TCP: Tag Correlating Prefetchers

Zhigang Hu
T.J. Watson Research Center
IBM Corporation
zhigangh@us.ibm.com

Margaret Martonosi
Dept. of Electrical Eng.
Princeton University
mrm@ee.princeton.edu

Stefanos Kaxiras
Comm. Sys. and Software
Agere Systems
kaxiras@agere.com

Abstract

Although caches for decades have been the backbone of the memory system, the speed gap between CPU and main memory suggests their augmentation with prefetching mechanisms. Recently, sophisticated hardware correlating prefetching mechanisms have been proposed, in some cases coupled with some form of dead-block prediction. In many proposals, however, correlating prefetchers demand a significant investment in hardware.

In this paper we show that correlating prefetchers that work with tags instead of cache-line addresses are significantly more resource-efficient, providing equal or better performance than previous proposals. We support this claim by showing that per-set tag sequences exhibit highly repetitive patterns both within a set and across different sets. Because a single tag sequence can capture multiple address sequences spread over different cache sets, significant space savings can be achieved. We propose a tag-based prefetcher called a Tag Correlating Prefetcher (TCP). Even with very small history tables, TCP outperforms address-based correlating prefetchers many times larger. In addition, we show that such a prefetcher can yield most of its performance benefits if placed at the L2 level of an aggressive out-of-order processor. Only if one wants prefetching all the way up to L1, is dead-block prediction required. Finally, we draw parallels between the two-level structure of TCP and similar structures for branch prediction mechanisms; these parallels raise interesting opportunities for improving correlating memory prefetchers by harnessing lessons already learned for correlating branch predictors.

1 Introduction

With the widening speed gap between processor and main memory, memory performance has become a major bottleneck for current microprocessors. To address this bottleneck, computer architects have mainly relied on fast on-chip caches. However, due to latency, power, and transistor budget constraints, on-chip cache sizes are not able to keep up with the growing data requirements of typical application programs, leaving many programs suffering high cache miss rates and subsequent performance degradation.

Aside from efforts to increase cache capacity or optimize cache organization, many architects have turned to prefetching mechanisms. Prefetching works by predicting what data will be required by the processor in the future and fetching them into caches *a priori*. Prefetching is somewhat similar to branch prediction, where addresses of to-be-executed instructions are predicted and associated instructions pre-loaded into the processor core. As in branch predictors, prefetching can be initiated in either hardware [2, 5, 8, 9, 10, 12, 17, 19] or software [13, 14, 15, 16]. Compared to software prefetching, hardware prefetchers have the advantage of transparency and run-time information availability. Due to the lack of program semantic information, however, many hardware prefetchers have relied on capturing specific recurring patterns observed in memory reference streams. For example, stride prefetchers [2] target load instructions that stride through the address space. Stream buffers [10] attempt to capture reference streams formed by consecutive cache lines.

Correlation-based prefetching [5, 8, 9, 12, 19] is a more general prefetching scheme, attempting to exploit any correlation between a future memory reference and past memory behavior, including memory reference streams, load instruction addresses, and branch history. A common drawback in previous proposals on correlation-based prefetching is the relatively large size of their correlation tables, often 1-2 MB [9, 12]. These size requirements are comparable to current on-chip L2 caches and therefore bring up concerns about latency, power, and transistor budget overhead. Moreover, some prefetchers require instruction addresses, in addition to address traces. Passing information about instructions from the processor core to prefetchers complicates the processor design.

The large table sizes of these correlation prefetchers are fundamentally necessitated by the fact that programs reference many addresses, and thus many items will need to be tracked and correlated. In this paper we show that *tag-based* correlation prefetching can be done effectively and more cost-efficiently than prior address-based schemes. We follow a sequence of steps to establish this claim:

- First, we show that L1 cache tags exhibit strong regularity. This also reiterates the well-known phenomenon of locality at the tag and page level [1, 11, 18].
- Second, we show that tag *sequences*, the necessary ingredient for tag correlating prefetching, are highly repetitive and thus form a solid basis for predictions.
- Third, we show that a *single* tag sequence covers *multiple* address sequences that would necessitate distinct entries in an address correlating prefetcher.
- Finally, we show with an example design that a tag correlating prefetcher can reconstruct prefetching addresses with the same accuracy of an address-based prefetcher *many orders of magnitude larger*. Specifically, we propose a small, simple, stand alone, correlation-based prefetcher which outperforms previous proposals, requiring only few kilobytes of storage. This prefetcher, called

a Tag Correlating Prefetcher (TCP), keeps track of per-cache-set tag sequences and exploits recurring tag correlation patterns for prediction.

Another important contribution of our paper is to investigate the placement of such a prefetcher. In general, we propose the TCP to be positioned between the L1 and the L2 data caches. There it can observe miss address streams from the L1 data cache and issue prefetches to the L2 data cache. The prefetches only update the L2 data cache, and therefore do not disrupt the L1 data cache. This position is where we get the most benefit for the least disturbance to the overall design.

The TCP has a two-level structure: The first level table stores the tag history at each cache set, while the second level table tracks tag correlation patterns. This structure closely resembles the well-known two-level branch predictors [22]. This similarity is important; potentially TCP can benefit from the large body of mature research on branch predictors.

Using a cycle-accurate simulation of a wide-issue out-of-order superscalar processor and the whole SPEC2000 benchmark suite, we show that a tag correlating prefetcher with a relatively small 8KB history table, achieves a 14% performance improvement over the whole SPEC2000 benchmark suite. This outperforms a previous proposal with 2MB table, which is based on correlations of both addresses and PC traces. The basic TCP prefetches only up to the L2 level, for some benchmarks, further improvements are possible by incorporating an accurate dead block predictor and prefetching into L1.

The rest of the paper is organized as follows. Section 2 introduces the experimental setup used in the paper. Section 3 describes the recurring behavior of single cache tags and tag sequences, which motivates tag correlating prefetchers. Section 4 details the structure and operations of tag correlating prefetchers, while Section 5 gives simulation results to demonstrate the effectiveness of TCP. In Section 6, some design issues related to TCP are discussed and several directions for future work are outlined. In Section 7 we discuss the related work. Finally, Section 8 offers our conclusions.

2 Simulation Methodology

To evaluate our proposals, we use a modified version of Simplescalar 3.0 [3, 4] to simulate an aggressive 8-issue out-of-order processor. The main processor and memory hierarchy parameters are shown in Table 1. Because contention can have important influence on performance, we have incorporated a simulator modification that accurately models contention at the L1/L2 and memory buses [12].

We evaluate our results using the SPEC CPU2000 benchmark suite [21]. The benchmarks are compiled for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings. For each program, we skip the first 1 billion instructions to avoid unrepresentative behavior at the beginning of the program's execution. We then simulate 2 billion instructions using the reference input set. We include some overview statistics here for background. Figure 1 shows how much the performance (IPC) of each benchmark would improve if all accesses to the L2 data cache are cache hits. This is the target we aim for in our memory optimizations. The programs are sorted from left to right according to the amount they would speed up with an ideal L2 data cache. Starting from the next section, we will present simulation results according to this benchmark order.

