2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Loop-Aware Memory Prefetching Using Code Block Working Sets

Adi Fuchs† Shie Mannor† Uri Weiser† Yoav Etsion†§
†*Electrical Engineering* §*Computer Science*
Technion - Israel Institute Of Technology
{*adi@tx, shie@ee, uri.weiser@ee, yetsion@tce*}*.technion.ac.il*

***Abstract*—Memory prefetchers predict streams of memory addresses that are likely to be accessed by recurring invocations of a static instruction. They identify an access pattern and prefetch the data that is expected to be accessed by pending invocations of the said instruction. A stream, or a prefetch context, is thus typically composed of a trigger instruction and an access pattern. Recurring code blocks, such as loop iterations may, however, include multiple memory instructions. Accurate data prefetching for recurring code blocks thus requires tight coordination across multiple prefetch contexts.**

This paper presents the code block working set (CBWS) prefetcher, which captures the working set of complete loop iterations using a single context. The prefetcher is based on the observation that code block working sets are highly interdependent across tight loop iterations. Using automated annotation of tight loops, the prefetcher tracks and predicts the working sets of complete loop iterations.

The proposed CBWS prefetcher is evaluated using a set of benchmarks from the SPEC CPU2006, PARSEC, SPLASH and Parboil suites. Our evaluation shows that the CBWS prefetcher improves the performance of existing prefetchers when dealing with tight loops. For example, we show that the integration of the CBWS prefetcher with the state-of-the-art spatial memory streaming (SMS) prefetcher achieves an average speedup of 1.16× (up to 4×), compared to the standalone SMS prefetcher.

I. Introduction

Memory prefetching is extensively used in modern processors to ameliorate the effect of memory latency on performance. Memory prefetchers decompose a program's memory accesses into multiple data streams, which are identified by the program counter (PC), a spatial pattern, or a combination of both. The number of concurrent streams that a processor can track is determined by the availability of hardware resources [18].

Computer programs, on the other hand, incorporate the semantics of well-defined programming language constructs such as procedures and loops. But these semantics are not plainly expressed as PC or spatial cues.

Existing prefetchers, therefore, do not associate a compound code segment with the entire group of memory addresses it requires, namely its working sets. In particular, they ignore opportunities to target repetitive code in programmer-defined tight loops, which consume a considerable fraction of a program's runtime execution.

In this paper we introduce the concept of a *code block working set* (CBWS), which is the ordered vector of cache lines accessed by an invocation of a loop iteration. A CBWS provides the complete memory access trace of a loop iteration. We further present the *CBWS* prefetcher that uses compiler annotations of loop iterations to track and predict the CBWSs of pending loop iterations.

Predicting an iteration's entire working set is made possible with the formulation of *CBWS differentials*, which are the result of an element-wise subtraction of two CBWS vectors generated by separate loop iterations. The differentials thus represent the evolution of a loop's cache footprint across iterations.

The CBWS prefetcher uses CBWS differentials to prefetch an iteration's entire working set ahead of time (skipping addresses that are already cached). The compiler annotations of loop iterations allow the prefetcher to aggressively track and prefetch only memory accesses within a loop. In contrast, existing prefetchers that target similar address patterns (e.g., GHB PC/DC [22]) track memory operations individually and do not bind instructions into groups. As a result, they cannot tune prefetch policies (e.g., prefetch on hits or misses, prefetch depth, track all L1 accesses or only misses) for specific instruction groups.

The compiler annotations tag loop iterations, and each code block is given a unique static identifier. The process of annotating and identifying code blocks is performed using a dedicated compiler pass implemented in the LLVM framework [16]. The proposed prefetcher is designed as an add-on component and is integrated into the *spatial memory streaming* (SMS) prefetcher [29].

The key contributions of this paper are as follows:

- We formalize the concept of *code block working sets* (CBWSs) as the set of disparate addresses that are accessed by a loop iteration.
- We quantify the tight interdependence across CBWSs in tight loops. Using CBWS differentials, we demonstrate the predictability of CBWSs.
- We highlight the benefits of using compiler hints to tune the aggressiveness of the prefetcher based on high-level programming constructs.
- We propose a prefetcher that operates at the CBWS level. The proposed prefetcher is designed as an add-on that falls back on the SMS prefetcher when a CBWS prediction has a low confidence.

