2013

Analyzing and Quantifying Dynamc Program Behavior in Terms of Regularities and Patterns

Celal Ozturk
University of Rhode Island, celalozturk@my.uri.edu

ANALYZING AND QUANTIFYING DYNAMIC PROGRAM BEHAVIOR IN TERMS OF REGULARITIES AND PATTERNS

BY

CELAL OZTURK

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

UNIVERSITY OF RHODE ISLAND

2013

DOCTOR OF PHILOSOPHY DISSERTATION

OF

CELAL OZTURK

APPROVED:

Dissertation Committee:

Major Professor Resit Sendag

Gerard Baudet

Yan L. Sun

Nasser H. Zawia
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2013

ABSTRACT

Current processors employ aggressive prediction mechanisms to improve performance and reduce power. Most optimizations, however, are as a result of fairly ad-hoc observations or they primarily rely on heuristic. It is increasingly important to understand and quantify a program's dynamic behavior to effectively design next-generation prediction mechanisms. Although quantifying frequent behavior in an application's dynamic execution behavior is trivial in cases such as observing the frequency of each type of instruction, it is very challenging to summarize dynamic data reference behavior. As a result, most prediction mechanisms (data prefetchers, branch predictors, and other) employed in current processors today rely on heuristic-based analysis or ad-hoc observations. After some patterns are observed, a hardware decision is made and the design space of the predictor or multiple predictors is explored through simulation to determine the best performing predictor and its configuration. However, because the design is targeted for observed and/or anticipated patterns, some dynamic behavior is not captured and remains undetected.

In this study, Idesigned and implemented two comprehensive analysis tools to quantify dynamic program behavior in terms of regularities and exact patterns. My specific emphasis in developing these tools has been on processor design and computer architecture although the tools are sufficiently general to also be used by others in software development and security.

My PatternFinder tool integrates algorithms and mechanisms inspired by DNA discovery tools. I developed three flavors of this tool that required different

implementations due to specific optimizations for faster speed and smaller space. The first implementation targets the analysis of branch outcome patterns, which are sequences of 1s (ones) and 0s (zeros). The second implementation is a generalized version that allows 64-bit integers instead of 1-bit values as in the first implementation and thereby can be used to evaluate address and instruction patterns. Finally, the third implementation extends the second implementation to find patterns common to different input sequences.

My automatic source code analysis tool maps instructions to their corresponding data structuresat run-time without the need to analyze the program source code by hand. This tool is linked to the PatternFinder in that when specific instruction or data structure access patterns are targeted, automatic source-code analysis tool generates necessary input trace for the PatternFinder tool. Together the two tools that I develop can quantify pattern behavior in programs' dynamic execution.

Finally, I have demonstrated the use of the abovementioned two tools in summarizing branch and address patterns, and to identify the data structures that causes branch mispredictions for a set of program traces and SPEC CPU 2006 benchmarks.

ACKNOWLEDGMENTS

During the 5 years of my graduate work, I have received support from a great number of people. Dr. Sendag has not only been a great advisor to me, but also has been a good friend and has been supportive in my personal life as well. I would like to thank my brothers, Erdinc & Sinan Ozturk and my parents Gonul & Dursun Ozturk for their infinite support and encouragement which motivated me significantly towards getting my graduate degrees. I would like to thank my dear friends and long time roommates Danielle Dragon and Matthew Scarcella for their personal support and helping me get through tough times.

I would like to thank my best friends, Nevzat Atakli and Ozlem Kocabas, for being able to support me even from 5000 miles away and providing me the best vacation I could ask for when I needed time off from work.

I also would like to thank my landlord, Amy Higbie, for not not kicking me out of my house when she put the house on sale, just because she didn't want me to struggle with housing when I was about to finish my degree.

TABLE OF CONTENTS

ABSTRACT ..... **ii**

ACKNOWLEDGMENTS ..... **iv**

TABLE OF CONTENTS ..... **v**

LIST OF TABLES ..... **vi**

LIST OF FIGURES ..... **vii**

CHAPTER 1 ..... **1**

INTRODUCTION 1

CHAPTER 2 ..... **4**

REVIEW OF LITERATURE 4

CHAPTER 3 ..... **10**

METHODOLOGY 10

CHAPTER 4 ..... **31**

FINDINGS 31

CHAPTER 5 ..... **65**

CONCLUSION 65

REFERENCES **67**

BIBLIOGRAPHY **71**

LIST OF TABLES

TABLE PAGE

Table 1. Maximum non-overlapping patterns length and its coverage. 32

Table 2. Weighted average pattern lengths and weighted average repetition intervals 36

Table 3. Classification of benchmarks based on their spatial and temporal branch outcome regularities 40

Table 4. Maximum non-overlapping pattern length and its coverage 45

Table 5. Weighted average pattern lengths and weighted average repetition intervals 48

Table 6. Average percentage of mispredictions for each class 55

Table 7. Sample output showing top 10 most mispredicted branches 55

LIST OF FIGURES

FIGURE PAGE

Figure 1. Discovering the subsequences of sequence S→caabaaabacd having at least 2 occurrences in S.. 12

Figure 2. The block diagram of PatternFinder. 13

Figure 3. Misprediction contribution of Top 5, 10 and 15 most mispredicted branches in SPEC2000 18

Figure 4. Dynamic branch misprediction classification mechanism 20

Figure 5. Methodology Methodology for identifying changing function input type mispredictions 21

Figure 6. Methodology for identifying loop type mispredictions 24

Figure 7. Methodology for identifying array access and linked list type mispredictions 27

Figure 8. PatternFinder input/output size comparison 31

Figure 9. Cumulative distribution of hot pattern sizes (spatial regularity) for Spec CPU 2006 benchmarks 34

Figure 10. Cumulative distribution of hot pattern sizes (spatial regularity) for CBP traces 35

Figure 11. Coverage and confidence values for overlapping patterns in CLIENT02 39

Figure 12. Coverage and confidence values for non-overlapping patterns in CLIENT02 39

FIGURE PAGE

Figure 13. Coverage and confidence values for overlapping patterns in INT03 41

Figure 14. Coverage and confidence values for non-overlapping patterns in INT03 41

Figure 15. Coverage and confidence values for overlapping patterns in MM05 42

Figure 16. Coverage and confidence values for non-overlapping patterns in MM06 42

Figure 17. Coverage and confidence values for non-overlapping patterns in SERVER01 43

Figure 18. Coverage and confidence values for non-overlapping patterns in SERVER01 44

Figure 19. Cumulative distribution of hot pattern sizes for SPEC2006..... 47

Figure 20. Code snippet for hot load PCs in mcf..... 51

Figure 21. Longest common patterns changing over stream number threshold. 52

Figure 22. Breakdown of misprediction types for 4kB gshare predictor for SPECint, SPECfp benchmarks 54

Figure 23. Breakdown of the other category..... 55

Figure 24. Code snippet for a hot PC in gap, (a) assembly, (b) C code..... 56

Figure 25. Code snippet for top hot PC in gap, (a) assembly, (b) C code 58

Figure 26. Code snippet for top second hot PC in parser (a) assembly, (b) C code ... 60

Figure 27. Code snippet for one of the hot PCs in gcc, (a) assembly, (b) C code 62

CHAPTER 1

INTRODUCTION

Making the common case fast is a design principle that has been used in microprocessor design for decades. This principle applies when determining how to spend resources, since the performance impact on making some occurrence faster is higher if the occurrence is frequent. Although quantifying frequent behavior in an application's dynamic execution behavior is trivial in cases such as observing the frequency of each type of instruction, it is very challenging to summarize dynamic data reference behavior [1]. As a result, most prediction mechanisms (data prefetchers, branch predictors, and other) employed in current processors today rely on heuristic-based analysis or ad-hoc observations. After some patterns are observed, a hardware decision is made and the design space of the predictor or multiple predictors is explored through simulation to determine the best performing predictor and its configuration. However, because the design is targeted for observed and/or anticipated patterns, some dynamic behavior is not captured and remains undetected.

It is increasingly important to have a complete understanding of dynamic program behavior in order to make more informed decisions early in the design process. An attempt to quantify regularities in a memory address trace was made by Chilimbi in [1]. This was the first study to quantify the observation that extended memory access sequences recur using a hierarchical compression algorithm, called SEQUITUR [2]. Several researchers then proposed ways to exploit this behavior [3-5]. Surprisingly,

after a decade, their analysis has remained one of the most detailed for quantifying hot memory streams. Unfortunately, frequent pattern analysis for hot streams with SEQUITUR is very limited and can be misleading. SEQUITUR forms a grammar to summarize an input sequence in compressed form; however, there is no guarantee in finding most important or relevant non-overlapping patterns. It is also not suited for finding overlapping or approximate patterns. In this study, inspired by DNA discovery tools [6-8], I adapt and revise the methods motivated by suffix trees [6] in order to develop comprehensive pattern discovery tools targeted for computer architecture. Suffix trees have several advantages over SEQUITUR in designing such a tool, which I discuss in Section 3.

In this study, I present a pattern analysis tool ,PatternFinder, and the results produced by the tool that quantify exact overlapping and non-overlapping patterns in dynamic program behavior. Exact patterns are most relevant to analyze branch outcome behavior, and provide insights into the predictability and relative importance of patterns. PatternFinder can also quantify exact patterns in dynamic data reference behavior (and do so more rigorously than SEQUITUR-based analysis shown in [1]). However, a true insight can only be gained by discovering approximate patterns because a few changes in a particular data reference pattern must not nullify importance of that pattern. My observations with memory access patterns suggest that one must target a different set of pattern language. Unlike branch prediction in which the next outcome must be predicted correctly, prefetching system predictions are assumed successful if prefetched data is accessed by the processor in near future and thus predictions must not need to be correct for the next consecutive access to be

useful. Since my current version of the PatternFinder is not capable of analyzing approximate patterns, in this study, I focus on exact overlapping and non-overlapping branch outcome patterns and left approximate pattern analysis as a future work .

This thesis makes the following contributions:

1) It presents design and implementation of a novel pattern analysis tool for computer architecture research.

2) It explores and quantifies non-overlapping patterns in dynamic branch outcomes for spatial and temporal branch stream behavior.

3) It explores and quantifies non-overlapping patterns in address request patterns for spatial and temporal address stream behavior.

4) It quantifies overlapping branch outcome patterns that have implications on predictability.

5) It presents a methodology for dynamic source code analysis

6) It explores and quantifies non-overlapping patterns commonly seen in multiple streams.

CHAPTER 2

REVIEW OF LITERATURE

I first motivate my effort by examining the state-of-the-art in data prefetching and branch prediction. Although many prediction mechanisms' success depends on frequent patterns, there are no comprehensive studies for finding and summarizing patterns for dynamic execution behavior of programs.

Much processor design research is based on observing regularities in benchmark applications and design mechanisms to exploit this behavior. There are many examples. Caches are based on temporal (code and data reuse) and spatial (arrays, etc.) locality of instruction and data reference accesses. Branch prediction is based on regularities in branch outcomes and targets (e.g. loops, local and global correlations). Prefetching is based on data reference regularities (stride patterns, etc.).

2.1 Branch Prediction

Modern microprocessors use aggressive branch predictors to minimize the performance impact of control-flow changes. Two-level branch predictors, explicitly track global or local branch history patterns, and for each branch, make different predictions depending on the recent history [5-7]. Within most programs, some branches are best predicted using global history, while others are best predicted using local history. A processor that only implements one or the other type of predictor therefore penalizes some branches. A hybrid predictor includes multiple predictors

[11-13], with some way to choose which predictor to use at any given time. Recent works, such as O-GEHL [14] and LTAGE [15] exploit much longer histories than prior predictors. These predictors employ multiple prediction tables indexed with different length folded histories. Several others [16-20] target longer histories based on neural networks.

Although there is extensive work in branch prediction, most analysis done has been heuristic-based. After observing some patterns in benchmark programs, a hardware design decision is made and design space of the predictor is extensively explored through simulation to determine usefulness. However, because design is targeted for observed and/or anticipated patterns, some dynamic behavior is not captured and remain undetected. In this study, I present a framework for pattern discovery in branch outcomes (or different events in the dynamic execution behavior of benchmarks) to guide in making more informed decisions early in the design process.

2.2 Data Prefetching and Memory Access Patterns

Hardware data prefetching is a well-known technique to help alleviate the memory wall problem [22]. Many general purpose microprocessors rely on data prefetching to improve performance for memory-intensive workloads. Most of the early prefetchers [23, 24] were based on sequential prefetchers, which prefetch sequential memory blocks relying on the fact that many applications exhibit spatial locality. Although sequential prefetchers work effectively in many cases, applications with non-sequential data access patterns do not benefit from sequential prefetching.

That motivated the research on more complex prefetchers that try to capture the non-sequential nature of these applications. Prefetching techniques targeting pointer-based applications have been studied in [25, 26]. Joseph and Grunwald [27] study Markov-based prefetchers. In recent years, [4, 5, 28] advocate memory streaming for arbitrarily irregular yet repetitive address patterns. These papers provide a way to exploit the fact that there are hot data streams (observed by SEQUITUR), which arise as applications iterate over data structures, even arbitrarily irregular ones. The success of these methods depends on understanding complete access pattern behavior of applications. A preliminary version of a memory trace analysis was done by Chilimbi in [1]. This was the first study to quantify the observation that extended memory access sequences recur using a hierarchical compression algorithm (SEQUITUR), developed by Nevill-Manning and Witten [2]. Larus [29] used SEQUITUR in his earlier work to construct Whole Program Paths (WPP), which are a compact, yet analyzable representation of a program's dynamic control flow. However, analysis in [1] and [29] is limited in that only exact patterns are investigated. As mentioned in Section 1, approximate patterns provide better insight into memory access behavior. Unfortunately, myPatternFinder tool can also not discover approximate patterns at this time. Therefore, in this study, I focus on analyzing dynamic branch behavior and left exploration of approximate patterns as future work. PatternFinder explores overlapping patterns for predictability as well as non-overlapping patterns for stream behavior. SEQUITUR is not suited for finding overlapping patterns. As a result, instead of using Sequitur algorithm as in [29] and [1], for reasons described above, I adapt and revise suffix tree [6] algorithms, which have been successfully used in text processing and bioinformatics.

2.3 Pattern Discovery Algorithms

Sequence pattern discovery is a research area aiming at developing tools and methods for finding a priori unknown patterns in a given set of sequences, patterns that are frequent, unexpected, or interesting according to some formal criteria. Brazma et al. [31] describes the overall pattern discovery with three sub-problems.

1) Choosing the appropriate language to describe patterns.
2) Choosing the scoring function for comparing patterns.
3) Designing an efficient algorithm.
4) Customizing the pattern finding process

Choosing the appropriate language to describe patters is very important because it has direct impact on the formation of the output. In many cases, the results of the pattern tool must be post-processed in order to extract the desired information. If the language is not carefully chosen, the program output may not be as useful.

Choosing the scoring function is very crucial for the pattern tool. In a long stream there can be thousands of patterns overlapping with each other and for a non-overlapping pattern analysis, only one can be chosen to be included in the output. Thus, a decent scoring function must be implemented in order to choose the best pattern possible among the overlapping ones.

Due to the nature of benchmarks, the input streams can be very long. Because of that, the efficiency of the pattern finding algorithms is really important. An inefficient algorithm would not be able to process long streams in a reasonable amount of time.

And finally, for the pattern finding process, it is very useful to have a customizable one for the tools which are meant to be available open source. This type of tool can easily modified per users' needs and target more user specific information. Tools like SEQUITOR really suffer in this case because they use a Context Free Grammar in order to find patterns, which is a fixed algorithm and does not allow the user to modify the algorithm easily.

I followed Brazma's methodology for developing the PatternFinder.

IBM Bioinformatics Research Group developed the TEIRESIAS algorithm for discovery of patterns in biological sequences that operate in two phases: scanning and convolution [32]. During the scanning phase, elementary patterns with sufficient occurrence frequency are identified. These elementary patterns constitute the building blocks for the convolution phase and are combined into progressively larger patterns until all the existing, maximal patterns have been generated.

Some of the most efficient algorithms capable of discovering discrete patterns such as substrings of any length, are based on the suffix tree data structure [6, 33]. Suffix trees are used to accelerate many string operations [34] by indexing texts (sequences) in a way so that query times would not depend on the size of the indexed text. In the suffix tree all possible sub-words can be read from the top of the tree-structured index regardless of original text size. There are many bioinformatics applications of suffix trees [78, 80-81]. The direct link to pattern discovery methods is given by the fact that all possible substrings (patterns) are presented in this tree

structure. Suffix tree based approaches and extensions have been used for approximate string matching, finding the longest common substring of two strings and finding all common substrings in a database of strings. Such queries are essential for many applications such as bioinformatics [6], time series analysis [35], document clustering [36] and compression [37].

