In Proceedings of the 30th International Symposium on Computer Architecture (ISCA), June 2003.

Phase Tracking and Prediction

Timothy Sherwood Suleyman Sair Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,ssair,calder}@cs.ucsd.edu

Abstract

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior, at run-time, can unlock a multitude of optimization opportunities.

In this paper, we present a unified profiling architecture that can efficiently capture, classify, and predict phase-based program behavior on the largest of time scales. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. Our analysis shows that our design can capture phases that account for over 80% of execution using less that 500 bytes of on-chip memory.

1 Introduction

Modern processors can execute upwards of 5 billion instructions in a single second, yet most architectural features target program behavior on a time scale of hundreds to thousands of instructions, less than half a μS. While these optimizations can provide large benefits, they are limited in their ability to see the program behavior in a larger context.

Recently there has been a renewed interest in examining the run-time behavior of programs over longer periods of time [10, 11, 19, 20, 3]. It has been shown that programs can have considerably different behavior depending on which portion of execution is examined. More specifically, it has been shown that many programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase. Taking advantage of this time varying behavior can lead to, among other things, improved power management, cache control, and more efficient simulation. The primary goal of this research is the development of a unified run-time phase detection and prediction mechanism that can be used to guide any optimization seeking to exploit large scale program behavior.

A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being an interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency. Simply put, if a phase of execution is correctly identified, there should only be small variations between any two execution intervals identified as being part of the same phase. A key point of this paper is that the phase behavior seen in any program metric is directly a function of the way the code is being executed. If we can accurately capture this behavior at run-time through the computation of a *single* metric, we can use this to guide many optimization and policy decisions without duplicating phase detection mechanisms for *each* optimization.

In this paper, we present an efficient run-time phase tracking architecture that is based on detecting changes in the *proportions* of the code being executed. In addition, we present a novel phase prediction architecture that can predict, not only when a phase change is about to occur, but also the phase to which it is will transition. Since our phase tracking implementation is based upon code execution frequencies, it is independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics allows us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

We demonstrate the effectiveness of our hardware based phase detection and classification architecture at automatically partitioning the behavior of the program into homogeneous phases of execution and to identify phase changes. We show that the changes in many important metrics, such as IPC and energy, correlate very closely with the phase changes found by our metric. We then evaluate the effectiveness of phase tracking and prediction for value profiling, data cache reconfiguration, and re-configuring the width of the processor.

The rest of the paper is laid out as follows. In Section 2, prior work related to phase-based program behavior is discussed. Simulation methodology and benchmark descriptions can be found in Section 3. Section 4 describes our phase tracking architecture. The design and evaluation of the phase predictor are found in Section 5. Section 6 presents several potential applications of our phase tracking architecture. Finally, the results are summarized in Section 7.

2 Related Work

In this Section we describe work related to phase identification and phase-based optimization.

In [19], we provided an initial study into the time varying behavior of programs, showing that programs have repeatable phase-based behavior over many hardware metrics – cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy for all the SPEC 95 programs. Looking at these metrics over time, we found that many programs have repeating patterns, and that important metrics tend to change at the same time. These places represent phase boundaries.

In [20], we proposed that by profiling only the code that was executed over time we could automatically identify periodic and phase behavior in programs. The goal was to automatically find the repeating patterns observed in [19], and the lengths (periods) of these patterns. We then extended this work in [21], using techniques from machine learning to break the complete execution of the program into phases (clusters) by only tracking the code executed. We found that intervals of execution grouped into the same phase had similar behavior across all the architecture metrics examined. From this analysis, we created a tool called SimPoint [21], which automatically identifies a small set of intervals of execution (simulation points) in a program to perform architecture simulations. These simulation points provide an accurate and efficient representation of the complete execution of the program.

The work of Dhodapkar and Smith [10, 9] is the most closely related to ours. They found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They propose that by detecting phases and phase changes, multi-configuration units can be re-configured in response to these phase changes. They have used their working set analysis for instruction cache, data cache and branch predictor re-configuration to save energy [10, 9].

The work we present in this paper identifies phases and phase changes by keeping track of the proportions in which the code was executed during an interval based upon the profiler used in [20]. In comparison, Dhodapkar and Smith [10, 9] track the phase and phase changes solely upon what code was executed (working set), without weighting the code by its frequency of execution. Future research is needed to compare these two approaches.

Additional differences between our work include our examination of architectures for predicting phase changes, and different uses from [10, 9], such as value profiling and processor width reconfiguration. We provide an architecture that can fairly accurately predict what the next phase will be, along with predicting when there will be a phase change. In comparison, Dhodapkar and Smith do not examine phase-based prediction [10, 9], but concentrate on detecting when the working set size changes, and then reactively apply optimization.

Merten et al. [15] developed a run-time system for dynamically optimizing frequently executed code. Then in [3], Barnes et al. extend this idea to perform phase-directed complier optimizations. The main idea is the creation of optimized code "packages" that are targeted towards a given phase, with the goal of execution staying within the package for that phase. Barnes et al. concentrate primarily on the compiler techniques needed to make phase-directed compiler optimizations a reality, and do not examine the mechanics of hardware phase detection and classification. We believe that using the techniques in [3] in conjunction with our phase classification and prediction architecture will provide a powerful run-time execution environment.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 4 operations per cycle, 64 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

3 Methodology

To perform our study, we collected information for ten SPEC 2000 programs `applu`, `apsi`, `art`, `bzip`, `facerec`, `galgel`, `gcc`, `gzip`, `mcf`, and `vpr` all with reference inputs. All programs were executed from start to completion using SimpleScalar [5] and Wattch [4]. Because of the lengthy simulation time incurred by executing all of the programs to completion, we chose to focus on only 10 programs. We chose the above 10 programs since their phase based behavior represents a reasonable snapshot of the SPEC 2000 benchmark suite, along with picking some of the programs that showed the most interesting phase-based behavior. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used was derived from the SimpleScalar 3.0 tool set [5], a suite of functional and timing simulation tools for the Alpha AXP ISA. The baseline microarchitecture model is detailed in Table 1. In addition to this, we wanted to examine energy usage optimizations, so we used a version of Wattch [4] to capture this information. We modified all of these tools to log and reset the statistics every 10 million instructions, and we use this as a base for evaluation.

4 Phase Capture

In this section we motivate the occurrence of phase-based behavior, describe our architecture for capturing it, and examine the accuracy of using the program behavior in our phase-tracking architecture to identify phase changes for various hardware metrics.

