Existing and Novel Spatio-Temporal Prefetchers

Will Dey, Dr. Akanksha Jain, Dr. Calvin Lin
Department of Computer Science
The University of Texas at Austin

***Abstract*—Prefetching is a very important technique in modern computer architecture that hides the large performance gap between today's processors and main memory by predicting the memory accesses the processor will make in the future and requesting them ahead of time. *Spatio-temporal* prefetching is a new strategy that observes patterns in proximity of requested addresses as well as patterns of accesses over time to form a prediction.

This report provides the first independent implementation and evaluation of the well-known Spatio-Temporal Memory Streaming prefetcher (STeMS) and also introduces the novel idea of branch-guided *basic-block prefetching*, which holds promise as a new avenue for designing inexpensive, highly accurate spatio-temporal prefetchers in the future. These prefetchers are evaluated along with other designs using the industry-standard SPEC CPU 2006 benchmarks to provide a comprehensive overview of the current state-of-the-art in prefetching. A simple basic-block prefetcher is shown to achieve the highest accuracy of all tested designs: 69.6%, as opposed to the near-50% accuracy of others. STeMS implementation results are further analyzed, with new insights that recommend the removal of its complicated streaming engine that poses minimal benefit. However, the results encourage addition of a prefetch buffer, like STeMS's SVB, for irregular prefetchers.

1. INTRODUCTION

Modern processors have become orders of magnitude faster than their memory counterparts, since hardware and cost limitations force main computer memory to be produced from the relatively slow Dynamic RAM (DRAM). Waiting for data to arrive from DRAM can stall a processor for thousands of cycles, resulting in a very noticeable slowdown, especially in database systems where more than half of the time can be wasted in these data stalls [1]. *Prefetching* has proven to be a very effective technique to hide these long latencies: by observing patterns in memory accesses, a prefetcher can predict future accesses and request them ahead of time. Data will ideally be loaded by the time the processor requests it, hiding DRAM latency.

A number of prefetcher types have been proposed, which can be broadly categorized into three groups: spatial, temporal, and spatio-temporal.

Spatial prefetchers attempt to learn patterns in proximity of addresses, i.e. when block `A+4` is accessed after some cache block `A`. Such designs are often well-suited for situations like accessing fields of a `struct`, which are a static distance apart in memory.

Temporal prefetchers observe patterns of memory accesses by recording the order of accesses over time. Since they can predict patterns of addresses that are large spacial distances apart, temporal prefetchers can capture more complex patterns, like iterating linked-lists. However, they are unable to predict spatially correlated addresses if they are requested too far apart in time, and have much larger space requirements to store order. As a result, temporal prefetchers can generally cover more access patterns overall but are not effective at covering fine-grained spatial patterns.

Spatial-temporal prefetchers aim to combine both approaches to cover a wider variety of memory access behavior and achieve greater system speedups. Motivating examples for a spatio-temporal prefetcher which pure-spatial or pure-temporal strategies have trouble with include traversing trees of `struct`s for critical database systems, or managing they dynamically allocated objects in the heaps of garbage-collected languages.

The ideal spatio-temporal prefetcher would also degenerate to pure spatial or temporal prediction when necessary and handle spatially-correlated elements embedded within temporally-correlated patterns, therefore covering all possible program behaviors.

This report makes the following contributions:

1. The first independent implementation and evaluation of Wenisch et al.'s well-known Spatio-Temporal Memory Streaming (STeMS) prefetcher [2].
2. Clarification on many important details of STeMS which were previously either unspecified or only described Wenisch et al.'s more specialized papers [3][4][5], to ease future implementation and promote further research in spatio-temporal techniques
3. The introduction of basic-block prefetching using the program structure information from observing branches, a low-cost but promising technique which achieves the highest accuracy out of all current state-of-the-art prefetchers: 69.6%.
4. A number of observations on the behavior of STeMS, including the minimal benefit from its

complicated streaming engine and the performance gains which can be attributed to a prefetch buffer.

The rest of this report is organized as follows: Section 2 puts the work in this report in context and provides useful background information on referenced papers, Section 3 provides a more complete description of STeMS with previously undescribed implementation details and concerns, Section 4 presents a novel branch-guided prefetching solution and provides the rationale behind this technique, Section 5 describes our methodology and evaluates STeMS and the potential of basic-block prefetching, before concluding in Section 6.

2. RELATED WORK

Spatial Prefetching

One example of a very simple prefetching scheme that exploits spatial locality is stride prefetching, which always prefetches the `A±n` line, with some fixed `n`, when some cache line `A` is accessed.

Many designs have been proposed that attempt to use the commonplace spatial relationships in programs, but with wider adaptability. This report focuses on the Spatial Memory Streaming (SMS) prefetcher [3], which is representative of the current state-of-the-art for pure-spatial prefetching. SMS splits the entire address space into *spatial regions* and records accesses to regions in bit-vectors, prefetching recorded patterns when the program is on the same instruction (using the *Program Counter*, or *PC*) and access the same relative location inside a region. In this way, SMS can predict the same patterns a stride prefetcher can, as well as patterns that are not the same in are regions of memory and involve multiple cache lines to prefetch.

SMS is described in greater detail in Section 3.1, as it is a subsystem in the implementation of STeMS.

Temporal Prefetching

Recent temporal prefetchers utilize the Global History Buffer (GHB) [6], a large circular buffer that stores the sequence of L1 misses (L2 accesses) as they arrive. An index table maps each address to a pointer for its latest occurrence in the buffer. When a previously-recorded address is missed in the L1, the GHB looks up its entry in the index table, and begins prefetching the stream following the miss in the buffer.

The Temporal Memory Streaming (TMS) prefetcher [5] uses a GHB for its core function, but refers to it as a Region Miss Order Buffer (RMOB). The two data structures are equivalent in their function, but this report uses "RMOB" from here onward to avoid confusion when referencing Wenisch et al.'s work. TMS also features a complex system for managing multiple candidates for prefetch streams on top of the RMOB, labeled the Temporal Streaming Engine (TSE) and a prefetch buffer (with metadata) termed the Streamed Value Buffer (SVB).

Wenisch et al. have published two papers on the TMS design: pure-TMS for shared-memory multiprocessors [5], and a slightly altered version for space reduction which only maps a sample of the RMOB into index table, called Sampled TMS (STMS) [4]. As these papers essentially describe the same prefetcher with minor changes, this report refers to them both as "TMS."