In this study, I apply the methods motivated by the suffix trees for pattern discovery from dynamic program execution traces. For the discovery of the most frequent patterns I adapt the write-only top-down algorithm for constructing the suffix trees [38]. This approach is simple and easily modifiable, as different branches of the suffix tree can be constructed independently from each other. In its implementation, only those branches of the suffix tree need to be constructed which are actually accessed by search procedures. Traditional linear-time algorithms [33, 39] maintain complex data structures and they all construct the tree in a very specific order, thus making modifications into the search order hard or impossible.

CHAPTER 3

METHODOLOGY

PatternFinder generalizes the WOTD algorithm for constructing and reporting all the patterns from the defined pattern language. Efficient pruning of the search space guarantees that only these patterns that are frequently present in input data, are constructed and evaluated.

PatternFinder takes as input a sequence of numbers and reports all patterns that occur in this input sequence, their pattern lengths, where they occur, their input coverage, their user-defined importance, and some other user-specific metrics. On average, 99.9% of the input sequence is covered with a minimum pattern length of 2 because subsequences that occur only once and single data points are not considered patterns. Therefore, in terms of compression, unlike SEQUITUR, PatternFindercan only provide lossy compression and therefore one cannot use PatternFinder output to fully reconstruct the input sequence.

PatternFinder can perform customized queries for finding patterns of interest based on pattern lengths, coverage and randomness. The run-time is dependent on this customization. On average, it is fast and provides results within minutes for 100M-long input traces that have been analyzed for this study.The tool is carefully designed for speed and minimal memory space requirements. Although faster implementations are possible, they require vast memory space for keeping the whole suffix tree in

memory. My implementation allows us evaluate the 100M-long traces with a workstation using 8GB of memory.

In this study, the focus patterns are the ones that occur at least k times in a sequence,S. I aim at a solution that is faster for larger values of k, keeps the space requirement relatively low, and at the sametime is simple to understand and implement. The solution is motivated by the WOTD algorithm for suffix tree construction. I represent the algorithm for constructing the$O(n^2)$timeand space, suffix trie instead of the compact suffix tree. The trie variant is easierto describe and implement, as well as it allows us to generalize this algorithm fordiscovering patterns from more complex pattern classes.

My algorithm builds the suffix trie for the input sequence S in a systematic order, e.g., in the breadth-first order, level by level. An advantage in constructing the tree in this way is that all children of a node are inserted in one step. There is no need for multiple visits to nodes in differentparts of the trie and the physical implementation of tree nodes can be optimizedby knowing exactly how many children the node will have. Example of such atrie construction is in Figure 1 for an input sequence S→caabaaabacd.

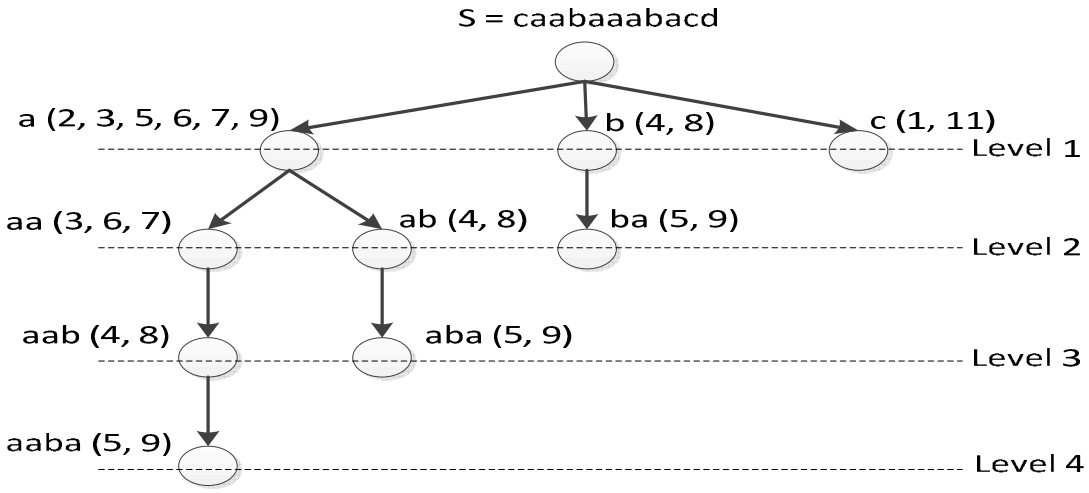

Figure 1: Discovering the subsequences of sequence S→caabaaabacd having at least 2 occurrences in S.

3.1 Summarizing the Input Sequence with Non-Overlapping Subsequence Patterns

I used the suffix tree described in Figure 1 as the main data structure to also find the non-overlapping patterns. The overall process of finding non-overlapping patterns is shown in Figure 2. To summarize the input sequence with non-overlapping subsequence patterns, PatternFinder first finds the longest pattern/s according to some user-defined criteria (e.g., occur at least k times or maximum pattern length of L that occur at least k times, etc.). Occurrences of this pattern cover parts of the input sequence. This step is repeated, each time in the remaining parts of the input, until no patterns longer than some user-defined length are found (e.g., minimum pattern length of 2) or some input coverage criteria is met (e.g., 90% input coverage). Each of these steps are called an iteration. PatternFinder increases its coverage of the input sequence by running iterations until no patterns are left or a predetermined stopping condition is reached. Each of the iterations covers some parts of the remaining input, which is

shown as pattern placement and input reduction in Figure 2. First iterations are slower since they go deeper in the tree finding longer patterns.

Eliminating infrequent patterns and overlapping occurrences of pattern within a node:

As shown in Figure 2, during the construction of the suffix tree for a particular iteration,at each level, nodes for patterns that occur less than k times are deleted and are not evaluated further. In addition, at each level, the algorithm detects and eliminatesoverlapping occurrences of patternswithin each node. This can be done in linear time because position lists for each node is kept in order.This eliminates significant number of patterns from the suffix tree, which in turn improves processing times, without significantly changing the pattern behavior observed in the program trace.

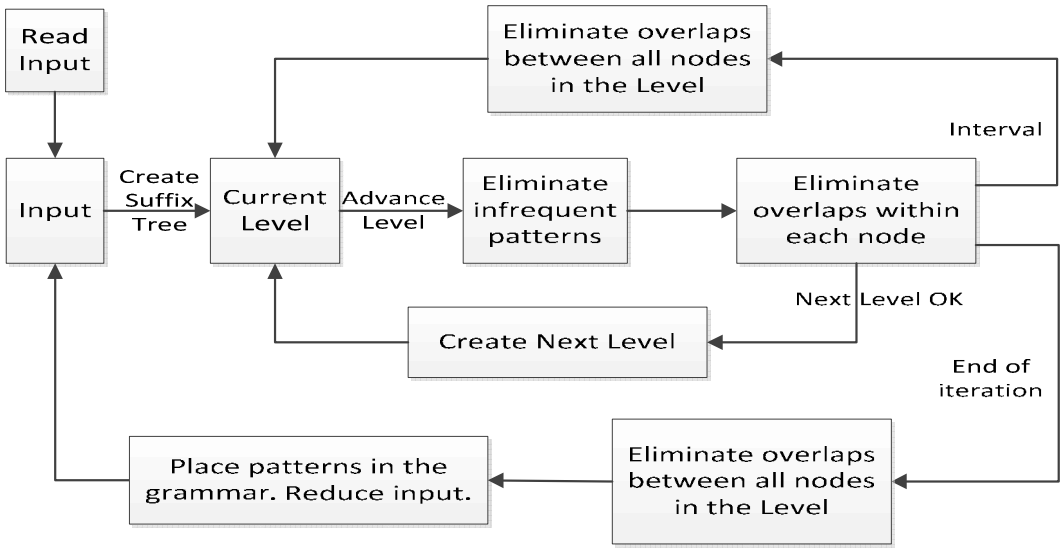

Figure 2: The block diagram of PatternFinder

Eliminating overlaps between nodes:

Although the overlapping patterns are eliminated, as described above, for each node's patterns, different patterns in different nodes at each level of the tree may haveoverlaps with each other. These overlaps are not eliminated at every level because it is not possible to determine which patterns are more valuable without going deeper in the suffix tree. However, for reasonable processing times, there is not much need to eliminate these overlapping patterns. For these reasons, a hybrid solution is chosen:an input parameter, interval, specifies the elimination interval for overlapping patterns at different nodes of the suffix tree. By doing this at every such interval, patterns are given more chance to grow and to stay in the tree longer until it is more clear to observe if they are valuable. My experiments show that doing this every 20 levels produced the best results. To lower chances of eliminating important patterns, my algorithm also computes the earliest level this overlap eliminationcan be started. Hence, often elimination starts after the program reaches level 100(pattern length of 100), at which point,overlap elimination is applied every 20 levels. Finally, my algorithm also uses an input parameterthat specifies the minimum number of elements required in the suffix tree to enable eliminationof the overlaps. Because if the suffix tree is not very large, there is no need for elimination – this process can hurt the performance instead.

The overlap elimination operation needs to decide which patternsat the particular level of the suffix tree under investigation are more valuable. Different scoring functions (e.g., pattern with highest occurrence frequency) can be applied to

determine the hot patterns. Starting with the hottest pattern in the list of patterns at the tree level, the algorithm reserves the space (that corresponds to the locations in the input sequence) covered by this pattern's occurrences. Other patterns which partially or fully reside in that space(i.e., overlapping occurrences of other patterns with the hottest pattern's occurrences) need to be eliminated. After this elimination, new hottest pattern in the remaining list is found and the procedure is repeated until there are no more frequently occurring patternsremain in the list.

Early termination of an iteration:

An iteration terminates at the longest pattern (that is, next level does not have any frequently occurring patterns) if there are no conditions to terminate it earlier. This gives priority to longer patterns even if their coverage might be too small. It also increases the processing time. I introduced three conditions where iterations must be terminated early.

1) The first stopping condition for an iterations to reach the user-given maximum pattern length.
2) Another parameter allows the program to stop the iteration when the level's non-overlapping coveragefallsunder a certain threshold. I define non-overlapping coverage as the minimum area of the input that is covered with the current patterns without any overlaps. Non-overlapping coverage is computed at each iterationrelative to the remaining input which has not been covered by the previous iterations.

3) In either case when there are no early termination conditions or with conditions mentioned in 1 and 2 above, there isno guarantee to cover the best area with the best combination of patterns at each iteration. In order to automatically find a good spot to stop the iteration, after observing fluctuations in coverage between levels, I define another parameter; average percent drop in coverage per eliminated pattern. If this drop is over a certain threshold, it suggests that significant patterns have been deleted from the tree at this level, so placement must be done for the previous level and therefore iteration terminates at the previous level.

Early termination of PatternFinder:

The final parameter for early termination is for the whole process. Because 90/10 locality rule states that a program spends 90% of its execution time in only 10% of the code, a user may want coverage for only 90% of the input, which greatly improves the processing time. Therefore, I introduced a new parameter for minimum overall input coverage. According to this parameter, the program stops looking for patterns when the desired coverage, usually chose as 90%, has been reached. Which saves a lot from execution time and also prevents very small patterns from being included in the output.

3.2 Targeting Specific Instructions by Dynamic Source Code Analysis

The input streams for pattern analysis tools can come from many different sources. They could be branch outcome patterns of whole programs, branch outcome patterns for a single branch, address request patterns, address request patterns for a single load instruction, function call chains, etc. A user might even be interested in just using patterns for instructions doing linked list traversals. If the source code and debug symbols for the benchmark/program is available, one can easily extract this information and collect the specific trace needed. But if the source code and debug symbols are not available, it would be extremely difficult to gather this information. In order to solve this problem, I've implemented an extension to the tool, which identifies branches which are dependent on array accesses, pointer references, linked lists, constant loops, varying count loops and function calls. Using this extension, one can easily generate a trace for specific targets like; function call chain in a program,address tracefor the linked list traversals, branch outcome trace for branches dependent on array accesses, etc. It is also possible to detect most mispredicted branches and generate a trace for each one of them. This is very useful because few most predicted branches cover most of the branch misprediction in the whole benchmark for almost all benchmarks.

Figure 2 shows how hot branch PCs contribute to the overall mispredictions for SPECint, SPECfp, and Mibench benchmarks, respectively, when a 4kB gshare branch predictor is used. On average, for SPECint, top 5, top 10, top 20 static branches cause 39%, 53%, 65% of all mispredictions, respectively. For SPECfp, top 5, top 10, top 20 static branches cause 71%, 83%, 92% of all mispredictions, respectively. Finally, for Mibench, top 5, top 10, top 20 static branches cause 67%, 79%, 87% of all

mispredictions, respectively. Majority of mispredictions are caused by few hot branches.

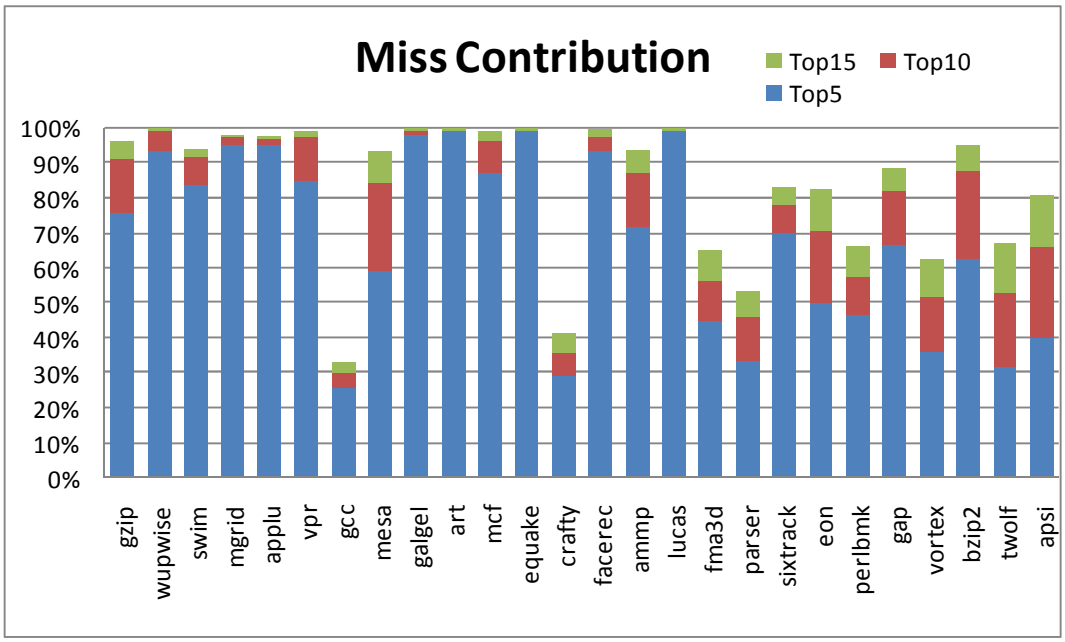

Figure 3.Misprediction contribution of Top 5, 10 and 15 most mispredicted branches in SPEC2000

For the branch misprediction classification, I repeat Skadron's run-time branch misprediction classification for SPEC CPU 2000 and Mibench benchmarks with a 4kB (i.e., 16K entries) gshare [6] predictor. Mispredictions are classified into five groups: conflict, training, wrong-history, needs both history, and other. To classify a branch's misprediction type, the program performs a sequence of tests as described in [3]. Each branch flows down this sequence of tests until it is categorized or falls through as a misprediction that could not be categorized. The classification progress is goes with this flow:

1. The prediction starts with a gshare predictor. If the prediction is incorrect, misprediction classification starts.
2. The first step is to test if a gshare predictor with no aliasing could predict the branch. When the gshare predictor that is free of aliasing is implemented, the number of table entries is kept the same (i.e., same history size is used). However, each table entry remembers all branch references to that entry by

updating their corresponding 2-bit counters. Therefore, the predictor is free of destructive interference. If this predictor was able to provide correct prediction, the misprediction falls into the conflict category. That is, the predictor under test would predict the branch correctly, but a destructive interference prevented the predictor from doing so, and as a result, a conflict misprediction has occurred.

3. The second step uses a 2-bit predictor to predict the branch. If this prediction is correct, it suggests that the branch has not been predicted correctly before because the branch predictor under test has long training time. This is a misprediction due to training (as mentioned in [3], this is an approximation.)
4. If the branch misprediction has not been classified in the previous steps, it may have happened because the branch needs local history. If a local predictor of the same size, but free of interference (logically infinite sized predictor), predicts this branch correctly, it suggests that global history is not appropriate for this branch because it needs local history, i.e., it is a wrong type history misprediction.
5. If still not classified, an interference-free predictor that uses both global and local histories is tested if it can provide correct prediction for this branch. A correct prediction in this case suggests the branch needs both types of history, and the misprediction is classified as "needs both types of history". However, if the branch mispredicts with this predictor also, it falls into the group of other mispredictions as it cannot be classified by this taxonomy.