4.1 Phase-Based Behavior

The goal of this research is to design an efficient and general purpose technique for capturing and predicting the run-time phase behavior of programs for the purpose of guiding any optimization seeking to exploit large scale program behavior. Figure 1 helps to motivate our approach to the problem. This figure shows the behavior of two programs, `gcc` and `gzip`, as measured by various different statistics over the course of their execution from start to finish. Each point on the graph is taken over 10 million instructions worth of execution. The metrics shown are the

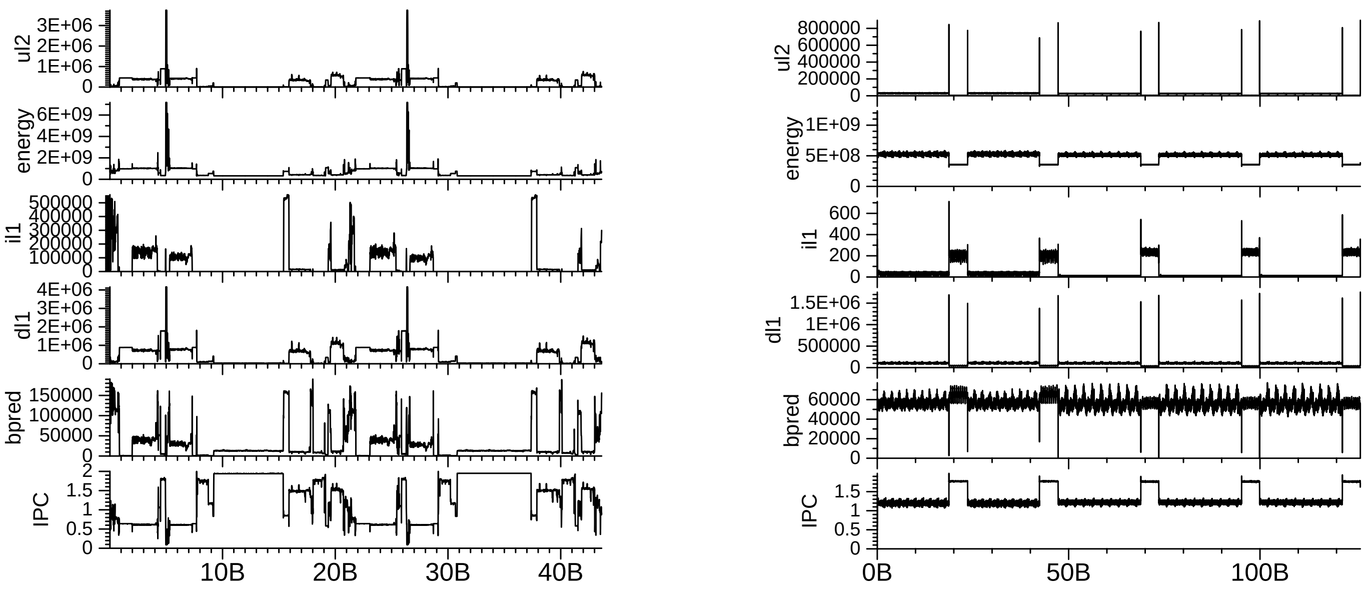

Figure 1: To illustrate the point that phase changes happen across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the programs `gcc` *(shown left) and* `gzip` *(shown right). Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.*

number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC. The results show that all of the metrics tend to change in unison, although not necessarily in the same direction. In addition to this, patterns of recurring behavior can be seen over very large time scales.

As can be seen from these graphs, even at a granularity of 10 million instructions (which is at the same time scale as operating system time slices) there can be wildly different behavior seen between intervals. In this paper, we concentrate on a granularity of 10 million instructions because it is both outside the scope of normal architectural timing and is small enough to allow for many complex phase behaviors to be seen.

4.2 Tracking Phases by Executed Code

Our phase tracker architecture operates at two different time scales. It gathers profile information very quickly in order to keep up with processor speeds, while at the same time it compares any data it gathers with information collected over the long term. Additionally, it must be able to do all that while still being reasonable in size.

Our phase profile generation architecture can be seen in Figure 2. The key idea is to capture basic block information during execution, while not relying on any compiler support. Larger basic blocks need to be weighed more heavily as they account for a more significant portion of the execution. To *approximate* gathering basic block information, we capture branch PCs and the number of instructions executed between branches. The input to the architecture is a tuple of information: a branch identifier (PC) and the number of instructions since the last branch PC was executed. This allows us to roughly capture each basic block executed along with the weight of the basic block in terms of the number of instructions executed, as we did in [20, 21] for identifying simulation points.

Classifying phases by examining only the code that is executed allows our phase tracker to be independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics is also very important to allow us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

At this point it is worth making more explicit the differences between our technique and that of Dhodapkar and Smith [10, 9]. Dhodapkar and Smith use a bit vector to track the working set of the code for a particular interval. While our technique is based on the basic block vectors used in [20]. The bit vectors of Dhodapkar and Smith track a metric that is related to which code blocks were *touched*, whereas our metric tracks the *proportion* of time spent executing in each code block. This is a subtle but important distinction. We have found that in complex programs (such as `gcc` and `gzip`) there are many instructions blocks that execute only intermittently. When tracking the pure working set, these infrequently executed blocks can disguise the frequently executed blocks that dominate the behavior of the application. On the other hand, by tracking the frequency of code execution it is possible to distinguish important instructions (basic blocks) from a sea of infrequently executed ones. Examining these differences in more detail is a topic of future research.

Another advantage of tracking the proportions in which the basic blocks are executed is that we can use this information to

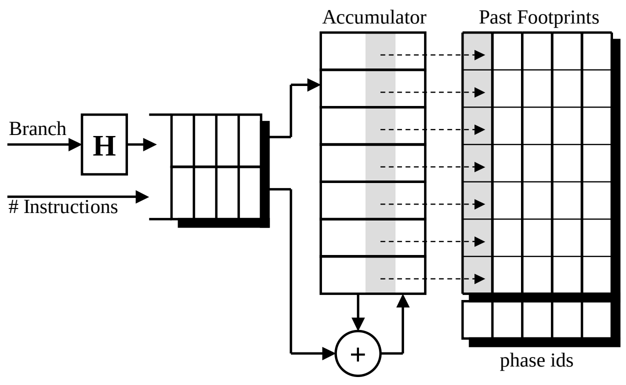

Figure 2: Our phase classification architecture. Each branch PC is captured along with the number of instructions from the last branch. The bucket entry corresponding to a hash of the branch PC is incremented by the number of instructions. After each profiling interval has completed, this information is classified, and if it is found to be unique enough, stored in the past footprint table along with its phase ID.

identify not only when different sections of code are executing, but also when those sections of code are being exercised differently. A simple example is in a graphics manipulation program running a parameterized filter on an input image. If you run a simple 3x3 blur filter on an image you get very different behavior than if you run a 7x7 filter on the same image despite the fact that the same filter code is executing. The 7x7 filter will have many more memory references and those memory references conflict very differently in the cache than in the 3x3 case. We have seen this very behavior in examining the interactive graphics program `xv`. Using the proportion of execution for each basic block can distinguish these differences, because in the 3x3 filter the head of the loop is called more than twice as frequently as in the 7x7 filter.

The same general idea applies to other data structures as well. Take for example a linked list. As the number of nodes in the linked list traversal changes over different loop invocations, the number of instructions executed inside the loop versus the time spent outside the loop also changes. This behavior can be captured when including a measure of the proportion of the code executed, and this can distinguish between link list traversals of different lengths.

