for IPC as a complex procedure is to be followed in the fetch stage to predict a particular branch. They do not take into account any phase behavior for choosing the predictor and the predictor selection needs to take place for every single branch prediction. In contrast, our scheme incurs small overheads at the beginning of a new phase but then run the *chosen one* for that entire phase. As our experiments validate, applications often choose simple predictors depending on the phase of execution.

9. CONCLUSIONS

In this paper, we have demonstrated that a variety of workloads depict large scale repetitive behavior and that the characteristics of the program remains fairly uniform within single phase. This uniform behavior within a phase can be exploited to both reduce the power consumption and increase the accuracy, as the hardware is specifically suited to a particular phase. We have tried to exploit this feature in selecting an appropriate branch predictor configuration. In the case of branch predictor selection, there is significant reduction in the number of branch prediction misses (up to 44.35%) and up to 4.7% increase in IPC. As discussed earlier in Section 7, power consumption due to execution of mis-speculated instructions and flushing multi-stage pipelines can be reduced by using the adaptive phase-based branch predictor.

10. REFERENCES

[1] D. H. Albonesi. Dynamic ipc/clock rate optimization. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 282–292, Washington, DC, USA, 1998. IEEE Computer Society.

[2] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259, Washington, DC, USA, 1999. IEEE Computer Society.

[3] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with

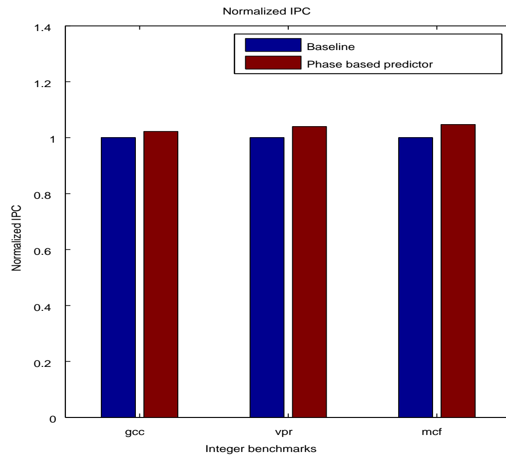


Figure 8: Improvements in IPC. ***This graph shows the improvements in IPC due to phase-based branch predictor selection for the*** **gcc, vpr** ***and*** **mcf** ***benchmarks. The improvements in IPC range from around 2% to 4.5%.***

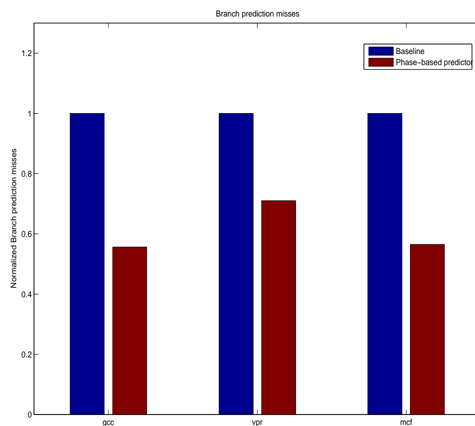


Figure 9: Reduction in branch predictor misses. ***This graph shows the reduction in branch prediction misses due to phase-based branch predictor selection for the*** **gcc, vpr** ***and*** **mcf** ***benchmarks. The reductions are significant, ranging from 30% to 45%.***

adaptive processing. *IEEE Computer*, 36(12):49–58, December 2003.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, New York, NY, USA, 2000. ACM Press.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[6] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. An adaptive issue queue for reduced power at high performance. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 25–39, London, UK, 2001. Springer-Verlag.

[7] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[9] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 202–213, New York, NY, USA, 2000. ACM Press.

[10] M. C. Huang, D. Chaver, L. Piñuel, M. Prieto, and F. Tirado. Customizing the branch predictor to reduce complexity and energy consumption. *IEEE Micro*, 23(5):12–25, 2003.

[11] D. A. Jiminez and C. Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, 2002.

[12] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 155–166, Washington, DC, USA, 1998. IEEE Computer Society.

[13] S. McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGARCH Comput. Archit. News*, 30(5):45–57, 2002.

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, New York, NY, USA, 2003. ACM Press.

[16] J. E. Smith. A study of branch prediction strategies. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, New York, NY, USA, 1998. ACM Press.

[17] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 145–154, New York, NY, USA, 1999. ACM Press.

[18] B. Xu and D. H. Albonesi. Runtime reconfiguration techniques for efficient general-purpose computation. *IEEE Design & Test of Computers*, 17(1):42–52, 2000.

[19] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO*, pages 51–61, 1991.