Processor Core	
Clock rate	2GHZ
Instruction Window	128-RUU, 128-LSQ
Issue width	8 instructions per cycle
Functional Units	8 IntALU,3 IntMult/Div, 6 FPALU,2 FPMult/Div, 4 Load/Store Units
Memory Hierarchy	
L1 Dcache Size	32KB, 1-way, 32B blocks, 64 MSHRs
L1 Icache Size	32KB, 4-way, 32B blocks
L1/L2 bus	32-byte wide, 2GHZ
L2 I/D	each 1MB, 4-way LRU, 64B blocks,12-cycle latency
Memory Latency	70 cycles

Table 1: Configuration of Simulated Processor

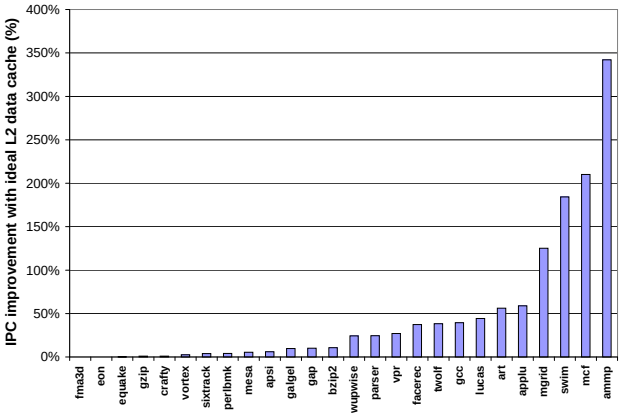

Figure 1: Potential IPC improvement with an ideal L2 data cache for SPEC2000 benchmarks.

3 The Recurring Behavior of Cache Tags and Cache Tag Sequences

In this section, we examine the behavior of single cache tags, and then expand it to tag sequences in each cache set. The patterns we observe in this section will be exploited in later sections to build effective hardware prefetchers.

3.1 The Behavior of Cache Tags

The locality of memory references is well-known: programs tend to access addresses that match or are close to previously-accessed addresses. Traditionally memory reference locality has been interpreted in terms of complete addresses. Since cache tags are formed by the high order bits of memory addresses, intuitively the rule of locality should also apply to cache tags. (Note that locality of tags are in accordance with locality found for virtual pages [1] and TLB [11, 18]). In the following paragraphs we further formalize the locality of tags, with formula "$A \rightarrow B$" representing the relationship that "if A appeared in the recent past, then B will likely appear in the near future".

First, temporal locality states that recently re-accessed addresses are likely to be accessed in the near future. When re-references to an address occur, the corresponding tag and index will also re-appear. So temporal locality indicates that cache tags tend to recur within the same cache set. This line

of thought can be represented by the following formula.

$$A \rightarrow A$$
$$\Rightarrow tag(A) \rightarrow tag(A) \ \ and \ \ index(A) \rightarrow index(A)$$

Second, spatial locality says that items whose addresses are near each other tend to be referenced close together in time. This correlation can be formalized as follows:

$$A \rightarrow A + \delta$$

Depending on the size of δ, three situations could occur:

1. $tag(A) = tag(A+\delta) \ \ and \ \ index(A) = index(A+\delta)$. This happens when δ is so small that A and $A + \delta$ remain in the same cache line. In this situation $tag(A)$ re-appears in the same set, accompanying the occurrence of $A + \delta$.

2. $tag(A) = tag(A+\delta) \ \ but \ \ index(A) \neq index(A+\delta)$. This happens if δ is big enough to change the index but not enough to affect the tag. In this situation $tag(A)$ re-appears in another cache set when $A + \delta$ is referenced.

3. $tag(A) \neq tag(A+\delta)$. This happens if δ is big enough to change the tag. In this situation $tag(A)$ will not re-appear when $A + \delta$ is referenced.

Spatial locality typically refers to two addresses that are near each other, therefore δ is usually small enough so that the third situation rarely occurs. Combining situation 1 and 2, we can interpret spatial locality as: "cache tags tend to re-appear either in the same cache set, or in other cache sets." This interpretation also applies to temporal locality, where cache tags recur only in the same cache set. Thus, for both temporal and spatial locality:

$$A \rightarrow A \ \ or \ \ A \rightarrow A + \delta$$
$$\Rightarrow tag(A) \rightarrow tag(A)$$

To confirm that cache tags do exhibit recurring behavior, we profiled the SPEC2000 benchmark suite and recorded the tag access history, both within and across cache sets. Note that we only track miss address traces from the L1 data cache: tags corresponding to cache hits are not counted in the profiling. Since cache hits are all instances of tag recurrences, the tag repetition in the miss address trace represents a lower bound on the degree of repetition that would be seen in a full reference trace. We use miss traces in our study because they are much more amenable for the hardware we propose in Section 4.

The top graph of Figure 2 shows the number (log scale) of unique cache tags in the miss streams of a 32 KB direct-mapped L1 data cache. The graph on the bottom gives the average number of times each tag recurs. Taking the art benchmark as an example, it has only 98 unique tags, but on average each tag re-appears about 3 million times in the miss stream. This means that art actually misses repeatedly on a very small set of tags, indicating a moderate history table would capture the whole set of tags. Since each 32 KB address range shares a unique tag in our experiments (see the L1 cache configuration in Table 1), the number of unique tags roughly indicates the size of the program working set: benchmarks with the largest working sets are apsi, gap, wupwise, lucas, applu, and swim. The bottom graph shows that tags are highly repetitive, often recurring thousands of times.

Figure 3 gives corresponding results for complete addresses. As expected, the number of unique addresses is much

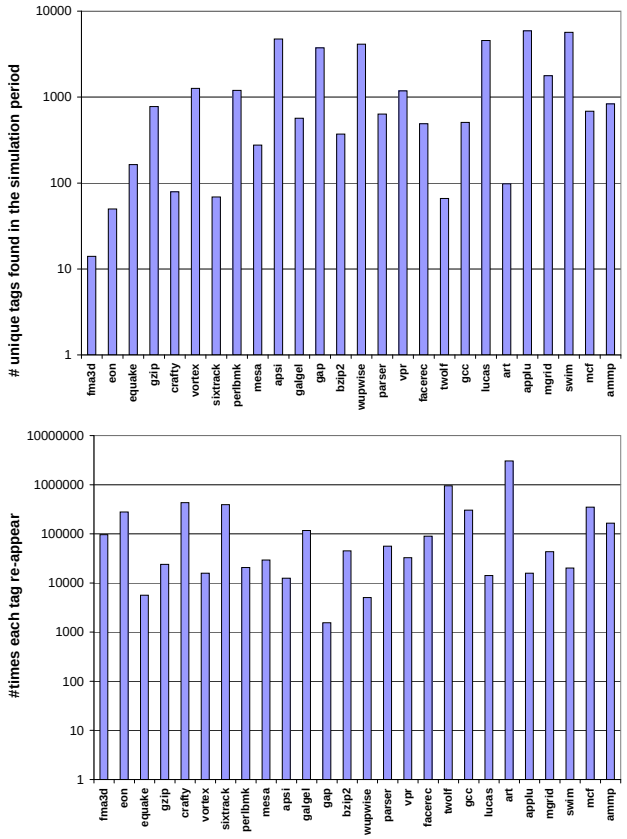

Figure 2: Number of unique tags (top) and average number of times each tag appears (bottom) in the miss streams of a 32KB direct-mapped L1 data cache for SPEC2000 benchmarks. Both graphs are in log scale. Benchmarks are ordered from left to right according to their performance potential with an ideal L2 data cache.

