An Analysis of Address and Branch Patterns with PatternFinder

Celal Ozturk, Ibrahim Burak Karsli and Resit Sendag
Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island
Kingston, USA 02881
cozturk@ele.uri.edu, bkarsli@ele.uri.edu, sendag@ele.uri.edu

***Abstract*—Current processors employ aggressive prediction mechanisms to improve performance and reduce power. It is increasingly important to understand and quantify a program's dynamic behavior to effectively design next-generation prediction mechanisms. In this paper, we develop algorithms and mechanisms inspired by DNA discovery tools to analyze and quantify program dynamic behavior in terms of regularities and patterns. We describe our PatternFinder tool and analyze its results to summarize most important branch and data address pattern behaviors for a set of program traces and SPEC CPU 2006 benchmarks.**

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, 4]. 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 paper, inspired by DNA discovery tools [5, 6], we adopt and revise the methods motivated by suffix trees [5] 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 we discuss in Section 3.

In this paper, we present a novel pattern analysis tool, the PatternFinder, which is designed to discover and quantify patterns, and the results produced by the tool that quantify data address and branch patterns in dynamic program behavior. PatternFinder is an offline analysis tool that can summarize input sequences using patterns, quantify spatial and temporal regularities in input sequences and enumerate hot patterns based on specified scoring criteria (e.g., coverage). It allows user customizable search with a number of input parameters. PatternFinder can be used to gain insights into hardware and software optimization opportunities as well as perform pattern-centric benchmark classifications.

This paper makes the following contributions:

1) It presents design and implementation of the PatternFinder: 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. It also quantifies overlapping branch outcome patterns that have implications on predictability.
3) It quantifies last-level cache miss data address and address-delta patterns and discusses their implications on hardware prefetching.

The remainder of this paper is organized as follows: Section 2 describes the related work. Section 3 details the PatternFinder tool. Section 4 describes the experimental setup. In Section 5, we present pattern analysis for branch outcomes and address patterns, and discuss their implications on branch prediction and data prefetching. Finally, Section 6 concludes.

2. BACKGROUND AND RELATED WORK

2.1. Patterns and Processor Design

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.).

A. 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 [7, 8]. 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 [9], with some way to choose which predictor to use at any given time. Recent works, such as and `ISL-TAGE` [10] exploit much longer histories than prior predictors. These predictors employ multiple prediction tables indexed with different length folded histories. Several others [11-12] target longer histories based on neural networks.

B. Data Prefetching and Memory Access Patterns

Hardware data prefetching is a well-known technique to help alleviate the `memory wall` problem [13]. Many general purpose microprocessors rely on data prefetching to improve performance for memory-intensive workloads. Most of the early prefetchers [14, 15] 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 [16, 17]. In recent years, [4, 18] 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 [19] 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 [19] is limited in that only exact non-overlapping patterns are investigated. As mentioned in Section 1, approximate patterns provide better insight into memory access behavior.

2.2. 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. [20] 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. We followed Brazma's methodology for developing the `PatternFinder`.

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 [5, 21]. Suffix trees are used to accelerate many string operations [22] by indexing texts (sequences) in a way so that query times would not depend on the size of the indexed text. 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 [5], time series analysis [23], document clustering [24] and compression [25].

In this paper, we apply the methods motivated by the suffix trees for pattern discovery from dynamic program execution traces. For the discovery of the most frequent patterns we adapt the `write-only top-down (WOTD)` algorithm for lazy construction of the suffix trees [26]. 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 [21] 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. Although the `WOTD` algorithm is an $O(n.logn)$ average time algorithm for the text length n, it has been shown to on average work in $O(n)$ time and sometimes to outperform the linear-time algorithms [27].

3. PATTERNFINDER ANALYSIS TOOL

As a first step to enable pattern-aware design, we have implemented a pattern tool to find overlapping and non-overlapping patterns and analyze patterns of branch outcomes and data addresses for a set of benchmarks. In this section, we first discuss why `SEQUITUR` is not our first choice for frequent pattern analysis of programs. We, then, describe our `PatternFinder` tool.

3.1. Limitations of the SEQUITUR algorithm

`SEQUITUR` is a string compression algorithm that constructs a context-free grammar for its input. This algorithm has been used to find hierarchical structures in a variety of sequences, ranging from `DNA` sequences to

genealogical databases. The insight underlying the algorithm is that log N rules can generate N occurrences of a subsequence. For example, the string: S→xyzxyzxyzxyzxyz is produced by the grammar: S→112, 1→22, 2→xyz. This grammar requires fewer symbols (11 versus 15), and, explicitly captures repetitions of the pattern xyz. Nevill-Manning and Witten proved that SEQUITUR runs in time linear in the length of the input string [2].

Chilimbi [1] applied SEQUITUR algorithm to an input trace. Once the grammar is generated, they further processed the SEQUITUR output to quantify hot data streams. Although SEQUITUR provides a compressed yet relatively easy to understand grammar to summarize an input sequence, it is not suited for finding overlapping or approximate patterns. As we discuss in later sections, one might be interested in overlapping patterns for branch outcomes, while non-overlapping patterns are more important for streaming behavior. Furthermore, approximate patterns are important for memory streaming since prefetching does not have to be perfect and observing the overall behavior is important to make early design decisions.

Finally, with SEQUITUR, there is no guarantee in finding the most important or relevant non-overlapping patterns in an input sequence. For example, consider the string S→caabaaabacd. SEQUITUR generates the following grammar: S→c1b1abacd, 1→aa, which may be misleading. For the same input sequence, WOTD suffix tree approach will exhaustively find all frequent patterns including the longest sequence aaba. WOTD algorithm can also be tailored for finding overlapping, non-overlapping and approximate patterns as well as for finding similar behavior in multiple input sequences with the user's required scoring criteria.

3.2. PatternFinder Implementation Details

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. In terms of compression, unlike SEQUITUR, PatternFinder can only provide lossy compression because subsequences that occur only once and single data points are not considered patterns. Therefore, it cannot 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 to few hours for 100M-long input traces that we analyzed for this paper. Although this may seem slow, the pattern analysis results presented in this paper can only be done, otherwise, by running thousands of simulations. And in many cases, it is not possible to do similar analysis with an architecture simulator.

A. *Finding Overlapping Patterns*

In this study, we are interested in patterns that occur at least k times in a sequence, S. We aim at a solution that is faster for larger values of k, keeps the space requirement relatively low, and at the same time is simple to understand and implement. We represent the algorithm for constructing the $O(n^2)$ time and space, suffix trie instead of the compact suffix tree. The trie variant is easier to describe and implement, as well as it allows us to generalize this algorithm for discovering patterns from more complex pattern classes.