TMS is the second of the two subsystems embedded in the STeMS design and, like SMS, is described in greater detail later, in Section 3.2.

Spatio-Temporal Prefetching

The most widely known spatio-temporal prefetcher is Somogyi and Wenisch et al.'s Spatio-Temporal Memory Streaming (STeMS) design [2], which links together spatial predictions from SMS temporally with the TMS predictor, combines the predictions in a complex reconstruction process, and feeds the combined spatio-temporally predicted stream into the streaming engine from TMS for prefetching to a prefetch buffer. The primary insight STeMS contributes is its reconstruction processes which aims to prefetch addresses in the order the processor would request them, rather than naïvely prefetching all spatial elements at once (see Figure 1), which can waste prefetch bandwidth and cause contention in the cache very quickly [2].

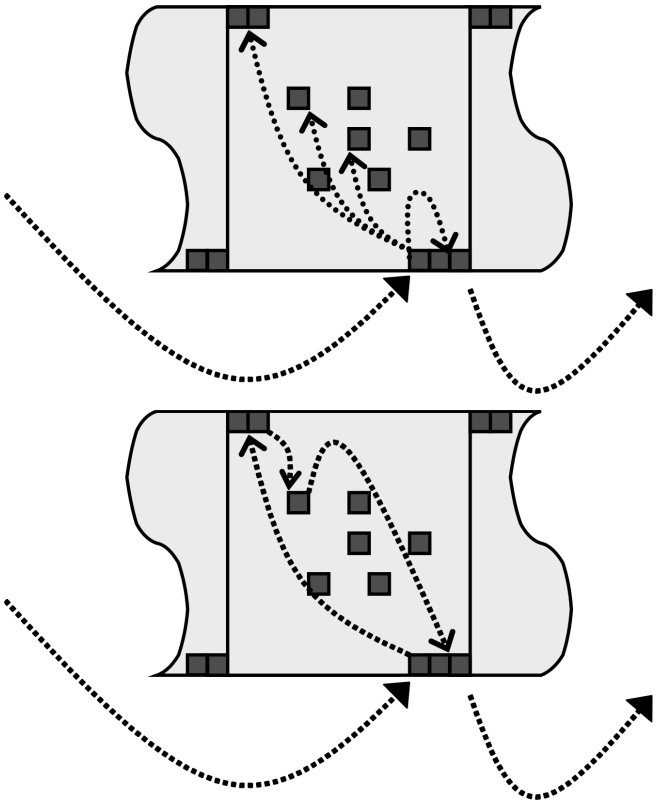

Figure 1: Naïve spatio-temporal prediction order versus the true access order and target prediction of STeMS. Small dark-gray squares represent data spread out spatially in memory. Large gray containers represent spatial regions of memory. Arrows show the order of access to data. [2]

However, as this report describes, the STeMS scheme is not the only way to achieve spatio-temporal prediction, nor is it necessarily the best way.

Prefetchers like the Irregular Stream Buffer (ISB) [7] can be configured in a dual coarse-grained/fine-grained fashion that can capture spatio-temporal behavior, with an extra layer of indirection in prediction ensure proper prefetch order, rather than reconstruction [7].

A highly accurate, novel design that captures the benefits of spatio-temporal prediction without the complicated and traffic-heavy reconstruction process is introduced in Section 4, opening promising avenues for speedups with prediction guided by the program structure information afforded by observing branches.

Branch-Guided Prefetching

Some previous proposals have used branch information to augment prefetch predictions, such as B-Fetch [8], which attempts to extrapolate future access addresses based on current register data and branch predictor information. However, the novel Basic-Block (BB) prefetcher that is introduced in this report is the first to model program structure from branch taken/not-taken information and use the model to classify memory elements as spatially-correlated or temporally-correlated.

3. SPATIO-TEMPORAL MEMORY STREAMING (STeMS)

No implementation of the STeMS spatio-temporal prefetcher has ever been provided since its in 2009 [2], which has been a roadblock preventing research of what in particular contributes to STeMS's performance and spatio-temporal prefetching in general.

STeMS is a combination of two of Wenisch et al.'s previous prefetching designs: Spatial Memory Streaming (SMS) and Temporal Memory Streaming (TMS). In order to provide a complete description of the work that goes into the operation of STeMS, this section first covers the SMS subsystem then the TMS subsystem. It then describes how they link together from start to finish of the prefetching pipeline: from training, to recall and reconstruction, to the streaming engine, and finally to what is essentially a prefetch buffer (the SVB).

3.1 Spatial Memory Streaming (SMS)

SMS [3] observes spatial correlations by splitting all memory into fixed-size spatial regions (Wenisch et al. found 2048-long spatial regions to provide maximum coverage [2][3]), and recording all L1 accesses to each region in a bitset over the course of a *generation*. A *generation* in SMS the series of accesses between the first access to a spatial region (the *trigger access*) and eviction of any address in that spatial region from the cache (see Figure 3). The rationale behind also requesting notifications of cache evictions from the core is so that SMS does not record addresses to be prefetched later, only to have them deemed too unused to be kept and evicted on arrival.

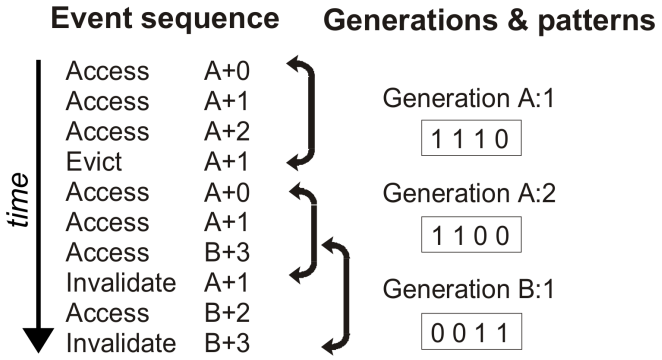

Figure 3: Interleaved generations in SMS, with spatial regions `A` and `B`. Each access to a cache block sets the corresponding bit in that spatial region's bitset. Generations start on first access and end on eviction/invalidation. [3]

Active Generation Table (AGT)

During training, the bitsets of all the generations which are currently active (able to record additional accesses) is stored in the Active Generation Table (AGT), whose operation in the original SMS paper is