We evaluate the CBWS prefetcher using the gem5

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.27

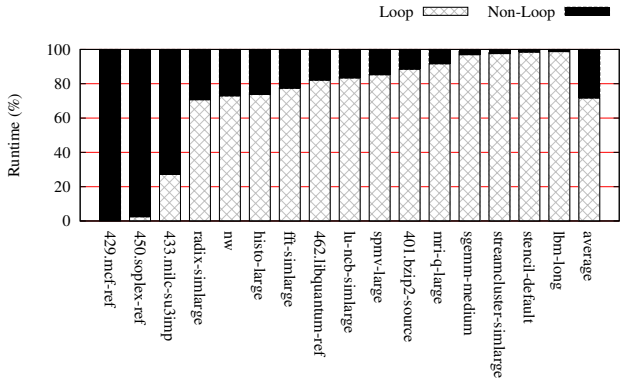

Figure 1. Fraction of runtime spent executing tight, innermost loops for the memory intensive benchmarks used in this paper.

simulator [3] and memory intensive benchmarks from SPEC2006 [1], PARSEC [2], SPLASH [33], [25] and Parboil [31]. The performance is compared to other well-known prefetchers such as stride [8], [14], GHB G/DC + GHB PC/DC [22] and SMS [29].

The proposed scheme requires less than 1KB of storage, which is small in comparison to the other evaluated schemes. The evaluation shows that the CBWS prefetcher speeds up a high-performance, 4-way out-of-order core by an average of 1.16× (up to 4×) for a collection of both memory-intensive and regular workloads.

The rest of the paper is organized as follows: Section II presents the concept of code block working sets (CBWS) and their correlations. Section III discusses related work. Section IV presents the prediction scheme using CBWS differentials, followed by a description of the prefetcher architecture in Section V. Finally, we describe our experimental methodology in Section VI, the evaluation of the CBWS prefetcher in Section VII and conclude in Section VIII.

II. The Case for Code Block Working Sets

The execution of tight loops tends to dominate application runtime. For example, Figure 1 shows the fraction of time spent in tight, innermost loops for the most memory-intensive benchmarks (out of ∼30 benchmarks) we evaluated in this paper. The figure shows that, on average, over 70% of the benchmarks' runtime is spent executing tight loops. Optimizing the micro-architecture for tight loops thus carries substantial performance benefits.

Tight loop iterations are typically composed of a small number of static instructions, which suggests that their working set is limited in size and can be tracked in hardware. As a result, we argue that code blocks can be associated with their working sets, and that hardware can reason about the predictability of the working sets.

The rest of this section demonstrates the tight correlation between CBWSs, i.e., the working sets accessed by successive iterations. Given that two CBWSs are correlated through a pattern of stride vectors (as described below), we

```
#define IDX(nx,ny,x,y,z) (x+nx*(y+ny*z))
...
void cpu_stencil
(float c0,float c1,float *A0,float *A,int nx,int ny,int nz)
{
  for(int i=1;i<nx-1;i++)
  {
    for(int j=1;j<ny-1;j++)
    {
      for(int k=1;k<nz-1;k++)
      {
        BLOCK_BEGIN(0);
        A[IDX(nx, ny, i, j, k)] =
            c1 * (A0[IDX(nx, ny, i, j, k + 1) ] +
                  A0[IDX(nx, ny, i, j, k - 1) ] +
                  A0[IDX(nx, ny, i, j + 1, k) ] +
                  A0[IDX(nx, ny, i, j - 1, k) ] +
                  A0[IDX(nx, ny, i + 1, j, k) ] +
                  A0[IDX(nx, ny, i - 1, j, k) ]) -
            c0 * A0[IDX(nx, ny, i, j, k)];
        BLOCK_END(0);
      }
    }
  }
}
```

Figure 2. Main loop of the Parboil Stencil benchmark.

	PC_0	PC_1	PC_2	PC_3	PC_4	PC_5	PC_6	
$CBWS_0$ = (	80,	81,	**6515**,	4467,	5499,	5483,	5491	)
$CBWS_1$ = (	80,	81,	7539,	**5491**,	**6523**,	6507,	6515	)
$CBWS_2$ = (	80,	81,	8563,	6515,	7547,	7531,	7539	)
$CBWS_3$ = (	80,	81,	9587,	7539,	8571,	8555,	8563	)
$CBWS_4$ = (	80,	81,	10611,	8563,	9595,	9579,	9587	)
$CBWS_5$ = (	80,	81,	**11635**,	9587,	10619,	10603,	10611	)
$CBWS_6$ = (	80,	81,	12659,	**10611**,	**11643**,	11627,	11635	)
$CBWS_7$ = (	80,	81,	13683,	11635,	12667,	12651,	12659	)

Figure 3. The access pattern exhibited by the Stencil code. The columns depict PC-based access streams and the rows illustrate CBWSs. The highlighted addresses indicate potential PC/DC misses.

