ABSTRACT

MUKUNDAN, JANANI Instruction Cache Checkpoints Using Phase Tracking and Prediction. (Under the direction of Professor Paul D. Franzon).

The Memory wall is standing taller than ever. There is an ever growing imbalance between memory bandwidth and processor speeds. Due to these diverging rates most applications are limited by memory performance. Various aggressive techniques to hide memory latency have done little to hide this gap. Clearly, we will need better optimization techniques to bridge the gap between processor and memory speeds. In future it will be necessary for us to understand program patterns and behavior at run time, so that we can efficiently utilize various optimization techniques. Past research [10] has suggested that program’s tend to have cyclic patterns of execution. They tend to execute in phases, which repeat over time. It is possible to efficiently capture, classify and predict phase based program behavior at run time [13].

We propose using Phase Tracking and Prediction to bridge the memory gap. We introduce the concept of Instruction Cache Checkpoints that exploit program behavior to prefetch into the Instruction Cache. The intuition behind this scheme is that since phase behavior can be predicted, we can effectively pre-fetch instructions according to phase transitions. We also propose a new improved Phase Prediction architecture based on phase run-lengths. We begin by studying and evaluating phase behavior in SPEC2k FP benchmarks. The observed phase behavior is then exploited by creating Instruction Cache Checkpoints that use prefetching based on phase changes. Detailed simulation of five of the SPEC 2k FP benchmarks show that using Instruction Cache Checkpoints gives us an average reduction of 17.8% in the number of Instruction Cache misses.
Instruction Cache Checkpoints Using Phase Tracking and Prediction

by

Janani Mukundan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master Of Science

Computer Science

Raleigh

2005

Approved By:

Dr. Edward Davis

Dr. Vincent Freeh

Dr. Paul D. Franzon
Chair of Advisory Committee
To

Amma, Appa and Nandu
Biography

Janani Mukundan was born on 28th April 1982 in Neyveli, India. She received the Bachelor of Engineering (B.E.) Degree in Computer Science and Engineering from SRM Engineering College, University of Madras, in 2003. Janani has been a graduate student in the Computer Science Department at North Carolina State University, Raleigh, NC since Fall 2003. She will receive the Master of Science degree in Computer Science from NCSU with the defense of this thesis.
Acknowledgements

At the outset I would like to thank God for giving me the strength and endurance to successfully complete this task.

Next I would like to thank my parents for their continuous support and motivation. I could not have realised my goals without their love and encouragement.

I sincerely thank my advisor Dr. Paul Franzon for giving me an opportunity to work with him. I would also like to thank Dr. Vincent Freeh and Dr. Edward Davis for agreeing to be on my master’s thesis committee and helping me complete this work.

My heartfelt gratitude to Dr. Eric Rotenberg and Dr. Gregory T. Byrd for their invaluable suggestions and support throughout my graduate studies. Finally I would like to thank my friends for always being there for me whenever I needed them.
# Contents

List of Figures vi

List of Tables viii

1 Introduction 1

2 Background: Phase Behavior 4
   2.1 Cyclic Behavior of Programs 4
   2.2 Static Phase Tracking and Classification 5
      2.2.1 Basic Block Distribution Analysis 6
      2.2.2 Automatically Characterizing Large Scale Program Behavior 9
   2.3 Dynamic Phase Tracking and Prediction 10
      2.3.1 Dynamic Phase Tracking 12
      2.3.2 Dynamic Phase Classification 13
      2.3.3 Dynamic Phase Prediction 15
      2.3.4 Run-Length Encoding Markov Predictor 17

3 The Candidate Predictor and Instruction Cache Checkpoints 19
   3.1 Need for a New Phase Prediction Architecture 19
   3.2 A New Improved Phase Predicting Architecture based on Prediction Nodes:
       The Candidate Predictor 22
   3.3 Instruction Cache Checkpoints: Introduction and Working 25

4 Simulation Methodology and Results 31
   4.1 Simulation Methodology 31
   4.2 Results and Discussion 32

5 Conclusion and Future Work 42
   5.1 Concluding Remarks 42
   5.2 Future Work 42

Bibliography 44
List of Figures

2.1 Time varying behavior of wave. X-axis is in terms of 100 million committed instructions [10] .......................... 5
2.2 Architectural Framework for a Modern Super Scalar Processor ...................... 7
2.3 Architectural Framework for the Static Phase Tracking and Classification Scheme .................................................. 8
2.4 Architectural Framework for the Dynamic Phase Tracking and Classification Scheme .................................................. 11
2.5 Phase Tracking and Classification Unit [13] .................................................. 13
2.6 Minimum number of phases that need to be captured versus the amount of program execution they cover [13] ...................... 15
2.7 Architectural Framework for the Markov Phase Prediction Scheme ................. 16
2.8 Run-Length Encoding Markov Predictor [13] .............................................. 18

3.1 Phase Transitions in swim .......................................................... 20
3.2 Run Length Comparison for Phase 13 in swim ........................................ 21
3.3 The New Improved Candidate Phase Predictor ....................................... 23
3.4 Modified Phase Prediction Architecture using Prediction Nodes .................. 24
3.5 Prediction Nodes with run-length of 6 for Phase 3 .................................... 24
3.6 Efficiency of the new improved Candidate Phase Predictor ....................... 26
3.7 Architectural Framework for Prefetching using Instruction Cache Checkpointing .................................................. 27
3.8 Instruction Cache Checkpointing Unit .................................................. 28

4.1 mgrid – Normalized I-Cache Misses .................................................. 33
4.2 mgrid – I-Cache Misses (prefetch vs. no-prefetch) .................................. 34
4.3 ammp: Phase Transitions in the Initialization Stage .............................. 35
4.4 ammp: Phase Transitions in the Execution Stage(1) ............................. 36
4.5 ammp: Phase Transitions in the Execution Stage(2) ............................. 37
4.6 ammp: Normalized I-Cache misses (with Initialization phase) ............... 38
4.7 ammp: Normalized I-Cache misses (skip Initialization phase) ............... 38
4.8 swim I-Cache misses ............................................................. 39
4.9 swim: Phase Transitions in the Initialization Stage ................................ 39
4.10 swim: Phase Transitions in the Execution Stage. .................. 40
4.11 quake: I-Cache misses. .................................................. 40
4.12 quake: Normalized I-Cache misses. ................................. 41
List of Tables

4.1 Simulation Parameters for Base line Architecture Using Only Caches. . . 31
4.2 Simulation Parameters for Architecture using Prefetcher. ............... 32
Chapter 1

Introduction

Moore’s law states that computing performance doubles every 18 months. Processor speeds keep rising dramatically. In contrast memory speeds double only every seven to ten years. There is an ever growing imbalance between memory bandwidth and processor speed. These diverging rates imply an impending memory wall [14] due to which most applications will be limited by memory performance. This has necessitated the use of aggressive techniques to hide memory latency.