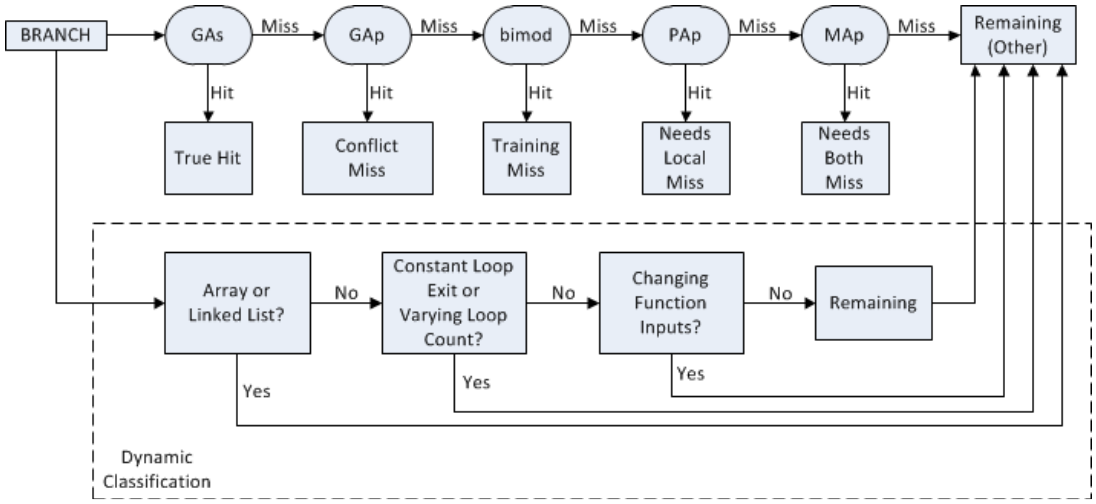

Figure 4. Dynamic branch misprediction classification mechanism

By running several predictor organizations of increasing sophistication simultaneously, the simulator performs the abovementioned cascade of tests until the branch either predicts correctly, or the misprediction fails all tests. Remaining branches are either inherently difficult to predict, or fall into a category not included in this scheme (e.g., need longer history). This process categorizes each dynamic branch's behavior for gshare branch predictor.

In addition to Skadron's classification, I add 5 new categories; changing function inputs, varying loop counts, constant loop exits, array accesses and linked list traversals. In this section I describe how I define these classes and how branches are classified using these new classes.

3.2.1 Changing Function Inputs

Many branches are dependent on the values of parameters that are passed to the function which they belong to. Due to optimizations done by compilers, it's not a

straight-forward task to know if a register is a function input, in some cases it's not even possible. Due to this problem, I simplify what should be considere a function input. I define a function input as a register which is read but has never been written within that function before. Then, for every instruction that uses this register as a source register, the algorithm mark the destination register as being dependent on the function input. The algorithm keep following this chain until the function returns. By doing this the algorithm is able to check if a register is dependent on a function input immediately. Let's say a program executes a branch that uses R3as a source register. The algorithm checks the data structure to find out if the R3 has its "Function Input" flag set. If it's set, branch gets identified as a candidate for being classified as "Changing Function Inputs". The algorithm has to update these values for every instruction and it has to create new data structures for each function call, and not destroythese data structure until that function returns. Every function has its own data structures preserved even if they make calls to other functions.

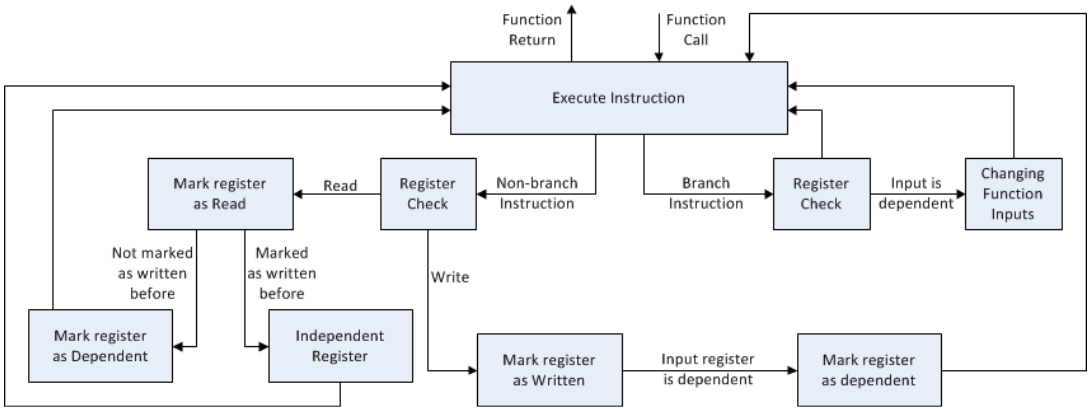

Figure 5. Methodology for identifying changing function input type mispredictions

The algorithm stores the data for each function instance separately. Let's say function A calls function B. Right before the call we have the data specifically for the Function A. The algorithm keeps them stored because B will eventually return and A

will continue executin. After B is called, new data structures are created for the function B, and those data structures will be used until B calls another function or B returns. When the execution comes back to A, we continue with A's data structures from where we left. We use a stack for the function call chain. Every time a function is called, we insert a function node on top of the stack. Every time a function returns, we remove the top function node from the stack. For every instruction other than CALL or RETURN, we do the computations on the top element of the stack, because we know it's the function being executed currently. These are the data structures we use to follow the function dependency chain:

- Array of 32 for integer register writes
- Array of 32 for floating register writes
- Array that has a flag called "Function Input"

This is the flow of the algorithm:

Fetch the instruction and figure out what registers are written and what registers are read.If a register is read, check the "register writes" array for that specific register. If the "register writes" array says it's not written in the function: It's considered as a function input because the function uses it without initializing. We should visit the "register data structure array" and set this register's "Function Input" flag. Since the destination register's new value is also dependent on this register, we should set the destination register's "Function Input" flag as well.

If the "register writes" array says it's written in the function: We should check the "register data structure array". If this source register's "Function Input" flag has been

set, then we should set the "Function Input" flag of the destination register, because it's new value depends on a register that has been marked as a "Function Input" before. If a register is written: We should check if any of the source registers has the "Function Input" flag set. If so, we should set the "Function Input" flag of this destination register. Otherwise we should clear the "Function Input" flag of this register since it's now written with registers that are not dependent on the function input.

3.2.2 Constant Loop Exits and Varying Loop Counts

Many of the branch mispredictions are caused by loop branches. Especially loops with small iteration counts counts have significant branch misprediction counts. Many predictor designs have targeted loop branches to predict loop exits to eliminate these mispredictions. It's not a straightforward task to identify loop branches because of compiler optimizations and varying iteration counts of loops. In order to identify a loop branch , we cumulatively store counters for taken and not taken information. If the branch is taken and the previous branch outcome was taken as well, we increment the last counter by one, which is the last taken counter. If the branch is taken and the previous branch outcome was not taken, we add a new counter to the branch and give1 as the value, which is the new not taken counter for the branch. If the branch is not taken and the previous branch outcome was not taken, we increment the last counter by 1, which is the last not taken counter. If the branch is not taken and the previous branch outcome was taken, we create a new counter which is the new taken counter for the branch. After enough data is collected, we look at these counters to see if we

can identify a loop. If the branch outcome counters follow a pattern as taken counters are always more than 1 and not taken counters are always 1, we identify the branch as a loop. Also if the branch outcome counters follow a pattern as taken counters are always 1 and not taken counters are always more than 1, we identify the branch as a loop. After identifying the branch as a loop, we investigate the counters to see if the loop has a constant iteration count or a varying iteration count. If the branch is following a pattern of taken-nottaken count pairs, we mark the branch as a constant loop exit branch. If the branch is following varying iteration counts, we mark the branch as varying loop counts. It's important to note that some none-loop branches may also be identified as loop branches because of their outcome.

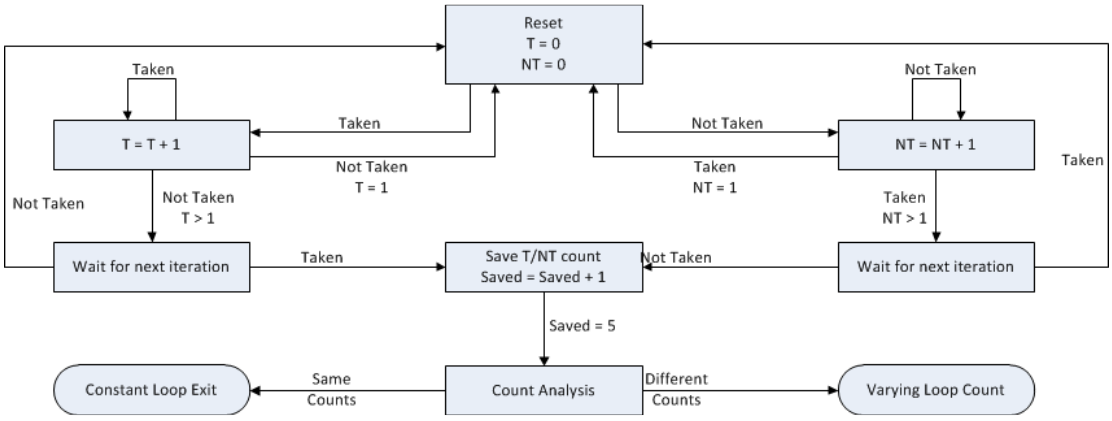

Figure 6. Methodology for identifying loop type mispredictions

3.2.3 Linked List Traversals and Array Access/Pointer Reference

Many branches depend on the values loaded from the memory. Whether the value comes from a pointer or an array a linked list, these branches are correlated with a load instruction. Therefore the main idea behind identifying array access/pointer reference and linked list traversal is detecting load-branch correlations. We define a

load branch correlation when a branch's source registers depend on the value loaded by a load instruction, directly or indirectly. The first step to identifying a load branch correlation is marking every load instruction's destination register as load dependent. Every time an instruction executes, we look at the registers read and written. If the instruction writes to a register, we mark this register as dependent to the registers that were read by this instruction using a data structure which holds dependency variables. In the future, if another instruction reads this written register and writes to another register, we mark this new written register as dependent to the previous written register. Since the previous written register already holds the information that it's also dependent on other registers, we have this dependency information like a chain, and are able to keep track of instructions which are far away being depended on each other's values. We use this tracking method because branches can be dependent on load instructions indirectly, in other words they could use a modified value loaded by a load instruction. Every time we see a branch, we look at the source registers and follow their dependency chain. When we're following the dependency chain, if we find out that there's a load instruction's destination register in the chain, then we mark the branch as having a load correlation and store the information for the load instruction in the data structure for branches and also mark the load instruction to be investigated. Every time we see a load instruction, we store the address being read and also the value which is read from that address. After collecting enough information, we investigate these address-value pairs. First we look at the distance between the values of the addresses if these values follow a constant stride, we mark the branch as being an array access/pointer reference. If most of the distances are the same, but there

are different distances every now and then, we also investigate the values loaded from those addresses. This change in the stride could happen for 2 reasons:

- We could be accessing partial data from and array, then accessing another partial data but from a different starting point later.
- We could be traversing a linked list which has nodes added to it at different times, causing the address distance pattern to have spikes in distance rather than having a constant stride of the node size.

At this point, we look at the values loaded from those addresses. Since we have all the address-value patter information stored, we can investigate if the values that are loaded by the load instruction are used to compute the source address of that load instruction for future execution which is a very common linked list traversal behavior. If that's the case, we identify the branch as being a linked list traversal. If these values loaded from those addresses are not used to compute the future addresses for that load, we mark the branch as being an array access/pointer reference. If the distances between the load addresses are varying frequently, this could happen for 2 reasons:

- We could be accessing tiny portions of an array at different times and different indexes.
- We could be traversing a linked list which has nods added to it frequently, causing the distances between the nodes varying frequently.

Again in this case, we look at the values loaded from

those addresses and try to find a linked list behavior. If the values that are loaded from those addresses are used to compute future addresses for the load instruction, we mark

the branch as linked list traversal. Otherwise we mark the branch as array access/pointer reference.

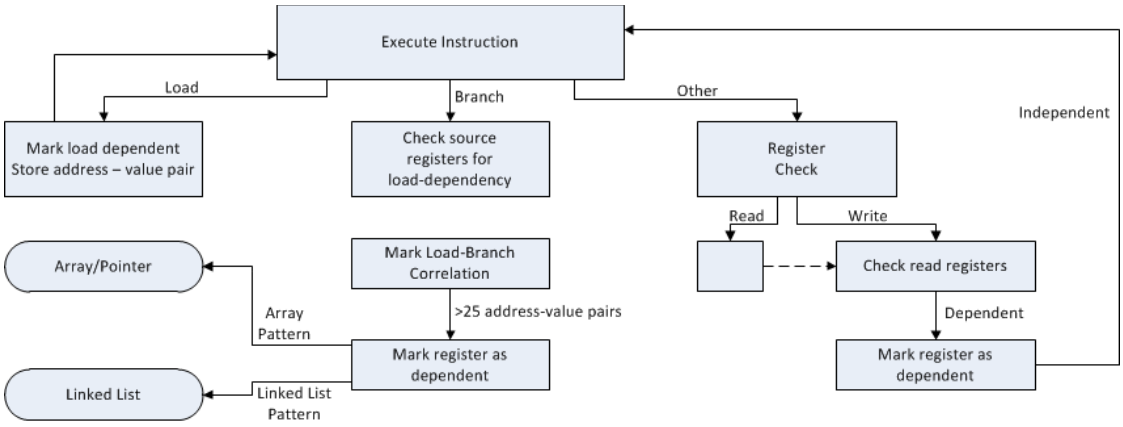

Figure 7. Methodology for identifying array access and linked list type mispredictions

3.3 Experimental Methodology

In this study, the programs I used include several of SPEC CPU 2006 benchmarks [28] and a set of 40 benchmark traces (16 client, 6 integer, 7 multimedia, 5 server, 6 workstation applications) provided with the 2011 Third Championship Branch Prediction (CBP) Competition [29] framework. SPEC benchmarks were compiled with gcc full-optimization. For SPEC benchmarks, I used 100M-size representative samples, which is found by SimPoint tool [30]for the reference input sets and the traces were generated using the MASE-alpha simulator [31]. Each CBP benchmark trace is for a 50M dynamic instructions. Table 1 lists the benchmarks I studied and their dynamic branch counts.

I ran the best performing (winner of the CBP competition) state-of-the-art TAGE [10] branch predictor on the benchmarks to be able to correlate PatternFinder's results. TAGE predictor uses a number of prediction tables (16 for my simulations)

with increasing branch outcome history. For the simulations, I used 16 different length histories to form a geometric series (as suggested by TAGE) between 4 and 1024 bits. TAGE favors long history predictions. For example, if there are multiple prediction table hits, the prediction of longest history table is selected if confidence exceeds a predetermined threshold.

Finally, all measurements in this study were performed on an Intel Xeon X5460 quad-core processor with 8GB of memory

The simulations I have performed for this study uses the PatternFinder tool with several different command line parameters. These parameters allow the user to pinpoint the appropriate patterns according to the goal of the simulation. In order to make the tool user friendly, PatternFinder implements many command line parameters.

List of parameters for the PatternFinder:

- Minimum number of occurrence
- Minimum pattern length
- Maximum pattern length
- Minimum coverage per iteration
- Maximum coverage for the complete run
- Interval for collision elimination
- Pattern length to start collision elimination
- Minimum number of unique patterns for collision elimination
- Single iteration
- %Coverage loss per pattern tolerance

- Output format parameters

CHAPTER 4

FINDINGS

ThePatternFinder output representations are very detailed providing importance of each individual pattern of any length in terms of their coverage of the input trace; where they occur in the input trace and their frequencies. This section presents the results found by thePatternFindertool. First, I discuss non-overlapping pattern analysis and implications on temporal and spatial branch outcome locality followed by overlapping pattern analysis and implications on branch predictability.

Output Information

The PatternFinder tool inputs a sequence of symbols and outputs detailed pattern information extracted from the input sequence. The output consists of information such as; lengths of the patterns, positions of the patterns, number of occurrence for each unique pattern, coverage of each unique pattern, coverage of all the patterns, average distance between each occurrence of a pattern. Since the output has a lot of information, it needs to be post-processed using scripts/programs in order to extract the specific information needed. Even though output packs a lot of information, it's much smaller than the input sequence, which makes it very fast to parse. Table [blabla] shows the input/output sizes for the CBP Framework and SPEC2006 benchmarks. The y-axis of the chart is logarithmic.