	PC_0	PC_1	PC_2	PC_3	PC_4	PC_5	PC_6	
$CBWS_1$-$CBWS_0$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)
$CBWS_2$-$CBWS_1$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)
$CBWS_3$-$CBWS_2$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)
$CBWS_4$-$CBWS_3$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)
$CBWS_5$-$CBWS_4$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)
$CBWS_6$-$CBWS_5$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)
$CBWS_7$-$CBWS_6$ = (	0,	0,	1024,	1024,	1024,	1024,	1024	)

Figure 4. The correlated loop strides exhibited by the Stencil code, shown as CBWS differential vectors.

show that the distribution of all observed patterns is highly skewed and that only a small fraction of all observed patterns differentiate most CBWS instances. This property points to memory access regularities that, combined with compact representation of CBWSs, can be used to predict the CBWSs that will be accessed by pending loop iterations.

A. Evolution of working sets across iterations

We now turn to explore the evolution of working sets across loop iterations and the benefits provided by the CBWS prefetcher. This is demonstrated using a (simplified) code snippet taken from the main loop of the Parboil Stencil benchmark [31], which applies a Jacobi stencil operator to a 3D-grid. The code is shown in Figure 2 (including block boundary annotations that are discussed later in the paper).

The code consists of three nested loops, such that each index calculation (calculated via the *IDX* macro) depends on all three loop variables. As a result, some of the memory instructions access the same cache lines.

Figure 3 depicts the memory addresses accessed over eight iterations of the innermost loop. The figure illustrates the addresses as a matrix, whose columns represent addresses accessed by each static instruction (PC), and whose rows represent CBWSs, or the sequence of addresses accessed by each iteration. Furthermore, Figure 4 shows the access pattern as vectors of address deltas between iterations, illustrating that the access pattern is very predictable and consists of simple strides between CBWS vectors.

Interestingly, even though the access pattern can be identified by PC-based prefetchers such as GHB PC/DC [22], the prefetchers' static, conservative configuration prevents it from completely eliminating cache misses throughout the execution of the loop.

This limitation of the PC/DC prefetcher is demonstrated in Figure 3. The figure highlights the cache misses incurred by the prefetcher when it is configured to prefetch only on cache misses and to use a prefetch depth of 4 (as originally described by Nesbit and Smith [22]; note that the last two accesses in each iteration reuse previously prefetched data). Because the prefetcher is only triggered by cache misses, it must wait for one to act. And when it acts, it only fetches the next 4 predicted addresses. As a result, the prefetcher misses on every 5th memory access of the stream it prefetches. Alternatively, an aggressive configuration of the PC/DC prefetcher, one that also prefetches on cache hits and uses a larger prefetch depth, could greatly reduce the number of misses incurred by the loop. However, such a configuration would be too aggressive for other program phases, where it may pollute the caches and degrade the overall performance.

In a similar manner, region-based prefetchers, such as the *spatial memory streaming* (SMS) prefetcher [29], are limited by their design parameters. Specifically, these prefetchers use limited region sizes, whereas addresses in the 3D Stencil code may span regions that are input dependent, whose range triplicates with the dimensions of the grid.

In contrast, the CBWS prefetcher uses explicit compiler annotations to track the working sets of complete loop iterations. Inside the annotated loops, the prefetcher uses *CBWS differentials* to predict the working set required by complete iterations. As depicted in Figure 4, the differentials are vectors of address deltas computed as element-wise subtractions of two CBWS vectors. The delta-vectors align multiple streams and facilitate accurate prediction of the entire CBWS for each loop iteration. Moreover, CBWS differentials are not limited to consecutive loop iterations and, using a CBWS history buffer, can be used to predict patterns that span non-consecutive iterations.