4.3 Capturing the Code Profile

To index into the accumulator table in Figure 2, the branch PC is reduced to a number from 1 to $Nbuckets$ using a hash function. We have found that 32 buckets is sufficient to distinguish between different phases even for some of the more complex programs such as `gcc`. A counter is kept for each bucket, and the counter is incremented by the number of instructions from the last branch to the current branch being processed. Each accumulator table entry is a large (in this study 24-bit), saturating counter, which will not saturate during our profiling interval of 10 million instructions. Updating the accumulator table is the only operation that needs to be performed at a rate equivalent to the processor's execution of the program (once for every branch executed). In comparison, the phase classification described below needs to only be performed once every 10 million instructions (at the end of each interval), and thus is not nearly as performance critical.

We note that the hashing function we use is fundamentally the same as the random projection method we used to generate phases in [21]. In this prior work, we make use of random projections of the data to reduce the dimensionality of the samples being taken. A random projection takes trace data in the form of a matrix of size $L \times B$, where L is the length of the trace and B is the number of unique basic blocks, and multiplies it by a random matrix of size $B \times N$, where N is the desired dimensionality of the data which is much smaller than B. This creates a new matrix of size $L \times N$, which has clustering properties very similar to the original data. The random projection method is a powerful technique when used with clustering algorithms, and for capturing phase behavior as we showed in [21]. The hashing scheme we use in this paper is essentially a degenerate form of random projection that makes a hardware implementation feasible while still having low error. If all the elements of the random projection matrix consist of either a 0 or a 1, and they are placed such that no column of the matrix contains more than a single 1, then the random projection is identical to this simple hashing mechanism. We have designed our phase classification architecture around this principle.

Figure 3 shows the effect of applying the above mentioned technique for capturing the phase behavior of the integer benchmark `gzip`. The x-axis of the figure is in billions of instructions, as is the case in Figure 1. Each point on the y-axis represents an entry of the phase tracker's accumulator table. Each point on the graph corresponds to the value of the corresponding accumulator table entry at the end of a profiling interval. Dark values represent high execution frequency, while light values correspond to low frequency. The same trends that were seen in Figure 1 for `gzip` can be clearly seen in Figure 3. In both of these figures, when observing them at the coarsest granularity, we can see that there are at least three different phases labeled `A`, `B` and `C`. In Figure 3, the phase tracker table entries `2, 5, 7, 13 and 17` distinguish the two identical long running phases labeled `A` from a group of three long running phases labeled `C`. Phase table entries `12 and 20` clearly distinguish phase `B` from both `A` and `C`. This figure is pictorial evidence that the phase tracker is able to break the program's execution into the corresponding phases based solely on the executed code, and that these phases correspond to the behavior seen across the different program metrics in Figure 1.

4.4 Forming a Footprint

After the profiling interval has elapsed, and branch block information has been accumulated, the phase must then be classified. To do this we keep a history of past phase information.

If we fix the number of instructions for a profiling interval, then we can divide each bucket by this fixed number to get the percentage of execution that was accounted for by all instructions mapped to that bucket. However, we do not need to know the exact percentages for each bucket. Instead of keeping the

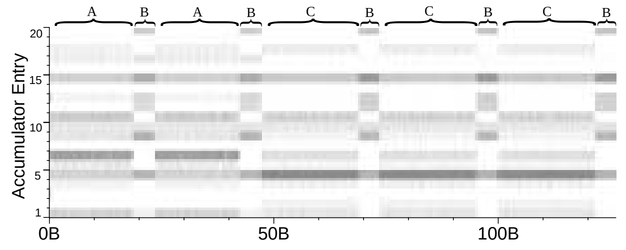

Figure 3: Visualization of the accumulator table used to track program behavior for `gzip`*. The x-axis is in billions of instructions, while the y-axis is the entry of the accumulator table. Each point on the graph corresponds to the value of the accumulator table at the end of a profiling interval where dark values correspond to more heavily accessed entries. The same trends that were seen in Figure 1 can be clearly seen in Figure 3.*

full counter values, we can instead compress phase information down to a couple of the most significant bits. This compressed information will then be kept in the Past Footprint table as shown in Figure 2.

The number of counter value bits that we need to observe is related to $Nbuckets$. As we increase the number of buckets, the data is spread over more buckets (table entries), making for less entries per bucket (better resolution) but at the cost of more area (both in terms of number of buckets and more bits per bucket). To be on the safe side, we would like *any* distribution of data into buckets to provide useful information. To achieve this we need to ensure that even if data is distributed perfectly evenly over all of the buckets, we would still record information about the frequency of those buckets. This can be achieved by reducing the accumulator counter by:

$$(bucket[i] \times Nbuckets)/(intervalsize)$$

If the number of buckets and interval size are powers of two, this is a simple shift operation. For the number of buckets we have chosen (32), and the interval size we profile over, this reduces the bucket size to 6 bits, and thus requires 24 bytes of storage for each unique phase in the Past Footprint table. In practice we see that the top 6 bits of the counter are more than enough to distinguish between two phases. In the worst case, you may need one or two more bits to reduce quantization error, but in reality we have not seen any programs that cause this to be an issue.

If too few buckets are used, aliasing effects can occur due to the hashing function, where two different phases will appear to have very similar Footprints. Therefore, we want to use a sufficiently large number of buckets to uniquely identify the differences in code execution between phases, while at the same time use only a small amount of area.

To examine the aliasing effect and determine what the appropriate number of buckets should be, Figure 4 shows the sum of the differences in the bucket weights found between all sequential intervals of execution. The y-axis shows the sum total of differences for each program. This is calculated by summing the differences between the buckets captured for interval i and $i-1$ for each interval i in the program. The x-axis is the number of distinct buckets used. All of the results are compared to the ideal case of using an infinite number of buckets (or one for each separate basic block) to create the Footprint. On the program `gcc` for example, the total sum of differences with 32 buckets was 72% of that captured with an infinite number of buckets. In general we have found that 32 buckets was enough to distinguish between two phases.

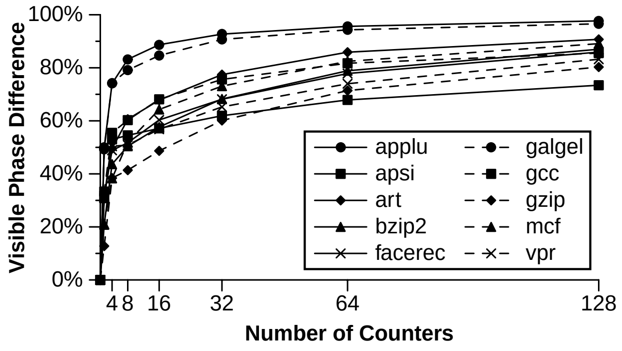

Figure 4: The percent difference found between Footprints from sequential intervals of execution, when varying the number of counters used to represent the footprints. The results are normalized to the difference between intervals found when having an infinite number of buckets to represent the footprint; 32 buckets captures most of the benefit.

4.5 Classifying a Footprint to a Phase ID

After reducing the vector to form a footprint, we begin the classification process by comparing the footprint to a set of representative past footprint vectors. We compare the current vector to each vector in the table. The next section details how we perform the comparison and determine what a match is. If there is a match, we classify the profiled section of execution into the same phase as the past footprint vector, and the current vector is not inserted into the past footprint table. If there is no match, then we have just detected a new phase and hence must create a new unique phase ID into which we may classify it. This is done by choosing a unique phase ID out of a fixed pool of IDs. When allocating a new phase ID, we also allocate a new past footprint entry, set it to the current vector, and store with that entry the newly allocated phase ID. This allows future similar phases to be classified with the same ID. In this way only a single vector is kept for each unique phase ID, to serve as a representative of that phase. After a phase ID is provided for the most recent interval, it is passed along to prediction and statistic logging, and the phase identification part of our algorithm is completed.

