Not Quite My Tempo: Matching Prefetches to Memory Access Times

Mark Sutherland Ajaykumar Kannan Natalie Enright Jerger
Dept. of Electrical and Computer Engineering, University of Toronto
{suther68,kannanaj,enright}@ece.utoronto.ca

Abstract

Modern prefetchers can generally be divided into two categories, spatial and temporal, based on the type of correlations they attempt to exploit. Although these two types have different advantages, and perform well on different application sets, a design that utilizes both types of information will be able to achieve greater prefetch accuracy. We address the lack of temporal information in the state-of-the-art Spatial Memory Streaming (SMS) prefetcher by proposing Tempo*, a novel banked implementation of SMS that further classifies cache accesses within the same physical page based on the repetitive miss latency, or tempo, present in the local access stream. Evaluated on the SPEC CPU2006 benchmark suite,* Tempo *reduces useless prefetches by 17.6%, and achieves 1.45% and 2.57% increase in IPC on high and low bandwidth memory configurations over a purely spatial SMS design.*

1. Introduction

In a modern, energy-constrained design space, a good prefetcher should prioritize improving IPC through *accurate* and *timely* prefetches, rather than simply opting for a very aggressive configuration that may pollute the on-chip caches and/or waste energy due to wrongly prefetching useless blocks. For example, the seminal Stride prefetcher [1]captures a very common data access pattern that often arises from tight loops in the instruction stream. Due to its simplicity, it has been included in many commercial microarchitectures such as IBM's POWER4 [12] and POWER5 [3]. However, Srinath et al. demonstrate the over-prediction inherent in these simple prefetchers, proposing Feedback Directed Prefetching (FDP) which increased performance by 6.5% and reduced memory bandwidth by 19% by dynamically monitoring cache pollution and adjusting prefetch aggressiveness [11]. Sandbox prefetching is another method that relies on dynamically tracking accuracy, with the important addition of a Bloom filter that tracks prefetches previously confirmed as "useful" from 16 candidate prefetchers [7].

The Spatial Memory Streaming (SMS) prefetcher [9] partially addresses the problem of over-prediction. Proposed to capture the more complex memory behaviour present in commercial workloads, SMS exploits the spatial correlation oft present in the access pattern of the same code fragment [4], and tracks these correlations over large regions of memory (i.e., an operating system page). However, the design of SMS lacks information about the *temporal* characteristics of an application's memory access stream. This is a significant impediment to the performance of SMS on scientific and engineering workloads, as their memory access patterns can be more accurately predicted by temporally-ordered prefetchers based on a Global History Buffer (GHB) [2, 6]. Later work augments SMS with temporal information also in the form of a GHB; this STeMS prefetcher achieves approximately 3% speedup over the original SMS design [10] and requires megabytes of off-chip storage for its temporal metadata.

This paper presents *Tempo*, an improvement on the SMS prefetcher, which is based on the observation that many applications have memory access patterns that follow a repetitive tempo, or time between cache accesses. Time-Aware Stride (TAS) prefetching embeds counters within a Stride prefetcher to distinguish between different access times [13]. With *Tempo*, we learn from their insights and apply similar principles to selectively issue timely prefetches, increasing performance while judiciously conserving energy expended on off-chip accesses.

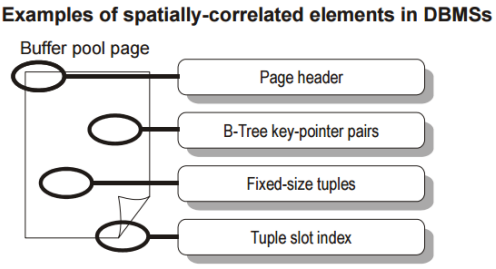

Figure 1: Example of spatial correlation in a database page [9].

2. Spatial Memory Streaming Background

The SMS prefetcher tracks repeated access patterns within a variable size region of memory, generally chosen to be one physical page [9]. This can capture complex access patterns exhibited in commercial and server workloads. Figure 1 shows an example of these patterns occurring in a database's buffer pool, where each page will have specific elements that are *always* read before accessing and modifying the data. SMS captures these page-level access patterns by storing bit vectors (referred to as spatial patterns) where each set bit represents a block that was accessed and can apply to any page at prefetch time. At prefetch time, the first access in any spatial region (regardless of its location inside the page), is referred to as the trigger access, and will generate prefetches for each block whose bit is set in the pattern vector.