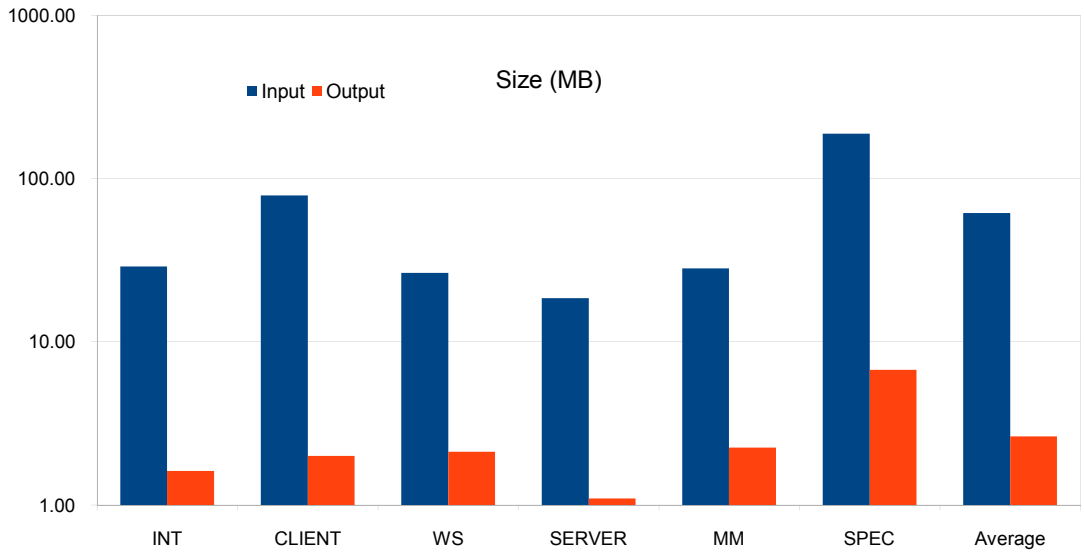

Figure 8.PatternFinder input/output size comparison

4.1 Longest Non-overlapping Patterns

Table 1 reports the longest non-overlapping patterns for each benchmark that is seen at least twice in the input sequence and their individual coverage. Long patterns are indicative of better spatial regularityand provide better spatial streaming opportunity. PatternFindergives priority to long patterns, that is, long patterns are found first and placed before shorter patterns are searched in the remaining parts of the input sequence. This is very different than what SEQUITUR does for compression. Therefore, SEQUITUR cannot usually find longest patterns. As we can see from Table 1, extensive pattern lengths are observed. The highlighted entries in the table show patterns longer than 200K.In the case of CLIENT02, a 2.9M length pattern exists (and covers 39% of the input trace). The longest pattern that SEQUITUR reports for this benchmark is only 1.6K. PatternFinder can find near optimal pattern summary, and queries can be customized by the user. For instance, instead of longest pattern first placement, longest pattern with best coverage (magnitude of regularity) provides a

Benchmark	Dynamic Branches (K)	Max. Length (K)	Max. Length Coverage %	Benchmark	Dynamic Branches (K)	Max. Length (K)	Max. Length Coverage %
INT01	6.0M	67.7	2.26	SERVER01	4.2K	655.4	30.69
INT02	5.4K	26.6	0.99	SERVER02	4.0K	21.8	1.10
INT03	5.1K	236.6	9.25	SERVER03	3.7K	8.7	0.46
INT04	7.9K	1.5K	38.67	SERVER04	3.8K	9.2	0.49
INT05	3.0K	8.1	0.54	SERVER05	3.7K	40.3	4.33
INT06	2.9K	8.3	0.58	MM01	4.0K	55.1	2.77
CLIENT01	3.9K	15.2	1.56	MM02	4.0K	24.2	2.44
CLIENT02	15.1K	2.9K	39.11	MM03	4.5K	25.7	1.14
CLIENT03	4.8K	33.2	1.38	MM04	3.8K	25.5	1.35
CLIENT04	4.4K	1.7	0.08	MM05	5.4K	0.9	0.03
CLIENT05	3.8K	152.0	7.86	MM06	1.8K	46.9	5.34
CLIENT06	8.6K	84.2	1.96	MM07	6.1K	0.1	0.00
CLIENT07	5.7K	289.7	10.24	bzip2	9.1K	65.5	1.43
CLIENT08	3.5K	33.9	1.94	mcf	23.3K	159.0	1.37
CLIENT09	3.5K	49.3	2.85	zeusmp	4.1K	660.6	32.31
CLIENT10	3.2K	15.4	0.96	gromacs	16.4K	1.2	0.01
CLIENT11	4.8K	3.3	0.27	cactusADM	0.4K	154.4	82.35
CLIENT12	3.7K	14.2	0.77	namd	16.3K	119.6	1.47
CLIENT13	4.1K	24.8	1.21	gobmk	13.0K	54.1	0.83
CLIENT14	4.2K	115.6	5.53	hmmer	11.3K	137.4	2.44
CLIENT15	4.7K	52.0	2.23	sjeng	16.4K	7.8	0.09
CLIENT16	4.4K	70.2	3.15	libquantum	21.6K	2.8K	26.66
WS01	4.8K	17.9	0.74	h264ref	5.6K	249.1	8.87
WS02	3.6K	40.4	2.25	omnetpp	18.2K	10.5	0.12
WS03	7.3K	5.4	0.15	astar	15.5K	524.3	6.74
WS04	4.1K	102.9	5.02	sphinx3	7.6K	161.6	4.26
WS05	3.5K	12.0	0.69	xalancbmk	18.7K	228.1	2.45
WS06	4.4K	23.9	1.08				

Table 1: Maximum Non-overlapping Pattern Length and Its Coverage

bettermetricfor quantifying stream behavior. It also gives faster simulation results as described in Section3. Overall, many benchmarks has long non-overlapping branch outcome patterns.

4.2 Spatial and Temporal Branch Outcome Streams

In this subsection, I discuss spatial and temporal regularities. Spatial regularity is defined as the number of data points in the regular subsequence. Temporal regularity is defined as the average number of references between successive non-overlapping occurrences of the subsequence that exhibits regularity.

Figures 3 and 4illustrate the cumulative distribution of hot pattern sizes, which summarize the spatial regularity of SPEC CPU 2006 benchmarks and CBP traces, respectively. In these figures, the maximum pattern length is limited to 1000. Overall coverage threshold is set to 90%. That is, at least 90% of the data points in the input must participate in patterns. The figure shows the weighted average pattern length across all of the sequence's patterns, where a pattern's weight is its individual coverage. Long patterns indicate good spatial regularity. Since PatternFinder finds long patterns first, only when long patterns cannot cover 90% of the input sequence, short patterns are given opportunity. Therefore, in Figures 3 and 4, curves closer to the top left corner represent benchmarks with the worst spatial locality. For example, gromacs (top line) has the worst spatial locality in Spec CPU 2006 benchmarks as 99% of its patterns are less than 80 references long. Similarly, top left cluster of lines in Figure 4, MM07, WS03, WS04, INT02, MM05 andINT01 have the worst spatial behavior as more than 95% of their patterns are less than 100 references long. On the

other hand, INT04, xalancbmk, namd, CLIENT02, CLIENT03, CLIENT06, libquantum, zeusmp, MM06, INT03 have best spatial locality with better distribution of pattern lengths: 60% or more of their patterns are longer than 700 references long.By analyzing Figures 3 and 4, programs can be divided into seven classes as listed in Table 3.This classification is done by examining the slopes at specific points in the figures and using this information to decide for the boundaries between the groups.

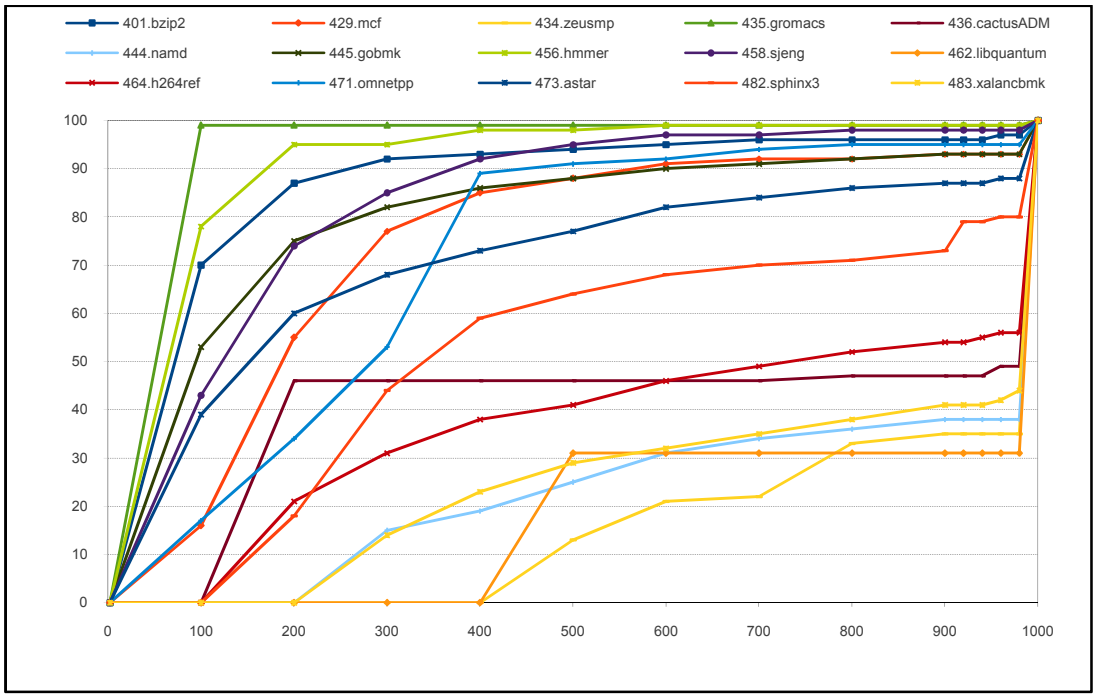

Figure 9: Cumulative distribution of hot pattern sizes (spatial regularity) for Spec CPU 2006 benchmarks. x-axis: pattern length, y-axis: % number of patterns. Simulations are run for minimum pattern length of 2, maximum pattern length of 1000 and for 90% coverage.

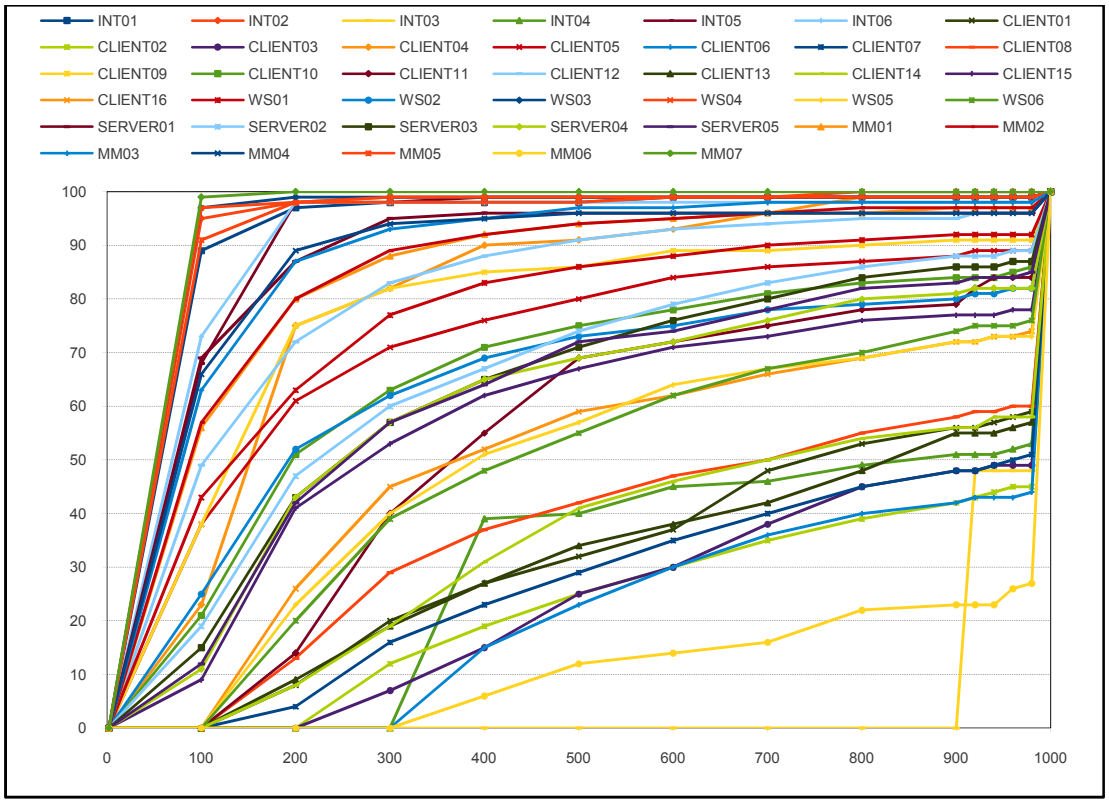

Figure 10: Cumulative distribution of hot pattern sizes (spatial regularity) for CBP traces. x-axis: pattern length, y-axis: % number of patterns. Simulations are run for minimum pattern length of 2, maximum pattern length of 1000 and 90% coverage.

Table 2 shows weighted average pattern lengths, where the coverage of a pattern is used as its weight, so hotter patterns have a greater influence on the reported average value. As expected, the benchmarks with worst spatial locality, such as MM07, gromacs, WS03, and MM05 have the smallest average pattern length. Table 2 also presents the weighted average pattern repetition (temporal regularity) intervals expressed in terms of number of references. Based on temporal regularities, the programs divide into five categories as shown in Table 3. The first groups are formed by benchmarks with higher weighted average repetition intervals, e.g., gromacs, mcf and hmmer, representing the low temporal locality groups. From group 1 to group 5,

the temporal locality increases.

Bench.	WAPL	WARI	TAGE misp. %	Bench.	WAPL	WARI	TAGE misp. %
INT01	184.74	1.11M	9.03	SER01	344.96	0.47M	3.79
INT02	157.42	1.12M	12.1	SER02	553.92	0.82M	1.02
INT03	917.60	8.41K	0.01	SER03	573.63	0.71M	0.96
INT04	751.28	0.27M	0.09	SER4	610.25	0.47M	0.92
INT05	159.97	0.65M	4.1	SER05	592.10	0.54M	0.86
INT06	191.51	0.62M	4.21	MM01	343.61	0.19M	2.49
CL01	748.56	0.32M	0.49	MM02	342.98	0.37M	2.62
CL02	793.99	1.35M	5.05	MM03	263.19	0.36M	2.62
CL03	775.84	0.46M	0.43	MM04	344.02	0.26M	2.62
CL04	310.26	1.09M	2.66	MM05	73.81	1.33M	14.3
CL05	547.55	0.17M	1.73	MM06	846.06	47.4K	0.14
CL06	788.23	0.43M	0.14	MM07	46.63	1.54M	14.0
CL07	775.86	0.24M	1.29	bzip2	367.27	1.20M	4.65
CL08	739.19	0.24M	0.78	Mcf	424.19	3.33M	2.21
CL09	495.56	0.33M	1.62	Zeus.	836.88	38.5K	0.41
CL10	584.23	0.26M	1.19	gromacs	55.98	4.53M	13.6
CL11	591.68	0.39M	0.76	Cactus	880.86	4.52K	0.07
CL12	404.53	0.4M	1.98	Namd	810.31	0.12M	1.17
CL13	760.73	0.32M	0.41	gobmk	476.88	0.89M	10.1
CL14	740.62	0.14M	0.56	hmmer	177.29	2.57M	5.7
CL15	643.41	0.17M	1.0	Sjeng	295.90	1.46M	4.93
CL16	674.58	0.21M	0.71	Libq.	873.60	1.7K	0.0
WS01	488.12	0.33M	1.99	h264ref	767.18	0.12M	1.45
WS02	626.00	0.15M	1.46	omnetpp	391.44	1.89M	1.67
WS03	69.72	1.28M	14.0	Astar	568.82	0.87M	2.88
WS04	232.13	0.63M	22.3	sphinx3	634.82	0.69M	2.1
WS05	667.93	0.27M	0.45	xalan	799.70	0.64M	0.54
WS06	656.28	0.23M	0.6				

Table 2. Weighted average pattern lengths and weighted average repetition intervals

Better temporal locality suggests better history table prediction opportunity because when the pattern repetition interval is large, it is more likely that table history is polluted with other patterns. Spatial and temporal classifications in Table 3 correlates well with branch misprediction rates. Benchmarks with better spatial and temporal localities tend to have lower misprediction rates.