The CBWS prefetcher is, therefore, an aggressive, pattern-based prefetcher that aligns correlated prefetch streams and issues multiple prefetch operations in lock-step. Its key benefit over PC/DC lies in its use of compiler hints to focus on tight loops, which allows it to use a more aggressive prefetch strategy. Concretely, it enables the prefetcher to track L1 hits and use larger prefetch depths.

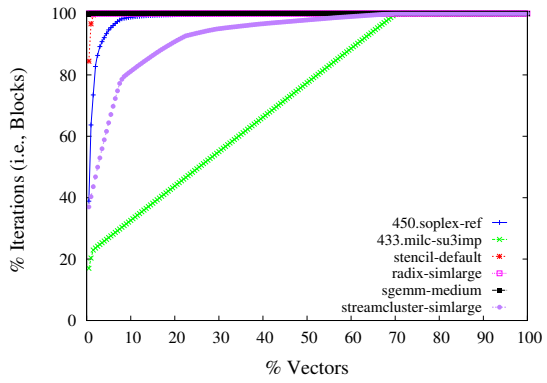

Figure 5. The correlation between the number of distinct CBWS differential vectors and the fraction of loop iterations they correlate. The figure shows that most iterations (high values of vertical axis) are predicted by a tiny fraction of the differential vectors (small values of horizontal axis).

In summary, this section demonstrates the promise of the CBWS concept, as well the benefits of using compiler hints to tune prefetching policies for tight loops. We next turn to motivate the dedicated handling of tight loops.

B. The skewed distribution for CBWS differentials

The aggregation of multiple address deltas into a single CBWS differential vector is problematic if a large number of distinct vectors is in use at any given time. In such a case, a prefetcher will require significant hardware resources to store the distinct CBWS differential vectors. Interestingly, the distribution of vectors is highly skewed and suggests that only a few vectors dominate the inter-CBWS correlation.

Figure 5 depicts this distribution as the correlation between the fraction of distinct vectors (horizontal axis) and the fraction of loop iterations they predict (vertical axis). For brevity, the figure only shows the distribution for a subset of the benchmarks. The figure shows that the vast majority of loop iterations are served by a tiny fraction of the differential vectors. In the SPEC Soplex benchmark, for example, about 90% of the loop iterations are differentiated by only 5% of the distinct CBWS differential vectors. A loop-aware prefetcher can leverage this skewed distribution to accurately predict CBWSs in pending loop iterations. In the following sections we will present and evaluate the design of a CBWS-based prefetcher.

III. Related Work

The prominence of memory latency as a performance bottleneck has motivated numerous studies on data prefetching.

A. Prefetching streams of addresses

Probably the most common hardware prefetchers are the block [14] and stride prefetchers [8]. Block prefetchers target blocks that consecutively follow the recently accessed address, whereas stride prefetchers target accesses to blocks that are correlated by fixed strides.

Several studies demonstrate the importance of prefetching multiple streams concurrently. Palacharla and Kessler [23] show that it is common for programs to access several arrays in a single loop iteration. They propose to use stream buffers and tables to detect concurrently accessed streams without using the PC as an identifier. Similarly, Iacobovici et al. [10] target coordinated strided streams. Their findings highlight the non-uniform behavior of stride groups (shown in this work as well). Joseph and Grunwald [13] target cache prefetching using a Markovian model, which is a probabilistic model that correlates consecutive pairs memory addresses. Our study shows that associating address sets with code blocks improves accuracy and enables a longer prefetching window.

Nesbit and Smith propose the *global history buffer* (GHB) [22] to improve the accuracy of stride, Markov, and correlated delta schemes. GHB stores cache misses in a buffer. On a memory access, the buffer is indexed by a key (e.g., PC) and, using previous miss patterns, may initiate a memory prefetch. The global history buffer was later extended by Nesbit et al. [21] to support adaptive correlation prefetching within concentration zones. This scheme detects program phases and dynamically adapts itself. A more recent design by Sharif et al. [27] integrates both global and local history buffers to improve prefetch accuracy.

Mutlu et al. [19] propose the address-value delta (AVD) predictor. This prefetcher manages pointer-based access patterns as streams by maintaining a table of data-PC deltas to capture pointer based behavior in iterative code.

Somogyi et al. [29] propose *spatial memory streaming* (SMS). This scheme groups cache lines that reside in a single memory segment and thus likely belong to the same data structure. This scheme groups lines by tracking cache invalidations and evictions to mark spatial regions. *Spatio temporal memory streaming* (STeMS) [28] extends SMS by adding a temporal tracing scheme to prevent cache pollution caused by untimely prefetch transactions. STeMS imposes a fairly large storage overhead ($\sim$640KB).