larger than that of unique tags: typically 2-3 orders of multitude more. Nevertheless, the number of unique tags correlates well with the number of unique addresses: programs with the most unique addresses, such as apsi, gap, wupwise, lucas, applu, and swim, also have the largest number of unique tags. These benchmarks would likely stress both tag correlating prefetchers and address correlating prefetchers. On the other hand, addresses recur much less frequently than tags. Because there are fewer unique tags than addresses, history tables for tags can be much smaller than history tables for addresses. Because tags recur more frequently than addresses, each tag history entry has more potential reuse, increasing the effectiveness of each tag history entry.

A key difference between tags and addresses is that a tag can appear in different cache sets while an address is confined to one set. Therefore, the number of recurrences in the bottom graph of Figure 2 could result from both intra-set and inter-set recurrences. If a tag re-appears 100 times, it could be that it shows up in 10 cache sets, and re-appears 10 times in each cache set it has resided in. It could also be that it appears in only 1 cache set, but re-appears 100 times in that cache set, or *vice versa*. To separate these cases, we show on the top of Figure 4 the average number of sets each cache tag touches and on the bottom the average number of times each tag appears in each set it touches. Based on the previous analysis on locality, the top graph roughly indicates degree of spatial locality

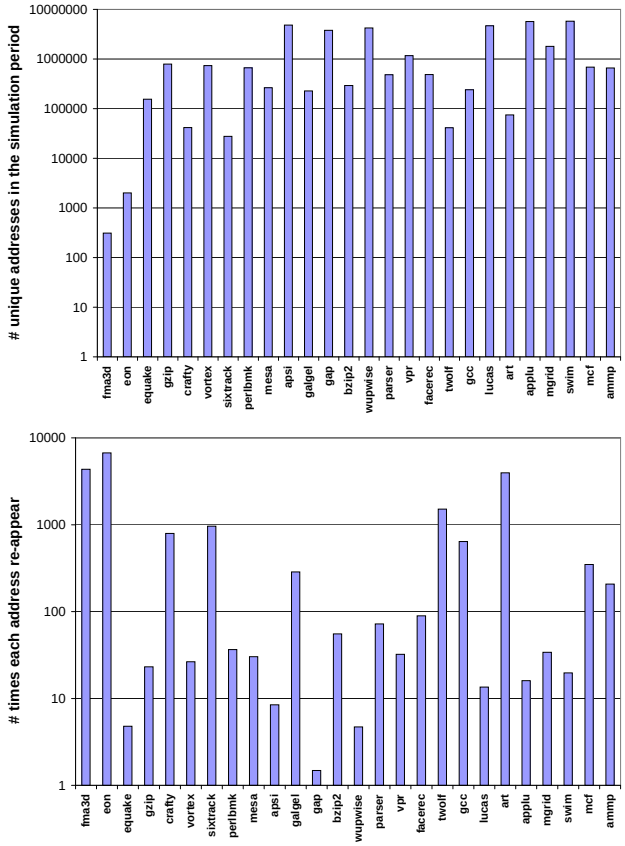

Figure 3: Number of unique addresses (top) and average number of times each address appears (bottom) for SPEC2000 benchmarks. Both graphs are in log scale. Note the difference in y-axis scales compared to Figure 2.

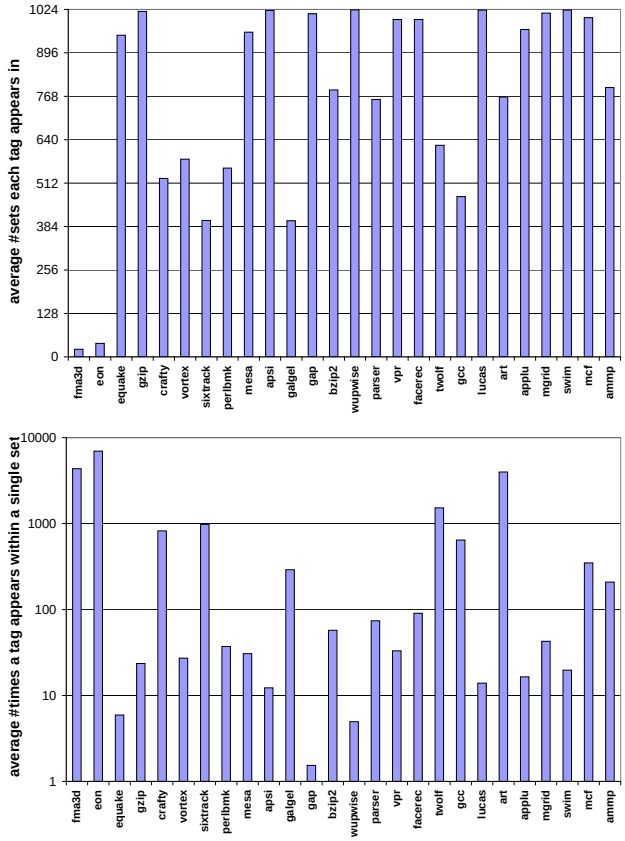

Figure 4: Average number of sets in which a cache tag appears (top) and average number of times a cache tag appears in a single set (bottom) for SPEC2000 benchmarks.

of the programs, while the bottom graph correlates to temporal locality. In the top graph, the upper limit is 1024, which is the total number of cache sets in our L1 data cache (See Table 1). Many benchmarks, such as gzip, apsi, wupwise, lucas and swim, are near this upper limit. In these benchmarks each tag can be found in almost every set of the L1 data cache, indicating a high degree of spatial locality. On the other hand, tags in these benchmarks repeat rather infrequently within each sets they touched: only a few 10s of times, indicating a low degree of temporal locality. In benchmarks such as fma3d and eon, each tag touches only a small number of cache sets, but it repeats thousands of times in each set, indicating that these benchmarks have good temporal locality but relatively poor spatial locality.