4.3 Overlapping Branch Patterns

Unlike SEQUITUR, where the analysis must start after the whole grammar is produced, with suffix trees, the analysis can start as the tree is constructed. As the children of each level of the tree gives a set of unique patterns of same length and the patterns of the next level of a node is only one bit longer, a branch outcome analysis with a sliding window can be done during tree construction.

For each unique branch pattern of any length greater than l, the approximate confidence to predict the next branch outcome gets computed with the PatternConf in Eq. 1. For example, for a frequent pattern of 10110, the frequency of patterns101100 and 101101 is checked. If one of them occurs for 90 times and the other for 10 times, the confidence of 10110 is assumed to be 90%. To compute the overall confidence of an entire pattern length (with many unique patterns forming one level of the tree), one needs to sum all the max outcomes and divide the sum to number of total occurrences of all unique patterns of same length, as shown by PatternLengthConf in Eq. 2.

$$PatternConf = \frac{\max\left(count(out0), count(out1)\right)}{count(alloutcomes)} \qquad \textbf{Eq. 1}$$

$$PatternLengthConf = \frac{\sum_{n=0}^{N} \max(pattern\_nOutcome)}{\sum_{n=0}^{N} pattern\_nFreq} \quad \textbf{Eq. 2}$$

Although Eq. 1 and Eq. 2 compute approximate confidence and is not representative of branch predictability because it assumes infinite resources and ignores the order in which the branch outcomes arrive, it is still a good relative confidence metric to compare different pattern lengths. Other pattern length information, such as NumUniquePatterns of length l and the OverlappingCoverage – the ratio of number of unique patterns of length l and the number of input data points, together with Eq. 1 quantifies the importance of each pattern length.

For each level of the tree, the algorithm also finds non-overlapping patterns and their importance is computed as NonOverlappingCoverage – the product of number of unique non-overlapping patterns and pattern length. Note that OverlappingCoverageandNonOverlappingCoverageare very different metrics: one corresponds to outcome prediction opportunity of a pattern length while the other is the streaming opportunity of a pattern length.

Due to space limitations, I focus on few of the benchmark results with different behaviors as shown in Figure 5. As also shown in Table 1, extended pattern lengths are observed. CLIENT02 has a pattern of 2.9M length. In addition, two unique patterns of length greater than 2M recur multiple times to give 80% non-overlapping coverage as shown in Figure 5. Checking this behavior further, I observe (looking at the points where these patterns occur, which is provided by the pattern tool) that each of these twonon-overlapping patterns repeats itself back to back in two different

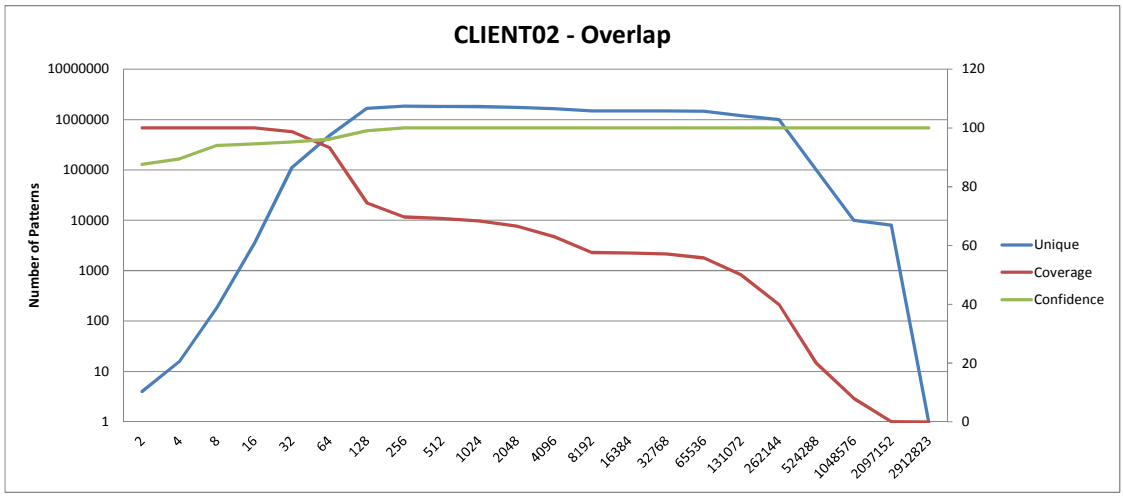

Figure 11. Coverage and confidence values for overlapping patterns in CLIENT02

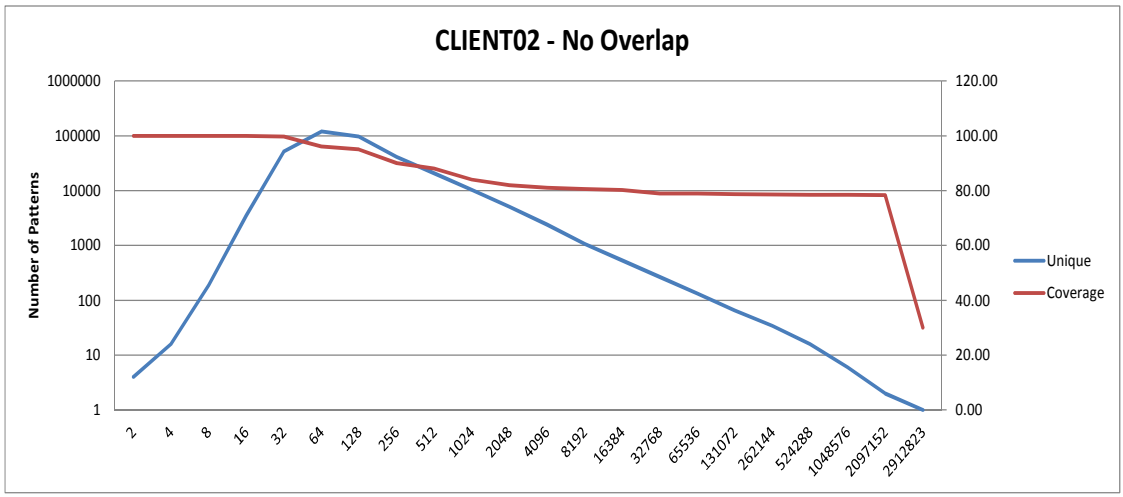

Figure 12. Coverage and confidence values for non-overlapping patterns in CLIENT02

phases of the program. CLIENT02 is one of thebenchmarks that TAGEmisprediction rate is relatively high, which is not surprising looking at the figure for overlapping patterns. Even with long histories, the number of unique patterns is very high (more than 1M). This means that TAGE is not able to provide predictions with the long history tablesbecause there are many conflicts which prevent long historytables to exceed the threshold due to frequent evictions. I modified the branch predictor

simulator and confirm this. Most predictions are provided by short history tables. Long non-overlapping histories with high coverage suggest that branch streaming is worth pursuing in cases such as CLIENT02.

Spatial	**Group 1:** MM07, gromacs, WS03, WS04, INT02, MM05, INT01 **Group 2:**hmmer, INT06, INT05, bzip2, SERVER01 **Group 3:** MM04, MM03, MM02, MM01, gobmk **Group 4:**CLIENT12, sjeng, WS01, CLIENT09, CLIENT05, astar, CLIENT04, mcf, omnetpp **Group 5:** WS02, CLIENT10, SERVER02, SERVER03, SERVER05, SERVER04, CLIENT15 **Group 6:** CLIENT16, sphinx3, WS05, CLIENT11, WS06 **Group 7:** INT04, xalancbmk, namd, CLIENT02, CLIENT03, CLIENT06, libquantum, zeusmp, MM06, INT03, 464.h264ref, CLIENT08, CLIENT14, CLIENT01, CLIENT13, CLIENT07
Temporal	**Group 1:** gromacs, mcf, hmmer **Group 2:**omnetpp, MM07, sjeng, CLIENT02, MM05, WS03, bzip2, INT02, INT01, CLIENT04, gobmk, astar, SERVER02, SERVER03, sphinx3, INT05, xalancbmk, WS04, INT06 **Group 3:** SERVER05, SERVER04, SERVER01, CLIENT03, CLIENT06, CLIENT12, CLIENT11, MM02, MM03, WS01, CLIENT09, CLIENT13, CLIENT01 **Group 4:** INT04, WS05, MM04, CLIENT10, CLIENT07, CLIENT08, WS06, CLIENT16, MM01, CLIENT15, CLIENT05, WS02, CLIENT14, h264ref, namd **Group 5:** MM06, zeusmp, INT03, cactusADM, libquantum

Table 3: Classification of benchmarks based on their spatial and temporal branch outcome regularities

Second important result that is observed from Figure 11 is that for TAGE-like predictor, one can estimate the range of the history lengths for best performance. A common practice today is to simulate the design space for a predictor (or a prefetcher) to find the best configuration, which is very time-consuming (it may also be misleading since predictors are designed by ad-hoc observations and it is quite possible that they do not cover a significant amount of benchmark behavior.)

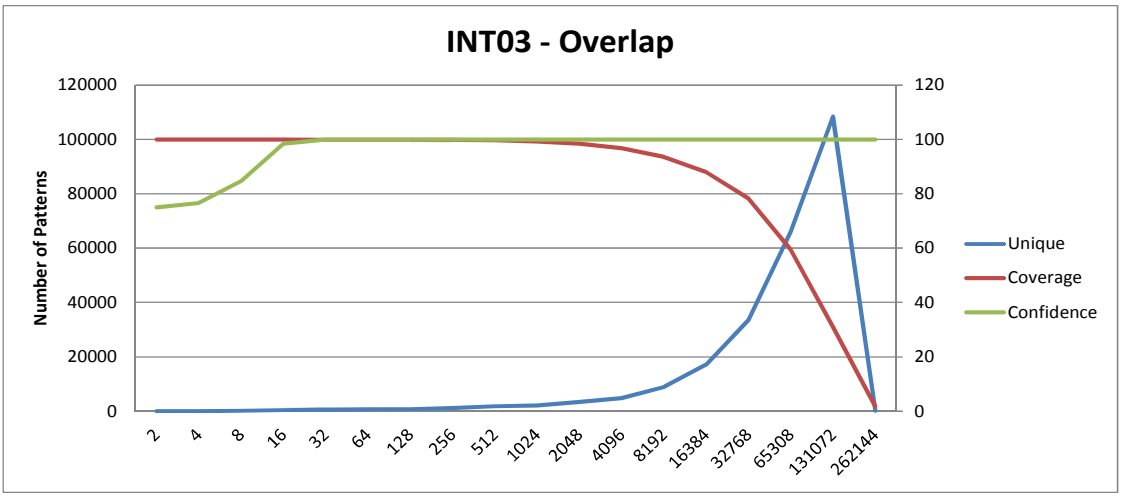

Figure 13. Coverage and confidence values for overlapping patterns in INT03

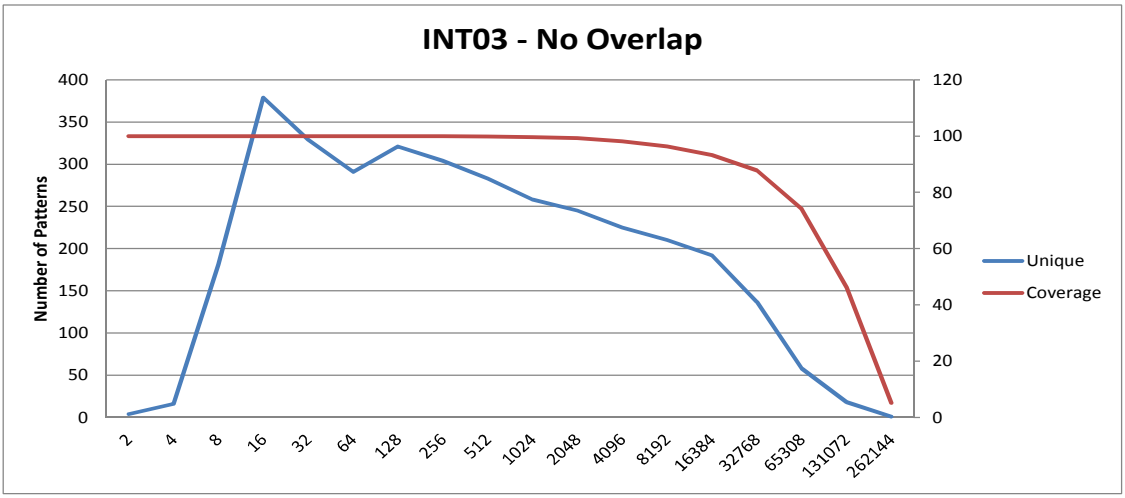

Figure 14. Coverage and confidence values for non-overlapping patterns in INT03

Looking at Figure 13 and Figure 14, results for INT03-Overlap, number of unique patterns is very low, less than 500 for pattern length of 128. This suggests that there will not be many conflicts in TAGE tables. Number of uniquepatterns is only about 2000 for the pattern length of 1024. The confidence goes to 100% at pattern length 16. Coverage is 100% up to pattern length 512. Overall, with these numbers, I conclude the following: the best historylengths for TAGE predictor is a geometric series

between history length 16 and 512, which is also confirmed by simulation. One can also expect to see very low misprediction rate because a 100% confidence/coverage betweenthe suggested history lengths is seen with relatively very small number of unique patterns.

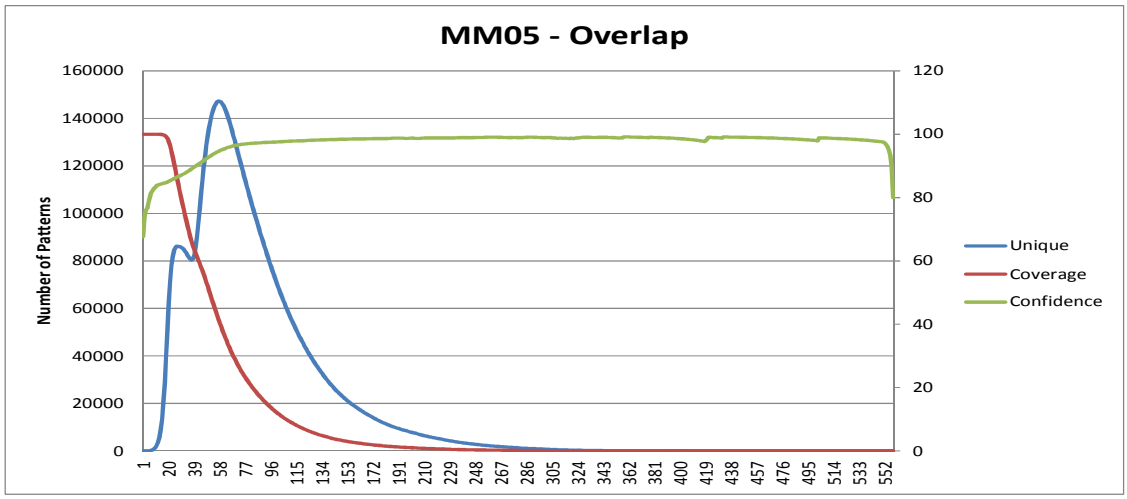

Figure 15. Coverage and confidence values for overlapping patterns in MM05

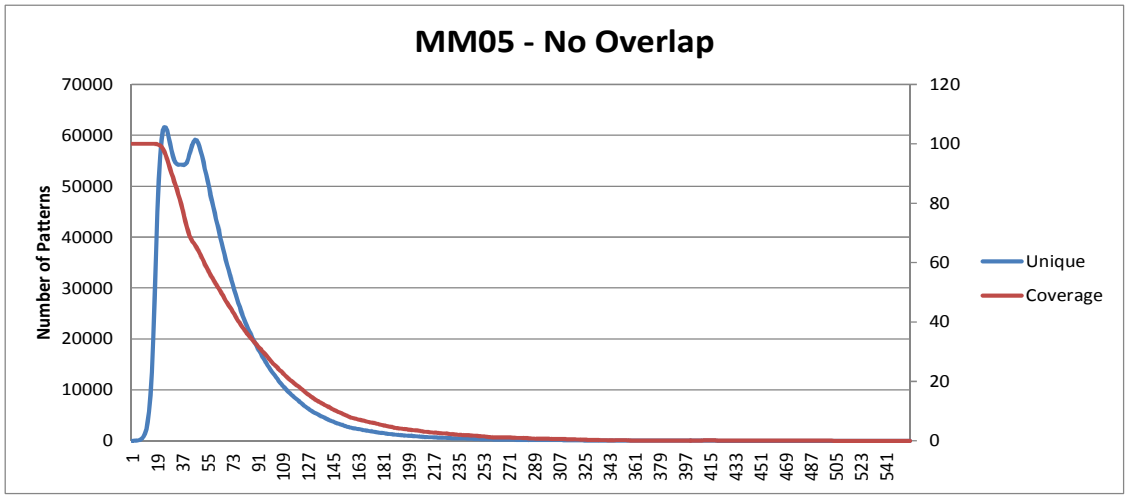

Figure 16. Coverage and confidence values for non-overlapping patterns in MM06

It can be seen from Figure 15 and Figure 16 that MM05 results show a very different behavior. Much shorter patterns are seen, large number of unique patterns and coverage drops very quickly to small numbers as the patternlength increases. If we increase the history length for TAGEbeyond 100, it will not have a good impact; this favors shorter history lengths (decreasing the history length from 1024 to 256, TAGE performed better). However, because of the large number of unique patterns, onecan expect to see high misprediction rate; in fact, TAGE has a 15% misprediction rate. Although, MM05's unique patternsdoubles from pattern length 30 to 65, the coverage drops from 80% to about 20%. Further investigation, using k=1 for finding patterns, reveal that there is a scan behavior, where within a region the same behavior is not seen again, so it is impossible to have large coverage with longer histories. The mispredictions experienced due to lack ofrecurrence of a pattern cannot be eliminated. Thus, MM05 presents itself as a not history-prediction friendly benchmark.