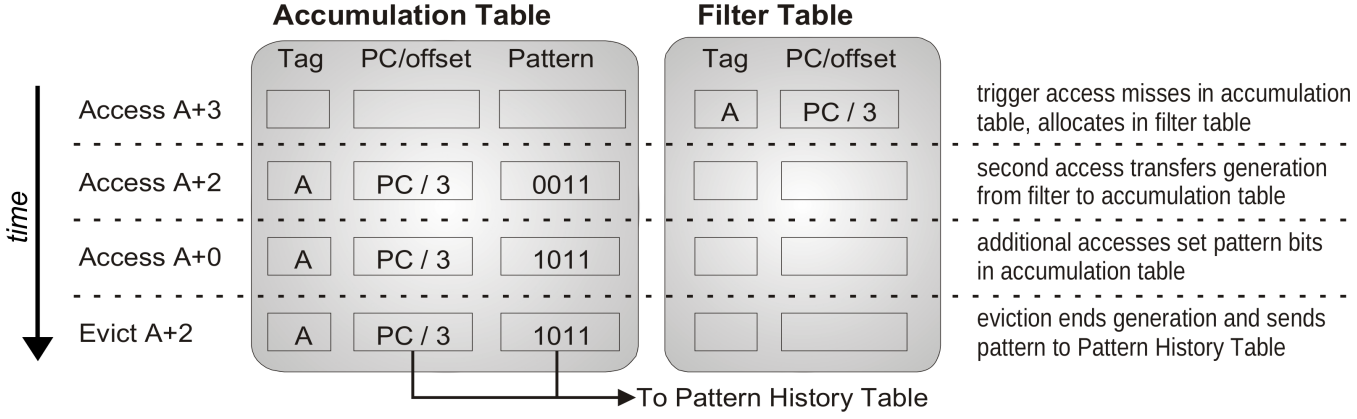

Figure 2: Operation of the Active Generation Table (AGT) in SMS. The AGT is composed of two smaller tables as an optimization, to avoid storing extra data for generations that only span one access. Generations are stored in a Filter Table first and move to the main Accumulation Table on second access. [3]

illustrated in Figure 2. The AGT is tagged by spatial region number, and records the *PC+offset* and access bitvector for each active generation entry. The *PC+offset* value is created by joining the Program Counter (PC) of the instruction which initiated the trigger access and the *offset* of the trigger access, the relative location of the trigger access within the spatial region. In implementation, this is done by shifting the PC right and bitwise `OR`ing it with the low-order bits of the address, a process which is formally described in Line 11-12 of Algorithm 1. The PC+offset value becomes significant later as it is used to tag the second SMS data structure, the Pattern History/Sequence Table (PHT/PST) and has implications on the coverage of the SMS recall process.

Though the AGT is logically one table, it is implemented as two as an optimization. Generations are recorded in a Filter Table first, and are moved to the main Accumulation Table only on a second access, effectively filtering out single-access generations. An important implementation detail that was not mentioned by Somogyi and Wenisch et al. is that when SMS is used inside of STeMS as a subsystem, there must be no Filter Table to remove single-access generations because the first access to a generation is always missed by SMS and recorded by TMS instead. Any access that the SMS subsystem actually receives to record must be a second access and must always be recorded since the first access was captured by TMS.

Another very important modification made to the AGT when used inside STeMS is that it no longer uses bitvectors to record spatial patterns and uses a bounded list instead to also record the order of spatial accesses in the stream, which becomes significant in therms of the SMS recall and later reconstruction procedure.

Finally, a *reconstruction delta* which provides information about the relative position of the access in the overall access stream is stored along with all recorded elements in STeMS. For spatial elements, the reconstruction delta is the number of other accesses that occurred between the current spatial element and the previous element in the generation, which could be the trigger access. The spatial sequences in Figure 4 illustrate the deltas for each element. (Note that the use of the reconstruction delta does *not* amount to delta correlation, like to type referred to be Nesbit and Smith [6], because it is not a delta in address space, but rather in time).

Pattern History/Sequence Table (PHT/PST)

On eviction of any block in a generation in the AGT, it is moved to the Pattern History Table (PHT) for more permanent storage. The PHT is tagged by PC+offset and stores spatial patterns.

However, the PHT now requires more space to store order information to prefetch data in a more accurate stream as shown in Figure 1. Its increased size forces it to be moved off-chip. The off-chip version of the PHT is renamed by Wenisch et al. to the Pattern Sequence Table (PST) for STeMS to reflect the fact that the table now stores sequential order information. Because information about order is inherently temporal, the version of SMS included with STeMS acts more as a fine-grained temporal prefetcher rather than a true spatial prefetcher. The modified SMS can nevertheless provide some spatial prediction benefit because it primarily tracks accesses to fixed spatial regions and will pick up on spatial correlations that would be lost in a pure-temporal prefetcher.

3.2 (Sampled) Temporal Memory Streaming (STMS/TMS)

Region Miss Order Buffer (RMOB)

Temporal Memory Streaming (TMS) [5] is essentially a Global History Buffer (GHB) [6] at its core but refers to its circular data structure as a Region Miss Order Buffer (RMOB) instead, as noted previously in Section 2.

Like SMS, TMS also stores a reconstruction delta when used in STeMS. In the TMS case, the reconstruction delta represents the number of accesses between