To learn the access patterns inside any spatial region, SMS inspects the cache access stream and records each block that is accessed in the current region inside the Active Generation Table (AGT), which tags each entry with the upper page bits. Upon evicting of any block in the current region being tracked, SMS transfers the pattern history vector and its tag to the Pattern History Table (PHT). This eviction serves as a delimiter for the current spatial pattern being generated, due to the desire to have all of the blocks accessed by one pattern simultaneously in the cache. Upon seeing a new trigger access, SMS searches through the PHT for a tagged entry matching the current PC. If a match is found, the blocks in the spatial pattern are prefetched into the local cache.

2.1 Limitations of SMS

A disadvantage of using SMS is that it lacks temporal information about the *order* of the blocks that are part of a spatial region generation. This problem is exacerbated when we increase the spatial region size, since a large number of bits may be set inside this

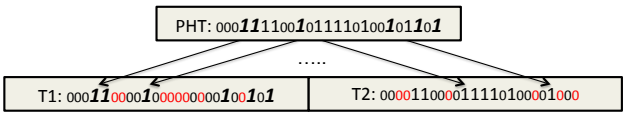

Figure 2: Decomposition of a PHT entry into multiple tempo-based patterns.

pattern vector. Additionally, we observe that many applications exhibit traversals of the same set of cache lines, but in a different order than during the AGT training process. Note that during the later traversals, although the same PC does access all the predicted cache blocks recorded by SMS, the prefetcher does not have the capability to predict the correct *order* to prefetch these blocks in. Given that we do not want to saturate the memory system with prefetch requests, significant performance gains could be attained from a system that would enable SMS to only issue prefetch requests for a *timely subset* of cache blocks that will be referenced in the near future.

3. Tempo-Based SMS Prefetcher

The *Tempo* prefetcher further classifies the cache accesses within a spatial region into multiple sets, based on a new *timeliness* component. Adding this information during training gives the prefetcher the capability to request the most timely or useful blocks at prefetch time. Our notion of timeliness is similar to the TAS prefetcher, in the form of a global miss counter that ticks whenever an L2 miss occurs [13]. We define the *cache time delta* as the difference between the global miss counter between two consecutive cache accesses. By gathering memory traces and statistics from the applications comprising SPEC CPU2006, we observe the same PC generating misses in the L2 cache with varied cache time deltas, hereafter referred to as *tempos*. An example pattern might be PC A generating three cache misses to blocks A, A+5, and A+15 with the miss counter equal to 2, 6, and 11; this access sequence has tempos of 4 and 5, respectively.

3.1 Implementing PC-Localized Tempo Filtering

Now that we have added temporal information to each memory access, we can use this information while training the *Tempo* prefetcher. In Figure 2, we show how a dense pattern representing a large number of memory accesses can be filtered into multiple pattern vectors, each representing a distinct access tempo. This is implemented in hardware by splitting both the AGT and PHT into multiple disjoint slices, where the AGT only sets the bits for the cache accesses that were recorded by that PC *with that particular tempo*. Our tempo values are similar to those of the TAS prefetcher [13]: a cache access tempo of less than 4 is classified as fast, from 4-9 is medium, from 9-15 is slow, and over 15 is very slow. We implement this filtering process in hardware by including a small number of local tempo buffers (LTBs) in the *Tempo* prefetcher, which are indexed by a strong hash of the PC and store the last three tempos observed for that particular PC. Therefore, during a new spatial region generation, each L2 cache access calculates the average tempo inside its respective LTB, and sets the correct bit in the AGT slice which corresponds to that LTB value. Figure 3 shows this process. During training, each PC will hash into one of 32 LTB's, calculate the average of the last 3 observed access tempos, and record the access to the current cache block in the pattern vector that corresponds to this access tempo. Upon observing an eviction to any cache block in this spatial region, all slices of the AGT that are usable by this access tempo are sent to the PHT. Usable slices are those which correspond to the access tempo at eviction, including those with slower tempos.

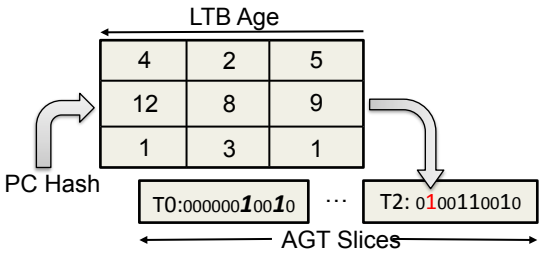

Figure 3: Updating the AGT based on a hash of the local tempo buffer.