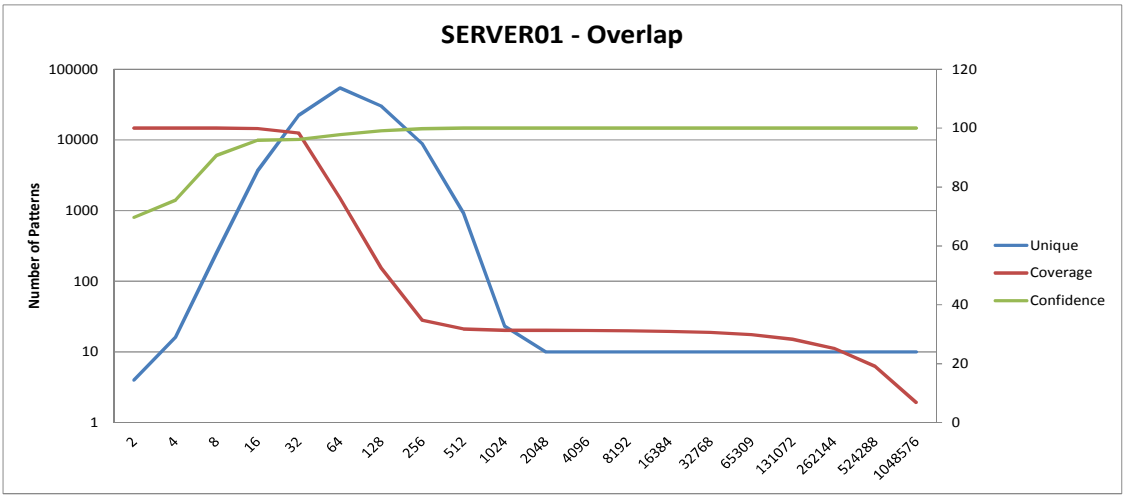

Figure 17. Coverage and confidence values for non-overlapping patterns in SERVER01

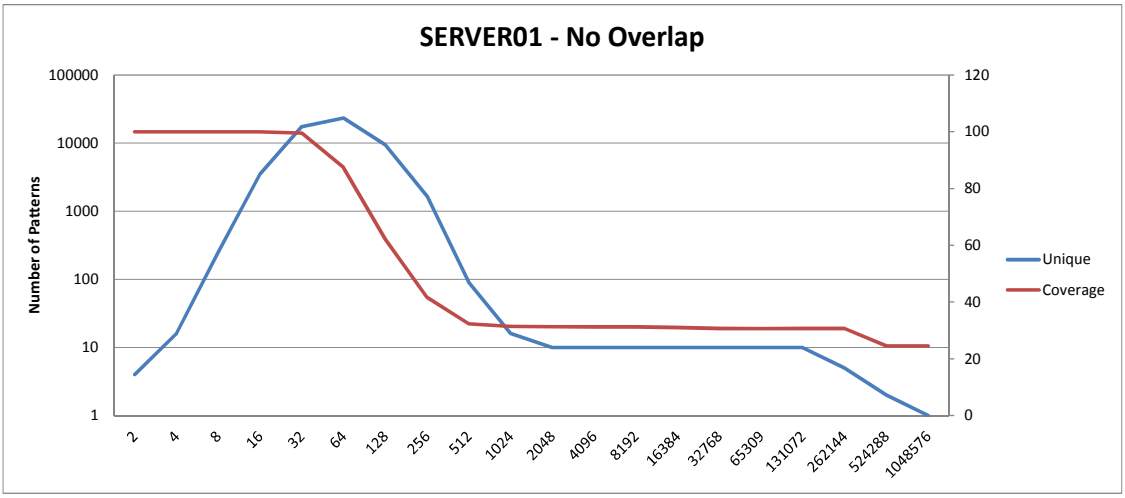

Figure 18. Coverage and confidence values for non-overlapping patterns in SERVER01

4.4 Address patterns

Aside from branch outcomes, memory addresses requested by programs also show various pattern behaviors and it's very important to analyze and understand them. Following section will address the analysis of non-overlapping address patterns.

4.4.1 Longest Non-overlapping Patterns

Table 4 reports the longest non-overlapping patterns for each benchmark in SPEC2006 that is seen at least twice in the input sequence and their individual coverage. Long patterns are indicative of better spatial regularity and provide better spatial streaming opportunity. PatternFinder gives priority to long patterns, that is, long patterns are found first and placed before shorter patterns are searched in the remaining parts of the input sequence. Overall, some benchmarks have very small patterns. Which would be expected for address patterns. But some of them have really

long patterns with length hundreds of thousands. It's also very interesting that the longest pattern in 462.libquantum has around 1.3 million length, which corresponds to 22.71% of the whole input stream with only 2 occurrences. This is a really interesting behavior. It shows that not only does the program execute the same piece of long code twice, but the data structure which is accessed in this piece of code is kept fully intact after the first execution.

Benchmark	Input Length	Max. Length	Occurrence	Coverage %
401.bzip2	20714659	85780	2	0.83%
429.mcf	33956785	238564	2	1.41%
434.zeusmp	19498172	5091	26	0.68%
435.gromacs	9803632	848	2	0.02%
436.cactusADM	40146082	243	33730	20.42%
444.namd	23031245	997	2	0.01%
445.gobmk	20220532	50045	2	0.49%
456.hmmer	28626061	84566	2	0.59%
458.sjeng	24178018	9465	2	0.08%
462.libquantum	12124879	1376881	2	22.71%
464.h264ref	27126245	280170	2	2.07%
471.omnetpp	24347407	780	3	0.01%
473.astar	25882958	524287	2	4.05%
482.sphinx3	20494556	445101	2	4.34%
483.xalancbmk	22168782	9769	2	0.09%

Table 4. Maximum non-overlapping pattern length and its coverage

4.4.2 Spatial and Temporal Address Streams

In this subsection, I discuss spatial and temporal regularities in the address streams.

Figures 19illustrates the cumulative distribution of hot pattern sizes, which summarize the spatial regularity of SPEC CPU 2006 benchmarks for their address request streams. In these figures, the maximum pattern length is limited to 100. Overall coverage threshold is set to 90%. That is, at least 90% of the data points in the input must participate in patterns. The figure shows the weighted average pattern length across all of the sequence's patterns, where a pattern's weight is its individual coverage. Long patterns indicate good spatial regularity. Since PatternFinder finds long patterns first, only when long patterns cannot cover 90% of the input sequence, short patterns are given opportunity. Therefore, in Figure 19, curves closer to the top left corner represent benchmarks with the worst spatial locality. For example, 401.bzip2 (top line) has the worst spatial locality in Spec CPU 2006 benchmarks as 99% of its patterns are less than 20 references long. Similarly, top left cluster of lines in Figure 19, 429.mcf, 482.sphinx3, 462.libquantum, 473.astar and483.xalancbmk have the worst spatial behavior as more than 85% of their patterns are less than 20 references long. On the other hand, 435.gromacs, 444.namd and 434.zeusmp have best spatial locality with better distribution of pattern lengths: 30% or more of their patterns are longer than 20 references long.

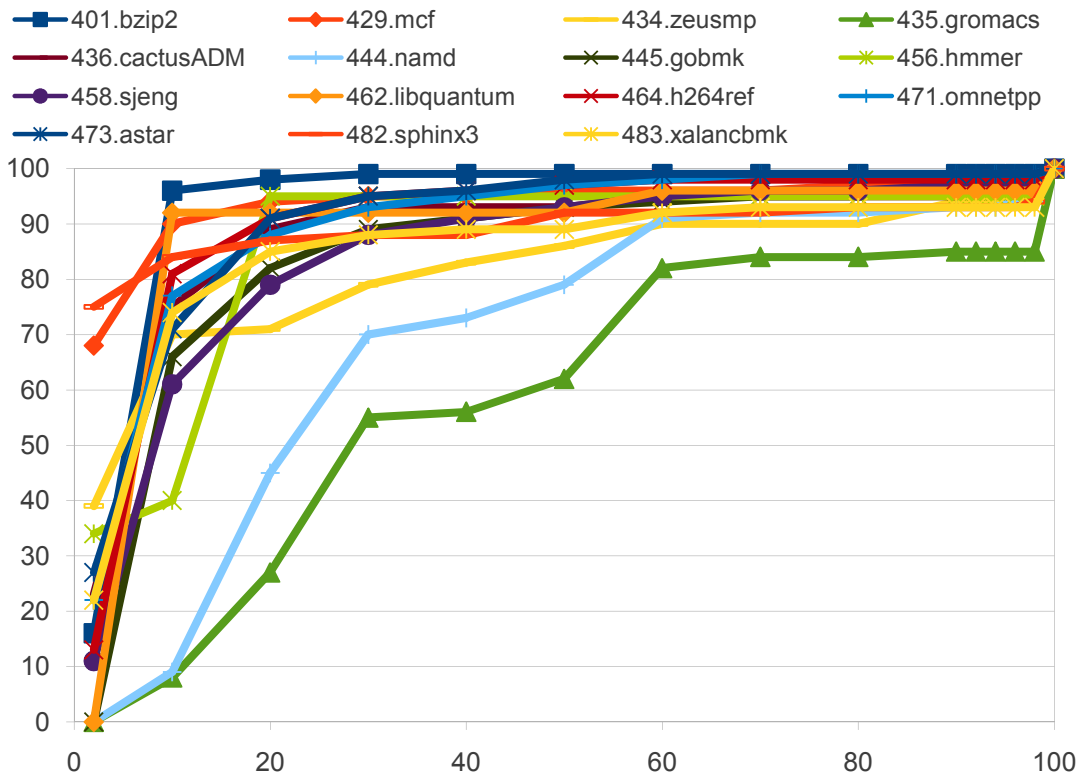

Figure 19. Cumulative distribution of hot pattern sizes for SPEC2006

Table 5 shows weighted average pattern lengths, where the coverage of a pattern is used as its weight, so hotter patterns have a greater influence on the reported average value. As expected, the benchmarks with worst spatial locality, such as 401.bzip2, 473.astar, have the smallest average pattern length. Table 2 also presents the weighted average pattern repetition (temporal regularity) intervals expressed in terms of number of references. Better temporal locality suggests better history table prediction opportunity because when the pattern repetition interval is large, it is more likely that table history is polluted with other patterns. Spatial and temporal classifications in Table 3 correlates well with branch misprediction rates. Benchmarks with better spatial and temporal localities tend to have lower misprediction rates.

Benchmark	Weighted Average Length	Weighted Average Repetition Interval
401.bzip2	13.472462	105932.0105
429.mcf	34.097126	718311.8393
434.zeusmp	54.500971	51961.20459
435.gromacs	57.323758	444341.0402
436.cactusADM	41.612383	43622.56799
444.namd	45.170923	203356.3083
445.gobmk	42.720648	671608.5027
456.hmmer	42.023024	6191908.166
458.sjeng	40.775792	861955.9451
462.libquantum	30.257393	4784850.124
464.h264ref	31.233398	1450808.097
471.omnetpp	24.452185	2558691.564
473.astar	21.082339	1683093.062
482.sphinx3	63.514376	2053108.821
483.xalancbmk	53.581694	342271.6953

Table 5. Weighted average pattern lengths and weighted average repetition intervals

4.4.3 Coarse-grain Triggers for Hot Streams

As extended hot streams exist, it is important to find efficient ways to exploit it. Memory streaming [36] was proposed to exploit long recurring memory access patterns by recording the memory addresses into main memory and replaying when needed. Most efficient implementations separate storage of address sequences (history buffer) and correlation data (index table), each of which is stored in main memory due to their large sizes. For significant performance gains, correlation table must be made larger than 64MB. Although size is manageable in main memory, practicality challenge arises from long latency lookups for prefetch meta-data and its bandwidth-hungry maintaining cost. To prefetch data, two round-trip memory accesses occur

initiated by a last-level cache miss. The first access searches the index table to retrieve the pointer to the history table entry where the stream is stored. The second access is to the history buffer to start reading the stream. Streamed addresses are buffered on chip waiting to be prefetched.

The fundamental problem here is that the trigger for starting a stream is a miss address. This fine-grain trigger causes many extra bookkeeping for the index table and consumes significant bandwidth resources, especially critical for today's multi-core processors. I propose course-grain triggers for memory streaming. Based on the initial analysis with few Spec CPU 2006 benchmarks and pointer-intensive Olden benchmarks, coarse-grain triggers can be found. With course-grain triggers, such as, a function call or call-chains, a long hot stream traversal can be initiated without incurring extra index table lookup in memory. It is possible to develop pattern tools that discover triggers for hot data streams, and thereby guide design of efficient prefetchers for memory streaming.

Another problem the current memory streaming method has is that it does not separate the easy-to-detect patterns, such as constant deltas from the streams which consumes a lot of extra bandwidth. These constant patterns are easily caught by state-of-the-art stride prefetchers that are employed in current high-performance processors. Memory streaming must be coordinated to work together with currently-employed stride prefetchers for greater performance. Ebrahimi et al. [40] presents an efficient technique for coordinating multiple prefetchers.

Although above discussion is based on prefetching, course-grain triggers are equally applicable to any streaming method. Below, I give an example of course-grain trigger

from mcf benchmark followed by a discussion on how course grain triggers can be found with pattern discovery methods.

An example of a hot address stream and a course-grain trigger

I analyzed mcf benchmark since it is one of the hardest to improve performance with conventional prefetching. I simulated a 100M simulation point selected by SimPoint tool with Gem5 simulator. The following code shown in Figure 20 corresponds to the hottest section of the mcf benchmark where most last-level cache misses occur. In the simulation, two of those load instructions marked as LD1 and LD2 are both executed 2.4M times. Since they are consecutive load instructions that access the same object, there is always a constant stride between the address loaded by LD2 and the address loaded by LD1. Even though these instructions are seen 2.4M times, both of them only load from 34173 different addresses. For each load instruction, 32108 of these addresses are seen exactly 72 times, which covers about 97% of their whole execution, which suggests that data structure does not change significantly (few additions/deletions).

Running Algorithm 1 on the address traces that I generated, the longest pattern found is about 13K long occurring 15 times. There are, however, shorter patterns, still more than a thousand of addresses long that occur 72 times. Since, I have not yet implemented a tool that can find approximate patterns; it is not known if longer patterns occur. However, with small directed scripts, some longer approximate patterns are observed. One important observation about the patterns that the exact pattern tool (Algorithm 1) found is that almost all the long patterns are non-

overlapping and recur at a significant distance. More detailed analysis reveal that every time PC=120009e5c (psimplex.c:127 -- refresh_potential(net)) executed, pattern repeats (72 times). Here PC address 120009e5c is the course-grain trigger. The analysis shows us that we need approximate pattern finding tools to automatically analyze this benchmark. It also shows us that course-grain triggers can be found.

```
   mcfutil.c:84
   120007e50:   ldq     t7,56(t2)  → LD2
     120007e54:   xor     t7,0x1,t7
     120007e58:   bne     t7,120007e80
<refresh_potential+0xf0>
   mcfutil.c:85
     120007e5c:   ldq     a0,64(t2)
   .
   .
   .
   mcfutil.c:92
     120007e9c:   mov     t2,t6
   mcfutil.c:93
   120007ea0:   ldq     t2,24(t2)→ LD1
   mcfutil.c:82
     120007ea4:   bne     t2,120007e50
<refresh_potential+0xc0>
   mcfutil.c:98
     120007ea8:   ldq     a0,16(t6)
```

```
    80    while( node != root )
    81    {
    82       while( node )
    83       {
    84          if( node->orientation == UP )  → LD2
    85             node->potential = node->basic_arc-
>cost + node->pred->potential;
    86          else /* == DOWN */
    87          {
    88             node->potential = node->pred-
>potential - node->basic_arc->cost;
    89             checksum++;
    90          }
    91
    92          tmp = node;
    93          node = node->child;  → LD1
    94       }
```

Figure 20. Code snippet for hot load PCs in mcf: (a) assembly, (b) C code.