Our 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 different parts of the trie and the physical implementation of tree nodes can be optimized by knowing exactly how many children the node will have. Example of such a trie construction is in Figure 1 for an input sequence S→caabaaabacd. In this figure, each node corresponds to one specific pattern and each level may have many nodes (patterns). The numbers in parentheses correspond to the positions (end pointers instead of start pointers are used for algorithmic convenience) where a subsequence pattern occurs in the input sequence. Note that all patterns, overlapping and non-overlapping, are explored.

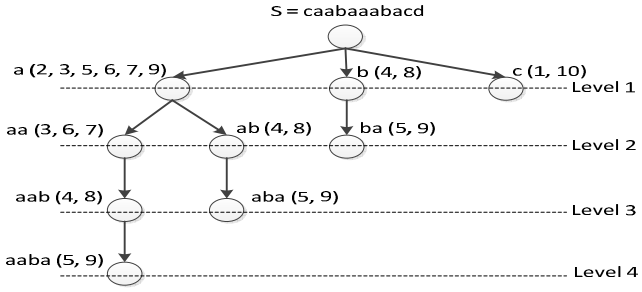

Figure 1: Discovering the subsequences of sequence S→caabaaabacd having at least 2 occurrences in S.

B. *Non-Overlapping Patterns*

We 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 an input sequence with non-overlapping subsequence patterns, PatternFinder first finds the longest pattern/s according to some user-defined importance criteria (e.g., occur at least k times or maximum pattern length of L that occur at least k times, etc.). Non-overlapping occurrences of this pattern (or patterns) 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

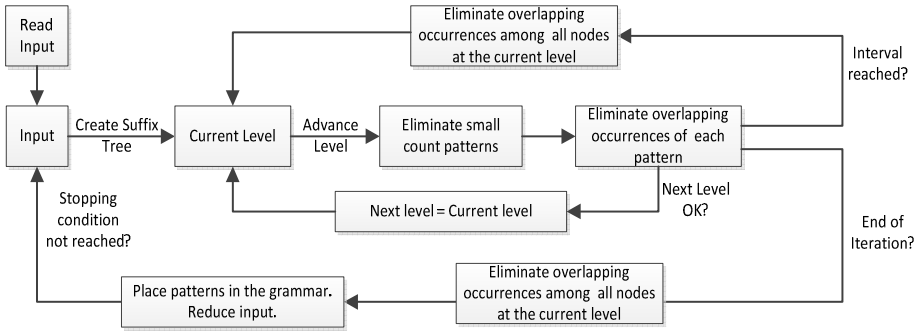

Figure 2: Summarizing an input sequence with non-overlapping patterns with `PatternFinder`

coverage). We call each of these steps, an *iteration*. `PatternFinder` increases its coverage of the input sequence by running many 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[1]. First iterations are slower since they go deeper in the tree finding longer patterns. Later iterations can be much faster depending on how many previous levels of the suffix trie can be saved during construction.

The process shown in Figure 2 is very computationally intensive. To reduce the simulation time, we eliminate overlapping occurrences of the pattern within a node during the construction of the suffix tree for a particular iteration. For example, in Figure 1, at Level 2 (i.e., the pattern length of 2), for pattern *aa*(3, 6, 7), the second and third occurrences of it overlap, therefore, its third occurrence is eliminated to update the occurrences list to be non-overlapping as *aa*(3, 6). This eliminates significant number of patterns from the suffix tree, which in turn improves processing times. To further reduce the execution time, at every *N* (e.g., 20) levels (called `interval` in Figure 2) of the suffix tree, we eliminate the overlaps between different nodes (patterns) at that level. An example for overlapping patterns can be seen in Figure 1 at Level 3 between *aab*(4, 8) and *aba*(5, 9). The overlap elimination is not done at every level to lower chances of eliminating important patterns. Other speed optimizations include three conditions where an iteration must be terminated early: 1) when reached user-given maximum pattern length, 2) when a level's (pattern length's) coverage falls under a user-given threshold, and 3) when average percent drop in coverage is over a user-given threshold. Finally, we introduce a parameter for early termination of the whole process in Figure 2: when a user-given maximum input coverage is reached the program terminates.

[1] When an iteration ends, the patterns at the current level are placed in the grammar and input is reduced. Later iterations find (increasingly shorter) patterns in the reduced input. The pattern placement starts with the highest score pattern (e.g., most frequent). Overlapping occurrences of other patterns with this pattern are eliminated. Then the next pattern with the highest score is placed. If all occurrences of a pattern at this level overlap with patterns placed before it, it never gets placed. This placement method is not optimal. However, the optimal placement operation is NP-hard. Different placement methodologies that we have tested perform similarly, better methods need further research.

4. EXPERIMENTAL METHODOLOGY

In this study, we used a set of 40 benchmark traces (16 client, 6 integer, 7 multimedia, 5 server, 6 workstation applications, and each benchmark trace for 50M dynamic instructions) provided with the 2011 Third Championship Branch Prediction (CBP) Competition [29] framework to study branch outcome patterns. We studied address patterns for SPEC CPU 2006 benchmarks [28] using the traces generated by Gem5 [30] simulator targeted for x86 ISA. All SPEC benchmarks were compiled with gcc full-optimization.

To study branch outcome patterns, we generate the `PatternFinder` input trace using the CBP framework. The traces record branch PCs and their outcomes (1/0) although, in this study, we only investigate global branch outcome patterns. We ran the best performing (winner of the CBP competition) state-of-the-art `TAGE` [10] branch predictor on our benchmarks to be able to correlate `PatternFinder`'s results. `TAGE` predictor uses a number of prediction tables (16 for our simulations) with increasing branch outcome history. For our simulations, we use 16 different length histories to form a geometric series (as suggested by `TAGE`) between 3 and 2000 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.

To study SPEC CPU 2006 address patterns, we generate the address traces for 2MB last-level cache (LLC) misses. The benchmarks were run for 100M instructions after fast-forwarding 40B instructions with the reference input sets (although we present results for only one input set, we did not observe significant changes in patterns with different input sets). The `PatternFinder` input traces include the LLC-missed PCs and their addresses. We investigated both global and local address and delta patterns.

Finally, all measurements in this paper were performed on an Intel Xeon X5460 quad-core processor with 8GB of memory running Ubuntu Linux 10.04 operating system.

5. EVALUATION

In this section, we use the `PatternFinder` to study branch outcome and memory address patterns and discuss their implications on branch prediction and data prefetching, respectively.