Traditionally caches have been used to reduce the impact of memory latency. Caches work because of two reasons, a) Temporal locality and b) Spatial locality. Nowadays modern uni-processor designs have multiple levels of caches. But most programs still spend a lot of time stalling on memory accesses. In particular Instruction Cache misses can result in significant performance losses in highly pipelined processors [3]. When an instruction cache miss occurs, instruction issue is stalled for a period equal to the cache miss latency. This is in contrast to data cache misses, where hardware scheduling helps to partly hide the miss latency.

One of the most popular techniques used to hide memory latency is cache prefetching. The idea behind cache prefetching is to predict and prefetch (into the cache) in advance, future memory accesses, so that these accesses hit in the cache when the processor needs it. Conventional prefetching algorithms generally start prefetching only after a miss occurs.
For example, Sequential Prefetchers fetch the missing block and prefetch the next sequential block when a block misses in the cache, while Stride Prefetchers, prefetch the next 'n' blocks after a cache miss occurs. These techniques might not completely hide memory latency. We need a scheme that predicts cache blocks required in the future, and prefetches them in advance, so that they will be present in the cache when the program needs it. This will completely eliminate the latency of going to main memory or to lower level caches. This requires a clear understanding of program behavior and execution sequence.

Understanding program patterns and behavior at run time is the key to efficient utilization of various optimization techniques. Past research [10] has suggested that programs generally tend to have widely different behavior during different parts of their execution. More specifically it has been shown that programs execute as a series of phases [10, 11]. Past definitions indicate that a phase is an interval of execution during which a measured metric remains relatively stable. In [11], this idea was expanded to include all similar sections of execution irrespective of temporal adjacency. But the key point that must be understood is that the phase behavior seen in any metric at any given time is directly related to the code that is being executed. In [13], Sherwood, et al have come up with a unified profiling architecture that efficiently captures, classifies and predicts phase based program behavior at run time.

In this thesis, we propose a scheme that will use phase prediction at runtime to create Instruction Cache checkpoints that exploit program behavior to prefetch into the Instruction Cache. The intuition behind this scheme is that since phases tend to repeat fairly often during a program’s execution, we can effectively determine (using the phase predictor) not only when we transition to a different phase, but also the phase that we will enter into. Once we know this, we can pre-fetch instructions according to phase transitions. This will potentially eliminate all inter-phase Instruction Cache cold misses.

The following are the contributions reported in this thesis.

- A new improved Phase Prediction Architecture
- A novel approach of check pointing Instruction Cache values for every phase to potentially eliminate all inter-phase Instruction Cache cold misses
The organization of the thesis is as follows. Chapter 2 describes background work with respect to Static and Dynamic Phase Classification and Prediction Techniques. Chapter 3 focuses on a New Phase Prediction Technique that makes use of inherent program behavior. It then introduces the concept of an Instruction Cache Checkpoint and describes how it can be used to prefetch into the Instruction Cache. Chapter 4 presents the Simulation methodology and discusses results. Chapter 5 summarizes the conclusions of this thesis and describes future work.
Chapter 2

Background: Phase Behavior

2.1 Cyclic Behavior of Programs

Programs tend to execute in phases. They tend to go through different stages of execution, beginning with the Initialization or the startup phase, which is used to set up the data structures and generally prepare the system for execution. Once we complete Initialization and enter program execution, there are still sub-phases to be found. Programs are written in a modular fashion, often as a set of functions called within a loop, and where each function calls other functions again within loops and so on [11]. When programs are written in this manner, they have a strong cyclic trend of execution. In [10], it was demonstrated that most programs show cyclic behavior across various architectural features like Branch Prediction, Cache Performance, Value Prediction, and Address Prediction. Cyclic behavior can be defined as a repeatable or sometimes even a periodic pattern that can be seen throughout the program’s execution. For example, Figure 2.1 shows the time varying behavior of the SPEC95 program wave obtained from [10]. We can see that wave exhibits two distinct IPC patterns, a low IPC phase of 2 and a high IPC phase of 3. These two patterns repeat throughout its execution. Similar patterns are also exhibited by other architectural metrics during the course of the program’s execution.

So program execution as a whole can be considered as a collection of phases re-
peating over time. It is possible for us to automatically identify this phase behavior. The key point that must be understood is that the phase behavior seen in any metric at any given time is directly related to the code that is being executed, i.e. phase behavior at any point in a program’s execution sequence can be identified by examining the code being executed. It can also be said that phase behavior can be found and classified by examining only the ratios in which different regions of code are being executed over time.

### 2.2 Static Phase Tracking and Classification

This section provides a brief overview of how phase tracking and classification is done statically. A technique called Basic Block Distribution analysis [11] that is used to determine the cyclic behavior of an application is described. An introduction on how program behavior is characterized and classified based on observed patterns is also discussed. The terminology used in the rest of the thesis is defined below.
• interval: An interval is a section of continuous execution within a program. For the results presented in this thesis, the length of the interval is assumed to be 10 million instructions.

• phase: A phase is a set of intervals within a program’s execution that have similar behavior, regardless of temporal adjacency. Essentially what this means is that a phase can reappear multiple times through the program’s execution.

• Phase Tracking and Classification: Phase Tracking and Classification breaks down a program’s intervals of execution into phases with similar behavior.

• Phase Prediction: Phase Prediction determines what phase the next interval will be grouped into based on the phase behavior that has been seen in the past.