In order to capture common pattern behavior in multiple streams, I have extended the tool to let user search for patterns which occur commonly in different streams. The user can specify the minimum number of occurrences for the pattern in each stream. The user can also specify the percentage of the input streams that must have the pattern within their boundaries.

In order to show an example behavior, I've chosen the refresh_potential function call as a trigger and split the input trace accordingly. This resulted in having

18 different streams which is because this function is called 18 times during the execution of the trace. The first thing to look at would be the longest pattern length that can be found which is common to these streams. But since a pattern does not have to be seen in all of the streams, I've run the simulation with 10 different parameters, changing the percentage of streams to have the common pattern.

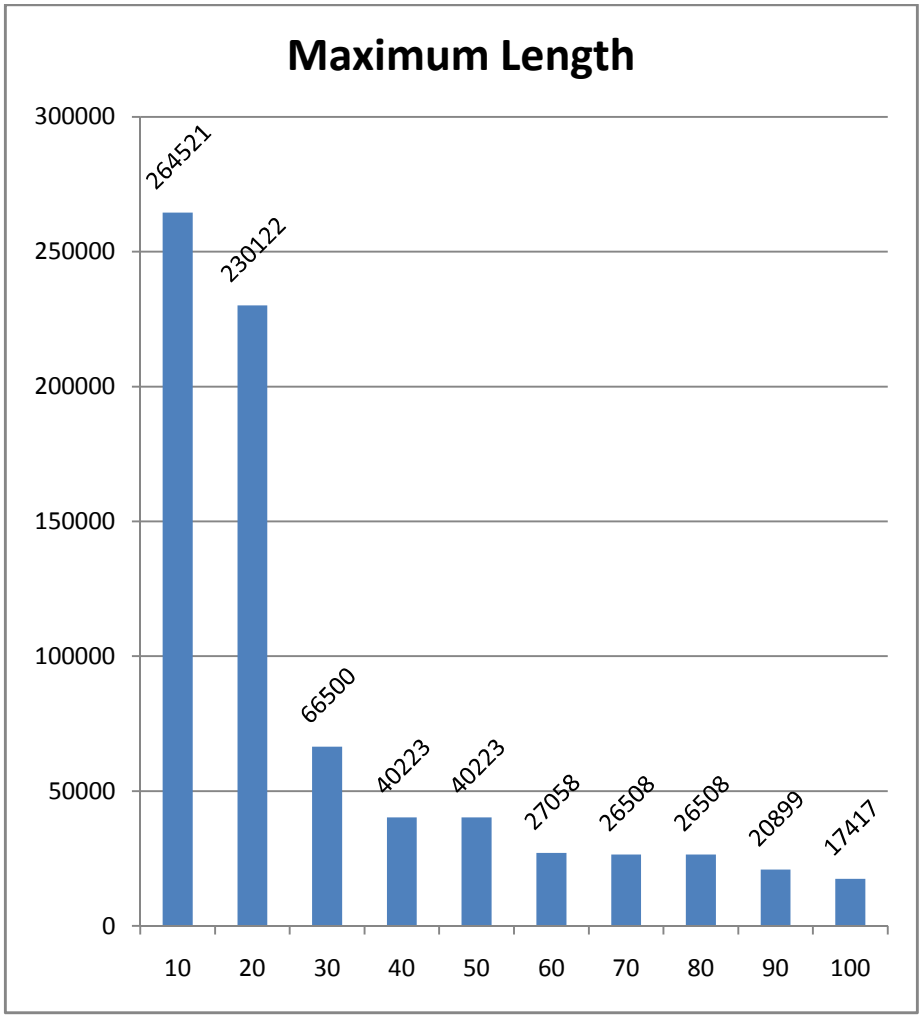

Figure 21. Longest common patterns changing over stream number threshold.

It can be seen from Figure 21 that for 10% of the streams and for 20% of the streams, the longest pattern length is quite large, which is over 200K. In the 20% case, a pattern

of 230K length is seen in 4 different streams, which is quite an interesting behavior, which covers the whole outer loop of tree traversal within that function. This shows that at 4 calls of this functions, the exact address sequence of length 230K is followed.

Another potential analysis was looking at the commonality between all the streams, which would mean looking at patterns which are seen at least once in every single stream. For this analysis, the max length is chosen as 100. At the end of the simulation, 42% of the whole stream was covered, which means there's at least 42% similarity between all the streams generated using that function call.

4.5 Dynamic source code detection by mispredictions

Figure 22 shows the breakdown of the branch misprediction categories for SPECint, SPECfp benchmarks, respectively. An interesting observation is that, for most of the benchmarks, the “other” is the largest category. This is more pronounced for the following benchmarks: for bzip2, vpr, mcf, parser, perl, and twolf, about 50% of the mispredictions fall into the “other” category; And for art, swim, mgrid, lucas, sixtrack, dijkstra, susan, sha, and bitcount, more than 75% of the mispredictions fall into the “other” category.

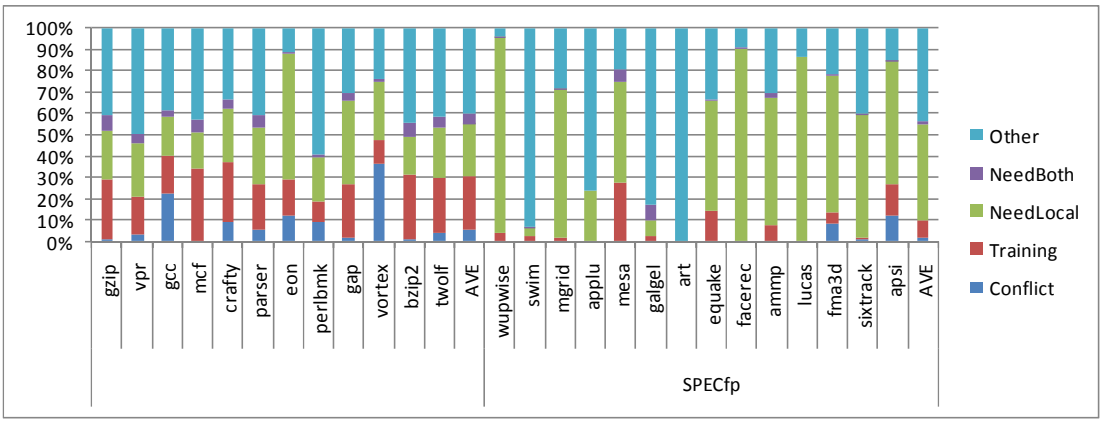

Figure 22. Breakdown of misprediction types for 4kB gshare predictor for SPECint, SPECfpbenchmarks.My experiments also show that larger global history decreases mispredictions in conflict, need other and other categories while increasing mispredictions in training category, which is expected.

Table 6 summarizes the results by showing average percentages of each class of mispredictions per benchmark suite. These results show the importance of wrong type history along with well known problems of conflicts and training times. However, one can also see that a large percentage of mispredictions (about 40% on average) can not be categorized as being from one of the abovementioned misprediction types using this taxonomy. It must also be noted that, with this taxonomy, a branch's mispredictions may fall into different categories for different dynamic instances of the branch. Therefore, this taxonomy can not provide detailed information about a specific branch. This suggests a further investigation for important branch instructions. In this study, after identifying hot branches through run-time profiling, I perform source-code analysis in order to provide more insights into why specific branches mispredict often. This also identifies branches, which cause mispredictions that go under the "other" category.

In this study, I analyze each branch individually, unlike Skadron does. Each branch can fall into many different categories during the execution of the program, therefore a counter is kept for each misprediction class for every single class and collect the results that way. Table 7 summarizes how the information is collected for each branch.

	Conflict	Training	Need Local	Need Both	Array	Linked List	Constant Loop Exit	Varying Loop Count	Changing Function Input	Undecided
SPECint	4.95%	25.30%	24.90%	5.05%	27.63%	8.50%	0.02%	2.88%	0.45%	0.16%
SPECfp	1.23%	8.49%	45.07%	1.38%	39.15%	1.08%	0.00%	2.74%	0.58%	0.24%

Table 6. Average percentage of mispredictions for each class

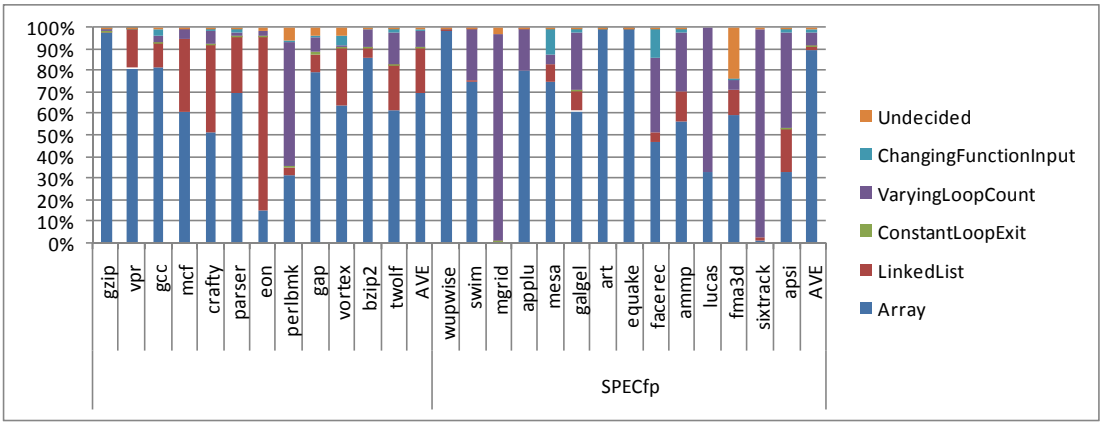

Figure 23. Breakdown of the other category

PC	Called	Misses	Conflict	Training	NeedLocal	NeedBoth	Other	Reason
200693CC	15428	15427	15427	0	0	0	0	Array
200A3E74	32487	7936	2	1217	4231	310	2175	Array
200AA68C	44218	5571	0	542	3822	64	1141	Linked List
200A3D80	40522	4690	78	502	2661	39	1407	Array
200AA65C	53545	4518	133	0	0	0	4385	Linked List
20061E60	12746	4417	0	0	3739	45	632	Array
20048348	17860	4194	139	0	2486	60	1508	Array
200A3D94	35877	3952	0	1774	1275	0	901	C. F. I.
2005FCE8	403820	3474	3457	16	0	0	0	Array
2008E394	12269	3316	0	2455	160	0	700	Array

Table 7. Sample output showing top 10 most mispredicted branches

4.5.1 Source Code Examples

Changing function inputs:

This example is from the gap benchmark. The branch is a for loop which starts from start+1 and executes until lenList. Both of these variables; start and lenList actually play a role in deciding how many times this loop will iterate. The start variable is directly a function input and the lenList variable is actually the lenght of the hdList variable which is a function input as well.

The branch is executed 47968 times. It's taken 35627 times and not taken 12341 times. The branch is correctly predicted for 38134 times and mispredicted for 9834 times. The function has 44 different function input pairs.

```
plist.c:533
  12007a4e8:   01 00 6b 21    lda    s2,1(s2)
  12007a4ec:   08 00 ad 21    lda    s4,8(s4)
  12007a4f0:   a2 0d 6c 41    cmple  s2,s3,t1
  12007a4f4:   d2 ff 5f f4    bne    t1,12007a440→ BR1
plist.c:546
  12007a4f8:   00 00 5e a7    ldq    ra,0(sp)
  12007a4fc:   08 00 3e a5    ldq    s0,8(sp)
plist.c:545
  12007a500:   ac 09 8b 41    cmplt  s3,s2,s3
```

(a)

```
520 long          PosPlist ( hdList, hdVal, start )
521     TypHandlehdList;
522     TypHandlehdVal;
523     long              start;
524 {
525     long              lenList;       /* length of <list>              */
526     TypHandlehdElm;         /* one element of <list>          */
527     long              i;             /* loop variable                 */
528
529     /* get the length of <list>                                        */
530     lenList = LEN_PLIST( hdList );
531
532     /* loop over all entries in <list>                                 */
533     for ( i = start+1; i<= lenList; i++ ) {  →  BR1
534
535         /* select one element from <list>                              */
536         hdElm = ELM_PLIST( hdList, i );
```

(b)

Figure 24.Code snippet for a hot PC in gap, (a) assembly, (b) C code

Constant Loop Exits:

A loop with a loop count of n, is often taken for n times followed by a not-taken at the loop exit. For cases when the loop counts are larger than what branch predictor can remember, prediction fails at the loop exits. Often, it is difficult for a global, local, or combined history predictor to keep sufficient history for this type of branches. Therefore, to target loop-exit mispredictions, loop predictor [26] was proposed. Constant loop exit mispredictions can also be put into insufficient history length or wrong-history type mispredictions categories.

This example is from the benchmark gap. The for loop which iterates from 0 to SIZE(hdSSeq)/SIZE_HD is marked as a constant loop exit. The branch is executed 22803 times. It's correctly predicted for 22295 times and it's mispredicted for 508 times. It's taken 15202 times and not taken 7601 times. In this example we see a nice case for a constant loop exit branch. The branch is always taken twice followed by a single not taken, which indicates the loop count is always 2. Even by looking at the C

code, it's obvious that the loop count will always be the same, since the it will run for SIZE(hdSSeq)/SIZE_HD. SIZE_HD variable is a constant and it's never changed. AloshdSSeq variable always has the same data type, which means SIZE(hdSSeq) will never change. Therefore the loop will always have the same iteration count.

```
12009889c:   48 00 5e a4     ldq     t1,72(sp)
statemen.c:233
  1200988a0:   08 04 e0 47     mov     v0,t7
statemen.c:231
  1200988a4:   01 00 4a 21     lda     s1,1(s1)
  1200988a8:   08 00 ce 21     lda     s5,8(s5)
statemen.c:234
  1200988ac:   01 08 01 44     xor     v0,t0,t0
  1200988b0:   47 00 20 e4     beq     t0,1200989d0
<EvFor+0x490>
statemen.c:231
  1200988b4:   00 00 42 a4     ldq     t1,0(t1)
  1200988b8:   82 76 40 48     srl     t1,0x3,t1
  1200988bc:   a2 03 42 41     cmpult  s1,t1,t1
  1200988c0:   dbff 5f f4     bne     t1,120098830→   BR1
  1200988c4:   00 00 fe 2f     unop
  1200988c8:   00 00 fe 2f     unop
  1200988cc:   00 00 fe 2f     unop
  1200988d0:   20 00 e0 c3     br      120098954 <EvFor+0x414>
  1200988d4:   00 00 fe 2f     unop
  1200988d8:   00 00 fe 2f     unop
  1200988dc:   00 00 fe 2f     unop
statemen.c:241
  1200988e0:   ffdf 9d 24     ldah    t3,-8193(gp)
  1200988e4:   00 00 fe 2f     unop
```

(a)

```
230          if ( TYPE(hdSSeq) == T_STATSEQ ) {
  231              for ( k = 0; k < SIZE(hdSSeq)/SIZE_HD; ++k ) {
→ BR1
  232                  StrStat = "";  HdStat = PTR(hdSSeq)[k];
  233                  hdRes = EVAL( HdStat );
  234                  if ( hdRes == HdReturn ) {
  235                      ExitKernel( hdRes );
  236                      return hdRes;
  237                  }
  238              }
  239          }
```

(b)

Figure 25.Code snippet for top hot PC in gap, (a) assembly, (b) C code

Linked List Traversal:

A pointer-chasing load, such as node=node→next, that determines the end of a linked list makes it hard to predict the branch that depend on it. If the linked list has n nodes, the loop iterates n times and the branch outcomes would be n-1 times "taken" followed by a "not taken". Branch predictors that exploit correlation in branch outcome histories often fail to predict these branches accurately. The analysis shows that mcf, parser, and dijkstra have significant amount of hard-to-predict branches of this type. However, at a closer look, these hard-to-predict branches may be predicted correctly because, although they do not have regular correlation in branch histories, they exhibit a type of locality that can be exploited with different mechanisms. Most components of data structures in SPEC CPU 2000 and Mibench benchmarks tend to remain stable. For example, after a linked list is initialized, the address of the end node remains the same until a new node is added to the end. In fact, even the order of the node addresses that is traversed remain the same until there is insertion or deletion. Therefore, if there are n nodes and if last m nodes of the linked list remain stable, once node n-m is accessed, one can predict that branch outcome that depends on this linked list traversal should be not taken when node n is reached. If a branch depends on such stable data, address of the data is sufficient to determine the branch outcome.
Figure 26 shows examples of linked-list-traversal-caused branch mispredictions for parser. Similar examples are also found in gcc, art, ammp, jpeg, bzip2, basicmath, dijkstra.