Overall, we find that for SPEC2000 benchmarks, on average (geometric mean) each benchmark has 576 unique tags: each tag spreads into 609 cache sets, and recurs 94 times within each cache set it touches. These numbers confirm our earlier interpretation of the rule of locality: that cache tags tend to re-appear either in the same set, or across different sets. This behavior can be summarized as the following formula:

$$tag(A) \rightarrow tag(A)$$

3.2 The Behavior of Cache Tag Sequences

In the previous section, we examined the recurrence behavior of single cache tags. Tags can be viewed as per-cache-set tag sequences, where the sequence length is 1. In this section we investigate the behavior of longer tag sequences. We focus on sequence lengths of 3 in our experiments. Targeting tag sequences at each cache set has many potential benefits for a prefetcher. First, when predicting the next tag based on previous tags in the same cache set, the index is implicitly designated so requires no prediction or tracking. Second, the time interval between consecutive misses in a cache set is typically larger than a memory access latency, providing sufficient time for a timely prefetching. Finally, the characteristics of per-cache-set sequences, as will be explored in this section, enable effective and hardware-efficient predictions of future tags.

As in the previous section, we start by measuring the number of unique sequences. If tag sequences were totally random, then the expected number (the upper limit) of three-tag sequences would be roughly the number of unique tags cubed. If the correlation between tags is strong, however, each tag would tend to appear together with some other specific tags. With correlation, the number of unique three-tag sequences will be much less than the upper limit. Figure 5 shows the number of unique three-tag sequences observed in our simulation, as a percentage of the upper limit. In most benchmarks, the number of unique sequences is much smaller than the up-

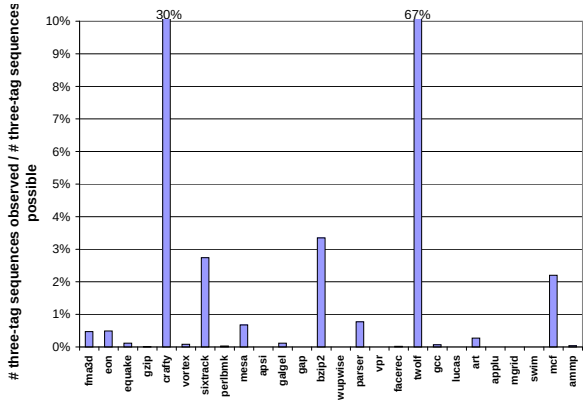

Figure 5: Number of three-tag sequences actually observed as a percentage of total number of possible three-tag sequences for SPEC2000 benchmarks.

per limit, typically less than 5%. This indicates strong tag correlations in most benchmarks. In crafty and twolf, the tag sequences behave quite randomly, so the number of unique sequences is large. We further note that these results do not particularly coincide with the application's memory footprint. In fact, crafty and twolf access fewer unique tags than most of the SPEC2000 benchmarks.

The top graph of Figure 6 shows the absolute number of unique three-tag sequences that appeared in the miss address streams of a 32 KB direct-mapped L1 data cache. The fma3d benchmark has the fewest unique three-tag sequences, while mcf has the most, with more than 7 million unique sequences. The bottom graph of Figure 6 gives the average number of times each three-tag sequence recurs. There is a wide variety among different benchmarks: from just above 10 for bzip2 to over 200,000 for art. In many benchmarks, each three-tag sequence appears thousands of times, indicating a very repetitive behavior that can be exploited by a history-based predictor.

Figure 7 further splits these recurrences into two categories: intra-set and across-set. The top graph gives the average number of sets in which each three-tag sequence appears. For example, in the swim benchmark, on average a tag sequence appears in 264 sets, about a quarter of total cache sets. These results have great impact on the space requirement of the history table. To illustrate this issue, let us consider two extremes. At one extreme, consider first if each three-tag sequence appears in every cache set, in other words, if each cache set has the same tag sequence history. Here, we can use a single table for all cache sets: no per-cache-set table is required. Sharing one history table among different cache sets greatly reduces space requirements. At the other extreme, consider if every three-tag sequence appears in only one set, so that each set has its own specific sequences. Here, history from different cache sets cannot be shared: they will simply contend for space thus leading to a larger capacity requirement.

The bottom graph in Figure 7 gives the average number of times a tag sequence appears within each set. The more frequently each sequence appears, the more effective it will be to prefetch based on them. For example, each sequence in fma3d is re-referenced about 75,000 times. This indicates that if we store a tag sequence the first time we see them, and

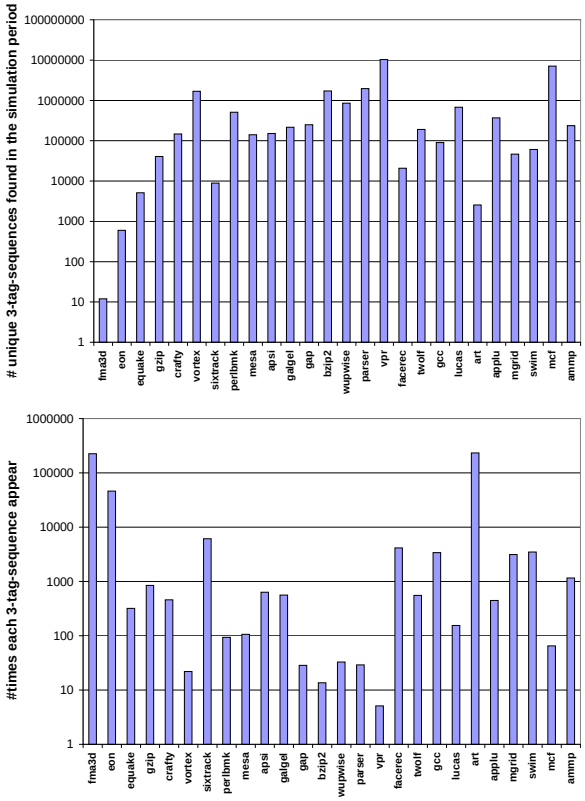

Figure 6: Number of unique three-tag sequences (top) and average number of times each sequence appears (bottom) for SPEC2000 benchmarks.

always keep it in the history, then potentially it can be reused thousands of times.

Figure 7 indicates a key difference between tag sequences and address sequences: a tag sequence can appear in different sets while an address sequence can not. In other words, a tag sequence that appears in different sets implies *multiple different address sequences*.

Overall in this section we demonstrated that per-cache-set tag sequences exhibit recurring behavior: if a tag sequence occurred in the past, it tends to re-appear in the future, either in the same cache set, or in other sets. This behavior can be formulated as follows:

$$tag(A1), tag(A2), ..., tag(Ak)$$
$$\rightarrow tag(A1), tag(A2), ..., tag(Ak)$$

4 TCP: Design Overview

In the previous section we described the recurring behavior of cache tag sequences. There is a strong correlation between a tag and its preceding tags, and this correlation is highly repetitive. In this section we describe a predictor that exploits this repetitive correlation to predict the next tag, according to previous tags just seen in the miss trace at a cache set. We name this prefetcher the "tag correlating prefetcher". We start by describing the structure of the prefetcher, and then present simulation results in the next section to show its effectiveness.