3.2 Prefetching Based on Memory Tempo

The process of generating prefetch addresses is similar to how addresses are classified based on access tempo. We make the key observation that we can choose which address candidates to issue as prefetches based on the current utilization of the memory system. *Tempo* has already classified the blocks recorded during this spatial generation into one of four distinct tempos, but that in itself is not enough to ensure prefetch timeliness. This is because the memory system may respond to our prefetches at different rates. For example, if the memory controller is currently heavily loaded with many outstanding requests, it will naturally respond to our demand accesses and prefetches more slowly, and vice versa if it has many free resources to handle incoming requests.

Therefore, as we defined the notion of the current *access tempo* for a particular PC, we also define the *memory tempo* by storing a checkpoint of the global miss counter (Section 3) in every L2 MSHR when it is allocated for a new demand request. When a demand request is filled into the cache and the MSHR is freed, we calculate the delta between the current miss counter and the value returned from the MSHR. This *memory tempo* is stored in a single global tempo buffer (GTB) which tracks the last seven observed memory tempos. We use this memory tempo information when *Tempo* observes a trigger access; the current local access tempo (from the LTB's) is compared against the current tempo of the memory system (in the GTB), allowing us to make an informed decision about which slices of the current spatial region that we wish to use for prefetching. Figure 4 further explains this process with a step by step walkthrough:

1. The current PC sees a demand request to address B+2.
2. We look up the current tempo for this PC in the LTB, and get an average tempo of 6.
3. All elements of the GTB are averaged to obtain a memory tempo of 11.
4. Therefore, the PC is *rushing* ahead of the tempo at which the memory is returning demand requests, making it likely that if we issue aggressive prefetches from the *fast* and *medium slices*, they will not return in time to cover the incoming demand miss.
5. *Tempo* therefore chooses to issue prefetches from the *slow* and *very slow* slices of the PHT, since it is likely that these cache blocks will be returned to the L2 cache in time to cover demand misses.

Also note that the inverse is true for the case where the GTB is returning requests faster than the current PC is issuing them (this is referred to as the PC *dragging* behind the memory's tempo). In this case, we can issue prefetches more aggressively from the faster tempo slices, knowing that the memory system will likely return these blocks in time for the upcoming demand accesses.

What differentiates the *Tempo* prefetcher from other existing priority schemes is the fact that our priority scheme is proactive

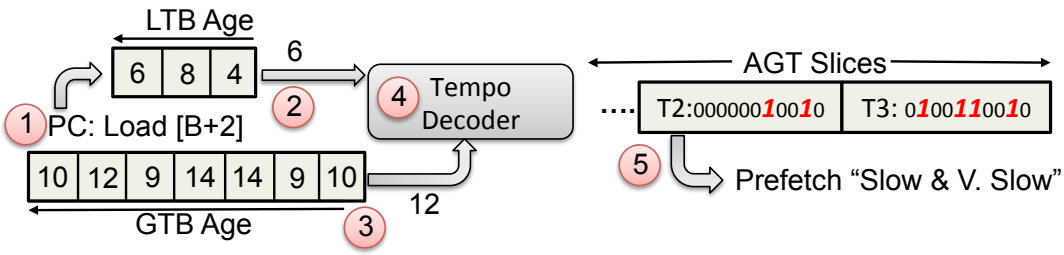

Figure 4: Tempo Prefetching Walkthrough

Data Structure	Components (each entry)	Budget
AGT (64 entries, 4 slices) 2-way Assoc	Tags: 64b addr - 12 Pattern Vectors: 63 Offset: 12 PC:14	36096b × 2 = 72192 bits
PHT (256 entries, 4 slices) 2-way Assoc	Tags: 14b PC Pattern Vectors: 63	78848b × 2 = 157696 bits
MSHR Structures (16)	Tags: 64b addr - 12 Valid: 1 Time Counter: 10 Cycle Counter: 64b	2032 bits
Local Tempo Buffers(32)	Each: 3-wide Time Counter: 10b	960 bits
Global Tempo Buffer (1)	Buffer: 7-wide Time Counter: 10b	70 bits
AMAT Registers(2)	Cycle Counters: 64b	128 bits
Total Bits:		**233,078 bits**

Table 1: Hardware Storage Breakdown for Tempo

rather than retroactive. Feedback Directed Prefetching [11] relies heavily on statistics such as prefetch accuracy and cache pollution to dynamically throttle its stream-based prefetcher based on the characteristics of the memory system. Although we do include a very simple throttling mechanism in *Tempo* based on average memory access time, the defining characteristic of our prefetcher is how we actively measure the access and memory tempos **during training**, and use that information to decide which prefetches to issue. This scheme is also in contrast to the filtering method of sandbox prefetching, which testing prefetches for accuracy and then issues only accurate prefetches during the next PC traversal of that respective cache region [7].