### 2.2.1 Basic Block Distribution Analysis

In [11], Sherwood, Perelman and Calder have come up with a static technique to divide programs into phases, based on Basic Block Analysis. A Basic Block is a piece of code that executes from start to finish, with a single entry and exit point. The frequencies with which basic blocks are executed in a program, is used as a metric to classify intervals into phases. The rationale behind this is that the behavior of a program at any given point is directly related to the code that is being executed. Basic Blocks provide us with this information. A program, when run for a certain amount of time will execute a certain number of basic blocks. Knowing this gives us a snapshot of that intervals execution behavior. It provides valuable information on where in the code the program is spending its time. By collecting the basic block distributions of two intervals, two separate execution snapshots are obtained, which can then be compared to determine how similar the two intervals are. If the snapshots are similar then these two intervals spend about the same amount of time in the same sections of code.

Basic Block distribution analysis works in the following manner. To characterize the program, a Basic Block Vector is created. A Basic Block Vector (BBV) is a single dimensional integer array that has an entry for each basic block in the program. The program’s execution is divided into intervals of 100 million instructions. Then each element in the Basic Block Vector is a count of the number of times the program has executed that
particular basic block in the corresponding interval. At the end of each interval, the basic block vector for that interval is normalized by dividing each element by the total sum of all elements in the vector. This gives us a proportion of the basic block execution. Once the Basic Block Vectors for all the intervals has been collected, it is possible to compare them, and classify them into phases. Figure 2.2 shows the architectural framework of a modern superscalar processor. Figure 2.3 describes how basic block distributions for intervals in a program are collected from a microarchitectural standpoint by using a superscalar processor. Since only basic block profiles of intervals are needed, a simple superscalar functional simulator can be employed (as opposed to using a timing simulator). The Basic Block Profiler obtains basic block information from the compiler at compile time. At run time, the superscalar functional simulator provides the Basic Block Profiler, PC information about every committed instruction. With the help of the basic block information provided by the compiler, the Basic Block Profiler determines whether this instruction is the first instruction of any basic block in the program. If so, it computes the basic block number for the current
committed instruction. This number is then used as an index into the Basic Block Vector and the corresponding entry is incremented by one, indicating that control has reached this basic block once.

**BASIC BLOCK VECTOR (BBV)**

Number of entries in a BBV ➞ basic blocks in an application
If # basic blocks = n, then size of BBV = n

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 0 | 1 | 2 | ... | ... | ... | n-2
|   |   |   |   |   |   | n-1 |

Super Scalar Architecture with Static Phase Classification

![Diagram](image)

Figure 2.3: Architectural Framework for the Static Phase Tracking and Classification Scheme

Comparing Basic Block Vectors is done by calculating either the Euclidean distance or the Manhattan distance. The Euclidean distance can be found by treating each vector as a single point in D-dimensional space. The distance between two points is simply the square root of the sum of squares. The Manhattan distance on the other hand is the distance between two points if the only paths you can take are parallel to the axes. This has the advantage that it weighs differences in each dimension more heavily (being closer in the x-dimension does not get you any closer in the y-dimension). The Manhattan distance is computed by summing the absolute value of the element-wise subtraction of two vectors. It was found in [12] that Manhattan distances accurately represented the differences in the Basic Block Vectors and hence is the preferred method when compared to the Euclidean method. Because all BBVs have been normalized, the Manhattan distance will always be
a single number between 0 and 2 (each BBV has been normalized to sum to 1). This number can then be used to compare how closely related two intervals of execution are to one another. If the Manhattan distance is closer to 0, the intervals tend to be similar, and if the values are closer to 2 then they tend to be different.

Basic Block Vector analysis was first introduced in [11] in order to find sections of code that closely represent the execution sequence of the entire program. This is because detailed simulation of an entire benchmark takes a lot of processing power and time. Often times researches simulate only a part of the program (for example the first 2 billion instructions) in order to speed up the process. But the simulated section does not necessarily represent the execution behavior of the program. We have seen from [10] that different parts of the program have different execution behaviors. So we need to find sections of code in the program that will accurately represent complete program behavior.

In order to determine these sections of code, Sherwood, Perelman and Calder introduce the notion of a Target Basic Block Vector (TBBV). This is nothing but the normalized basic block execution frequencies of the entire program. That is, the interval length for the Target Basic Block Vector is the length of the entire program. The aim of the study is to find a Basic Block Vector that closely matches the target BBV. This will then represent that section of the program that closely matches complete program execution behavior. Therefore, the Target Basic Block Vector is compared with the Basic Block Vectors of all intervals in the program. The interval that closely matches complete program execution will be the one that has the least Manhattan distance when comparing it with the Target Basic Block Vector and hence is the one that needs to be simulated.

### 2.2.2 Automatically Characterizing Large Scale Program Behavior

With the help of BBVs it is possible to analyze program behavior and hence discover common patterns that repeat over the course of program execution. In addition there is also a need for classifying these patterns so that this information can be used for various optimizations. Therefore, to find out how intervals of execution relate to each other a Basic Block Similarity Matrix is created [12]. This is a simple NxN matrix where N is the number of intervals in the program. Each entry (x, y) in the matrix gives us the Manhattan
distance between Basic Block Vector at interval x and Basic Block Vector at interval y. In order to group similar intervals into a phase a technique called Clustering is used. Clustering divides a set of points into groups such that points within a group are similar to one another (the metric used here is basic block execution frequency) and points in different groups are different from one another. In [12] the clustering method used was k-means. This was preceded by using random linear projection on the data set initially. The reason for this is because the complexity of the clustering problem increases with the increase in the number of dimensions (of the data set). Also the run time of the clustering algorithm is directly dependent on the number of dimensions. For clustering BBVs, the number of dimensions is the number of basic blocks. For the experimental data used in [12] the number of basic blocks ranges from 2756 to 102,038. This is obviously a very large working set and cannot be operated on. Random Linear Projection is therefore used to create a new low-dimensional space into which we project the data. Random Linear Projection breaks down the dataset into lower dimensionality without changing the underlying properties and similarity information the data carry.

Basic Block Vector analysis was used to find an interval in the program that closely represents the complete program behavior. In [12], this is further improved by obtaining multiple simulation points (i.e. multiple intervals that closely represent complete program execution). This is done by clustering the Basic Block Vectors into groups, wherein similar intervals are grouped into the same cluster, and intervals that are not similar to each other are placed in different clusters. Now a representative interval from each cluster is picked and executed. The interval that is closest to the centroid of a cluster is the one that closely matches the behavior of that cluster and hence is picked as a representative for that cluster. So the number of intervals that need to be executed in order to obtain a close representation of complete program behavior is equal to the number of clusters the program has been divided into.