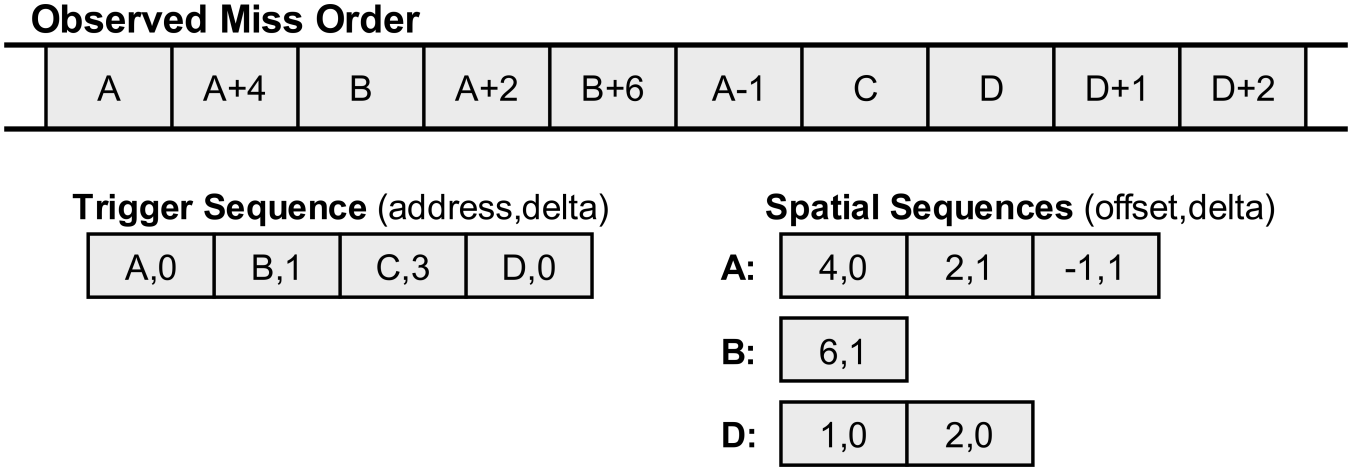

Figure 4: The STeMS process of breaking down an L1 miss/L2 access stream into spatially-correlated and temporally-correlated elements, with accesses to spatial regions `A`, `B`, `C`, and `D`. [2]

the current temporal element and the previous, as illustrated in Figure 4.

Finally, since the purpose of TMS in STeMS is to link together entries of the spatial PST in temporal space, STeMS's TMS also stores the PC of the instruction which initiated the trigger access so that a PC+offset tag may be recalculated later to look up the matching PST entry for a given temporal trigger (see full RMOB diagram in left of Figure 5).

The original TMS also included a Temporal Streaming Engine (TSE) that fed into its own prefetch-buffer-like Streamed Value Buffer (SVB), which is also employed by STeMS later in the pipeline. The operation of these rather involved mechanisms are described in further detail later in Section 3.5.

3.3 Training

STeMS training is more-or-less a simple combination of the SMS and TMS training routines, with SMS always taking precedence in recording the address in its data structures if possible, and TMS being a fallback that records anything that the SMS subsystem has never seen previously. In this way, the TMS subsystem only records the temporal sequence of never-before-seen spatial triggers, linking together entries in the PST across time (see Figure 4).

3.4 Reconstruction

The primary insight of STeMS was its reconstruction process, which mixed temporal and spatial predictions in a manner which tries to best match the interleaved way the processor will actually request the data. The goal of reconstruction is a prediction similar to the bottom of Figure 1.

The STeMS paper was lacking an implementation of this complicated process, which prevented a correct implementation for many years. In the interest of unambiguity and a formal description, pseudocode with specifics of the process is provided for the first time in this report as Algorithm 1.

When skimming this pseudocode, it should be noted that the **RMOB**, the RMOB's **Index** table, and the **PST** are all off-chip datastores, immediately making

```
1  procedure RECONSTRUCT(trigger)
2      i ← Index_trigger                ▷ position in RMOB
3      j ← 0                            ▷ temporal cursor
4      while j < size(Buffer) do
5          if j ≠ 0 then                ▷ ignore first delta
6              j ← j + RMOB_i.delta
7          end if
8          PLACE(RMOB_i.address, j)
9          j ← j + 1
10         k ← j                        ▷ spatial cursor
11         offset ← RMOB_i.address mod 2048
12         tag ← (RMOB_i.PC × 2048) + offset
13         for element in PST_tag do
14             k ← k + element.delta
15             address ← RMOB_i.address +
                         element.offset
16             PLACE(address, k)
17             k ← k + 1
18         end for
19         i ← (i + 1) mod size(RMOB)
20     end while
21 end procedure

22 procedure PLACE(address, position)
23     for offset in { 0, 1, -1, 2, -2 } do
24         if Buffer_position+offset is empty? then
25             Buffer_position+offset ← address
26             return success
27         end if
28     end for
29     return failure
30 end procedure
```

Algorithm 1: STeMS Reconstruction.

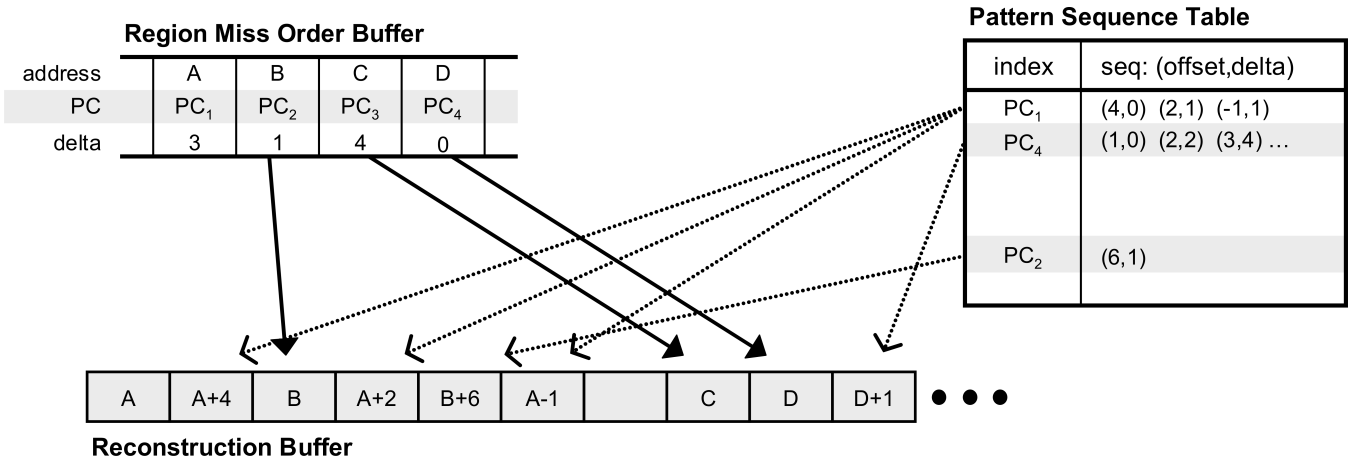

Figure 5: STeMS Reconstruction. [2]

reconstruction a very expensive, traffic-heavy procedure. STeMS does not provide any mechanisms specifically to mitigate this slowdown, but the impact of numerous off-chip accesses may be hidden because ongoing process performed off of the critical path and because data is being streamed into the chip from the RMOB and PST for reconstruction. This report's implementation has been tuned to use such optimizations wherever possible.