To examine the number of phase IDs we need to track, Figure 5 shows the percentage of execution that can be accounted for by the top p phases, where p is shown on the x-axis. Results are graphed for the programs that had the min (`galgel`) and max (`art`) coverage, `gcc`, `gzip`, and the overall average. These results show that most of the program's phase behavior can be captured using a relatively small number of phase IDs.

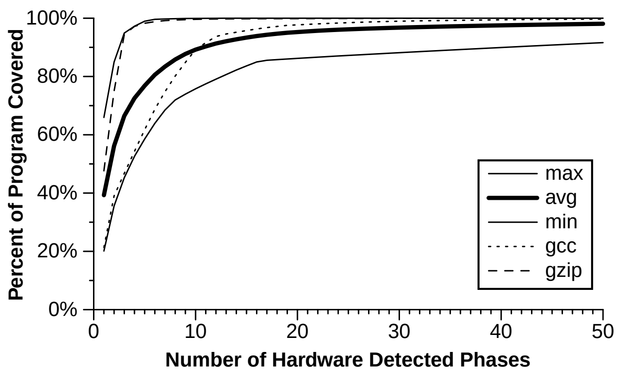

Figure 5: Results of the minimum number of phases that need to be captured versus the amount program execution they cover. The y-axis is the percent of program execution that is covered. The x-axis is the minimum number of phases needed to capture that much program execution.

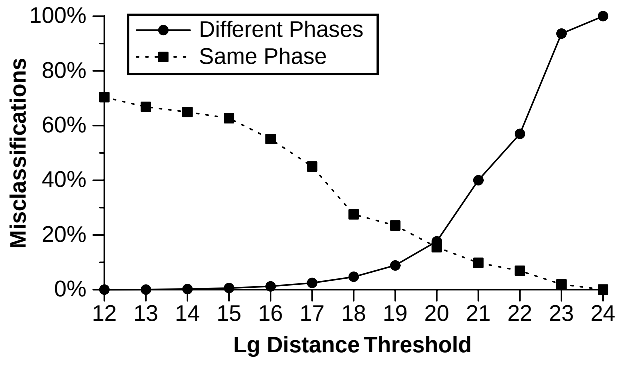

Figure 6: Results showing how well our hardware phase tracker classifies two sequential intervals of execution as being from "Different" or the "Same" phase of execution. The percent of misclassifications are shown in comparison to the phase classifications found using the off-line clustering SimPoint tool [21].

If we only track and optimize for the top 20 phases in each application, we will capture and be able to accurately apply phase prediction/optimizations to over 90% of the program's execution on average. In the worst case (min), we are able to optimize most of the program (over 80%) by only targeting a small number (20) of important recurring phases.

4.5.1 Finding a Match

We search through the Footprint histories to find a match, but this query is complicated by the fact that we are not necessarily searching for an exact match. Two sections of execution that have very similar footprints could easily be considered a match, even if they do not compare exactly. To compare two vectors to one another, we use the Manhattan distance between the two, which is the element-wise sum of the absolute differences. This distance is used to determine if the current interval should be classified as the same phase ID as one of the past footprint intervals.

If we set the distance threshold too low, the phase detection will be overly sensitive, and we will classify the program into many, very tiny phases which will cause us to lose any benefit from doing run-time phase analysis in the first place. If the threshold is too high, the classifier will not be able to distinguish between phases with different behavior. To quantify this effect, we examine how well our hardware technique classifies phases for a variety of thresholds compared to the phases found by the off-line clustering algorithm used in SimPoint [21].

The SimPoint tool is able to make global decisions to optimize the grouping of similar intervals into phases. The off-line algorithm makes no use of thresholds, instead its decisions are based solely on the structure found in the distribution of program behaviors. Our technique must be far more simplistic because it must be performed on-line and with limited computational overhead. This reduction in complexity comes at the cost of increased error.

The Different Phases line in Figure 6 shows the ability of our hardware technique to find phase changes (transitions between one phase and the next) when different thresholds are used to perform the phase classification. For example, when using a Manhattan distance of 1 million as our threshold (shown as 20 on our x-axis because it is in $\log_2$), our hardware technique identified 80% of the phase changes that occurred in the more complex off-line SimPoint analysis. Conversely, 20% of the phase changes were incorrectly classified as having the same phase ID as the last interval of execution.

Likewise, the Same Phases line in Figure 6 represents the ability of our hardware technique to accurately classify two sequential intervals as being part of the same phase as a function of different thresholds (again as compared to the off-line clustering analysis). For example, when using a Manhattan distance of 1 million (shown as 20 on the x-axis), our hardware technique identified 80% of the intervals that stayed in the same phase as correctly staying in the same phase, but 20% of those intervals were classified as having a different phase ID from the prior phase.

A misclassification occurs when two sequential intervals of execution are classified as being in the same phase or in different phases using our hardware approach when the off-line clustering analysis tool found the opposite for these two intervals.

If we are too aggressive and our hardware phase analysis indicates that there are phase changes when there are actually no noticeable changes in behavior, then we will create too many phase IDs that have similar behavior. This can create more overhead for performing phase-based optimization. On the other hand, if we are too passive in distinguishing between different phases, we will be missing opportunities to make phase specific optimizations.

In order to strike a balance between having a high capture rate and reducing the percent of false positives, we chose to use a threshold of 1 million. When comparing this with the interval size of 10 million instructions, this means that a difference in the phase behavior will be detected if 10% of the executed instructions are in different proportions. In choosing 1 million, we have on average a 20% misclassification rate. Note, that a misclassification does not necessarily mean that an incorrect optimization will be performed. For example, if we have a "Same Phase" misclassification (the two intervals were really from the same phase, but were classified into different phases), then a phase change is observed using our hardware technique when there was not one in the baseline classifier. If the two hardware detected phases have the same optimization applied to them, then this misclassification can have no effect.

4.6 Per-Phase Performance Metric Homogeneity

Using the techniques presented above, we can perform phase classification on programs at run-time with little to no impact on the design of the processor core. One of the goals of phase classification is to divide the program into a set of phases that are fairly homogeneous. This means that an optimization adapted and applied to a single segment of execution from one phase, will apply equally well to the other parts of the phase. In order to quantify the extent to which we have achieved this goal, we need to test the homogeneity of a variety of architectural statistics on a per-phase basis.

Figure 7 shows the results of performing this analysis on the phases determined at run-time. Due to space constraints we only show results for two of the more complicated programs `gcc` and `gzip`. For both programs, a set of statistics for each phase is shown. The first phase that is listed (separated from the rest) as `full`, is the result of classifying the entire program into a single phase. The results show that for `gcc` for example, the average IPC of the entire program was 1.32, while the average number of cache misses was 445,083 per ten million instructions. In addition to just the average value, we also show the standard deviation for that statistic. For example, while the average IPC was 1.32 for `gcc`, it varied with a standard deviation of over 43% from interval to interval. If the phase-tracking hardware is successful in classifying the phases, the standard deviations for the various metrics should be low for a given phase ID.