Figure 8 depicts the structure of a tag correlating address predictor. It is organized as a two-level structure, similar to conventional two-level branch predictors [22]. The first level table, the Tag History Table (THT), tracks the previous k tags

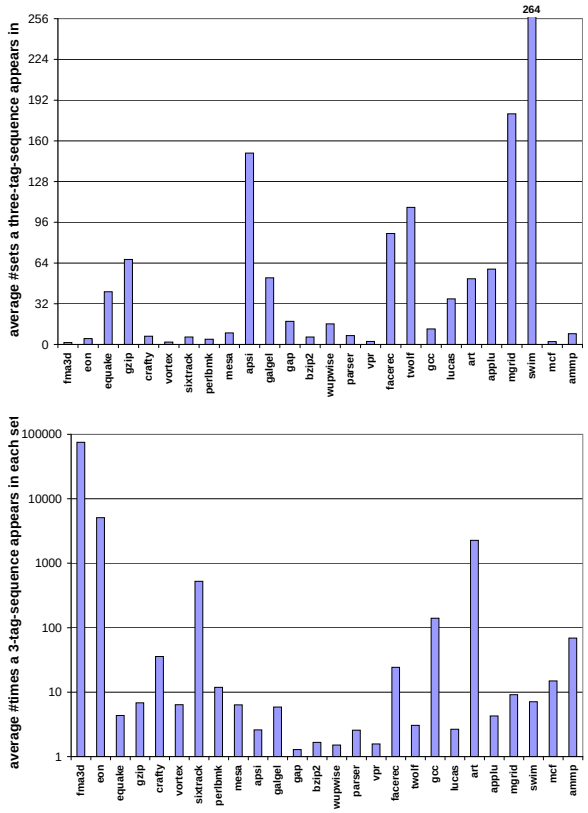

Figure 7: Average number of sets a sequence appears in (top) and average number of times a sequence appears in each set it touches (bottom) for SPEC2000 benchmarks. The upper limit in the top graph is 1024, which is the total number of sets in L1 data cache.

that appeared in the miss trace at each cache set. The second level table, the Pattern History Table (PHT), stores tag correlation patterns observed in the past.

The THT is indexed by the miss index (the index portion of current miss address). Each row in the table corresponds to a set in the L1 data cache. Thus, THT lookup can occur in parallel with an L1 cache lookup. There are k entries per row (set); each entry stores a previous tag in order of time. That is, tag1 is the oldest and tagk is the most recent. THT size, therefore, is:

$$(number\ of\ sets\ in\ d1) * k * sizeof(tag).$$

The index of the PHT is formed by the tag sequence obtained from the first level table, together with the current miss tag, and optionally the miss index. Figure 9 shows the indexing scheme that we use for the remainder of this paper. The higher m bits are taken from (the lower m bits of) a truncated addition (as in [12]) of all tags in the tag sequence, while

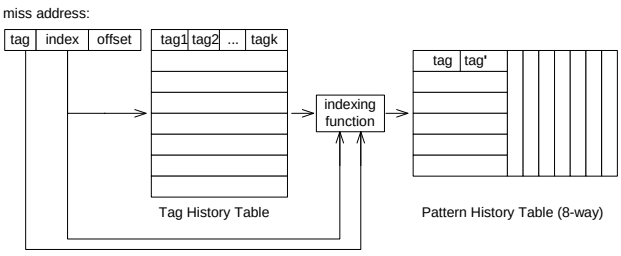

Figure 8: Structure of a two-level tag correlating address predictor.

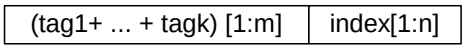

Figure 9: The indexing scheme of the pattern history table. "index[1:n]" stands for the lowest n bits from index.

the lower n bits are taken from the miss index. Choosing n between 0 and 10 achieves a trade off between sharing and separating history from different cache sets. In the case when n is 0, all cache sets share the history entries. On the other hand, when n is 10, which (for our L1 cache) means to use the full miss index, each cache set has its own private space for correlation history. Each entry in the PHT has two fields: tag and tag'. Tag' is the predicted successor to tag. The size of the PHT can be calculated using the following formula: $(number\ of\ PHT\ sets) * (ways\ per\ PHT\ set) * 2 * sizeof(tag)$.

Given such a structure, the operation of the TCP prefetcher consists of two basic functions: update and lookup. We describe these two operations in more details below, assuming a cache miss to L1 data cache is just observed. The miss index (the index part of the miss address) and the miss tag (the tag part of the miss address) are denoted as $missindex$ and $misstag$ respectively.

- *Update*: Update is the operation to refresh the THT and PHT when new misses occur so that the history information stored in these tables is always up-to-date.

  1. First, $missindex$ is used to access the THT, and a tag sequence $(tag1, tag2, ..., tagk)$ is located. This sequence is updated to $(tag2, ..., tagk, misstag)$, establishing $(tag2, ..., tagk, misstag)$ as the most recent tag history in this cach set.
  2. Second, tag sequence $(tag1, tag2, ..., tagk)$, combined with $missindex$ (see Figure 9), is used to index into the PHT and a set (row) in the PHT is located.
  3. Third, among all the entries within the PHT set, the one tagged with $tagk$ is located.
  4. Finally, the tag' (next tag) field of the entry is updated to $misstag$. This establishes $misstag$ as the most up-to-date next tag following the sequence of $(tag1, tag2, ..., tagk)$.

- *Lookup*: Lookup is the operation to decide a prefetch address based upon the knowledge that the immediate past tag sequence is $(tag2, ..., tagk, misstag)$ at this cache set.

  1. First, the sequence $(tag2, ..., tagk, misstag)$, combined with the $missindex$, is used as the PHT index to locate a PHT set.
  2. Second, from the PHT set, the entry tagged with $misstag$ is selected and its tag' field is predicted as the next tag that follows the tag sequence $(tag2, ..., tagk, misstag)$.
  3. Finally, the predicted next tag tag', combined with the current miss index $missindex$, forms a complete cache line address and subsequently a prefetch to this address is issued to L2.

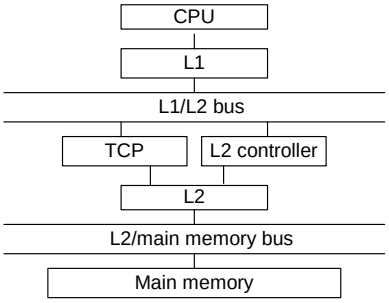

Figure 10: Overall system structure with a tag correlating prefetcher.

Since the address predictor requires the miss address streams, it must be placed after the L1 data cache, as shown in Figure 10. The predictor can be integrated with the L2 data cache controller. It observes the miss traces from the L1 data cache and issues prefetch requests to the L2 data cache. The L2 first checks whether the target data is already in itself. If found, the prefetch is completed and no further operation is required. Otherwise, a request is sent to the main memory to load the data into L2, but L1 is not updated.

Prefetching all the way up to L1 is more complex because not only do the benefits increase but also the risks from wrong prefetches. We discuss this case in a subsequent section.

Having studied the structure and operations of tag correlating prefetchers, in the next section we present some simulation results to show the effectiveness of TCP prefetchers.