While the pseudocode will be definitely useful for allowing further research and testing implementations of STeMS, an informal description is in order to explain the algorithm: On a miss that hits in the RMOB, the reconstruction process is started. For purposes of explanation, this report introduces the terms *temporal cursor* and *spatial cursor* to refer to the positions at which the next temporal or spatial element will be placed, respectively. Both are initialized to 0. The address for the temporal hit from the RMOB is placed in the reconstruction buffer at the temporal cursor. A PC+offset tag then is calculated from the RMOB data for the hit and used to look up a spatial pattern from the PST. The spatial cursor is set equal to the temporal cursor. For each element of the spatial pattern, the spatial cursor is advanced by the number of places indicated by the reconstruction delta, and the address for the spatial element (calculated by adding that element's spatial offset to the base temporal position) is placed at the spatial cursor. At any time during placement, the target slot may already be filled with an address. In this case, as Somogyi et al. describe [2], STeMS will search two places forward and backward from the target position in the reconstruction buffer for an empty spot, which evidently allows 99% of all addresses to be placed somewhere, and 92% placed in their original location [2].

This process continues, streaming more temporal elements from the RMOB until the reconstruction buffer is filled with addresses to be prefetched and moved to a *stream queue* for prefetching (see Section 3.5).

Spatial-Only Streams

One special degenerate case of reconstruction is when an L1 miss does not hit in the TMS subsystem, but *does* hit in the SMS subsystem. Since spatio-temporal prefetchers should ideally perform at least as well as pure-spatial prefetchers, STeMS degenerates to pure SMS in this case. All reconstruction deltas are ignored, and spatial elements from the hit entry in the PST are placed consecutively in the reconstruction buffer. Though spatial-only streams can approximate SMS, they are not equivalent because the reconstruction buffer is then moved to the TMS-like streaming engine. (see Section 3.5).

In this manner, STeMS also attempts to cover *compulsory misses* – misses on never-before-seen addresses – as SMS does. The PC+offset tag becomes significant here because, unlike TMS, it allows the SMS subsystem to hit in the PST when the exact address does not match, but the same part of the program accessed it and at the same offset in a spatial region, which acts as a type of fingerprint for a particular spatial pattern.

3.5 Streaming Engine

A second key source of some complexity in the STeMS design is the streaming engine, which manages *stream queues* and a prefetch-buffer-like structure known as the Streamed Value Buffer (SVB). These elements are borrowed directly from TMS, where the streaming engine is known as the Temporal Streaming Engine (TSE) and applied in STeMS to manage prefetching.

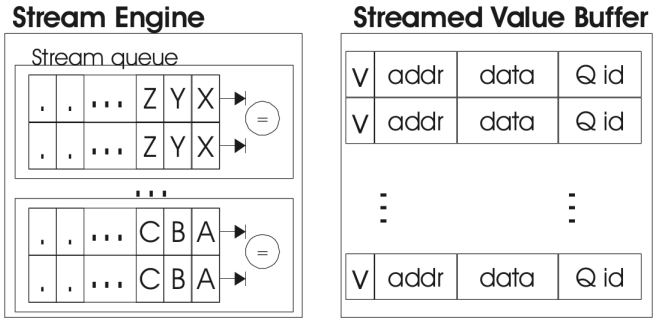

Figure 6: The STeMS Streaming Engine. [5]

A *stream queue* is a FIFO queue of addresses waiting to be prefetched. The STeMS streaming engine manages multiple stream queues with IDs assign to each, all of which are candidate streams of for prefetching (see Figure 6). At any given time, addresses from one stream queue are followed for prefetching into the SVB, which stores a valid bit, the memory address of the prefetched data, and the data itself, similar to a standard cache or prefetch buffer. However, the SVB additionally stores some STeMS-specific metadata: the ID of the stream queue the address was prefetched from. The SVB is searched in parallel with the L2 for data on an L1 data cache miss (see Figure 7). On a hit in the SVB, the streaming engine will begin prefetching from the queue ID of the hit entry of the SVB. Once a stream queue is half-full, reconstruction begins again for that stream from its last temporal trigger to refill the queue.

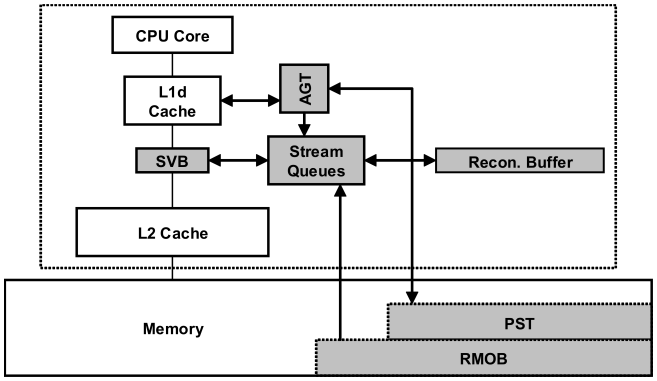

Figure 7: Block diagram illustrating all parts of STeMS. [2]

The streaming engine appears to be a method for mitigating the impact of inaccuracies in the core prefetching scheme, as it amounts to only prefetching predicted streams which appear to be correct after waiting for hits in the prefetch buffer. Limiting the streaming engine to 1 stream queue would degenerate into a direct pipeline into the SVB and can be used to nullify the effect, if any, of the streaming engine and determine how beneficial it actually is. This method of experimentation is followed and analyzed during this report's evaluation in Section 5.

4. BASIC-BLOCK PREFETCHER (BB)

In this section, a novel prefetcher design is presented which aims to be more general and adaptable to different program structures and workloads by recognizing that branching in programs is the sole cause of time-varying memory behavior (or any time-varying operation of the program for that matter), and applying this fact to guide spatio-temporal prediction.