Underneath the phase marked `full` are the five most frequently executed phases from the program as identified by our phase tracker. The phases are weighted by the percentage of the program's executed instructions they account for. For `gcc`, the largest phase accounts for 18.5% of the instructions in the entire program and has an average IPC of 0.61 and a standard deviation of only 1.6% (of 0.61). The other top four phases have standard deviations at or below this level, which means that our technique was successful at dividing up the execution of `gcc` into large phases with similar execution behavior with respect to IPC. Note, that some metrics for certain phases have a high standard deviation, but this occurs for architecture features/metrics that are unimportant for that phase. For example, the phase that occurs for 7.2% of execution in `gcc` has only 75 L1 instruction cache misses on average. This is an L1 miss rate of 0.00075%, so an error of 215% for this metric will not likely have any effect on the phase.

When we look at the energy consumption of `gcc`, it can be observed that energy consumption swings radically (a standard deviation of 90%) over the complete execution of the program. This can be seen visually in Figure 1, which plots the energy usage versus instructions executed. However, after dividing the program into phases, we see that each phase has very little variation within itself. All have less than 2% standard deviation. By analyzing `gcc` it can also be seen that the phase partitioning does a very good job across all of the measured statistics even though only *one* metric is used. This indicates that the phases that we have chosen are in some way representative of the actual behavior of the program.

5 Phase Prediction

The prior section described our phase tracking architecture, and how it can be used to classify phases. In this section we focus on using phase information to predict the next phase. For a variety of applications it is important to be able to predict future phase changes so that the system can configure for the code it will soon be executing rather than simply reacting to a change in behavior.

Figure 8 shows the percentage interval transitions that are changes in phase, for our set of benchmarks. For all of these programs, phase changes come quite often, but it should be noted that this statistic alone cannot gauge the complexity of the program behavior. The program `gcc` switches less than 10% of the time but switches between *many* different phases. The other extreme is `art` which switches almost half the time, but it is only switching between a few distinct phases. In this case, large repeating patterns can be observed. No two phases executing sequentially are that similar, but there is an order to the sequence. By adding in a prediction scheme for these cases, we not only take advantage of stable conditions as in past research, but actually take advantage of any repeating patterns in program behavior.

5.1 Markov Predictor

The prediction of phase behavior is different from many other systems in which hardware predictors are used. Because of this new environment, a new type of predictor has the potential to perform better than simply using predictors from other areas of computer architecture (branch and address prediction, memory disambiguation, etc.).

After observing the way that phases change, we determined that two pieces of information are important. First, the set of phases leading up to the prediction are very important, and second, the *duration* of execution of those phases is important.

A classic prediction model that is easily implementable in hardware is a Markov Model. Markov Models have been used in computer architecture to predict both prefetch addresses [13] and branches [8] in the past. The basic idea behind a Markov Model is that the next state of the system is related to the last set of states.

The intuition behind this design is that phase information tends to be characterized by many sections of stable behavior interspersed with abrupt phase changes. The key is to be able to predict when these phase changes will occur, and to know ahead of time what phase they will change to. The problem is that the changes are often preceded by stable conditions, and if we only consider the last couple of intervals we will not be able to tell the difference between sections of stable behavior that precede a phase change, and those sections that will continue to be stable. Instead, we need a way of compressing down stable phase

	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gcc	full	1.32	(43.4%)	27741	(135.5%)	445083	(110.7%)	50763	(203.2%)	6.44E+08	(90.0%)	227912	(139.7%)
	18.5%	0.61	(1.6%)	34665	(22.0%)	753382	(5.4%)	125091	(23.2%)	1.03E+09	(1.8%)	395997	(5.3%)
	18.1%	1.95	(0.3%)	13048	(3.9%)	28112	(15.1%)	43	(73.9%)	3.22E+08	(0.2%)	1006	(5.6%)
	7.2%	0.64	(0.2%)	843	(15.1%)	885081	(0.1%)	75	(215.5%)	9.78E+08	(0.3%)	443655	(0.1%)
	4.0%	1.49	(1.2%)	10145	(7.6%)	703554	(6.8%)	15591	(5.2%)	4.20E+08	(1.1%)	354084	(7.0%)
	3.9%	1.76	(1.6%)	2015	(13.6%)	98947	(5.9%)	102	(45.1%)	3.57E+08	(1.6%)	15595	(12.6%)
	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gzip	full	1.33	(16.3%)	56045	(11.1%)	90446	(58.2%)	60	(138.1%)	4.82E+08	(13.5%)	22880	(112.0%)
	17.1%	1.24	(3.4%)	53300	(10.8%)	96960	(10.1%)	12	(44.2%)	5.05E+08	(3.5%)	24218	(8.6%)
	9.4%	1.23	(3.8%)	54973	(11.5%)	99523	(11.3%)	11	(45.5%)	5.09E+08	(3.8%)	24518	(9.3%)
	8.8%	1.76	(0.6%)	56449	(4.8%)	37331	(5.6%)	241	(8.4%)	3.55E+08	(0.6%)	5617	(15.6%)
	8.0%	1.22	(4.3%)	54791	(6.8%)	99671	(11.9%)	40	(25.7%)	5.14E+08	(4.4%)	28153	(11.0%)
	7.4%	1.24	(3.1%)	55215	(11.1%)	96701	(9.6%)	12	(35.4%)	5.04E+08	(3.2%)	23701	(8.4%)

Figure 7: Examination of per-phase homogeneity compared to the program as a whole (denoted by `full`). For the two programs and each of the top 5 phases of each program, we show the average value of each metric and the standard deviation. The name of the phase is the percent of execution that it accounts for in terms of instructions. These results show that after dividing up the program into phases using our run-time scheme the behavior within each phase is quite consistent.

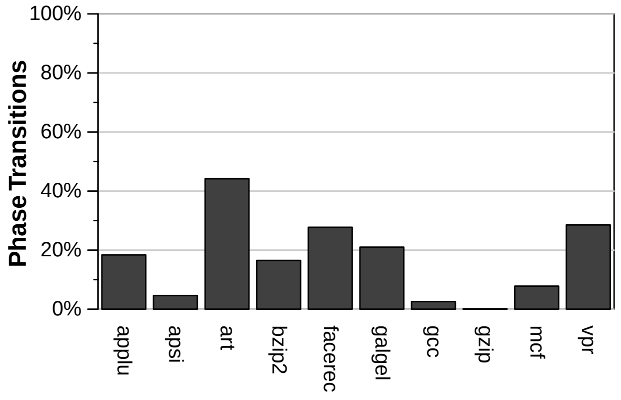

Figure 8: The percent of execution intervals that transition to a different phase from the prior execution interval's phase as found by our phase tracking architecture with 32 footprint counters using a 1 million Manhattan threshold.