### 2.3 Dynamic Phase Tracking and Prediction

The previous section talks about how phase classifications are done statically and provides an architectural framework for its implementation. Static Phase Classification has
shown that applications exhibit cyclic patterns of phase behavior. Although this information is useful, there are more opportunities to apply optimizations based on phase behavior if phase detection and classification can be done during run time when the program is executing. Section 2.3 describes an Online Phase Tracking and Classification scheme as proposed in [13]. Additionally it also presents an architecture that efficiently predicts phase based program behavior as implemented in [13]. Section 2.3 is divided into the following three subsections, (1) Dynamic Phase Tracking, (2) Dynamic Phase Classification and (3) Dynamic Phase Prediction. Figure 2.4 provides a microarchitectural framework for the online Phase Tracking and Classification scheme. It gives us an idea of how the phase tracking methodology can be implemented on a modern superscalar processor.

Figure 2.4: Architectural Framework for the Dynamic Phase Tracking and Classification Scheme
2.3.1 Dynamic Phase Tracking

In order to capture phase behavior, the basic block distribution for each profiling interval must be obtained. This information must be collected while the program is running without the help of a compiler. Also some basic blocks might have fewer instructions when compared to others. So the basic blocks need to be weighed according to the number of instructions in them. A close approximation of a basic block is the number of instructions between branches. Ideally all branches in the program as well as the count of the number of instructions between branches is tracked. This count gives us the size of each basic block so that it can be weighed accordingly. Since the metric of analysis is basic block distributions of intervals, code classification is based only on what is being executed. The classification process is completely unrelated to any architectural metric (like cache associativity or branch prediction strategy) and hence can be used as a code profiler for varying kinds of hardware and software optimizations.

The architecture of the dynamic phase tracker is shown in Figure 2.5. The Accumulator is a table with 32 entries that stores a snapshot of the execution sequence of the current profiling interval. The past Footprint table stores past phase behavior and has an entry for every phase that has been seen and recognized by the phase classifying architecture. The input to the architecture is the branch PC and the number of instructions executed since the last branch was seen. The branch PC is obtained from the superscalar fetch unit. The only other additional piece of hardware that is needed, is a counter that counts the number of instructions between branches. The branch PC is first converted into a number between 1 and 'N' buckets and this value is used to index into the Accumulator table. Experiments in [13] have suggested that 32 buckets are sufficient to distinguish between phases. Every entry in the Accumulator table contains a counter value corresponding to a bucket (or basic block). This counter is incremented (by the number of instructions in the basic block) every time control enters the corresponding basic block. The Accumulator table used in the Online Phase Tracking and Classification scheme is analogous to the Basic Block Vector that was used in the Static Phase Classification scheme. It gives us the basic block distribution for the current profiling interval. It must be noted that by reducing the branch PC to number between 1 and 32, and then indexing this into an Accumulator table we are effectively applying the random linear projection algorithm that was used in
the Static Phase Classification scheme. The random linear projection algorithm is used to reduce the dimensionality of the data set. The hashing scheme is a degenerate form of random projection. Indexing into the Accumulator table and incrementing the corresponding counter is the only operation that needs to be performed for every instruction in the execution sequence.

![Diagram](image)

Figure 2.5: Phase Tracking and Classification Unit [13]

2.3.2 Dynamic Phase Classification

After the profiling interval has elapsed, the interval must be classified. That is, it must be allocated (or grouped) into a particular phase. This requires a history of past phase information which is stored in the past footprint history table. The Accumulator values for a particular interval are first converted into a footprint for the corresponding interval. This footprint needs to provide a snapshot of the code that was executed in the current interval.
The best sanpashot that can be obtained is the complete Accumulator table data (as this accurately represents the code that was executed). But it is not necessary and is impractical to store the entire contents of the Accumulator as a footprint so that it can be compared with past values. It is enough to store compressed information from the Accumulator that still provides a snapshot of the code that was executed.

This footprint can be calculated by concatenating the most significant six bits of each entry in the Accumulator table. This provides a 24 byte (6 bits * 32 entries = 192 bits = 24 bytes) footprint entry that can be stored in the past footprint history table. Once the Accumulator has been reduced into a footprint, it needs to be compared with past footprints so that the current interval can be classified into a phase. The past footprint history table contains a footprint for every phase that has been seen and recognized. The current footprint vector is compared with each footprint vector in the table. If there is a match, the current interval is classified to be in the same phase as the past footprint vector. If there is no match, then a new phase has been detected. An entry for this new phase is created in the past footprint history table and current footprint is assigned to this entry. A new Phase Id is then given to this phase. Experiments in [13] have suggested that 20 is a reasonable number of entries that can be stored in the past footprint history table in order to distinguish between phases. Figure 2.6 shows that it is possible to capture over 90% of the program's execution if we track the top 20 phases in each application [13].

It must be noted that Dynamic Phase Classification is analogous to the clustering technique that was employed in the static phase classification scheme. Forming a footprint of the current interval and comparing it with past footprints is similar to grouping intervals into phases (or clusters) based on the clustering algorithm.

Finding a match in the past footprint history table requires a comparison of the current vector with the past vectors. Two intervals of execution that have very similar footprints can be considered a match even though they are not exactly equal. Therefore, a metric that defines closeness or similarity between vectors needs to be defined. This is nothing but the distance threshold. Setting a value for the distance threshold needs to be done carefully. If the distance threshold is too small, intervals will be classified into too many phases, and hence correct phase behavior will not be captured. On the other hand,
if the distance threshold is set too be too high, then intervals with different behavior will be classified into the same phase. In such a case applying optimizations to a phase that has varying behavior within itself might prove detrimental. In order to strike a balance between the two, a distance threshold of 1 million has been set in the simulation runs. Considering the fact that each interval has a length of 10 million, a distance threshold of 1 million indicates that a difference in the phase behavior will be detected if 10% of the executed instructions are in different proportions.