A *basic-block* is defined as the branch-less region between two branch instructions in a program. These basic building blocks are switched between using branch instructions, making up the entirety of the program (see Figure 8). The primary insight of BB is that the vast majority of memory accesses within a basic-block are static and therefore spatially-correlated (static elements cannot be temporally-correlated). The only case in which a temporally correlated element can exist well within a basic-block is in pointer arithmetic, of which array accesses are a subset, because they can change the location of a memory access based on external, time-varying factors. Pointer arithmetic is uncommon and discouraged in writing programs due to its unreliability when done by hand, but array accesses are a very common occurrence of safer, compiler-generated pointer arithmetic. Figure 8 shows one example of the pointer arithmetic implicit in array access (when `array` is an array of 4-byte-wide `int`s).

C	x86 Instructions
`int sum = 0;`	`MOV sum,0x0`
`int i = 0;`	`MOV i,0x0`
`do {`	`LOOP:`
`  sum += array[i];`	`MOV EAX,i`
	`MOV EBX,`
	`  array+EAX*4`
	`ADD sum,EBX`
`  i++;`	`ADD i,0x1`
`} while (i < n);`	`CMP EAX,n`
	`JL LOOP`

Figure 8: Side-by-side C program source and compiled machine instructions. Basic-blocks are highlighted in gray, branch instructions are in yellow, and embedded temporal elements in the basic-block are in red. *Italicized assembly* represents the memory address of a variable in C.

However, these cases are easily handled in the general case by removing a memory access from the spatial pattern and adding it as a lone, temporally-correlated trigger when that memory access is observed to be different when the same pattern is recalled later. Effectively, this will mark particular accesses as time-varying and temporally-correlated in a static basic-block on a second pass.

Only 2 extra bits must be pulled from the core to provide branch information to BB, as opposed to the 64 bits STeMS and SMS require to be informed of cache evictions. When the core encounters and then ultimately takes/skips the branch (ruling out any speculative paths), it will use one bit to inform BB that a branch was taken and another to indicate whether or not the branch was taken. All branch notifications take place off of the critical path.

BB is designed to be simple to provide a starting point for future basic-block prefetching advances and evaluate its potential. Essentially, it is the equivalent of a stride prefetcher that works on units of basic-blocks rather than singular addresses. These basic-blocks are then prefetched consecutively. The complicated and traffic-heavy reconstruction process is completely unnecessary in spatio-temporal prefetching with basic-blocks because basic-blocks can never be interleaved on a single core and are small enough that accounting for interleaved behavior with multiple cores poses no benefit.

Since branch information provides direct insight into the program structure, judging spatio-temporal locality based on branch encounters is likely to be be more accurate and adaptable to a wide variety of workloads than current solutions.

5. EVALUATION

Checking Correctness

Before simulation on workloads, the independent STeMS implementation was checked to verify that the reconstruction procedure put forth in this report (see Algorithm 1) faithfully reproduces the complex process that was intended and informally described by Somogyi et al. [2]. In particular, memory traces were generated whose accesses exactly match the illustrative example from Figure 2 and 5 from the original STeMS paper: `{A,A+4,B,A+2,B+6,A-1,C,D,D+1,D+2}`. The example is successfully decomposed during training as Figure 2 illustrates and reconstructed with full accuracy when the `A` cache block is missed in the L1 later. Direct inspection of simulation datastructures also proves the proper flow of addresses to prefetch data through the streaming engine and SVB.

5.1 Methodology

Both the independent STeMS implementation and BB are evaluated with ChampSim 2, a high-speed, accurate x86 simulator with particular attention to the memory subsystem [9] (see Table 1 for additional details). The simulation infrastructure faithfully models

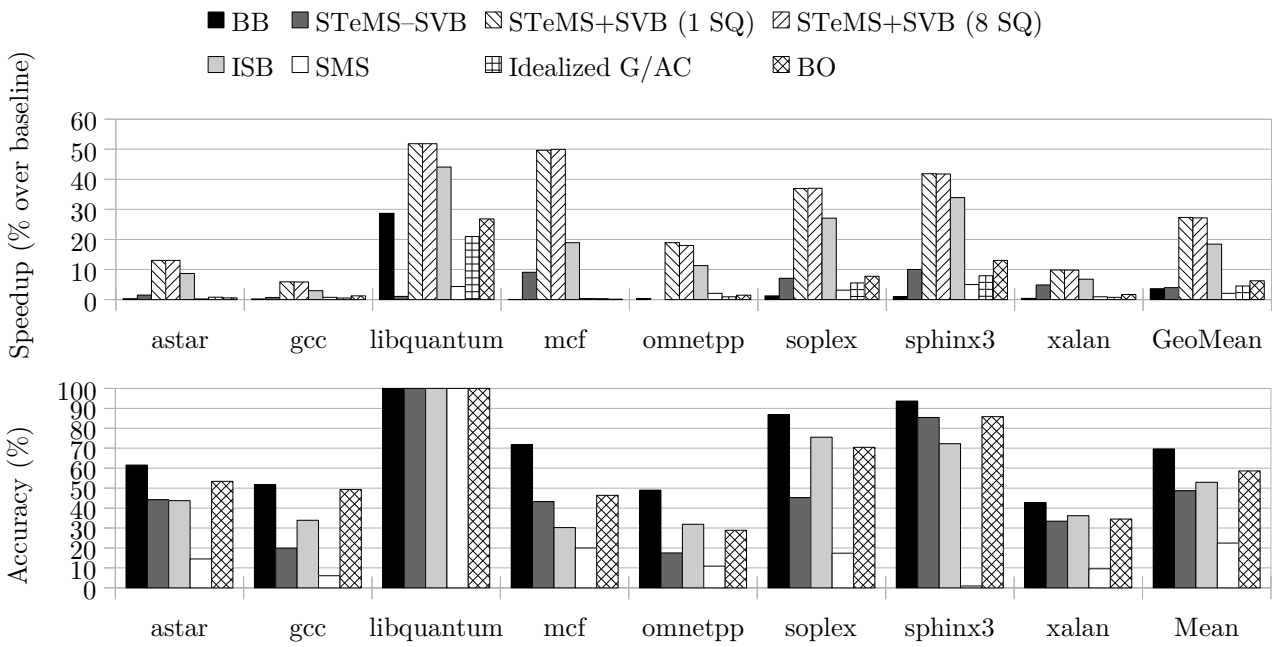

Figure 9: Comparison of speedup (based on instructions per cycle) and accuracy for irregular prefetchers on memory-intensive workloads of SPEC CPU 2006.

cache queue contention, memory traffic and off-chip latencies in prefetchers.

x86 Simulator	ChampSim 2 (from the Cache Replacement Championships [9] and Data Prefetching Championships [10])
Core	Out-of-order execution 256-entry ROB, 72-entry LSQ
Front-end	4-wide fetch Bimodal branch predictor 32KB 8-way L1 instruction cache
L1	32KB 8-way, 4-cycle latency
L2	256KB 8-way, 12-cycle latency 16 MSHRs
L3 (LLC)	2MB 16-way, 32-cycle latency 32 MSHRs
DRAM	55ns latency

Table 1: Baseline configuration parameters.