3.3 Hardware Storage Budget

Table 1 breaks down the hardware budget for *Tempo*. For simplicity's sake, we have chosen to implement each *Tempo* slice by replicating each of the SMS structures four times. However, we could represent each slice more economically by using a three bit counter for every cache block, where different values would represent this cache block being accessed by different tempos, saving 25% of the storage. We also include an extra 32-bit cycle counter in each MSHR to facilitate our simple throttling mechanism which is based on the average memory access time (AMAT). However, our experiments show less than 2% overall IPC impact from dynamic throttling, so this is an optional level of complexity.

4. Evaluation

Using the Pin [5] tool provided in the DPC-2 framework [8], we generate traces for SPEC CPU2006, compiled with GCC 4.0. We fast forward each benchmark for 10 billion instructions into the region of interest, before gathering a trace of 100 million instructions. The DPC-2 framework models an aggressive 6-wide pipeline that can issue two loads and one store per every cycle, as well as a three level cache hierarchy, with the following characteristics:

- 16kB, 8-way L1 cache, with 8 outstanding requests to the L2.
- 128kB, 8-way L2, with 10 cycle access latency to the tag array. The L2 can have 16 outstanding requests to the L3.
- 1 MB, 16-way LLC.

Our prefetcher trains on the L2 access stream, prefetches into the L2 cache, and can request its prefetches to be filled to the LLC if the MSHR occupancy is greater than 10 out of the 16 available.

4.1 Results

To evaluate the performance of *Tempo*, we compare its performance against a 32kB SMS prefetcher with pattern rotation using IPC, cache miss rate and the number of useless prefetches. In Figure 5, we compare the absolute IPCs achieved by *Tempo* and SMS, both compared to a baseline that does not include any prefetching. These results are for the "normal" configuration with a large LLC and a memory bandwidth of 1600 MT/s. We have omitted comparisons for *gromacs, namd, soplex, povray, calculix*, and *omnetpp*, as we see essentially no variation in IPC for these applications. *Tempo* achieves an average IPC increase of 1.45% over SMS for these applications, with a 20.4% benefit on the most memory intensive benchmark, *mcf*.

In Figure 7, we plot the number of useless prefetches issued by *Tempo*, normalized to the baseline SMS prefetcher (values greater than 1 correspond to an increase). Useless prefetches are undesirable for two reasons: they may simply never be referenced by the processor, which wastes energy in moving an unused block into a higher level cache, or potentially cause harmful cache pollution by evicting cache blocks which are still useful. We find that our reductions in useless prefetches are primarily due to the elimination of prefetch requests from the faster tempo slices, particularly when the memory is operating in the *very slow* tempo range.

However, we notice that in two benchmarks (*libquantum* and *omnetpp*), the number of useless prefetches actually increases. In both applications, Tempo issues a far greater number of prefetches than SMS due to the fact that AGT and PHT entries have a potentially longer lifetime than in an equally sized SMS prefetcher. In *Tempo*, each of the AGT and PHT slices can individually remain active as long as this PC is still generating requests at the matching tempo, where SMS evicts the entire pattern vector on the first eviction in the AGT, or tag conflict in the PHT. This discrepancy of how pattern vectors are managed between the two prefetchers gives *Tempo* the potential for more prefetch-generating PHT hits over time. Figure 6 compares the performance of *Tempo* and SMS in DPC-2's low memory bandwidth configuration of 400 MT/s. When operating in this restricted memory system, *Tempo* attains an IPC increase of 2.57% over SMS for these benchmarks, due to the fact that it judiciously issues more timely prefetches than SMS.