2.3.3 Dynamic Phase Prediction

This section describes how phase classification information can be used to predict the phase, the next profiling interval will be classified into. It is important for us to know phase changes, so that we can apply optimizations based on the phase we will transition into. For example, if we know the phase we are transitioning into it is possible for us to prefetch corresponding instructions into the instruction cache. Some other examples of optimizations
we can perform are, 1) Adapting the size of the data cache, 2) Dynamic processor width adaptation and so on. Figure 2.7 shows how the Phase Prediction scheme interacts with a modern superscalar processor and the Phase Classification Scheme described in the previous section. It gives us a microarchitectural view of how the Phase Predictor can be modelled on top of a superscalar processor.

Figure 2.7: Architectural Framework for the Markov Phase Prediction Scheme

Phase prediction is different from prediction of other architectural parameters like branch prediction and value prediction. Two pieces of information are necessary for phase prediction, 1) The last Phase Id seen and 2) The number of times prior to now that it has been seen in a row (run-length). A Markov predictor is perfectly suited for such predictions, as its prediction of the next state will depend on the last set of states that have been seen. The rationale behind using a Markov predictor is that phase behavior is characterized by many intervals of stable behavior interspersed with sudden and abrupt phase changes. The key is to be able to detect when these phase changes will occur and to know ahead
of time the phase we will be transitioning into. The problem is that phase changes are often preceded by stable behavior, and if only the last couple of intervals are considered it is not possible to tell the difference between sections of stable behavior that precede a phase change, and those sections that will continue to be stable. So we need to wrap up sections of continuous stable behavior into a single state. For this we can use a Run-Length Encoding Markov Predictor. The reason for this is because, run-length is a parameter that keeps track of constant phase behavior. For example, if the past 23 intervals have all been classified into the same phase, then the run-length is equal to 23. This parameter makes sure that constant phase behavior is captured as a single phase.

2.3.4 Run-Length Encoding 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. Figure 2.8, gives us the architecture of the predictor [13]. The Markov table consists of two fields, a tag field and a Phase Id field. The input to the architecture is the Phase Id of the current profiling interval (obtained from the Phase Classification Unit described in section 2.3.2), and the number of times prior to now, this Phase Id has been seen in a row (run-length). These two inputs are fed into a hash function. The lower order bits of the hash function provide an index into the prediction table. The higher order bits provide a tag. When there is a tag match, the Phase Id in the Markov Table provides a prediction of the next phase we will transition into. When there is a tag miss, the current Phase Id is used as the next Phase Id. The predictor is updated on only two occasions, 1) when there is a change in the Phase Id, and 2) when there is a tag match. An entry is inserted into the Markov Table when there is a change in the Phase Id, so that it can be predicted the next time around. For the second update case, the predictor is updated when there is a tag match because the observed run length may have changed. Intervals where the same Phase Id occurs several times in a row need not be stored in the table as they will be accurately predicted as “last Phase Id”.
Figure 2.8: Run-Length Encoding Markov Predictor [13]
Chapter 3

The Candidate Predictor and Instruction Cache Checkpoints

In chapter 3 we propose a New Phase Prediction Architecture that predicts multiple phase transitions at the end of every profiling interval. We then introduce the concept of an Instruction Cache Checkpoint that uses this new architecture to prefetch into the Instruction Cache. Chapter 3 is divided into three sections. Section 3.1 motivates the need for a New Phase Prediction Architecture. Section 3.2 introduces the New Phase Prediction Architecture and provides a framework for its implementation. Section 3.3 talks about the need for new prefetching techniques based on program behavior and proposes the concept of Instruction Cache Checkpoints.

3.1 Need for a New Phase Prediction Architecture

Experiments and simulations have suggested that when there is a phase transition, there are only a couple of phases that always occur as the next phase. The run lengths for most stable phases are also reasonably constant. Consider the benchmark swim. The phase transitions for swim are shown in Figure 3.1. The y-axis indicates the current Phase Id, the x-axis indicates Transition Phase Id and the z-axis indicates a Transition Counter. The
interpretation for the graph is as follows. Consider the current Phase Id to be 0. When we are in Phase 0, we can transition into either Phase 6 or Phase 17 as indicated by the Transition Phase Id axis. The Transition Counter gives us the number of times we have moved from current phase to transition phase. So by looking at the blue bars in the figure we can see that Phase 0 transitions into Phase 6, 158 times and to Phase 17, 4 times. The graph only shows those phases that have significant phase transitions. All other Phase Ids not shown in the graph have appeared less than two times in the execution sequence.

The point that must be noted from Figure 3.1 is that, all phases, transition to a fixed set of candidate phases, and the size of this fixed set is very small. For swim the maximum size of the Candidate Phase Id set is 4. This is for Phase Id 13, which transitions to Phases (1,6,8 and 15).

Figure 3.1: Phase Transitions in swim

If we further analyze the phase transitions of Phase 13, we can see that all transitions occur after a constant run-length (of Phase 13) has been seen. Figure 3.2 shows the run-lengths encountered when transitioning from Phase 13 to Phases (1, 6, 8 and 15). The
x-axis indicates the Candidate Phase Id set. This set is nothing but a set of Phase Ids that Phase 13 can transition into. The graph indicates that Phase 13 has either a run-length of 2 or a run-length of 3. A run-length of two means that, after seeing Phase 13 twice continuously, there is a transition to another phase. The y-axis indicates the frequency of the run-lengths noticed. This is a number that merely indicates the number of times we have seen a run-length.

![Run Length Comparisons For Phase Transitions in swim](image)

**Figure 3.2: Run Length Comparison for Phase 13 in swim**

From Figure 3.1 we can infer that phase transitions are fairly constant and Figure 3.2 shows that the run-lengths of phases are also fairly constant and are small in number. We can make use of these two characteristics of phase behavior to improve the prediction strategy. The run-length encoded Markov Predictor proposed in [13] and described in Chapter 2, gives out only a single prediction for every interval. This Phase Prediction architecture can be modified to make use of the inherent phase behavior that we have seen from Figures 3.1 and 3.2, to predict multiple phase transitions at the end of every interval. The architecture for the modified Phase Predictor is described in the next section.
3.2 A New Improved Phase Predicting Architecture based on Prediction Nodes: The Candidate Predictor

In order to improve the prediction strategy, we need to make use of the inherent phase transition characteristics described in the previous section. As indicated in the previous section the size of the Candidate Phase Id set is small. It is clear that phases, transition to only a small fixed number of Candidate Phases. So instead of predicting only a single Phase Id to transition into, we can predict multiple Candidate Phase Ids. This is feasible largely due to the fact that the maximum size of the Candidate Phase Id set seen in the benchmarks that were used for simulation does not exceed 4.