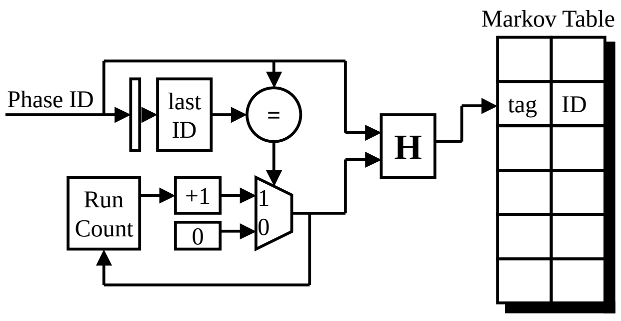

Figure 9: Phase Prediction Architecture for the Run Length Encoded (RLE) Markov predictor. The basic idea behind the predictor is that two pieces of information are used to generate the prediction, the phase id that was just seen, and the number of times prior to now that it has been seen in a row. The index into the prediction table is a hash of these two numbers.

information into a piece of information that we can use as state.

5.2 Run Length Encoding Markov Predictor

To compress the stable state we use a *Run Length Encoding* (RLE) Markov predictor. The basic idea behind the predictor is that it uses a run-length encoded version of the history to index into a prediction table. The index into the prediction table is a hash of the phase identifier and the number of times the phase identifier has occurred in a row.

Figure 9 shows our RLE Markov Phase ID prediction architecture. The the lower order bits of the hash function provide an index into the prediction table, and the higher order bits of the hash function provide a tag. When there is a tag match, the phase ID stored in the table provides a prediction as to the next phase to occur in execution. When there is a tag miss, the prior phase ID is assume to be the next phase ID to occur in the program's execution. We found that predicting the last phase ID to be 75% accurate on average.

We only update the predictor when there is (1) a change in the phase ID, or (2) when there is a tag match. We only insert an entry when there is a phase ID change, since we want to predict when the phase is going to change. Execution intervals where the same phase ID occurs several times in a row do not need to be stored in the table, since they will be correctly predicted as "last phase ID", when the there is a table miss. This helps table capacity constraints and avoids polluting the table with last phase predictions. For the second update case, when there is a tag match, we update the predictor because the observed run length may have potentially changed.

5.3 Predictor Comparison

We compare our RLE Markov phase predictor with other prediction schemes in Figure 10. This Figure has four bars for every program, and each bar corresponds to the prediction accuracy of a prediction architecture. The first and simplest scheme, `Last Phase`, simply predicts that the next phase is the same as the current phase, in essence always predicting stable operation. The prediction accuracy of this scheme is inversely proportional to the rate at which phases change in a given benchmark. For the program `gzip` for example, there are long periods of execution where the phase does not change, and therefore predicting no-change does exceptionally well.

In order to insure that we were not simply providing an

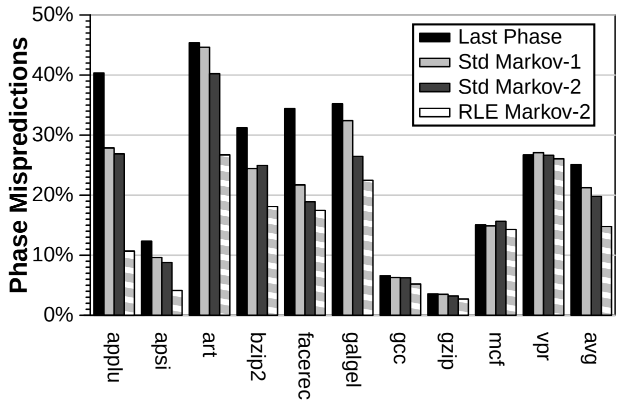

Figure 10: Phase ID Prediction Accuracy. This figure shows how well different prediction schemes work. The most naive scheme, `last`*, simply predicts that the phases never change. The bars marked* `Markov` *and* `RLE Markov` *show how well we can predict the phase identifiers if we use a Markov prediction scheme with a Markov table size of 256 entries.*

expensive filter for noise in the phase IDs, we also compared against a simple noise filter which works by predicting that the next phase will be the most commonly occurring of the last three phases seen. This is not shown, as it actually performed worse on all of the programs.

Additionally we wanted to examine the effect of a simple Markov model predictor for history lengths of 1 and 2. The Markov model predictor does a better job of predicting phase transitions than `Last Phase`, but it is limited by the fact that long runs will always be predicted as infinitely stable due to the history filling up. However, it is still very effective for `facerec` and `applu`, but does not provide much benefit for either `art` or `galgel`.

The final bar, `RLE Markov`, is our improved Markov predictor which compresses stable phases into a tuple of phase id and duration. All of the Markov predictors simulated had 256 entries taking up less than 500 bytes of storage. Using RLE Markov outperforms both the `Last Phase` and traditional `Markov` on all the benchmarks. It performs especially well compared to other schemes on both `applu` and `art`. Overall, using a Run-Length Encoded Markov predictor can cut the phase mispredictions down to 14% on average.

6 Applications

This section examines three optimization areas in which a phase-aware architecture can provide an advantage. We begin by examining the relationship between phase behavior and value locality. We then demonstrate ways to reduce processor energy consumption by adjusting the aggressiveness of the data cache and the instruction front end.

6.1 Frequent Value Locality

Prior work on value predictors has shown that there is a great deal of value locality in a variety of programs [14, 7]. Recently, researchers have started to take advantage of frequently loaded values for the purpose of optimizing caches. For example, Yang and Gupta [22] proposed a data cache organization that compresses the most frequently used program values in order to save energy. Another way of exploiting value locality is through value specialization, which can be done either statically or dynamically [6, 17, 16] to create specialized versions of procedures or code-regions based upon the values frequently seen. These techniques are built on the idea of finding the most frequent values for loads over the whole program, and then specializing the program to those frequent values.

We examine the potential of capturing frequent values on a per-phase basis and compare this to the frequent values aggregated over the entire program, as would be used in value code specialization [6]. To perform this experiment we first gathered the top 16 values that were loaded over the complete execution of the program and stored them into a table. We then examined the percentage of executed loads that found their loaded value in this table. This result is shown as `Static` in Figure 11. While significant portions of some programs are covered by just these few top values (such as `applu`), over half of the programs have less than 10% of their loaded values covered by these top values.

The question is: can we do better by exploiting hardware-detected phase information? To answer this question we take the top 16 values for each phase, as detected by the hardware phase tracker. These top values will be shared across a single phase even if it is split into two or more different sections of execution. Each load in the program is then checked against the top values for its corresponding phase. The `Phase Coverage` bar in Figure 11 shows the percent of all load values in the program that were successfully matched to it's per-phase top value set.

Without any notion of loads or values, our method of dividing up phases is very successful at assisting in the search for frequent values. By just tracking the top 16 values of each phase, we are able to capture the values from almost 50% of the executed loads on average. The `Perfect` bar shows percentage of loads covered if one captures the top 16 load values for each and every interval (i.e., 10 million instructions) separately. This is in effect the best that we could hope to achieve for an interval size of 10 million instructions, because the 16 entries in the value table are custom crafted for each interval individually. As shown in Figure 11, the phase-tracker compares favorably with the optimal coverage. Two thirds of the total possible benefit from per-interval value locality can be captured by per-phase value locality. It is important to point out this graph by itself is not a good indicator of usefulness as near perfect coverage could be achieved simply by making every interval a separate phase. However, as shown in Figure 5 only a few phases (around 20) are used to cover at least 80% of the program's execution.