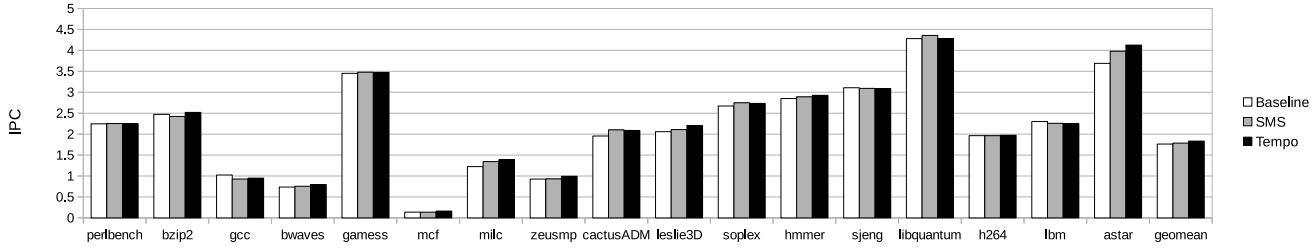

Figure 5: IPC for selected SPEC CPU2006 benchmarks.

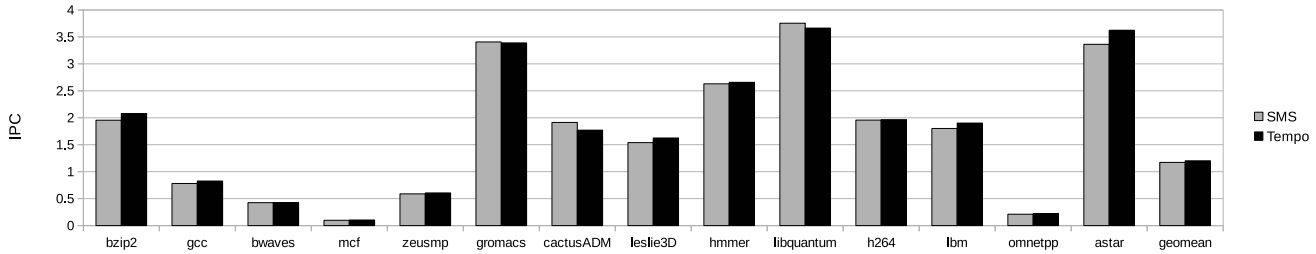

Figure 6: IPC for selected SPEC CPU2006 benchmarks, measured in low bandwidth mode.

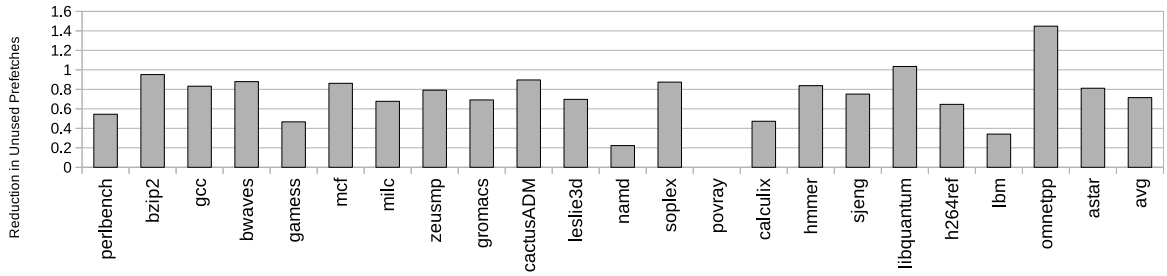

Figure 7: Reduction in useless prefetches for Tempo.

5. Conclusion

In this paper, we have described *Tempo*, an improvement to the Spatial Memory Streaming prefetcher. *Tempo* makes use of the observation that similar PCs tend to access the cache with regular frequency to temporally classify spatial patterns within physical memory pages, and significantly reduces untimely prefetches when compared against a baseline SMS design.

References

[1] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Tr. on Comp*, 1999.

[2] M. Dimitrov and H. Zhou. Combining local and global history for high performance data prefetching. *JILP 13*, 2006.

[3] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 2004.

[4] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proc. of. ISCA*, 1998.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of PLDI*, 2005.

[6] K. Nesbit, A. Dhodapkar, and J. Smith. AC/DC: an adaptive data cache prefetcher. In *Proc. of PACT*, 2004.

[7] S. Pugsley, Z. Chishti, C. Wilkerson, P.-F. Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proc. of HPCA*, 2014.

[8] S. Pugsley et al. Dpc-2 simulation framework, 2015. URL `http://comparch-conf.gatech.edu/dpc2/`.

[9] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proc. of ISCA*, 2006.

[10] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *Proc. of ISCA*, 2009.

[11] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. of HPCA*, 2007.

[12] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journ. Res. Dev.*, 2002.

[13] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: Improving timeliness in data prefetching. In *Proc. of ICS*, 2010.