5.1. Branch Patterns

We explored both overlapping and non-overlapping global branch outcome patterns. Overlapping patterns investigates branch predictability and gives insights into the design of next-generation branch predictors, while long non-overlapping patterns with good input coverage suggest branch streaming.

A. Non-overlapping Branch Patterns

We first investigate non-overlapping global branch outcome patterns for spatial and temporal regularities and branch streaming opportunities. We define *spatial regularity* 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.

Figure 3 illustrates the cumulative distribution of hot pattern sizes, which summarize the spatial regularity. In this figure, 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. Long patterns indicate good spatial regularity. The curves closer to the top left corner represent benchmarks with relatively worst spatial locality. For example, top left cluster of lines in Figure 3, `MM07`, `WS03`, `WS04`, `INT02`, `MM05` and `INT01` have the worst spatial behavior as more than 95% of their patterns are less than 100 references long. On the other hand, `INT04`, `CLIENT02`, `CLIENT03`, `CLIENT06`, `MM06`, `INT03` and `SERVER01` have best spatial locality with better distribution of pattern lengths: 60% or more of their patterns are longer than 700 references long. This analysis suggests that **branch streaming has great potential for benchmarks with good spatial regularity**. For example, `CLIENT02` and `SERVER01` have very long non-overlapping patterns with very good coverage, 80% and 35%, respectively. A mechanism similar to memory streaming proposals [3, 4, 31-35] can be used for also branch outcome streaming.

Table 1 presents important branch pattern characteristics for CBP traces. We can see that for most of the traces, extensive pattern lengths are observed. For `INT03`, `INT04`, `CLIENT02`, `CLIENT07` and `SERVER01`, maximum pattern length is larger than 200K. The longest pattern length observed is 2.9M for `CLIENT02`. Table 1 also shows the weighted average pattern lengths (WAPL) and repetition

Table 1: Branch Patterns: Maximum Non-overlapping Pattern Length and Its Coverage and Spatial and Temporal Regularity. **WAPL**: Weighted Average Pattern Length. **WARI**: Weighted Average Repetition Interval. WAPL and WARI are generated for maximum pattern length of 1000.

Benchmark	Dynamic Branches	Max. Length	WAPL	WARI	TAGE misp. %
INT01	6.0M	67.7K	184.74	1.11M	9.03
INT02	5.4M	26.6K	157.42	1.12M	12.1
INT03	5.1M	236.6K	917.60	8.41K	0.01
INT04	7.9M	1.5M	751.28	0.27M	0.09
INT05	3.0M	8.1K	159.97	0.65M	4.1
INT06	2.9M	8.3K	191.51	0.62M	4.21
CLIENT01	3.9M	15.2K	748.56	0.32M	0.49
CLIENT02	15.1M	2.9M	793.99	1.35M	5.05
CLIENT03	4.8M	33.2K	775.84	0.46M	0.43
CLIENT04	4.4M	1.7K	310.26	1.09M	2.66
CLIENT05	3.8M	152.0K	547.55	0.17M	1.73
CLIENT06	8.6M	84.2K	788.23	0.43M	0.14
CLIENT07	5.7M	289.7K	775.86	0.24M	1.29
CLIENT08	3.5M	33.9K	739.19	0.24M	0.78
CLIENT09	3.5M	49.3K	495.56	0.33M	1.62
CLIENT10	3.2M	15.4K	584.23	0.26M	1.19
CLIENT11	4.8M	3.3K	591.68	0.39M	0.76
CLIENT12	3.7M	14.2K	404.53	0.4M	1.98
CLIENT13	4.1M	24.8K	760.73	0.32M	0.41
CLIENT14	4.2M	115.6K	740.62	0.14M	0.56
CLIENT15	4.7M	52.0K	643.41	0.17M	1.0
CLIENT16	4.4M	70.2K	674.58	0.21M	0.71
WS01	4.8M	17.9K	488.12	0.33M	1.99
WS02	3.6M	40.4K	626.00	0.15M	1.46
WS03	7.3M	5.4K	69.72	1.28M	14.0
WS04	4.1M	102.9K	232.13	0.63M	22.3
WS05	3.5M	12.0K	667.93	0.27M	0.45
WS06	4.4M	23.9K	656.28	0.23M	0.6
SERVER01	4.2M	655.4K	344.96	0.47M	3.79
SERVER02	4.0M	21.8K	553.92	0.82M	1.02
SERVER03	3.7M	8.7K	573.63	0.71M	0.96
SERVER04	3.8M	9.2K	610.25	0.47M	0.92
SERVER05	3.7M	40.3K	592.10	0.54M	0.86
MM01	4.0M	55.1K	343.61	0.19M	2.49
MM02	4.0M	24.2K	342.98	0.37M	2.62
MM03	4.5M	25.7K	263.19	0.36M	2.62
MM04	3.8M	25.5K	344.02	0.26M	2.62
MM05	5.4M	0.9K	73.81	1.33M	14.3
MM06	1.8M	46.9K	846.06	47.4K	0.14
MM07	6.1M	0.1K	46.63	1.54M	14.0

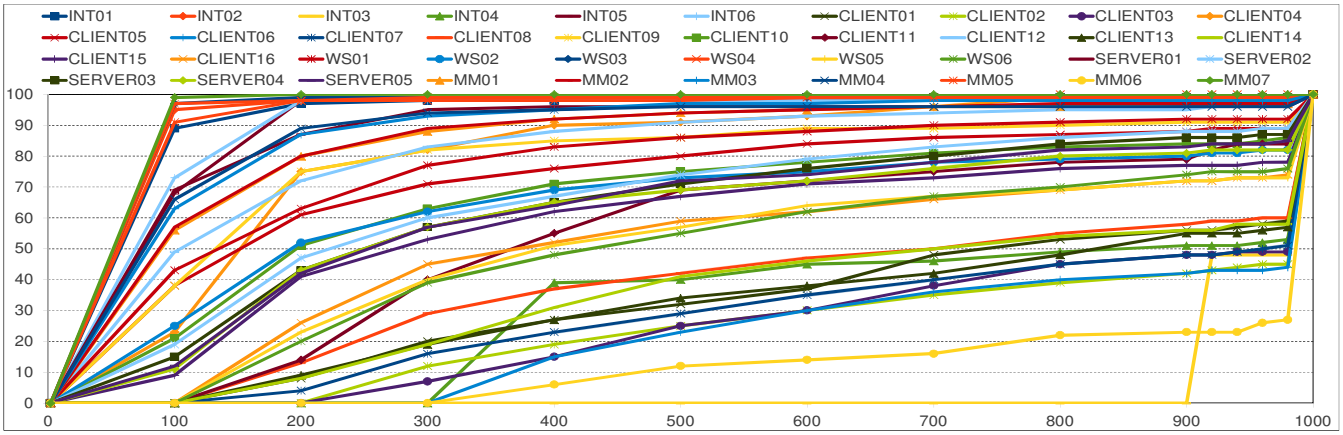

Figure 3: Cumulative distribution of hot pattern sizes (spatial regularity) for CBP traces. x-axis: pattern length, y-axis: % number of patterns. Simulations are run for minimum pattern length of 2, maximum pattern length of 1000 and 90% coverage.