Figure 3.3 provides a top level view of how the Candidate Predictor interacts with the rest of the microarchitectural framework. The only input that the Predictor needs is the current classified Phase Id (the current classified Phase Id is nothing but the Phase Id that has been assigned to the most recently executed interval). This is obtained from the Online Phase Tracking and Classification Unit indicated in the figure and described in sections 2.3.1 and 2.3.2.

The architecture for the Candidate Predictor scheme is shown in Figure 3.4. The inputs to the predictor are obtained from the Online Phase Tracking and Classification Unit (sections 2.3.1 and 2.3.2). Each phase has a set 8 Prediction Nodes associated with it. The choice of the number of prediction nodes allocated per phase is purely based on simulation results. It has been seen that a maximum of 8 prediction nodes per phase is sufficient to capture phase transitions for all phases (for the set of benchmarks used for simulation). These nodes store prediction information based on the run-lengths. Every Prediction Node has the following fields.

- run-length: indicates the number of times prior to now a particular Phase Id has been seen in a row.
- Predicted Id: indicates the Phase Id we will transition into when we see the corresponding run-length
- Usage Counter: indicates the number of times we have correctly predicted a transition
from current Phase Id to Predicted Id for the corresponding run-length (gives us the number of correct Phase Transition predictions).

The data enclosed within a Prediction node for a particular phase suggests that when seeing this phase for a run-length indicated in the prediction node, we will be transitioning into a phase indicated by the Predicted Id field. The usage counter is merely a confidence counter that is needed when there might be multiple phase transitions for the same run-length.

The working of the phase predictor is best explained with an example. At the end of each profiling interval, we classify the interval to a particular Phase Id (this is done by the phase classification unit explained in section 2.3.2). Let us assume that the current interval has been classified to be in Phase 3. Let us also assume that the last five intervals have also been classified to be in Phase 3. So if we include the current occurrence of Phase 3, the current run-length is 6. The next step will be to predict the next phase we will be seeing. For this we look into the prediction nodes of Phase 3 that have a run-length equal to the current run-length (which is 6). Figure 3.5 indicates the number of prediction nodes
Figure 3.4: Modified Phase Prediction Architecture using Prediction Nodes

that have a run-length of 6 for Phase 3. As indicated in Figure 3.5 there have been three

different phase transitions from Phase 3, each transition to a different Phase Id. In the case
of a Single Candidate Predictor, we need to predict a single Phase Id we can transition
into. For this we pick the prediction node that has the maximum usage counter. From the
figure, it can be seen that the first prediction node has the highest usage counter, and hence
we will transition into the Phase Id indicated in the predicted Id field which is Phase 10.
For a 2-Candidate Predictor, we need to predict two phase transitions. The first prediction
comes from the node that has the highest usage counter. The second prediction comes from
the node that has the next highest usage counter. In our example the prediction for the
second candidate is from node 2 as it has the next highest usage counter (of 9) and the
prediction Id that it indicates is Phase 2. In case of ties with respect to the usage counter the Phase Id that has been seen last is picked. Suppose there are no nodes that have seen the current run length, then we predict no phase change. That is, we predict the next phase to be equal to the current phase. We update the prediction nodes on two counts, (1) when the prediction is correct, or (2) when we mispredict and there is a phase change. When the prediction turns out to be correct we need to increment the usage counter for the corresponding prediction node. When a phase change occurs and we mispredict we need to add this information into the prediction node list for the current Phase Id. If there are no empty prediction nodes, then we need to delete the node that has lowest usage counter value for that particular run-length. But in practice we have not seen this happen in any of the benchmarks used for simulation, and the number of prediction nodes has always been sufficient.

Figure 3.6 shows the efficiency of the 2-Candidate Phase Predictor when compared with the run-length encoded Markov Predictor. The x-axis shows the benchmarks used for simulation. The y-axis indicates the ratio of the number of phase transition mispredicts seen in the 2-Candidate Phase Predictor with respect to the Markov Predictor (normalized mispredicts = mispredicts in 2-Candidate Predictor/mispredicts in Markov Predictor). It can be seen from the graph that the average number of mis-predicts is down by about 50%. It must be noted that the 2-Candidate Phase Predictor gives out two predictions for every profiling interval, while the Markov Predictor gives out only a single prediction for every profiling interval. The interesting idea that we can infer from Figure 3.6 is that predicting multiple phase transitions will greatly help us in prefetching into the instruction cache, as we have more choices to make about the address blocks we need to prefetch.

### 3.3 Instruction Cache Checkpoints: Introduction and Working

This section describes the concept of an Instruction Cache Checkpoint and how it exploits program behavior to prefetch into the Instruction Cache. In architectural terms a checkpoint is any structure that provides precise state. Checkpoints are generally used
Figure 3.6: Efficiency of the new improved Candidate Phase Predictor

when we execute speculatively and have to roll back when the speculation is incorrect. In such scenarios we will roll back to the most recently checkpointed precise state. Instruction Cache checkpoints however, take advantage of the phase behavior of programs to store the most frequently used instruction PC’s in every phase. The reasoning behind this is that when we see the same phase the next time around, we will have enough information about the instructions that will most likely be executed. The advantage therefore, is that we can prefetch those instructions in advance.

Figure 3.7 gives us a top level design of how the Instruction Cache Checkpointing Unit interacts with the Phase Classification and Phase Prediction Unit. The Checkpointing Unit needs two important pieces of information.

- The Phase Id of the current profiling interval (or the classified Phase Id as indicated in Figure 3.7): This is obtained from the Phase Classification unit described in section 2.3.2.
- The Candidate Phase Id set of the current profiling interval: This is obtained from the Phase Prediction Unit described in section 3.2.
Figure 3.7: Architectural Framework for Prefetching using Instruction Cache Checkpointing

Figure 3.8 gives us the organization of the Instruction Cache Checkpointing Unit. The design consists of the following structures.