5 Simulation Results

In this section we present simulation results for two configurations of TCP. In both these cases, the tag history table (THT) is organized as a 1024-set, direct-mapped structure, with each set storing 2 previous tags (k = 2 in Figure 8). The two cases differ by their pattern history tables. One (marked as TCP-8K) has a 8 KB PHT with 256-set, 8-way set associative and using no bits from the miss index. The other (marked as TCP-8M) has a 8 MB PHT with 262144-set, 8-way set associative and using the full miss index. Note that in both TCPs, each cache set can utilize a maximum of 8KB for storing history. The difference is that in TCP-8K, this 8KB storage is shared by all cache sets, while in TCP-8M, it is private to each cache set. Because of its size, we do not consider TCP-8M to be a realistic design point; rather we include it as an "idealized" view of how no-sequence-sharing affects each benchmark.

5.1 Basic Results

Figure 11 gives the performance results of TCP-8K and TCP-8M, compared to a DBCP with a 2 MB correlation history table. Dead-block correlating prefetcher (DBCP) [12], is a correlation based prefetcher that correlates the liveness of a cache line and the next tag with PCs of memory instructions, in addition to addresses. Note that in [12], a critical miss predictor [20, 6] is proposed to filter the correlation entries. In our experiment, this filter is not incorporated in either DBCP or TCP. In general, the two TCP prefetchers both out-perform DBCP. On average, DBCP achieves about 7% performance improvement, while TCP-8K and TCP-8M can achieve about

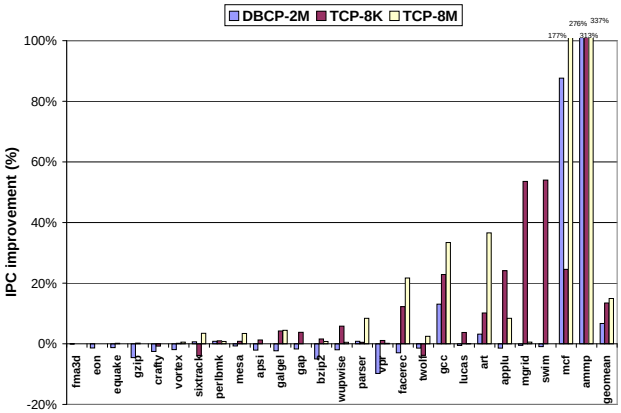

Figure 11: IPC for TCP with 8KB PHT and 8MB PHT vs. DBCP with 2MB correlation table.

14% and 15% respectively. Comparing TCP-8K and TCP-8M, we find that sharing history entries across cache sets leads to lower performance for some benchmarks, such as facerec, gcc, art, mcf, and ammp. On the other hand, it performs better for benchmarks like applu, mgrid, and swim. This difference can be explained by investigating the effect of sharing history entries between cache sets. For example, in the swim benchmark, each tag sequence appears in an average of 264 cache sets (see Figure 7). On the other hand, each sequence recurs rather infrequently within each set: an average of 7 times. In TCP-8K, where all entries are shared across all cache sets, each entry can be reused about $(264 * 7 - 1)$ times. However, in TCP-8M, since each set has its own storage, each entry can be reused only $(7 - 1)$ times. Since in swim sharing entries among cache sets is beneficial rather than detrimental, TCP-8K performs better than TCP-8M. Conversely, for the 5 benchmarks in which TCP-8M is better, each tag sequence is shared by a much less number of cache sets: in this case the benefit of sharing history entries is outweighed by the adverse effect of contention and aliasing between history entries from different cache sets.

In conventional caches without prefetchers, every L2 access is originated from a L1 cache miss, which means an associated load/store instruction is already stalled. If the L2 access hits, the overall latency is $\sim$10 cycles, which can usually be tolerated by an aggressive superscalar out-of-order core. However, if the L2 access misses, the long latency to the main memory, which could be hundreds of cycles, will fill the instruction window up with dependent instructions and thus stall the whole processor. With a tag correlating prefetcher, some of the original L2 accesses will be pre-issued by the prefetcher and thus will hit when accessed. The rest of the original L2 accesses are not captured by the prefetcher and are still initiated by L1 cache misses. We name these two categories of L2 accesses as "prefetched original" and "non-prefetched original" L2 accesses respectively. In addition, prefetches could lead to extra L2 accesses, for example, when the predicted addresses are never used later. Figure 12 gives the amount of "prefetched original", "non-prefetched original", and "prefetched extra" L2 accesses for SPEC2000 benchmarks with TCP-8K (top) and TCP-8M (bottom), all normalized to the number of original L2 cache accesses. An ideal prefetcher would have 100% "prefetched original", 0% "non-prefetched original", and no "prefetched extra" L2 ac-

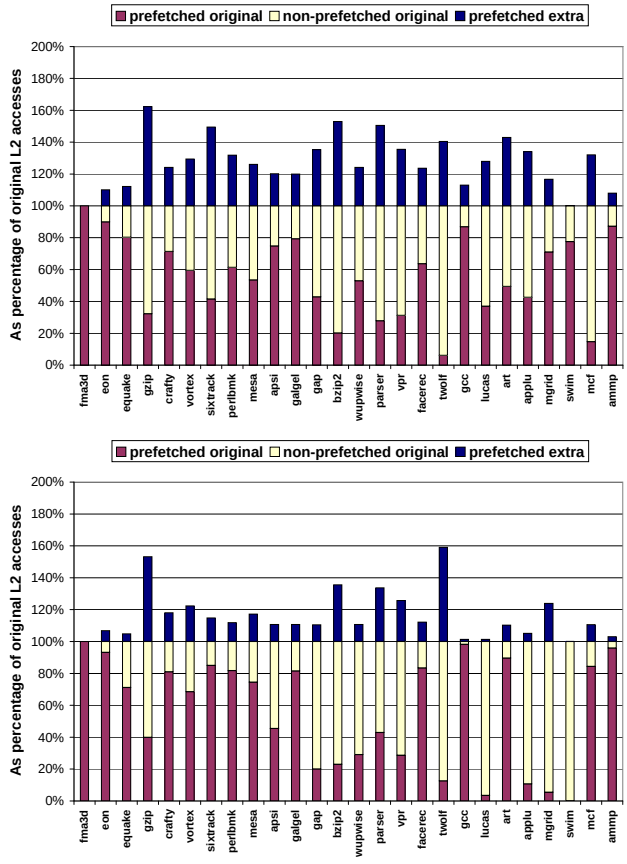

Figure 12: The amount of the three categories of L2 accesses for TCP-8K (top) and TCP-8M (bottom). Data is normalized to the number of original L2 accesses (when no prefetchers are used).

cesses. The prefetcher in the fma3d benchmark satisfies these requirements, even though the performance potential is small (recall that benchmarks are ordered from left to right by their performance potentials with ideal L2 caches). For TCP-8K, benchmarks like ammp, swim, mgrid, gcc benefit most from the prefetcher: the "prefetched original" percentages are high while the extra traffic is relatively light. For TCP-8M, the best performing benchmarks are ammp, mcf, art, gcc, and facerec.