intervals (WARI) (columns 3 and 4), 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, WS03, and MM05 (in Figure 3) have the smallest WAPL. WARI (temporal regularity) is expressed in terms of number of references. Benchmarks with higher WARI, e.g., MM07, CLIENT02, MM05, WS03, exhibit low temporal locality. Better temporal locality (e.g., MM06, INT03) suggests better history table prediction opportunity because when WARI is large, it is more likely that table history is polluted with other patterns. We can see that spatial and temporal regularities correlate well with branch misprediction rates (Table 1, last column). Benchmarks with better spatial and temporal localities tend to have lower misprediction rates.

B. Overlapping Branch Patterns

Overlapping branch outcome patterns provide the best information about branch predictability. This analysis can be performed as the suffix tree is constructed. For each unique branch pattern of any length, we compute the approximate confidence to predict the next branch outcome. For example, for a frequent pattern of 10110, we check the frequency of patterns 101100 and 101101 (the children nodes). If one of them occurs for 99 times (most frequently seen outcome) and the other for 1 time, the confidence of 10110 is assumed to be 99% (the larger outcome). For a user-given confidence threshold, we can then find the minimum pattern length that would correctly predict each instance of the branch outcomes in the input stream. An example analysis output of PatternFinder for overlapping exact branch outcome patterns is given in Tables 2 and 3, with and without PC, respectively, for WS04 benchmark. Figures 4a and 4b show the results, for prediction coverage with global history only and with global history and PC, respectively (due to the space limitations and readability, we show the results for a few CBP benchmark traces: 2 from each benchmark class with one easier to predict and one harder to predict). As expected, as the branch history length (L) increases, so does the coverage. INT03 reaches 99.99% coverage at L = 20, while CLIENT02 shows a more uniform prediction by pattern lengths and achieves 87.12% at L = 100. CLIENT02's prediction rate reaches 99.4% at L = 200. Only CLIENT01 benefits more than 1% from L > 400. Figure 4a shows that prediction coverage with global history only does surprisingly well, almost as well as when both global history and PC is used (Figure 4b) except that it requires longer history lengths.

Table 4 shows the prediction opportunity with long history lengths. It is important to note that only small percentage of unique patterns, on average, 0.007%, are used for prediction (the ones that satisfy high confidence level, e.g., 99% for this example). CLIENT01, INT03, SERVER04 and WS05 have relatively small percentage of unique patterns used, which suggest better prediction rates with limited size prediction tables. Most benchmarks benefit from L > 100, although few benefits from L > 400. Furthermore, by only using global history (without the PC), many benchmark can achieve more than 99% prediction rate, which is surprising. In Table 4, we also show the true upper-bound prediction rate that can be obtained if PC and global history are used together. For this analysis, we search for the most confident pattern length to predict each branch outcome instance in the input sequence. This upper-bound study can easily be done using PatternFinder during the suffix trie construction; however, it is not feasible to do the same using table-based prediction methods due to its memory requirements. Comparing the true upper-bound prediction rate with the current-state-of-the-art branch predictor ISL-TAGE [10]'s results (see Table 4), we see that there is still room for improvement.

Table 2: Example analysis output for overlapping patterns summary for WS04 with PC and global branch outcome history

	All Patterns					> 99% Confidence Patterns			
PatternLen	NumUniq	Coverage	Confidence	AvgOccurrence	AvgDist	NumUniq	CumulativeCoverage	LevelCoverage	NumUniqPCs
1	3110	100	73.8718	1318.13	1439.84	2752	28.99305	28.99305	2601
2	3382	100	75.3805	1212.58	1582.93	102	30.75537	1.762323	77
3	3700	99.9999	75.8246	1108.36	1775.4	91	30.90209	0.146722	71
…									

Table 3: Example analysis output for overlapping patterns summary for WS04 with global branch outcome history only

	All Patterns					> 99% Confidence Patterns	
PatternLength	NumUnique	Coverage	Confidence	AvgOccurrence	AvgDist	NumUnique	Coverage %
…							
8	256	99.9998	67.9695	16019.3	255.836	0	0
9	512	99.9998	68.9511	8009.66	511.27	1	0.824174
…							

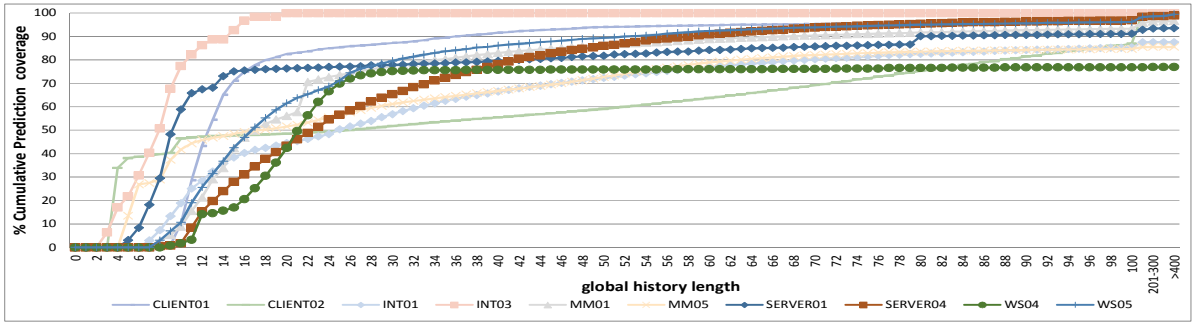

Figure 4a: Prediction coverage (global hist. only) for global branch history pattern lengths for 99% confidence threshold