Benchmarks

Since predicting regular access patterns is largely solved and advancements in spatio-temporal prediction aim to better predict irregular patterns, this evaluation focuses on the memory-intensive, irregular subset of benchmarks from SPECint2006. According to Jaleel's characterization of SPEC2006 [11], a benchmark is considered memory-intensive if it has a CPI > 2 and an L2 miss rate $> 50\%$. Two benchmarks from SPECfp2006 are also included, soplex and sphinx3, which have both regular and irregular memory access patterns. The benchmarks are compiled using GCC 4.2 with level 2 optimization (`-O2`). SSE3/4 instructions are disabled because the ChampSim simulation infrastructure does not support them. All benchmarks are run using the reference input set. The SimPoint sampling methodology is used to generate multiple 250-million-instruction-long SimPoints to capture all characteristic phases of each benchmark. SimPoints were generated using the SimPoint Tool [12]. A SimPoint length of 250 million instructions was chosen because it is large enough to capture long-range behavior, including multiple L2 cache misses on the same address.

Evaluated Prefetchers

BB is simulated along with multiple configurations of STeMS for more in-depth spatio-temporal analysis, as well as other prefetchers (see Figure 9-10) . All prefetchers are set to degree 1 prefetching into the LLC, with a couple exceptions below that are included for analysis.

First, a version of STeMS with the SVB (prefetch buffer with metadata) removed is simulated because no other prefetcher simulated here has the benefit of prefetching to a prefetch buffer parallel to the L2. Instead this SVB-removed version fetches to the LLC like other simulated designs. This prefetcher is labeled "STeMS–SVB."

Second, to remain faithful to the STeMS paper, a version with the SVB included is also simulated, labeled "STeMS+SVB." To quantitatively test the benefit of the complex streaming engine of STeMS, a variant of STeMS+SVB with 1 stream queue is simulated, which degenerates into no streaming engine and nullifies any of its benefits benefits. The single-stream-queue STeMS is labeled "STeMS+SVB (1 SQ)." To contrast, the "STeMS+SVB (8 SQ)" uses 8 stream

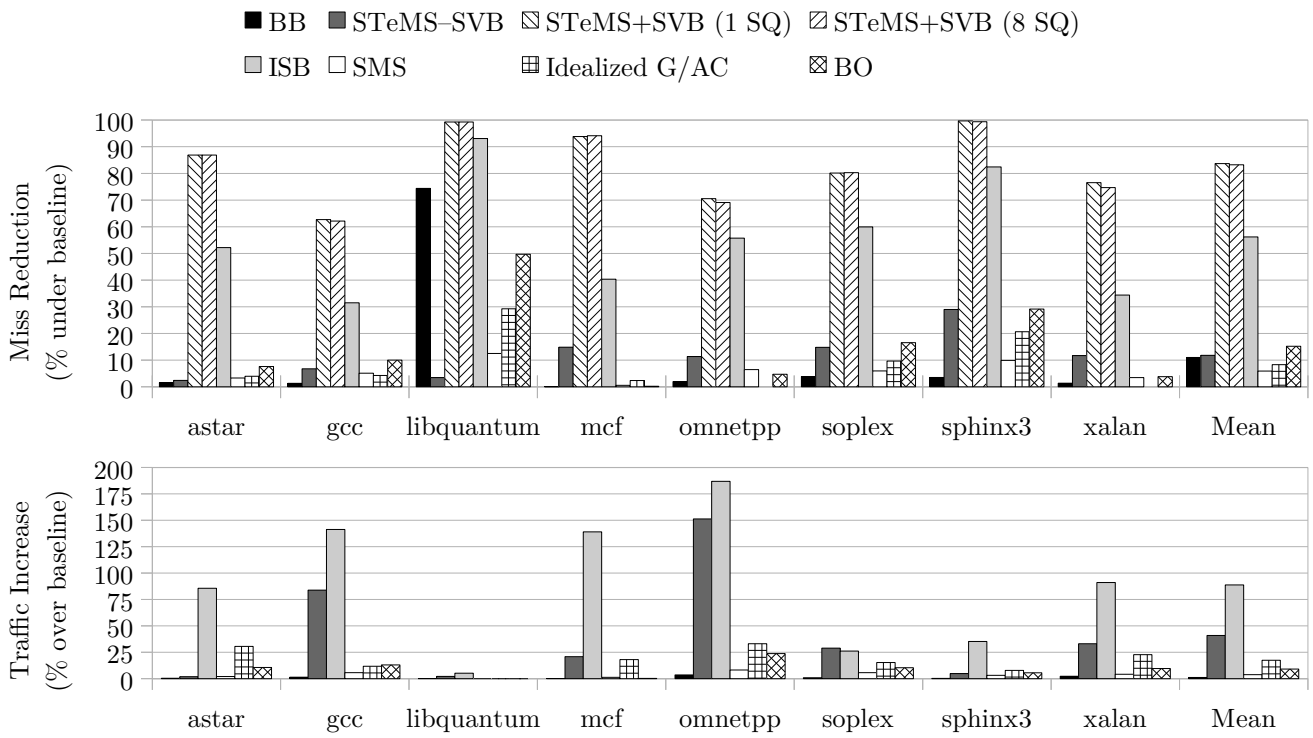

Figure 10: Comparison of miss reduction (coverage increase) and traffic increase for irregular prefetchers on memory-intensive workloads of SPEC CPU 2006. Lower is better for traffic increase.

queues and follows Somogyi et al.'s description of STeMS exactly [2]. Comparing these two variants will reveal any effect that the streaming engine has.

Third, Akanksha and Lin's Irregular Stream Buffer (ISB) [7], the current state-of-the-art the art in temporal stream prefetching, is simulated.