5.2 Design Variations

The basic results in the previous section focused on two particular configurations, yet there is still a large design space left to be explored. This section covers some interesting design variations.

5.2.1 Varying PHT Configurations

The size of the pattern history table decides how much tag correlation history can be stored. Enlarging the PHT can reduce the aliasing between different tag patterns and thus improve the prefetcher effectiveness. The top graph in Figure 13 gives the performance of SPEC2000 benchmarks using TCPs with varying PHT sizes. As shown in the graph, when the PHT is not indexed by the miss index, quadrupling the PHT size from 2KB to 8KB leads to a 6% performance improvement. Further increasing the PHT size beyond 8KB has a diminishing effect. For PHTs indexed with full miss index, the saturating

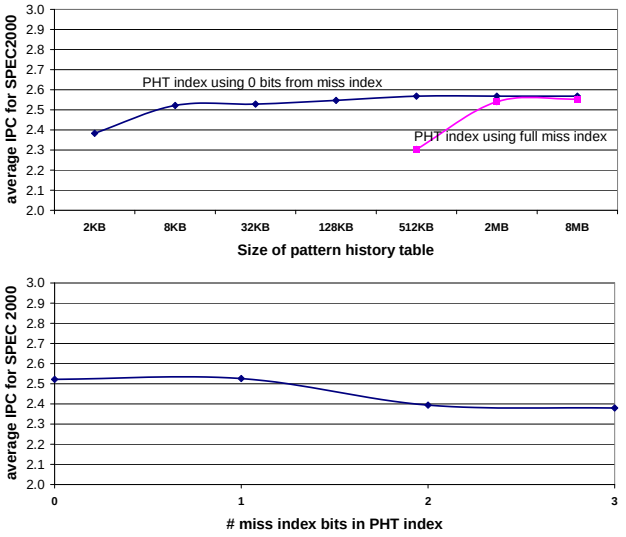

Figure 13: Average performance of SPEC2000 with different sizes of PHT (top) and different indexing schemes of PHT (bottom).

point is near 2 MB. The bottom graph in Figure 13 evaluates how the number of miss index bits used affects the performance of a 8 KB PHT. Using 0 or 1 bit from the miss index have similar performance, but using more bits will degrade the performance. As illustrated in Figure 9, using bits from the miss index essentially divides the PHT into separate sub-tables, each storing history from a group of cache sets. If the PHT is already small and the number of index bits is large, the sub-tables will eventually become too small to track tag history effectively, leading the performance to degrade.

5.2.2 Prefetching into L1

Basic tag correlating prefetchers only prefetch data up to the L2 data cache. For the highest performance possible we need to bring the data as close to the processor as possible, i.e., L1, *in a timely manner*. However, this is not a simple proposition because of the restrictions we face going up the memory hierarchy. Specifically, capacity, bandwidth, and scheduling considerations are considerable at that level and if not properly taken into account can invalidate many of the benefits of the prefetches, or even worse degrade performance by interfering with other critical data.

Because the L1 is much smaller than the L2, prefetching the wrong data into it can create significant disruption. In addition, prefetching the correct data at the wrong time is equally disruptive. Both these mishaps are not as pronounced in a large set-associative L2. Furthermore, because for the SPEC2000 the occupancy of L1-L2 bus is higher than the occupancy of the L2-memory bus, prefetches into L1 are competing with other accesses. If prefetches are given low priority on the L1-L2 bus, many of them can be delayed, canceled, superseded by accesses, overflow the outgoing prefetch buffer, etc. In all these situations the benefit of prefetching is lost.

To address the problem of timely prefetching, we incorporated a timekeeping dead block predictor from [8]. To address the second problem, we added an extra L1/L2 bus solely for prefetching. The result of these enhancements is a hybrid predictor that works in the following way: after a prediction is made, the predicted data is prefetched into L2 immediately,

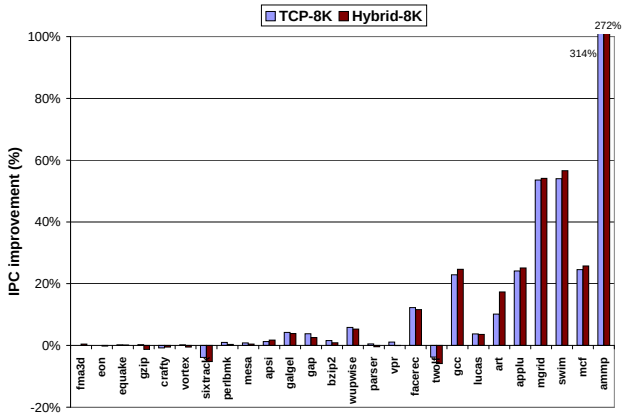

Figure 14: IPC for prefetching into L2 (TCP-8K) vs. prefetching into L1 (Hybrid-8K).

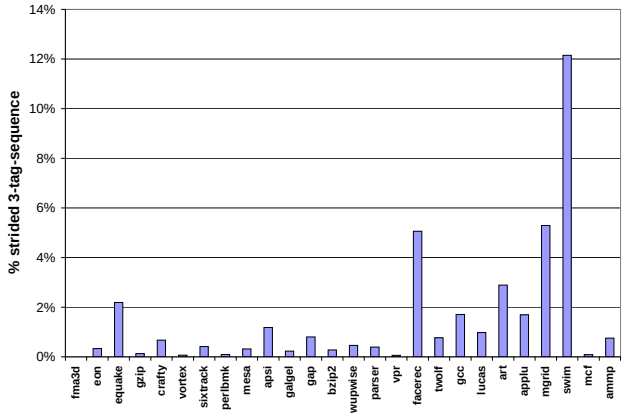

Figure 15: Percentage of strided three-tag sequences for SPEC2000 benchmarks.

but will update L1 only after the corresponding cache line is predicted dead. Figure 14 compares the performance of such a hybrid prefetcher to the base TCP prefetcher. The hybrid prefetcher further improves performance for gcc, art, applu, mgrid, swim, and mcf. From [8], we find that the timekeeping dead block predictor also works best for these benchmarks. Overall, Figure 14 tells us that, with an aggressive out-of-order superscalar processor, since the main memory latency is the major performance bottleneck while the L2 latency is tolerable, prefetching into L2 would gain the most benefit of prefetching. Prefetching further into L1 cache is beneficial, but only when an accurate dead block predictor and sufficient L1/L2 bandwidth is available.