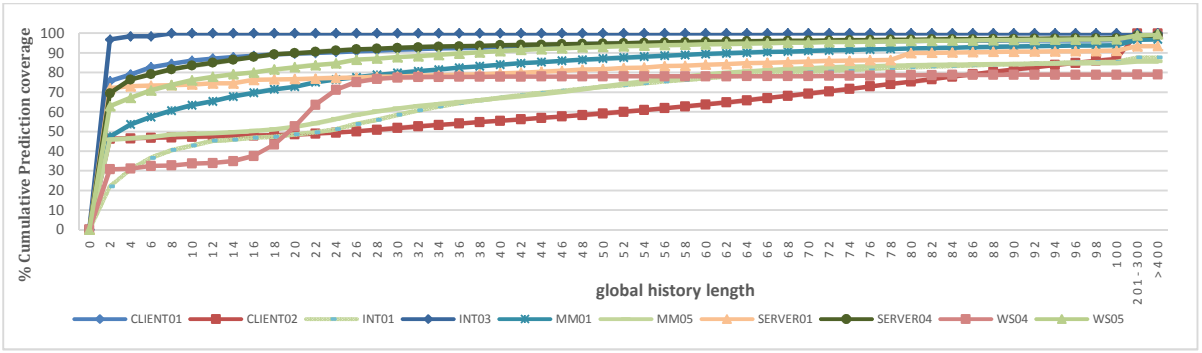

Figure 4b: Prediction coverage (PC+global hist.) for global branch history pattern lengths for 99% confidence threshold

Table 4: Prediction Opportunity with Long History Lengths, Overall Prediction Rates and ISL-TAGE misprediction rates

	CLIENT01	CLIENT02	INT01	INT03	MM01	MM05	SERVER01	SERVER04	WS04	WS05
>100	3.46	12.88	2.21	0.00	3.23	0.88	2.36	2.15	0.17	3.26
>400	1.42	0.00	0.01	0.00	0.14	0.00	0.02	0.33	0.06	0.66
>99% confident Branch Outcomes	99.37	99.79	87.46	99.99	96.73	85.54	93.39	98.88	78.95	99.38
Upper Limit Prediction Rate	99.80	99.90	95.24	99.99	98.99	94.45	97.39	99.56	91.91	99.56
Total Number of Unique Patterns	1,962M	29,422M	2,222M	126M	223M	73M	31M	609M	550M	1,233M
>99% confident Unique Patterns	28.2K	697.5K	480.8K	0.5K	111.8K	413.6K	113.4K	38.7K	573.5K	27.1K
Max Pattern Length	15.2K	2.9M	67.7K	236.6K	55.1K	0.9K	655.4K	9.2K	102.9K	12.0K
ISL-TAGE Prediction Rate	99.51	94.95	90.97	99.99	97.51	85.70	96.21	99.08	77.70	99.55

C. Branch Prediction

We discussed how branch predictability analysis can be performed by finding overlapping global branch outcome patterns. By finding the most confident pattern for each branch outcome instance, we can obtain the true upper-bound branch prediction rate as shown in Table 5. We can see that there is still room for improvement in branch prediction accuracy (between ISL-TAGE and upper-bound), especially for `CLIENT02`, `INT01`, `MM05`, `MM07`, `SERVER01`, `WS03` and `WS04`. There are important observations that we make from Table 5. First, for many benchmarks almost all branch outcomes can be predicted by very high confidence (>99%). Second, the benchmarks with lower percentage of ">99%-confident" prediction contribution perform worse with ISL-TAGE. `CLIENT02` is an exception where ISL-TAGE does not perform well and almost all prediction contribution comes from almost 100% prediction rate. It is important to understand how ISL-TAGE performs on branch outcomes that are not highly confident. We can then correlate the results in Table 5 with ISL-TAGE's behavior. For example, `WS04`'s behavior in Table 5 shows that ISL-TAGE performs very poorly. If ISL-TAGE does well on high confident branch outcomes (~79%), that means, it fails to predict all other branches.

When we analyze the results from 40 CBP traces (some analysis regarding number of unique patterns not shown), the results agree with some of the design decisions that the ISL-TAGE predictor [10] made. However, some of our results suggest modifications to TAGE's operation. For example, the following is in agreement with the TAGE's decisions: 1) While some branch outcomes can confidently be predicted with short histories, some others require longer histories. Therefore, a branch predictor must be able to keep track of multiple history lengths (using multiple tables). 2) Number of unique patterns is much larger than number of unique patterns useful for predictions (< 0.1%). Therefore, we must use tags in prediction tables for better utilization.

Table 5: Branch Prediction contribution with patterns of varying confidence, overall prediction rate (true upper-bound) and ISL-TAGE prediction rate

Benchmark	99%-100%	90%-99%	80%-89%	70%-79%	60%-69%	50%-59%	Pred Overall	ISL-TAGE Pred.
CLIENT01	99.37	0.25	0.07	0.04	0.04	0.02	99.80	99.51
CLIENT02	99.79	0.04	0.02	0.02	0.02	0.01	99.90	94.95
CLIENT03	99.69	0.16	0.03	0.01	0.01	0.00	99.91	99.57
CLIENT04	96.00	0.42	0.73	0.86	0.25	0.17	98.44	97.34
CLIENT05	98.54	0.35	0.18	0.09	0.09	0.04	99.30	98.27
CLIENT06	99.55	0.19	0.07	0.01	0.01	0.01	99.85	99.85
CLIENT07	99.32	0.23	0.06	0.04	0.04	0.02	99.70	98.70
CLIENT08	99.47	0.17	0.05	0.03	0.02	0.01	99.75	99.22
CLIENT09	97.48	0.51	0.30	0.33	0.28	0.10	99.00	98.38
CLIENT10	99.09	0.18	0.13	0.08	0.08	0.03	99.59	98.81
CLIENT11	99.09	0.17	0.14	0.07	0.09	0.05	99.60	99.24
CLIENT12	98.32	0.41	0.22	0.14	0.14	0.07	99.30	98.02
CLIENT13	99.62	0.16	0.03	0.02	0.01	0.01	99.86	99.59
CLIENT14	99.31	0.28	0.06	0.02	0.02	0.01	99.71	99.44
CLIENT15	98.93	0.35	0.11	0.06	0.05	0.02	99.53	99.00
CLIENT16	99.24	0.26	0.08	0.04	0.03	0.01	99.67	99.29
INT01	87.46	2.52	2.28	1.50	1.06	0.42	95.24	90.97
INT02	80.73	4.16	3.30	2.42	2.02	0.86	93.49	87.90
INT03	99.98	0.00	0.00	0.00	0.00	0.00	99.98	99.99
INT04	99.80	0.03	0.00	0.00	0.00	0.00	99.83	99.91
INT05	91.63	1.74	1.62	1.09	0.42	0.47	96.96	95.90
INT06	91.92	1.78	1.33	1.23	0.40	0.44	97.09	95.79
MM01	96.73	1.17	0.38	0.23	0.22	0.12	98.84	97.51
MM02	96.71	1.15	0.40	0.24	0.23	0.09	98.82	97.38
MM03	96.03	1.43	0.50	0.27	0.23	0.10	98.55	97.38
MM04	96.01	1.27	0.54	0.32	0.36	0.12	98.62	97.38
MM05	85.54	1.53	2.32	2.15	2.00	0.90	94.45	85.70
MM06	99.56	0.20	0.06	0.04	0.02	0.02	99.89	99.86
MM07	82.30	0.37	2.47	2.11	3.63	2.76	93.62	86.00
SERVER01	93.39	1.26	0.79	0.69	1.00	0.25	97.39	96.21
SERVER02	99.02	0.34	0.09	0.06	0.05	0.03	99.60	98.98
SERVER03	99.14	0.29	0.10	0.06	0.04	0.02	99.65	99.04
SERVER04	98.88	0.28	0.13	0.12	0.10	0.06	99.56	99.08
SERVER05	98.96	0.27	0.11	0.11	0.10	0.05	99.60	99.14
WS01	98.19	0.51	0.27	0.16	0.13	0.06	99.32	98.01
WS02	98.64	0.38	0.14	0.08	0.09	0.05	99.39	98.54
WS03	79.24	4.61	4.24	2.37	1.84	0.83	93.12	86.00
WS04	78.95	1.03	2.59	3.14	4.03	2.16	91.91	77.70
WS05	99.38	0.25	0.04	0.03	0.03	0.02	99.73	99.55
WS06	99.12	0.33	0.13	0.07	0.03	0.03	99.71	99.40