Fourth, Somogyi and Wenisch et al.'s Spatial Memory Streaming (SMS) prefetcher [3] is simulated. Since SMS is a subsystem embedded in STeMS, comparing STeMS–SVB and SMS results will provide an accurate view of the benefits of the added temporal prediction.

Fifth, an idealized G/AC prefetcher, using Nesbit and Smith's terminology [6], is simulated. An idealized G/AC prefetcher has been shown to approximate Wenisch et al.'s TMS well for long temporal streams (the best case for TMS) [4], and can be used to determine what benefit the spatial and compulsory miss prediction of STeMS has over a pure-temporal prefetcher like its TMS subsystem.

Sixth, Michaud's recent Best-Offset (BO) prefetcher [13], which won the 2nd Data Prefetching Championship [10], is also simulated to provide a comprehensive overview of the state-of-the-art in prefetching with these results.

5.2 Results

Figure 9 and 10 compare speedup, miss reduction (equal to increase in coverage), and traffic increase between evaluated prefetchers over a no-prefetcher baseline with the configuration listed in Table 1. Note that, with the exception of the STeMS+SVB variants, all results use degree 1 prefetching to the LLC with prefetchers training on the L2 miss stream for a fair comparison. In practice, these prefetchers may be set to a more aggressive prefetching configuration that increases speedup, but the relationships between the quantitative results for each prefetcher relative to others will remain the same, so all inter-prefetcher comparisons are valid.

Additionally, STeMS+SVB should not be compared to other prefetchers as a valid representation of the STeMS prefetching scheme because it benefits from prefetching into and L2 prefetch buffer unlike other designs. The STeMS+SVB results for speedup and miss reduction (coverage) are included to help analyze and draw conclusions about the benefit of a prefetch buffer by comparing them to STeMS–SVB, and about the benefit of a streaming engine by comparing the 1 stream queue and 8 stream queue variants. STeMS–SVB should always be used to compare the core STeMS scheme with other proposals.

BB, a simplistic start for basic-block prefetching attains a modest 4% mean speedup, matching that of STeMS–SVB. However, it consistently surpasses all other designs in accuracy in all benchmarks. BB attains the highest average accuracy over all the tests at 69.6%, compared to 48.6% for STeMS–SVB, 52.9% for ISB, 22.4% for SMS, and 58.6% for BO. BB's high accuracy indicates that basic-block-centered prefetching is a promising direction. High accuracy can be expected to translate to high speedup given more aggressive prefetching strategies than the current "stride"-like design of BB in the future.

Additionally, because of on-chip datastructures and the relatively cheap cost of branch notifications, BB causes the lowest traffic increase out of all designs: 1.1% for BB, compared to 40.8% for STeMS–SVB, 88.8% for ISB, 3.8% for SMS, 17.4% for an idealized G/AC and 9.1% for BO.

STeMS+SVB variants display massive increases speed and coverage when compared to the relatively modest gains of STeMS–SVB, demonstrating that the majority of the performance benefits of STeMS are due to the addition of a prefetch buffer, rather than the core prefetching strategy itself. The performance of the 8 stream queue variant and 1 stream queue variant (which enjoys no benefit from the streaming engine) are virtually equal, showing that the streaming engine and the extra metadata of the SVB has little to no effect on speed or overage (at least on a single core). Therefore, the majority of the performance of STeMS+SVB can be attributed to the inclusion of a pure prefetch buffer (SVB metadata demonstrably has almost no effect).

Finally, STeMS–SVB does show a slight increase in coverage when compared to the idealized G/AC (an upper bound on TMS performance) due to spatial prefetches and coverage on compulsory misses: idealized G/AC attains an 8.3% coverage increase while STeMS–SVB reaches 11.8%. However, the speedups remain roughly equal: 4.5% for idealized G/AC and 4% for STeMS–SVB.

6. CONCLUSIONS

This report has described the first independent implementation of the STeMS prefetcher [2] and clarified a number of important details in its most complicated areas, including pseudocode to aid in future research.

A novel branch-guided technique using basic-blocks for spatio-temporal prefetching is introduced which is likely more applicable to a wider variety of workloads because it can adapt to many program structures.

Results show that the basic-block technique is very inexpensive, breaking the trend of heavyweight spatio-temporal and is highly accurate, with 69.6% accuracy compared to the near-50% accuracy of current state-of-the-art prefetchers. However, this accuracy has yet to cause a large final speedup for basic-block prefetching.

Surprisingly, the results also demonstrate that the complicated streaming engine of TMS is unnecessary for STeMS, while its prefetch buffer causes a majority of its performance gains.

Looking to the future, further work on basic-block prefetching with more aggressive prefetch strategies is planned to transform the highly accurate nature of the technique into a similarly large overall speedup.

7. REFERENCES

[1] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood, "DBMSs on a modern processor: Where does time go?," In Proceedings of the International Conference on Very Large Data Bases, 1999.

[2] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," In Proceedings of the 36th International Symposium on Computer Architecture, 2009.

[3] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial Memory Streaming," In Proceedings of the 33rd Annual International Symposium on Computer Architecture, 2006.

[4] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical Off-chip Meta-data for Temporal Memory Streaming," In 2009 IEEE 15th International Symposium on High Performance Computer Architecture, 2009.

[5] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal Streaming of Shared Memory," In Proceedings of the 32nd Annual International Symposium on Computer Architecture, 2005.

[6] K. J. Nesbit, and J. E. Smith, "Data cache prefetching using a global history buffer," In IEEE Micro, 2005.

[7] A. Jain, and C. Lin, "Linearizing Irregular Memory Accesses for Improved Correlated Prefetching," In Proceedings of The 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

[8] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors," In Proceedings of The 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.

[9] Texas A&M University, "The 2nd Cache Replacement Championship," Texas A&M University, 2017. [Online]. Available: http://crc2.ece.tamu.edu/.

[10] Georgia Tech, "The 2nd Data Prefetching Championship," Georgia Tech, 2015. [Online]. Available: http://comparch-conf.gatech.edu/dpc2/.

[12] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation – a Pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites," VSSAD Technical Report, Technical Report, 2007.

[13] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2003.

[14] P. Michaud, "Best-Offset Hardware Prefetching," In Proceedings of the International Symposium on High-Performance Computer Architecture, 2016.