6 Future Work

Tag correlating prefetchers capture recurring patterns of multiple-tag sequences in the miss address streams. There is one particular pattern worth special attention: (per-cache-set) strided tag sequences. As the name implies, strided tag sequences refer to the sequences in which the tags exhibit a constant stride. If strided sequences are common, more space-efficient designs could be devised for them. Figure 15 gives the percentage of strided three-tag sequences in SPEC2000 benchmarks. Swim sees the most strided sequences: over 12% of all three-tag sequences are strided. In other benchmarks, strided sequences are much less frequent, typically less than 2% of total sequences. Overall, even though per-cache-set strided sequences are typically infrequent, they do happen in most benchmarks and even frequent in some benchmarks. One possible future work is to further investigate strided and other special sequences and exploit them to improve the performance or hardware-efficiency of tag correlating prefetchers. Also note that only intra-set strided tag sequences are considered here, those strided sequences across cache set boundary are orthogonal to what we have discussed here and are an interesting phenomenon that we plan to explore in future work.

A second avenue for future work concerns the number of prefetch targets. In [9], Joseph and Grunwald proposed a Markov prefether that stores multiple targets for each prediction. Such a prefetcher can improve prediction accuracy but increase the memory traffic since multiple prefetches will be issued for each prediction. In the design of tag correlating prefetchers, there is a similar trade-off for storing multiple targets, which we hope to investigate.

Since not all cache misses affect performance equally, a critical miss filter may also be useful in future investigations. Many researchers have tried to identify those misses that are critical for program performance and devise optimizations targeting them [20, 6]. Prefetchers can benefit from such a critical miss predictor in many ways. For example, only prefetches for critical misses will be issued, so that the prefetch-induced extra traffic can be reduced. In [12], only the correlation patterns of critical misses are stored, so that the space-efficiency can be improved. We expect TCPs also benefit from a critical miss predictor, something we plan to explore in future work.

A final direction for future work is to utilize knowledge from the large body of work on two-level correlation-based branch predictors. Because of the similarity between the structure of TCPs and branch predictors, such knowledge would help to improve the indexing and correlating effectiveness of TCP.

7 Related Work

Software prefetching, and more generally, compile-time analysis of memory access behavior, has been studied by many researchers [7, 13, 14, 15, 16]. Mowry *et al.* successfully predict what data references will likely miss in scientific codes that mainly employ matrices [15]. Ghosh *et al.* describe methods for generating and solving equations that give a detailed representation of cache misses in loop-oriented scientific code. Such a framework can be utilized to decide what addresses should be prefetched and when to start the prefetches. Other work, [13, 14, 16], target pointer-intensive applications and applications with recursive data structure and propose to insert compile-time prefetch instructions.

Compared to software prefetching, hardware prefetching [2, 5, 8, 9, 10, 12, 17, 19] usually requires extra hardware to track correlations between memory references with previous memory references and other information, such as memory instruction addresses and branch history. Baer and Chen proposed a early notion of correlation-based hardware prefetching for paged virtual-memory systems [1]. They also investigated a prefetching mechanism that captures load instructions that have constant strides [2]. Jouppi proposed a stream buffer

that can be effective when there is a large amount of sequentiality in the reference stream [10]. Charney and Reeves are the first to propose a generalized correlation-based hardware prefetching for caches [5]. In their scheme, the prefetcher is positioned between L1 and L2, and prefetches to L2 only. Joseph and Grunwald proposed a Markov model for prefetching and proposed to store multiple targets with each prediction [9]. Lai *et al.* were the first to propose a hardware predictor for dead blocks based on both PC traces and previous memory addresses [12]. They were also the first to propose prefetching according to per-cache-set memory reference behavior. Solihin *et al.* proposed to use a user level thread for prefetching and store the correlation history in memory, instead of specific hardware tables [19]. Hu *et al.* proposed a general methodology, exploiting time information to analyze, predict, and optimize memory behavior and applied it to build a hardware-efficient dead block predictor, which can be used in a hardware prefetcher [8].

8 Conclusions

This paper began by characterizing the locality and predictability of tag sequences appearing in the L1 cache miss address traces of an aggressive superscalar processor. We have shown that tags appearing in L1 exhibit strong regularity. This is a reiteration of the well-known phenomenon of locality, but one that is particular useful because of the new hardware structures it suggests. Tag *sequences*, the necessary ingredient for tag correlating prefetching, are also highly repetitive and thus a solid basis for predictions. In particular, we observe and quantify how tag sequences kept on a per-cache-set basis can show repetitive patterns that recur not only in that cache set, but typically in many other sets as well.

These observations suggest first that tag correlations can be useful for memory prefetch. Second, they also suggest a two-level approach: tag sequences are tracked from the miss traces of individual cache sets in a tag history table (THT), but are made available to all cache sets through a second hardware structure called a pattern history table (PHT). Together, this two-level approach comprises our proposed scheme for Tag Correlating Prefetch (TCP).

By sharing patterns through the PHT, TCP can be very effective at improving performance even with very small table sizes. An 8KB TCP offers a 14% performance improvement for the SPEC2000 benchmark suite, and outperforms previous proposals that are megabytes in size.

The two-level THT-PHT structure of a tag correlating prefetcher is quite similar to the correlating tables used in hardware branch prediction. This paper has explored some of the indexing and PHT design questions first answered for branch prediction. We find that most benchmarks benefit from PHT sharing, although there are some that are better when each cache set indexes its own pattern histories. Overall, we feel that the parallels between two-level branch predictors and TCP are interesting, and also hint that even more performance improvements may be possible by further exploiting the parallels between them.

9 Acknowledgments

We would like to thank An-Chow Lai for insightful discussions and for providing us with detailed bus models for SimpleScalar. Martonosi's research is supported in part by NSF ITR Grant CCR-0086031, by research support from Intel Corp., and by an IBM University Partnership Award. We also wish to acknowledge the Hellenic Air Force for facilitating one of the authors to work during the time he served there. Our thanks to the anonymous referees for providing helpful comments as well.

References

[1] J.-L. Baer and T.-F. Chen. Dynamic Improvements of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, 1976.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. Supercomputing '91*, pages 176–186, Nov. 1991.

[3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Tech. Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., July 1996.

[5] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, 1995.

[6] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, July 2001.

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *International Conference on Supercomputing*, pages 317–324, 1997.

[8] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proc. 29th Annual Intl. Symp. on Computer Architecture*, May 2002.

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

[10] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. ISCA-17*, May 1990.

[11] G. B. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proc. 29th Annual Intl. Symp. on Computer Architecture*, May 2002.

[12] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, July 2001.

[13] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: Software prefetching in pointer and call-intensive environments. In *Proc. Micro-28*, pages 231–236, 1995.

[14] C.-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. In *Proceedings of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 222–233, Oct. 1996.

[15] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. ASPLOS-V*, pages 62–73, Oct. 1992.

[16] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *Proc. Micro-28*, Dec. 1995.

[17] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. ASPLOS-VIII*, Oct. 1998.

[18] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *Proc. ISCA-27*, June 2000.

[19] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proc. 29th Annual Intl. Symp. on Computer Architecture*, May 2002.

[20] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson. Locality vs. Criticality. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, July 2001.

[21] The Standard Performance Evaluation Corporation. WWW Site. http://www.spec.org, Dec. 2000.

[22] T. N. Yeh and Y. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *Proc. ISCA-20*, May 1993.