Our results suggest the following modifications to TAGE's operation:

History Length: The opportunity to predict better with history lengths greater than 200 is low (although in some cases, even a small percentage, such as 0.1%, could be important). We must utilize shorter history tables better. For example, our results show that `INT03` reaches a prediction

rate of 99.99% with history tables that target history lengths of 20 or shorter. That is, most of the tables (11 out of 15) that TAGE use for long histories are wasted.

Tables: 1) Prediction table sizes must not be uniform: shorter patterns could be given smaller tagged-tables because number of unique patterns is much lower for shorter histories. In fact, number of unique patterns with high confidence is very low for short history lengths. 2) Since longer histories have better confidence, smaller saturating counters may be used.

Table Allocation Policies: 1) Examining few of the benchmarks' pattern summaries and prediction provider tables in ISL-TAGE, we find the following: almost 12% of `CLIENT02`'s very confident (~100%) patterns are long patterns. However, ISL-TAGE does not provide any predictions using long-history tables. A closer look reveals that `CLIENT02` has too many (more than 2M) unique long patterns and recurrence distance is too long (>1M branches) and therefore long history tables experience many evictions. These tables could receive more hits if some number of patterns retain in the tables and not get evicted. This suggests considering recurring distance history in allocation decisions. 2) ISL-TAGE allocates entries in longer-history tables when a shorter-history table's prediction has been incorrect. In order to increase the number of predictions with higher confidence patterns, allocation policy (based on our observations from pattern analysis) can be changed such that longer-history table entries are allocated not only on a misprediction but also on a not very highly-confident prediction.

Provider Tables: For ISL-TAGE, if multiple tables have tag hits, the prediction is provided by the longest-history matching table. Based on our pattern analysis, this policy can be changed so that prediction gets provided with shorter history table if confidence is very high (above a threshold, e.g, 99%). This is done to reduce the pressure on longer history tables.

5.2. Address Patterns

In this section, we present the LCC-miss address patterns for SPEC CPU 2006 benchmarks. The following benchmarks were not studied because they did not have sufficient number of misses with 2MB LLC: `410.bwaves`, `416.gamess`, `444.namd`, `445.gobmk`, `447.dealII`, `456.hmmer`, `454.calculix`, `458.sjeng`, `464.h264ref`, `465.tonto`, `473.astar` and `481.wrf`.

A. *Address-Delta Miss Patterns*

Since, for efficient design of a hardware prefetcher, short delta patterns are more important, we have summarized the LLC-miss delta input sequences with delta pattern lengths of 2 to 8. In this and later discussions, **it is important to note that a delta pattern is not a single delta (stride) but a sequence of deltas of length between 2 and 8.**

Figure 5 shows the top 10 global delta patterns' contributions in miss coverage. We observe that `400.perlbench`, `429.mcf`, `459.GemFDTD`, `471.omnetpp` and `450.soplex` (cluster in the bottom right of the figure) have insignificant or no global delta patterns. On the other hand, for many others, there are global delta patterns with significant coverage. For `462.libquantum`, one delta pattern has 99% coverage. For `434.zeusmp`, `433.milc` and `470.lbm`, 5 unique global delta patterns have 70% coverage.

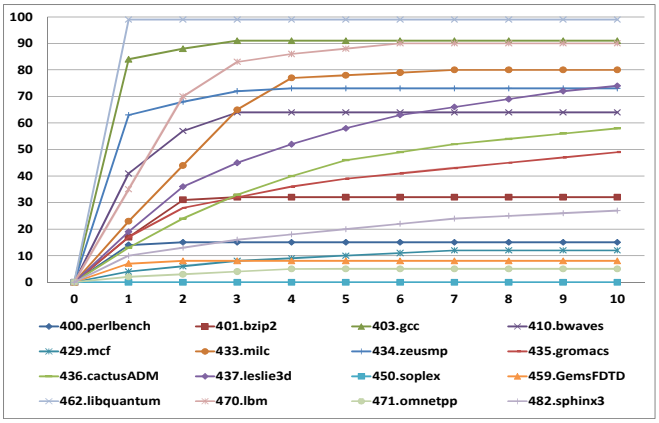

Figure 5: Top 10 delta patterns' cumulative contributions. x-axis: top 1 to 10 delta patterns, y-axis: % coverage. Simulations are run for minimum pattern length of 2, maximum pattern length of 8 and for 90% coverage.