Figure 26 shows the source code (assembly and C code) from parser that includes a tree structure. In this pointer-chasing code, the loaded values that determine the branch outcome are irregular, which makes this branch hard to predict. Conventional branch predictors fail to provide very accurate predictions for this type of branches. However, because BR1 in Figure 5 is mostly taken (92% of the time), a 4KB gshare predictor is still doing well. The misprediction rate is 8.67%. A 32KB gshare further reduces the misprediction rate to 7.4%. A 32KB PWL can achieve 6% misprediction rate.

```
post-process.c:746
 419d70:     28 00 00 00    lw $16,0($18)
 419d78:     05 00 00 00    beq $16,$0,419e80
post-process.c:747
 419d80:     28 00 00 00    lw $3,4($16)
 419d88:     55 00 00 00    sll $2,$3,0x2
 419d90:     42 00 00 00    addu $2,$2,$3
419d98:     55 00 00 00    sll $2,$2,0x2
 . . .          . . .          . . .
 419dc0:     02 00 00 00    jal 416ca0
 419dc8:     06 00 00 00    bne $2,$0,419de0
post-process.c:746
 419dd0:     28 00 00 00    lw $16, 8($16) → LD1
419dd8:     06 00 00 00    bne $16,$0,419d80 → BR1
```

(a)

```
//post-process.c
743    D_tree_leaf * dtl;
744    int d, count;
745    for (d=0; d<N_domains; d++) {
746        for (dtl = domain_array[d].child;
           dtl != NULL;   → BR1
           dtl = dtl->next) {  → LD1
747            if (ppmatch(selector, pp_link_array[dtl>link].name))
                break;
748        }
```

(b)

Figre 26.Code snippet for top second hot PC in parser (a) assembly, (b) C code.

In Figure 5, branch BR1 checks if the value in register $16 is not equal to NULL. $16 holds the address of dtl. BR1 is dependent on the pointer-chasing load, LD1 (dtl=dtl→next). LD1 often misses in cache, which means BR1 resolves late. BR1 is

accessed 476293 times. There are 1172 different values for register $16 (dtl addresses). Each address is accessed about 400 times on average. The values in these addresses do not change frequently. There are only a few changes throughout the simulation. Thus, address values ($16) instead of data loaded from these addresses are sufficient to know the branch outcome. There is also a pattern in which addresses follow each other, i.e., few node insertions or deletions for a long time. Since the data structures are very stable, register values that hold node address values in previous iterations of the loop can be used to predict the outcome of the branch instance that is dependent on the end node in the linked list.

Array Access and Pointer Reference:

In this example the branch has a load-branch correlation with the previous line of c code. The variable l is being checked if it's null. And l actually is a TypDigit pointer which is loaded from PTR(hdl)[i]. The branch is executed for 386035 times.

It's taken 108181 times and not taken 277854 times. The branch is correctly predicted 311925 times and mispredicted 74110 times.

The load instruction which produces the variable l, loads from 385795 different memory locations and there are only 9891 different values loaded from these addresses. And the values in those addresses are always consistent, they never change. This branch could be correctly predicted by using these addresses.

```
integer.c:796
  120053fb0:   08 00 89 a5     ldq     s3,8(s0)
  120053fb4:   00 00 fe 2f     unop
  120053fb8:   00 00 fe 2f     unop
  120053fbc:   00 00 fe 2f     unop
 120053fc0:   00 00 0c 31     ldwu    t7,0(s3)  → ld1
integer.c:799
  120053fc4:   12 04 e5 47     mov     t4,a2
integer.c:800
  120053fc8:   13 04 ff 47     clr     a3
integer.c:797
  120053fcc:   be 00 00 e5     beq     t7,1200542c8   → BR1
```

(a)

```
 792        /* run through the digits of the left operand                    */
 793         for ( i = 0; i< SIZE(hdL)/sizeof(TypDigit); ++i ) {
 794
 795            /* set up pointer for one loop iteration                      */
 796            l = ((TypDigit*)PTR(hdL))[i];  → ld1
 797            if ( l == 0 )  continue;  → BR1
 798            r = (TypDigit*)PTR(hdR);
 799            p = (TypDigit*)PTR(hdP) + i;
 800            c = 0;
 801
 802            /* multiply the right with this digit and add into the product */
 803            for ( k = SIZE(hdR)/(4*sizeof(TypDigit)); k != 0; --k ) {
 804               c = l * *r++ + *p + (c>>16);  *p++ = c;
 805               c = l * *r++ + *p + (c>>16);  *p++ = c;
 806               c = l * *r++ + *p + (c>>16);  *p++ = c;
 807               c = l * *r++ + *p + (c>>16);  *p++ = c;
 808            }
 809            *p = (c>>16);
 810         }
```

(b)

Figure 27.Code snippet for one of the hot PCs in gcc, (a) assembly, (b) C code

Based on the source-code analysis, one can summarize the findings as follows:

1. Since few branches correspond to a disproportional amount of mispredictions, it may be worth performing detailed source-code analysis on these hot branches.
2. In many cases, misprediction patterns exist. Most misprediction classes that we observed exhibit some sort of repeating patterns.

3. Some mispredictions are harder to correct than others. Mispredictions due to linked list traversals, randomly varying loop counts, changing inputs to functions are harder (will require more (and different type) history) than other types of mispredictions (conflicts, wrong-type history, constant loops, insufficient history length).
4. Address-value correlation provides some opportunity to correct mispredictions otherwise not possible, especially for linked list traversals and array accesses/pointer references. Data often do not frequently change in addresses. Many examples in benchmark programs.
5. Constant loop exit, insufficient history length, and wrong-type history mispredictions can usually be eliminated with relatively small size of misprediction histories because they often have regular misprediction patterns.
6. Given a constant hardware budget for branch prediction, it may be better to have a combination of branch predictor and a predictor that tracks and reduces mispredictions than having one complicated branch predictor.

CHAPTER 5

CONCLUSION

In this study, I presented the PatternFinder, a tool that we develop to analyze exact patterns. Using PatternFinder, we presented an example analysis of exact branch outcome patterns. The analysis have shown that extended data and branch patterns do exist in modern benchmark. It has also shown that hot patterns are useful in quantifying branch outcome locality. Spatial and temporal non-overlapping pattern locality provides useful insights into the branch streaming opportunities. Overlapping pattern behavior investigates branch predictability. The analysis has shown that the PatternFindertool can efficiently be used for summarizing a benchmark's dynamic branch behavior and thereby gives valuable insights into the design of future prediction mechanisms. The tool can also be used for pattern-centric classification of benchmarks and programs.

Due to the importance of finding common patterns between multiple streams, I've also extended the tool in order to capture common pattern behavior between different streams. Capturing this behavior could be useful in identifying coarse grain triggers and even validating the importance of coarse grain triggers.

I have also presented a methodology for dynamic source code analysis which gives the user insight about the data structures accessed by specific instructions without having any access to a source code or existing debug symbols in the executable. This type of analysis can give the user insight about the nature of the code

being executed such as array accesses, linked list traversals, tree traversals, pointer accesses, etc.

REFERENCES

[1] T. Chilimbi, "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality," PLDI 2001.

[2] Nevill-Manning, C.G. and Witten, I.H., "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," Journal of Artificial Intelligence Research, 7, 67-82, 1997.

[3] Wenisch et al "Temporal Streaming of Shared Memory" ISCA 2005.

[4] Somogyi et al. "Spatio-Temporal Memory Streaming," ISCA 2009.

[5] Wenisch et al. "Practical Off-chip Meta-data for Temporal Memory Streaming," HPCA 2009.

[6] D. Gusfield, "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology," Cambridge: Cambridge University Press, 1997.

[7] Bieganski et al "Generalized suffix trees for biological sequence data: Applications and implementation," 27th Ann Hawaii Int. Conf. on System Sciences. Vol. 5: Biotechnology Computing, 35-44, 1994.

[8] Gusfield et al "An efficient algorithm for all the pairs suffix-prefix problem," Inf. Process. Lett. 41(4):181-185, 1992.

[9] T.-Y. Yeh and Y. N. Patt. "Alternative implementations of two-level adaptive branch prediction," ISCA 1992.

[10] Pan et al "Improving the accuracy of dynamic branch prediction using branch correlation," ASPLOS 1992.

[11] S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

[12] Evers et al, “Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches,” ISCA 1996

[13] Chang et al, “Alternative implementations of hybrid branch predictors,” MICRO 1995.

[14] André Seznec, “Analysis of the O-GEometric History Length Branch Predictor,” ISCA 2005

[15] A. Seznec, “The L-TAGE predictor,” JILP, May 2007

[16] D. Jiménez and C. Lin, “Dynamic Branch Prediction with Perceptrons,” HPCA 2001.

[17] D. Jiménez, “Piecewise Linear Branch Prediction,” ISCA 2005.

[18] Gabriel H. Loh, “Deconstructing the Frankenpredictor for Implementable Branch Predictors,” JILP 2005.

[19] D. Jiménez, “Fast Path-Based Neural Branch Prediction,” MICRO 2004.

[20] A. Seznec, “Redundant History Skewed Perceptron Predictors: pushing limits on global history branch predictors,” IRISA Report No 1554, sept. 2003.

[21] Sendag et al, “Branch Misprediction Prediction: Complementary Branch Predictors,” IEEE Computer Architecture Letters, Dec. 2007.

[22] W. A. Wulf and S. A. Mckee, “Hitting the Memory Wall: Implications of the Obvious,” SIG. Comp. Arch. News, 23:20-24, 1995.

[23] J.-L. Baer and T.-F. Chen, “An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty,” SC 1991.

[24] N. P. Jouppi, Improving Direct-Mapped Cache Perf. by the Addition of a Small Fully-Assoc. Cache and Prefetch Buffers, ISCA 1990.

[25] Roth et al. Dependence Based Prefetching for Linked Data Structures. ASPLOS 1998.

[26] C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. ICS 2000.

[27] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. ISCA 1997.

[28] Wenisch et al. Temporal Streaming of Shared Memory. ISCA 2005.

[29] J. Larus, "Whole Program Paths," PLDI 1999.

[30] A. R. Pleszkun, "Techniques for compressing program address traces," MICRO 27, 1994.

[31] Brazma et al "Approaches to automatic discovery of patterns in biosequences," J. of Computational Biology 5(2):277-304, 1998.

[32] Rigoutsos, I., and Floratos, A., "Combinatorial pattern discovery in biological sequences: TEIRESIAS algorithm," Bioinformatics 1998.

[33] Weiner, P., "Linear pattern matching algorithms," In IEEE 14th Annual Symposium on Switching and Automata Theory, 1-11, 1973.

[34] C. Charras and T. Lecroq. Handbook of Exact String Matching Algorithms. King's College London Publications, 2004.

[35] Rasheed et al. Efficient periodicity mining in time series databases using suffix trees. IEEE TKDE, 23:79-94, 2011.

[36] H. Chim and X. Deng. A new suffix tree similarity measure for document clustering. In Proc. of ACM WWW, pages 121-130, 2007.

[37] Ferragina et al. Boosting textual compression in optimal linear time. Journal of ACM, 52:688-713, 2005.

[38] Giegerich et al, Efficient implementation of lazy suffix trees, 3rd Workshop on Algorithmic Eng (WAE99), 1999.

[39] Ukkonen, E., “On-line construction of suffix trees,” Algorithmica 14:249-260, 1995.

[40] Giegerich, R. and Kurtz, S. "A comparison of imperative and purely functional suffix tree constructions," Science of Computer Programming 25(2-3):187-218, 1995.

BIBLIOGRAPHY

J.-L. Baer and T.-F.Chen, “An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty,” SC 1991.

Bieganski et al “Generalized suffix trees for biological sequence data: Applications and implementation,” 27th Ann Hawaii Int. Conf. on System Sciences. Vol. 5: Biotechnology Computing, 35-44, 1994.

Brazma et al “Approaches to automatic discovery of patterns in biosequences,” J. of Computational Biology 5(2):277-304, 1998.

Chang et al, “Alternative implementations of hybrid branch predictors,” MICRO 1995.

C. Charras and T. Lecroq. Handbook of Exact String Matching Algorithms.King's College London Publications, 2004.

T. Chilimbi, “Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality,” PLDI 2001.

H. Chim and X. Deng. A new suffix tree similarity measure for document clustering. In Proc. of ACM WWW, pages 121-130, 2007.

Evers et al, “Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches,” ISCA 1996

Ferragina et al. Boosting textual compression in optimal linear time. Journal of ACM, 52:688-713, 2005.

Gabriel H. Loh, “Deconstructing the Frankenpredictor for Implementable Branch Predictors,” JILP 2005.

Giegerich et al, Efficient implementation of lazy suffix trees, 3rd Workshop on

Algorithmic Eng (WAE99), 1999.

Giegerich, R. and Kurtz, S. "A comparison of imperative and purely functional suffix tree constructions," Science of Computer Programming 25(2-3):187-218, 1995.

D. Gusfield, “Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology,” Cambridge: Cambridge University Press, 1997.

Gusfield et al “An efficient algorithm for all the pairs suffix-prefix problem,” Inf. Process. Lett. 41(4):181-185, 1992.

D. Jiménez and C. Lin, “Dynamic Branch Prediction with Perceptrons,” HPCA 2001.

D. Jiménez, “Piecewise Linear Branch Prediction,” ISCA 2005.

D. Jiménez, “Fast Path-Based Neural Branch Prediction,” MICRO 2004.

D. Joseph and D. Grunwald. Prefetching using Markov Predictors.ISCA 1997.

N. P. Jouppi, Improving Direct-Mapped Cache Perf.by the Addition of a Small Fully-Assoc. Cache and Prefetch Buffers, ISCA 1990.

J. Larus, “Whole Program Paths,” PLDI 1999.

S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

Nevill-Manning, C.G. and Witten, I.H., “Identifying Hierarchical Structure in Sequences: A linear-time algorithm,” Journal of Artificial Intelligence Research, 7, 67-82, 1997.

Pan et al “Improving the accuracy of dynamic branch prediction using branch correlation,” ASPLOS 1992.

A. R. Pleszkun, “Techniques for compressing program address traces,” MICRO 27, 1994

Rasheed et al. Efficient periodicity mining in time series databases using suffix trees. IEEE TKDE, 23:79-94, 2011.

Rigoutsos, I., and Floratos, A., "Combinatorial pattern discovery in biological sequences: TEIRESIAS algorithm," Bioinformatics 1998.

Roth et al. Dependence Based Prefetching for Linked Data Structures. ASPLOS 1998.

Sendag et al, "Branch Misprediction Prediction: Complementary Branch Predictors," IEEE Computer Architecture Letters, Dec. 2007.

A. Seznec, "Analysis of the O-GEometric History Length Branch Predictor," ISCA 2005

A. Seznec, "The L-TAGE predictor," JILP, May 2007

A. Seznec, "Redundant History Skewed Perceptron Predictors: pushing limits on global history branch predictors," IRISA Report No 1554, sept. 2003.

Somogyi et al. "Spatio-Temporal Memory Streaming," ISCA 2009

Ukkonen, E., "On-line construction of suffix trees," Algorithmica 14:249-260, 1995.

Weiner, P., "Linear pattern matching algorithms," In IEEE 14th Annual Symposium on Switching and Automata Theory, 1-11, 1973.

Wenisch et al "Temporal Streaming of Shared Memory" ISCA 2005.

Wenisch et al. "Practical Off-chip Meta-data for Temporal Memory Streaming," HPCA 2009.

Wenisch et al. Temporal Streaming of Shared Memory. ISCA 2005

W. A. Wulf and S. A. Mckee, "Hitting the Memory Wall: Implications of the Obvious," SIG. Comp. Arch. News, 23:20-24, 1995.

C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data

Structures. ICS 2000.

T.-Y. Yeh and Y. N. Patt."Alternative implementations of two-level adaptive branch prediction," ISCA 1992.