- Prefetch tables: There is a Prefetch table allocated for every phase. Prefetch tables are nothing but checkpoints of the Instruction Cache. They store the PC's of previously visited instructions in each phase. These prove as a starting point for instruction prefetching because Prefetch tables provide us information of address blocks of instructions that have been seen earlier and are likely to be seen in the future. Every time an interval has been classified into a phase, we essentially create a checkpoint of the instructions that have been executed in that interval. Therefore, when we see the same phase the next time around, we have already checkpointed the instructions that have executed the last time, and so we can exploit phase behavior and use this checkpoint to prefetch instructions that will be needed for the next profiling interval.

- Current Storage (CS) Table: During the execution of an interval, the Current Storage table learns and stores the most frequently used PC's for that interval. At the end
of every profiling interval, once the interval has been classified into a phase (by the Phase Classification Unit), we store the values of the Current Storage table into the Prefetch Table for the classified phase (the Phase Id for the classified phase is obtained from the Phase Classification Unit explained in section 2.3.2).

- **Checkpoint Instruction Cache (CI-Cache) and Superscalar Instruction Cache (I-Cache):** These are used to store the address blocks that have been prefetched from memory.

The working of the Instruction Cache Checkpointing Unit is as follows.

- **During the execution of an interval the most frequently used PC’s for that interval are captured and stored in the Current Storage Table.**

- **Once the profiling interval completes execution, the Checkpointing Unit interacts with the phase classifier to obtain the Phase Id for the profiled interval.**

- **The values in the Current Storage (CS) table are then copied into the Prefetch table that corresponds to the profiled interval.**
• The Checkpointing Unit simultaneously interacts with the Phase Predictor to obtain the set of Candidate Phases we will transition into.

• Finally the Checkpointing Unit uses the information stored in the Prefetch tables of the Candidate Phases to prefetch address blocks from memory and stores it in the I-Cache and CI-Cache.

Note that we prefetch only when we detect a phase change. Instead if we had prefetched after every profiling interval irrespective of whether a phase change, occurs or not then we will be thrashing an already warm cache for situations where we have phase run-lengths of more than 1.

The following example describes the prefetching strategy for a 2-Candidate Predicting architecture. The predictor provides us with two candidate predictions at the end of each profiling interval. The Checkpointing architecture consists of two cache structures.

• Superscalar Instruction Cache (I-Cache)

• Checkpoint Instruction Cache (CI-Cache)

At the end of the profiling interval, we access the Prefetch tables of both Candidate Ids. This gives us the address blocks that will be needed for the next profiling interval. We then prefetch these address blocks into both, the I-Cache and the CI-Cache. When the processor accesses the Instruction Cache in the fetch stage to retrieve Instructions it simultaneously looks into the CI-Cache as well. Now four different kinds of scenarios are possible

• I-Cache Hit, CI-Cache Miss

    Access block from I-Cache

• I-Cache Miss, CI-Cache Hit

    Access block from CI-Cache

• I-Cache Hit, CI-Cache Hit

    Access block from I-Cache

• I-Cache Miss, CI-Cache Miss

    Bring address block from memory into the I-Cache
Access block from I-Cache

Let us assume that the Phase Tracker and Classifier classifies the current interval into Phase 2. During the course of execution of the current interval, the Current Storage Table learns the most frequently used PC's. At the end of the profiling interval, the Phase Tracker and Classifier provides the current Phase Id (which is 2) to the Current Storage Table. The Current Storage Table now copies its contents into the Prefetch table for Phase 2. The Phase Predictor simultaneously provides the Checkpointing Unit, the Candidate Phase Id set for Phase 2. As we are considering a 2-Candidate Predictor, the Prediction results will contain 2 Candidate Phase Ids. Let us assume that this set now holds Phase 5 and Phase 17 as its first and second predictions respectively. We now start prefetching the contents of the Prefetch table for Phase 5 into the I-Cache (as this is the first prediction) and the contents of Prefetch table for Phase 17 into the CI-Cache (as this is the second prediction).

We must note that the combination of the I-Cache and the CI-Cache is analogous to using a 2-way I-Cache. But this combination works better than the 2-way cache in reducing the number of cache misses, primarily because of using a prefetching strategy that has been modelled on top of program behavior. Unlike traditional prefetching strategies that depend on the spatial locality of programs, Instruction Cache checkpointing exploits inherent phase behavior in a program. It prefetches instructions based on past phase behavior. Such a strategy is more superior because we know for a fact that the Prefetch tables hold address blocks of previously executed instructions, and since program behavior is inherently cyclic, we can confidently assume that the address blocks retrieved from the Prefetch tables are infact the ones that will be needed in the immediate future of the program's execution.
Chapter 4

Simulation Methodology and Results

4.1 Simulation Methodology

To perform the study, we collected information for 5 SPEC2k floating point benchmarks: ammp, mgrid, equake, swim and apsi. The above programs were chosen as they exhibited predictable and cyclic phase behavior. All programs were executed using the ECE 721 simulator. Phase Tracking, Phase Prediction and Prefetching architectures were modeled on top of the functional simulator. An I-Cache was plugged into the functional simulator to model Instruction Cache behavior. All intervals are of 10 million instructions. The Hit Latency modeled is 1 cycle while the Miss Latency is 100 cycles. Table 4.1 and Table 4.2, provide the basic simulation parameters.

Parameters for baseline architecture with 2-way cache

| I-Cache | 32k 2-way set associative, 128 byte blocks |

Table 4.1: Simulation Parameters for Base line Architecture Using Only Caches.
Parameters for 2-Candidate Prefetcher

<table>
<thead>
<tr>
<th>Checkpoint Instruction Cache</th>
<th>16k direct mapped, 128 byte blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-Cache</td>
<td>16k direct mapped, 128 byte blocks</td>
</tr>
<tr>
<td>Size of Each Prefetch Table</td>
<td>512 bytes</td>
</tr>
<tr>
<td>Size of Current Storage Table</td>
<td>3k 4-way set associative, 6 byte entries</td>
</tr>
</tbody>
</table>

Table 4.2: Simulation Parameters for Architecture using Prefetcher.