6.2 Dynamic Data Cache Size Adaptation

In a modern processor a significant amount of energy is consumed by the data cache, but this energy may not be put to good use if an application is not accessing large amounts of data with high locality. To address this potential inefficiency, previous work has examined the potential of dynamically reconfiguring the data caches with the intention of saving power. In [2], Balasubramonian et. al. present two different schemes with

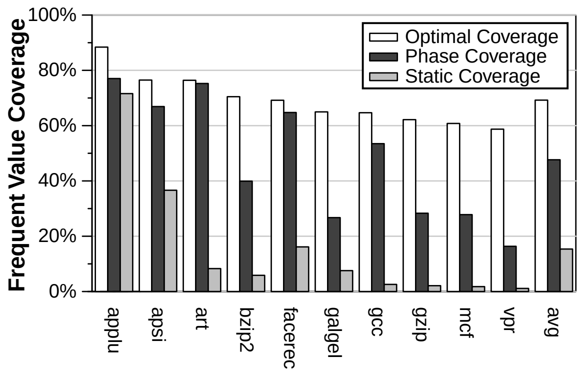

Figure 11: The percent of the program's load values that are found in a table of the most frequently values loaded over the whole program (Static Coverage), on a per-phase basis (Phase Coverage), and on a per execution interval basis (Optimal Coverage).

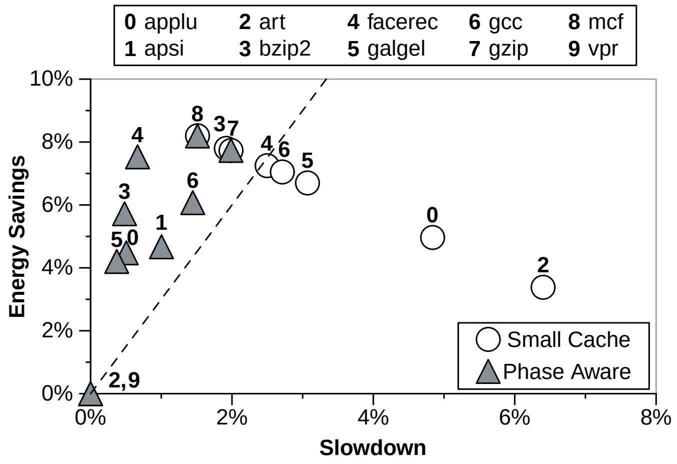

Figure 12: Data Cache Re-configuration. The tradeoff between energy savings and slowdown for two different cache policies. All results are relative to a 32KB 4-way associative cache. The circles in the graph (each labeled with a number for the program the data point is from) show the energy and performance of an 8KB direct mapped cache. The triangles show the tradeoff of intelligently switching between an 8KB direct mapped and a 32KB 4-way data cache based on phase classification and prediction.

which re-configuration may be guided. In one scheme, hardware performance counters are read by re-configuration software every hundred thousand cycles. The software then makes a decision based on the values of the counters. In another scheme, re-configuration decisions are performed on procedure boundaries instead of at fixed intervals. To reduce the overhead of re-configuration, software to trigger re-configuration is only placed before procedures that account for more than a certain percentage of execution.

Another form of re-configurable cache that has been proposed dynamically divides the data cache into multiple partitions, each of which can be used for a different function such as instruction reuse buffers, value predictors, etc [18]. These techniques can be triggered at different points in program execution including procedure boundaries and fixed intervals. The overhead of re-configuration can be quite large and making these policy decisions only when the large scale program behavior changes, as indicated by phase shifts in our hardware tracker, can minimize overhead while guaranteeing adequate sensitivity to attain maximum benefit.

We examined the use of phase tracking hardware to guide an energy aware, re-sizable cache. The energy consumption of the data cache can be reduced by dynamically shifting to a smaller, less associative cache configuration for program phases that do not benefit significantly from more aggressive cache configurations. By targeting only those phases that are predicted to have energy savings due to cache size reduction, our scheme is able to reduce power with very little impact on the performance.

We examined an architecture with two possible cache configurations, 32KB 4-way associative and 8KB direct mapped. In Figure 12, the trade off between these two configurations is plotted. For each program, we use the 32KB cache configuration as the baseline result. The labeled circles in Figure 12 show the total processor energy savings and performance degradation for each program if only the smaller (8KB) cache size is used. For example, a processor with a smaller cache configuration for the program `applu` is both 5% slower and uses 5% less energy. Two programs, `vpr` and `apsi`, actually use more energy with a smaller cache due to large slow downs. These two points are off the scale of this graph and are not shown.

While examining energy savings and slow down is interesting, it is important to note that there is more than one way to reduce both energy and performance. Voltage scaling in particular has proven to be a technology capable of reaping large energy savings for a relative reduction in performance. For our results, we assume that for voltage scaling a performance degradation of 5% will yield an approximate energy saving of 15%. We use this rule of thumb as our guideline for determining when to reduce the active size of the cache. In Figure 12, this simple model of voltage scaling is plotted as a dashed line. When the cache size is reduced, most programs fall far short of this baseline, meaning that voltage scaling would provide a better performance-energy tradeoff. There are a couple of exceptions, in particular `mcf`, `bzip`, and `gzip` do well even without any sort of phase-based re-configuration.

The shaded triangles in Figure 12 show what happens if we use phase classification and prediction to guide our re-configuration. When a new phase ID is seen, we sample the IPC and energy used for a few intervals using the 32KB 4-way cache, and a few intervals for the 8KB direct mapped cache. These samples could be kept in a small hardware profiling table associated with the phase ID. After taking these samples, if we find that a particular phase is able to achieve more than three times the energy savings relative to the slow down seen when using the 8KB cache, we then predict for this phase ID that the smaller cache size should be used. This heuristic means that the small cache size is used only if re-configuration would beat voltage scaling for that phase. After a decision has been made as to the con-

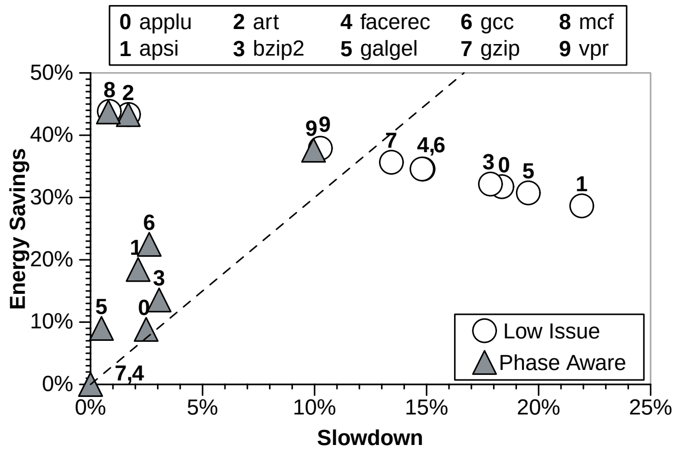

Figure 13: Processor Width Adaptation. The tradeoff between energy savings and slowdown for two different front end policies. All results are relative to an aggressive 8–issue machine. The circles in the graph (each labeled with a number for the program) show the energy and performance of a less aggressive 2–issue processor. The triangles show using the phase classifier and predictor for switching between 2–issue and 8–issue based on phase changes.

figuration to use for a phase ID, the corresponding cache size is stored in the phase profiling table/database associated with that phase ID. The phase classifier and predictor are then used to predict when a phase change occurs. When a phase change prediction occurs, the predicted phase ID looks up the cache size in the profiling table, and re-configures the cache (if it is not already that size) at the predicted phase change.