Table 6: LLC Miss Global Address-Delta Patterns

Benchmark	unique patterns	number of all patterns	Weighted Avg.Len	Weighted Avg. Rep. Interval	Coverage
400.perlbench	637	11374	2.0	1068.4	8.0
401.bzip2	164	5853	2.9	85.2	16.7
403.gcc	5	15225	7.4	34.4	85.2
429.mcf	45763	734902	4.4	184914.8	55.3
433.milc	13	257504	6.0	39.5	90.2
434.zeusmp	155	54265	6.3	602.4	84.9
435.gromacs	1797	42805	5.4	10160.6	76.3
436.cactusADM	171	56299	6.6	123.0	87.9
437.leslie3d	64	135397	6.8	184.2	99.8
450.soplex	0	0	0	0	0
459.GemsFDTD	4869	161187	6.6	56.3	93.3
462.libquantum	3	336634	8.0	23.6	100.0
470.lbm	21	447818	6.7	32.9	100.0
471.omnetpp	5678	236051	2.0	15069.4	27.4
482.sphinx3	12656	200207	6.0	72696.7	96.5

Tables 6 and 7 illustrate the details about global and local delta pattern summaries, respectively. We observe that `450.soplex` does not have any global delta patterns. It has relatively significant local delta patterns, with top 3 PCs (18% of miss coverage) having good regularity with very good coverage. `462.libquantum`, `470.lbm`, `433.milc`, `437.leslie3d` have very good global delta locality with few delta patterns covering over 90% of the LLC misses. `400.perlbench` does not have significant coverage with global delta patterns and it has relatively large number of unique patterns, which means that global delta patterns does not suggest prefetching. On the other hand, `400.perlbench` has significant local delta patterns that have good coverage, which suggests exploiting local delta patterns. `429.mcf` has 55% coverage with global delta patterns, however, too many unique patterns suggest that global delta regularity is low. As Table 7 shows, for `429.mcf`, only one PC has local delta regularity and is candidate for gain from prefetching. `436.cactusADM` has good local delta regularity (one delta pattern with 99% coverage for each top PC), however, there are too many streams (each PC's contribution too low), on the other hand,

Table 7: LLC Miss Local Address-Delta Patterns. For each top 5 hot PCs, its most important 1, 5 and 10 local delta patterns' miss coverage (in percentage) is shown. For example, for 400.perlbench, PC1 corresponds to 22% of all LLC misses and 1, 5 and 10 local delta patterns for that PC can eliminate 88, 91 and 91% of PC1's LLC misses. Last column shows each PC's percent contribution to the overall LLC misses and their sum (i.e., top 5 PCs' miss coverage).

Benchmark	PC1			PC2			PC3			PC4			PC5			Total (the sum of each PC's (PC1-PC5) individual Miss Coverage)
	top1	top5	top10	top1	top5	top10	top1	top5	top10	top1	top5	top10	top1	top5	top10	
400.perlbench	88	91	91	0	0	0	65	84	87	90	98	98	88	97	97	22 + 22 + 11 + 8 + 8 = **71**
401.bzip2	99	99	99	98	99	99	0	0	0	0	0	0	0	0	0	15 + 12 + 9 + 7 + 7 = **50**
403.gcc	99	99	99	90	90	90	85	97	97	85	97	97	85	97	97	45 + 9 + 5 + 5 + 5 = **69**
429.mcf	77	87	87	0	0	0	0	0	0	0	0	0	0	0	0	24 + 19 + 10 + 10 + 8 = **71**
433.milc	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	21 + 6 + 6 + 6 + 6 = **45**
434.zeusmp	41	96	96	41	95	95	41	95	95	41	95	95	42	92	92	4 + 4 + 4 + 4 + 4 = **20**
435.gromacs	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	4 + 2 + 2 + 2 + 2 = **12**
436.cactusADM	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	1 + 1 + 1 + 1 + 1 = **5**
437.leslie3d	50	95	95	50	94	94	99	99	99	99	99	99	99	99	99	11 + 10 + 2 + 2 + 2 = **27**
450.soplex	80	84	84	97	97	97	93	93	93	23	27	27	1	1	1	9 + 5 + 4 + 4 + 3 = **25**
459.GemsFDTD	88	97	97	88	97	97	88	97	97	88	95	95	88	95	95	6 + 6 + 6 + 6 + 6 = **30**
462.libquantum	99	99	99	99	99	99	0	0	0	0	0	0	0	0	0	58 + 42 + 0 + 0 + 0 = **100**
470.lbm	99	99	99	97	97	97	97	97	97	97	97	97	97	97	97	9 + 8 + 8 + 8 + 8 = **41**
471.omnetpp	9	29	35	20	59	64	8	27	35	0	0	0	0	0	0	3 + 2 + 2 + 2 + 2 = **11**
482.sphinx3	86	86	86	86	86	86	99	99	99	99	99	99	17	36	46	27 + 27 + 8 + 8 + 2 = **72**

it also has good global delta pattern coverage. `482.sphinx3` exhibits an interesting behavior with very good global delta pattern coverage, however, too many unique global delta patterns. Table 7 shows that, `482.sphinx` has good local delta regularity, with one pattern providing significant coverage for each of the top 4 PCs that corresponds to 70% of the LLC misses. So, although, it has too many unique global delta patterns, it does not have too many local delta patterns and suggest good performance with prefetching. `471.omnetpp` exhibit poor global and local delta regularities and thereby is not a good candidate for prefetching.

B. Spatial miss-address regularity

Figure 6 illustrates the cumulative distribution of hot address pattern sizes. `462.libquantum` and `429.mcf` have long address patterns (right bottom) that points to good spatial regularity. This is especially interesting for `429.mcf` which has most of its misses due to pointer chasing. On the other hand, `437.leslie3d`, `459.GemsFDTD` and `401.bzip2` (left top) have very short address patterns and thus exhibit worst spatial behavior. Finally, `400.perlbench` and `470.lbm` do not have any miss-address patterns (at least for the simulated 100M instructions). Also, for `433.milc`, `434.zeusmp` and `437.leslie`, there are no significant address patterns.

C. Data Prefetching Opportunities

Our results have shown that there are significant global and local address-delta patterns in most of the benchmarks. Furthermore, we observe extended address streams in some benchmarks. In this section, we discuss how to improve prefetching based on the pattern results that we presented.

Delta-Based Prefetching: We summarize global and local delta pattern potential for SPEC CPU 2006 benchmarks in Table 8. We also show the performance gains with a simple stride (one delta) prefetcher and with a perfect LLC with no misses. We can see that stride prefetching does not perform well on many of the benchmarks where there is great potential for prefetching. Our pattern analysis is able to show how much potential we can expect by exploiting local and global-delta patterns (Tables 6 and 7). It also shows that while prefetching using global-delta patterns is best for some benchmarks, for others local delta patterns are best. **Better coverage and prediction accuracy is possible by exploiting multiple delta-pattern history lengths, simultaneously, for both global and local-delta patterns.**

Finally, it is also important to note that pattern analysis can help us decide on how much hardware resource must be employed for the design. For example, by looking at the number of unique patterns and average repetition intervals, one can decide on the size of the predictor tables.