The AMPM prefetcher proposed by Ishii et al. [11] combines concentration zones with cache line bitmaps in order to identify spatial streams and predict future strides within zones. Importantly, the prefetcher is not PC-based and only targets global streaming patterns. Consequently, when applied to loops, it first identifies patterns inside an iteration and, only if such patterns are not found, may identify patterns across iterations.

The abovementioned schemes have no notion of code blocks, but rather track independent memory accesses. Decisions are made at instruction or stream level. In contrast, our proposed scheme operates at the code block level and maintains code block access history, thus preserving (partial) program context.

B. Using software and compiler hints

Several studies explore the use of compiler and software cues to guide hardware prefetchers. Wang et al. [32] and Ebrahimi et al.[7] propose using compiler hints to explicitly express known access patterns across independent memory references, such as recursion, linked structures or pointer indirection. Alternatively, other studies use coarse-grained prefetching techniques to mitigate the loss of cache state on thread switches [4], [15], in multiprogrammed/virtualized workloads [35] and in application-level prioritization [6].

Our scheme, in contrast, does not rely on deep compiler analysis of memory access patterns and supports fine-grained data sets by grouping instructions into code blocks.

C. Other prefetchers

Another software-oriented approach was proposed by Reinman et al. [26]. This scheme exploits the accuracy of modern branch predictors for directed prefetching of basic blocks to improve the performance of the instruction fetch. Similarly, Panda et al. presented B-Fetch [24], a scheme for in-order processors that uses the branch predictor and CPU registers to predict the effective addresses of memory instructions within basic blocks. Although guided by basic block invocations, this scheme analyzes memory instructions individually given a vector of previous processor states.

A different take on prefetching was proposed by Mutlu et al. [20]. Runahead execution avoids stalling the pipeline on long latency memory operations. Instead, the pipeline continues processing using speculative values and issues pending memory operations, thereby hiding their latencies.

Finally, a recent extensive study on prefetching techniques conducted by Lee et al. [17] concludes that the best prefetching methods accurately prefetch short, concurrently executing streams. Indeed, the proposed CBWS prefetcher follows this recommendation.

IV. Working Set Prediction Using Differentials

A prefetcher that operates at code block granularity requires hardware support to efficiently capture the changes in working sets across multiple executions of the same block. In previous sections, we presented CBWS differentials at high-level. This section presents the formal definition of CBWS differentials and the differential prediction algorithm.

A. Code block boundaries

The CBWS-based prefetcher employs software cues to group memory accesses. Special instructions tag code blocks and assign a static identifier to each block.

There are several benefits for compile time identification of loops and for embedding loop annotations in the code. Most importantly, it preserves the original loop semantics in the presence of compiler optimizations such as loop unrolling or loop splitting. This enables the CBWS prefetcher to remain agnostic to these compiler optimizations, which

Instruction	*Line*$_{\#}$	*CBWS*$_0$	*CBWS*$_1$	$\Delta_{0,1}$
BLOCK_BEGIN	-	$\emptyset$	$\emptyset$	$\emptyset$
LD 4800	120	120	$\emptyset$	$\emptyset$
LD 4804	120	120	$\emptyset$	$\emptyset$
LD FE50	3F9	120,3F9	$\emptyset$	$\emptyset$
LD 481C	120	120,3F9	$\emptyset$	$\emptyset$
ST FE50	3F9	120,3F9	$\emptyset$	$\emptyset$
LD 7FE0	1FF	120,3F9,1FF	$\emptyset$	$\emptyset$
ST 7FE0	1FF	120,3F9,1FF	$\emptyset$	$\emptyset$
BLOCK_END	-	120,3F9,1FF	$\emptyset$	$\emptyset$
BLOCK_BEGIN	-	”	$\emptyset$	$\emptyset$
LD 4900	124	”	124	4
LD 4904	124	”	124	4
LD FC50	3F1	”	124,3F1	4,-8
LD 491C	124	”	124,3F1	4,-8
ST 7FE0	1FF	”	124,3F1,1FF	4,-8,0
BLOCK_BEGIN	-	”	124,3F1,1FF	4,-8,0

Table I
EXAMPLE OF CBWS CONSTRUCTION AND DIFFERENTIAL CALCULATION (CACHE LINE SIZE IS 64B).