4.2 Results and Discussion

A single prediction strategy was initially simulated: The 2-Candidate Predictor Architecture. The results of the 2-Candidate Predictor are compared with the base line architecture with a 2-way Cache. Note that the 2-Candidate Predictor incorporates the Instruction Cache Checkpoint prefetcher while the base line architecture does not provide any prefetching. So essentially we are comparing the results of prefetching into the I-Cache vs. no-prefetching. Note that for all graphs indicating either number of I-Cache misses or Normalized I-Cache misses the x-axis indicates intervals in either multiples of 100 or 200. This means that every unit in the x-axis indicates either a billion or 2 billion committed instructions (as the length of the each interval is 10 million). When the y-axis indicates Normalized I-Cache misses, this indicates the ratio of the number of I-Cache misses when we prefetch with respect to the number of I-Cache misses when we do not prefetch (Normalized I-Cache misses = Misses when we prefetch/Misses when we don't prefetch).

Figure 4.1, is a graph indicating prefetch efficiency for the mgrid benchmark. The graph shows normalized values of prefetch vs. no-prefetch. So smaller is better. We see that the number of cache misses after prefetch is .97 times the number of cache misses without prefetch. We can see that although there is a decrease in cache misses when we prefetch, this decrease is not significant. This is due to the fact that mgrid does not show predictable phase behavior, although the marginal amount it shows is enough to bring the cache misses down. This can also be seen very clearly in Figure 4.2, which shows the number of cache misses with and without prefetching.
Figure 4.1: mgrid – Normalized I-Cache Misses.

Figure 4.6 shows Normalized Cache Misses for the benchmark ammp. ammp is a peculiar benchmark in the sense that it shows non-cyclic phase behavior in the initialization phase and then settles down to cyclic patterns in the execution phase. This is the reason for the first two bars in Figure 4.6, to be really high. The rationale behind such high values is due to the fact that the misprediction rate of the predictor is very high when it sees continuous non-cyclic phase changes in the initialization phase. Also the initialization phase for ammp is comparitively greater than all other benchmarks. It takes about 2 billion instructions for ammp to complete initialization and enter execution. The phase patterns between initialization and execution are also completely different. Therefore, initially the predictor starts learning the phase changes that take place in the initialization phase (which is really long - about 2 billion instructions). Once it enters execution, the phase transitions are completely different. So, the predictor has to start learning the new transitions from scratch. This can be seen in Figures 4.3, 4.4 and 4.5.
Figure 4.2: mgrid – I-Cache Misses (prefetch vs. no-prefetch).

Figure 4.3 shows the phase transitions in the initialization stage. Figures 4.4 and 4.5 show cyclic phase patterns in the execution stage. As we can see from these three figures, the initialization stage has different transitions when compared to the different execution stages, but there is a marked similarity between the phase transitions in the two execution stages. It must be noted that the same pattern follows throughout the entire program’s execution. Note that the x-axis indicates intervals.

Figure 4.7 on the other hand skips prefetching in the initialization phase and prefetches only during the execution phase. Now we can see a marked difference in the number of cache misses. The average normalized cache miss value reduces from 1.75 (Figure 4.6) to about 0.945 (Figure 4.7). This is because we first let the predictor settle down to cyclic patterns in phase behavior during the execution stage, and start prefetching only when the predictor is warmed up.
Figure 4.3: ammp: Phase Transitions in the Initialization Stage.

Figure 4.8 is a graph that compares I-Cache misses of swim with and without prefetching. swim, like ammp, shows rapid phase transitions in the Initialization stage. But this does not prove detrimental for prefetching as the length of the initialization phase is comparatively smaller. Also there is some cyclic behavior in the initial phases that the predictor is able to capture. Figure 4.9 shows phase transitions in the initialization stage. When we compare this graph with Figure 4.3 the length of the initialization stage is only about 500 million instructions. Once into the execution stage, swim shows excellent phase behavior as it keeps following a very periodic phase pattern as indicated by Figure 4.10

The benchmark equake on the other hand shows predictable cyclic phase behavior even in the Initialization stage. equake has a very periodic phase pattern that it follows. So the predictor warms up pretty quickly, and settles down to predicting cyclic trends that
Figure 4.4: ammp: Phase Transitions in the Execution Stage(1).

are seen throughout the program’s execution. Figure 4.11 and Figure 4.12 give us statistics that prove this point.
Figure 4.5: ammp: Phase Transitions in the Execution Stage(2).
Figure 4.6: ammp: Normalized I-Cache misses (with Initialization phase).

Figure 4.7: ammp: Normalized I-Cache misses (skip Initialization phase).
Figure 4.8: swim I-Cache misses.

Figure 4.9: swim: Phase Transitions in the Initialization Stage
Figure 4.10: swim: Phase Transitions in the Execution Stage.

Figure 4.11: equake: I-Cache misses.
Figure 4.12: \texttt{eqquake}: Normalized I-Cache misses.
Chapter 5

Conclusion and Future Work

5.1 Concluding Remarks

In this thesis we present a novel concept of Instruction Cache Checkpoints that exploits program behavior to scale the memory wall. This is accomplished by using the Phase Tracking and Prediction Architecture, that tracks, classifies and predicts phases and cyclic patterns during the programs execution. We then create checkpoints of the instruction cache for each phase that has been seen. These checkpoints provide a foundation for prefetching instructions into the Instruction Cache. With the help of the prefetching architecture we are able to reduce the number of Instruction Cache misses by an average of 17.8% for five of the SPEC 2k FP benchmarks. In addition to this we also present a new improved Phase Predicting Architecture that predicts multiple Candidate Phase Ids. Clearly, Instruction Cache Checkpoints is an initial move in trying to bridge the gap between processor speeds and memory bandwidth by automatically characterizing and exploiting program behavior at run time.

5.2 Future Work

- Data Cache Prefetching Using Checkpoints: The concept of prefetching into the Instruction Cache by exploiting program behavior can be extended so that we can
prefetch into the Data Cache.

- Branch Predictor Analysis using Phase Tracking and Prediction: Another area for further research includes analyzing branch predictor behavior for intervals in the program and checkpointing branch predictions for each phase that has been seen, so that this information can be used later on during program execution (when the same phase is seen again).
Bibliography


[8] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simu-