Address-Based Prefetching: We observed surprisingly long miss address patterns for a number of benchmarks,

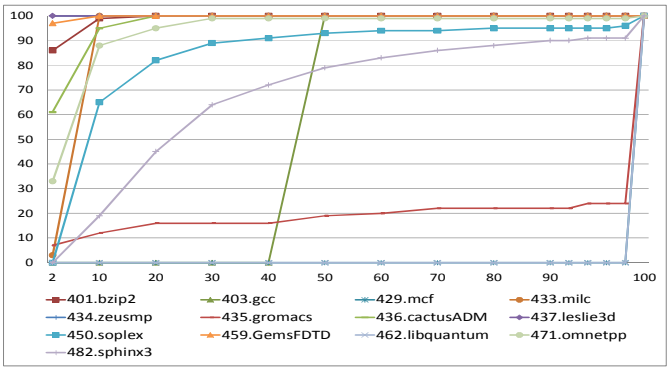

Figure 6: Cumulative distribution of hot pattern sizes (spatial regularity) for SPEC 2006 traces. x-axis: pattern length, y-axis: % number of patterns. Simulations are run for minimum pattern length of 2, maximum pattern length of 100 and 90% coverage.

Table 8. Summary of prefetching potential for SPEC CPU 2006 benchmarks derived from delta pattern analysis. Last two columns show the performance of simple stride prefetching and perfect LLC, respectively. As can be seen, stride prefetching does not perform well on many of the benchmarks where there is great potential for prefetching

Benchmark	Global delta (coverage /difficulty)	Top 10 PC Local delta (coverage/difficulty)	Stride prefet. Performance	Perfect LLC Perf.
400.perl	Low/hard	Medium/relatively easy	1.17x	1.32x
401.bzip2	Very low /relatively easy	Low/easy	1.00x	1.02x
403.gcc	High/easy	High/relatively easy	1.07x	1.76x
429.mcf	Medium/very hard	Low/medium	1.00x	6.12x
433.milc	High/easy	High/easy	1.06x	5.17x
434.zeusmp	High/relatively easy	Medium/medium	1.09x	1.44x
435.gromacs	High/relatively hard	low/easy	1.01x	1.11x
436.cactus	High/medium	Very low/easy	1.03x	1.35x
437.leslie3d	High/easy	Medium/relatively easy	1.02x	4.22x
450.soplex	None	Medium/medium	1.04x	5.29x
459.Gems	High/hard	High/relatively easy	1.01x	3.68x
462.libq	High/easy	High/easy	1.14x	5.90x
470.lbm	High/easy	High/relatively easy	1.06x	2.19x
471.omnetpp	Low/hard	Low/hard	1.01x	1.97x
482.sphinx3	High/relatively hard	High/relatively easy	1.02x	3.62x

`429.mcf` (13K), `450.soplex` (32K), `435.gromacs` (123K) and `462.libquantum` (904K), with significant coverage. Out of these benchmarks, `429.mcf` and `450.soplex` do not have significant potential exploiting delta-patterns for prefetching, therefore only by finding ways to efficiently design **memory streaming can significantly improve their performance**. However, it is not trivial to exploit streaming because for efficient implementation, where to start and stop streaming becomes critical.

Prior work on memory streaming [3, 4, 31-35] used the miss address as trigger for starting a stream. 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. However, as our results show that extended stream lengths exist, course-grain triggers are also possible, such as function call or call-chains, or outer loop iterations of nested loops that correspond to significant execution time. With course-grain triggers, a long hot stream traversal can be initiated without incurring extra index table lookup in memory. Further research is needed to study the potential of memory streaming with course-grain triggers.

6. CONCLUSION

In this paper, we presented an analysis of branch and data address patterns using our `PatternFinder` tool. We quantify spatial and temporal regularity in branch and data address patterns in terms of four associated concepts: pattern length, frequency, coverage and weighted average repetition interval. Our analysis has shown that hot patterns are useful in quantifying branch outcome, address and address-delta locality. Spatial and temporal non-overlapping branch outcome pattern locality provides useful insights into the branch streaming opportunities. Branch streaming can further improve branch prediction accuracy when long patterns cannot be captured efficiently with predictor tables. Overlapping branch outcome pattern behavior investigates branch predictability. Our analysis show that number of unique patterns and repetition interval are important metrics that must be taken into account designing branch predictors.

Our data address pattern analysis show that long hot address patterns provides insight into address streaming opportunities, while short hot address-delta patterns show potential of hardware prefetching. Our results show that successful hardware prefetcher must exploit multiple delta patterns simultaneously for greater performance. For some benchmarks that have small delta pattern opportunities, address streaming could be a solution.

Overall, our analysis has shown that the `PatternFinder` tool can efficiently be used for summarizing a benchmark's dynamic branch and data address behavior and thereby gives valuable insights into the design of future prediction mechanisms. The tool can also be used for pattern-centric classification of benchmark programs.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful suggestions. This work is partly supported by the National Science Foundation under grants CCF-1117467, CCF-1422516 and CNS-1405862.

REFERENCES

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

[2] C.G. Nevill-Manning, and I.H. Witten, "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," 32nd Ann. Int'l Symp. Computer Architecture, 2005.

[4] S. Somogyi et al., "Spatio-Temporal Memory Streaming," Proc. 36th International Symp. on Computer Architecture, 2009.

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

[6] 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.

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

[8] S. McFarling, "Combining branch predictors," Tech. Note 36, DEC WRL, 1993.

[9] Chang et al, "Alternative implementations of hybrid branch predictors,"MICRO 1995.

[10] A. Seznec, "64KB ISL-TAGE branch predictor," Championship Branch Prediction Competition, 2011.

[11] D. Jiménez, "Piecewise Linear Branch Prediction," ISCA 2005.

[12] Gabriel H. Loh, "Deconstructing the Frankenpredictor for Implementable Branch Predictors," JILP 2005.

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

[14] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty," SC1991.

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

[16] Roth et al., "Dependence Based Prefetching for Linked Data Structures," ASPLOS1998.

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

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

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

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

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

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

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

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

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

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

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

[28] www.spec.org

[29] www.jilp.org/jwac-2

[30] www.m5sim.org

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

[32] M. Ferdman et al., "Temporal Instruction Fetch Streaming," Proc. of the 41st Int'l Symp. on Microarchitecture, Dec. 2008.

[33] T. F. Wenisch, et al., "Temporal Streams in Commercial Server Applications," Proc. of the IEEE Int'l Symp. on Workload Characterization, 2008.

[34] T. F. Wenisch et al., "Mechanisms for Store-wait-free Multiprocessors," Proc. of the 34th Int'l Symp. on Computer Architecture, Jun. 2007.

[35] S. Somogyi et al., "Spatial Memory Streaming," ISCA 2006.