For all programs, our re-configuration is able to beat or tie voltage scaling. For example, using phase-based re-configuration results in a slowdown of 0.5% for `applu`, while the total energy savings is 4.5%. Even the program `apsi`, which had *increased* energy consumption in the small cache configuration, is able to get almost 5% energy savings with only a 1% slowdown.

6.3 Dynamic Processor Width Adaptation

One way to reduce the energy consumption in a processor is to reduce the number of instructions entering the pipeline every cycle [12, 1]. We call this adjusting the width of the processor. Reducing the width of the processor reduces the demand on the fetch, decode, functional units, and issue logic. Certain phases can have a high degree of instruction level parallelism, whereas other phases have a very low degree. Take for example the top two phases for `gcc` shown in Figure 7. The intervals classified to be in the first phase consisting of 18.5% of execution have an IPC of 0.61 with a high data cache miss rate. In comparison, the intervals in the second most frequently encountered phase (accounting for 18.1% of execution) have an IPC of 1.95 and very low data cache miss rates. We can potentially save energy without hurting performance by throttling back the width of the processor for phases that have low IPC, while still using aggressive widths for phases with high IPC.

In the current literature, decisions to reduce or increase the fetch/decode/issue bandwidth of the processor are made either at fixed intervals (relatively short intervals such as 1,000 cycles) [12] or, as in the case of branch confidence based schemes, when a branch instruction is fetched [1]. It can very difficult to design real systems that save energy by reconfiguring at these speeds, but a hardware phase-tracker can help make these decisions at a coarser granularity while still maintaining performance and energy benefits.

We examined an architecture that could be configured with 2 different widths - one where up to 2 instructions are decoded and up to 2 issued per cycle, and one where up to 8 instructions are decoded and up to 8 issued per cycle. When a new phase ID is seen by the phase tracker, we sample the IPC for three intervals with a width of 2 instructions, and three intervals with a width of 8 instructions. If there is little difference in the IPC between these two widths, then we assign a width of 2 instructions to this Phase ID in our profiling table, otherwise we assign a width of 8 instructions. During execution, we use the phase ID predictor to effectively predict the width for the next interval of execution and adjust the processor's width accordingly. Our results show that the chosen configuration for a given phase can be trained (1) with only a few samples, and (2) only once to accurately represent the behavior of a given phase ID. This requires very little training time due to the fact that 20 or fewer phase IDs are needed to capture 80% or more of a program's execution as shown in Figure 5.

Figure 13 is a graph of the results seen when applying phase-directed width re-configuration. The white circles in the graph show the behavior of running the programs on only a 2–wide machine relative to the more aggressive 8–wide machine. The dotted line again shows what could potentially be achieved if voltage scaling was used. While `mcf` and `art` save a lot of energy with little performance degradation on a 2–wide machine, the other programs do not fair as well. The program `apsi`, for example, has a slowdown of over 22% with an energy savings of around 30%. This does not compare favorably to voltage scaling (as discussed in Section 6.2). On the other hand if we use phase-directed width throttling on `apsi`, a total processor energy savings of 18% can be achieved with only 2.2% slowdown.

For all of the programs we examined, with one exception, the slowdown due to phase aware width throttling was less than 4%, while the average energy savings was 19.6%. This result demonstrates that there is significant benefit to be had in the re-configuration of processor front end resources even at very large granularities. In the worst case, this will mean a re-configuration every 10 million instructions, and on average every 70 million instructions. This should be designable even under conservative assumptions.

7 Summary

In this paper we present an efficient run-time phase tracking architecture that is based on detecting changes in the code being executed. This is accomplished by dividing up all instructions seen into a set of buckets based on branch PCs. This way we approximate the effect of taking a random projection of the basic block vector, which was shown in [21] to be an effective method of identifying phases in programs.

Using our phase classification architecture with less than 500 bytes of on-chip memory, we show that for most programs, a significant amount of the program (over 80%) is covered by 20 or less distinct phases. Furthermore, we show that these phases, while being distinct from one another, have fairly uniform behavior within a phase, meaning that most optimizations applied to one phase will work well on all intervals in that phase. In the program `gcc`, the IPC attained by the processor on average over the full run of execution is 1.32, but has a standard deviation of more than 43%. By dividing it up into different phases, we achieve much more stable behavior, with IPCs ranging between 0.61 and 1.95, but now with standard deviations of less than 2%.

In addition to this, we present a novel phase prediction architecture using a Run Length Encoding Markov predictor that can predict not only when a phase change is about to occur, but to which phase ID it will transition to. In using this design, which also uses less than 500 bytes of storage, we achieve a phase prediction miss rate of 10% for `applu` and 4% for `apsi`. In comparison, always predicting that the phase will stay the same results in a miss rate of 40% and 12% respectively.

We also examined using our phase tracking and prediction architecture to enable new phase-directed optimizations. Traditional architecture and software optimizations are targeted at the average or aggregate behavior of a program. In comparison, phase-directed optimizations aim at optimizing a program's performance tailored to the different phases in a program. In this paper, we examined using phase tracking and prediction to increase frequent value profiling coverage, and to provide energy savings through data cache and processor width re-configuration.

We believe our phase tracking and prediction design will open the door for a new class of run-time optimization that targets large scale program behavior. Even though we present a hardware implementation for phase tracking, a similar design can be implemented in software to perform phase classification for run-time optimizers, just-in-time compilation systems, and operating systems. Hardware and software optimizations that can potentially benefit the most from phase classification and prediction are (1) those that need expensive profiling/training before applying an optimization, (2) those where the time or cost it takes to perform the optimization is either slow or expensive, and (3) those that can benefit from specialization where they have the same code/data being used differently in different phases of execution. By using our dynamic phase tracking and prediction design, phase-behavior can be characterized and predicted at the largest of scales, providing a unified mechanism for phase-directed optimization.

Acknowledgments

We would like to thank Jeremy Lau and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, Semiconductor Research Corporation grant No. SRC-2001-HJ-897, and an equipment grant from Intel.

References

[1] J.L. Aragon, J. Gonzalez, and A. Gonzalez. Power-aware control speculation through selective throttling. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[3] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, December 2002.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, March 1999.

[7] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, pages 64–74, June 1999.

[8] I.-C. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, October 1996.

[9] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[10] 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.

[11] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[12] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pages 190–196, 2001.

[13] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[14] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[15] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[16] M. Mock, C. Chambers, and S.J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *33rd International Symposium on Microarchitecture*, pages 291–302, December 2000.

[17] R. Muth, S.A. Watterson, and S.K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[18] P. Ranganathan, S. V. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[19] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[20] 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.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[22] J. Yang and R. Gupta. Frequent value locality and its applications. *Special Issue on Memory Systems, ACM Transactions on Embedded Computing Systems*, 1(1):79–105, November 2002.