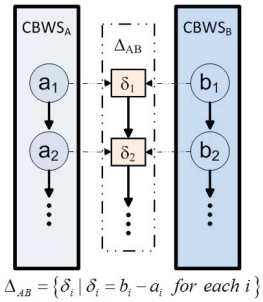

Figure 6. A CBWS differential Δ_{AB} is the stride vector computed by CBWS vectors $CBWS_A$ and $CBWS_B$, respectively.

were shown to affect hardware prefetchers [11]. In addition, compile-time annotations eliminate the need to dynamically identify code blocks (e.g., by tracing backward-branches) and thereby simplify the prefetcher design.

Two new instructions are added to the ISA to express block boundaries: $BLOCK\_BEGIN$, and $BLOCK\_END$. To limit the amount of state per code block, we limit code block tracing to include up to 16 distinct cache lines. Our experiments show that 16 lines are sufficient to map the entire working set of over 98% of the dynamic code blocks in the benchmarks tested.

B. CBWS and CBWS differentials

Each code block instance has a distinct code block working set (CBWS), which is a set of ordered cache blocks. CBWS differentials are vectors that consist of the element-wise subtraction of two CBWSs. Table I illustrates the creation of CBWSs and their differential from a trace of 2 block instances.

Formally, a CBWS is defined as follows: Let the memory accesses generated by code block *A* be defined as a sequence of tuples (a, t), where each tuple describes a memory access to cache block a at time t. The CBWS of code block *A* is a time-ordered set of unique line addresses:

$$CBWS_A = \{a_i \mid \forall (a_i, t_i) : i < j \Rightarrow t_i < t_j \,,\ a_i \neq a_j\}\,. \quad (1)$$

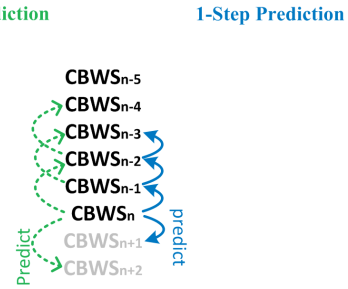

Figure 7. Prefetching using 1-step and 2-step differentials.

Figure 6 illustrates the *differential* of vectors $CBWS_A$ and $CBWS_B$ as the vector of their differences $\Delta_{A,B}$. This vector is formally defined as:

$$\Delta_{A,B} = \{\delta_i \mid \forall i : \delta_i = b_i - a_i,\ \ b_i \in CBWS_B,\ \ a_i \in CBWS_A\}\,. \quad (2)$$

Importantly, even though the differentiated working sets are generated by the same code block, they may be of different sizes (a result of branch divergence). In such cases, the two CBWSs are aligned and the size of the differential is defined by the smaller CBWS.

C. CBWS prediction

The prediction algorithm dynamically constructs the CBWS during the execution of the code block, and a prediction is made when the execution of a block completes.

Prefetching the CBWS only when the block begins executing poses a timing constraint. To mitigate this constraint, we support *multi-step differentials*. Instead of a single $\Delta_{n,n-1}$ differential, the prefetcher stores a history of $\Delta_{n,n-1}, \Delta_{n,n-2}, ..., \Delta_{n,n-k}$ differentials for the last k code block instances. This enables the prediction of blocks executed farther into the future. Figure 7 illustrates the prediction of either one or two CBWSs into the future. Specifically, we have found that a history of 4 differentials provides sufficient performance. In addition, the history differentials are generated progressively with each memory access, so the predictor requires only 4 adders.

Algorithm 1 outlines the operations performed when a BLOCK_BEGIN / BLOCK_END instruction is encountered and when a memory access is issued inside the code block:

- The *BLOCK_BEGIN* instruction clears the tracing of CBWS in preparation for the new code block. This operations is detailed in Figure 9 below.
- When a memory access is issued inside a code block, the address of the cache block is pushed to the CBWS that is being generated. In parallel, the CBWS differential history table is updated on each memory access by subtracting the correlated entries in the previous N CBWSs from the current address. This operation is detailed in Figure 10.
- The *BLOCK_END* instruction updates the CBWS *differentials* buffer and issues a prefetch for the next CBWSs. This operation is detailed in Figure 11 below.

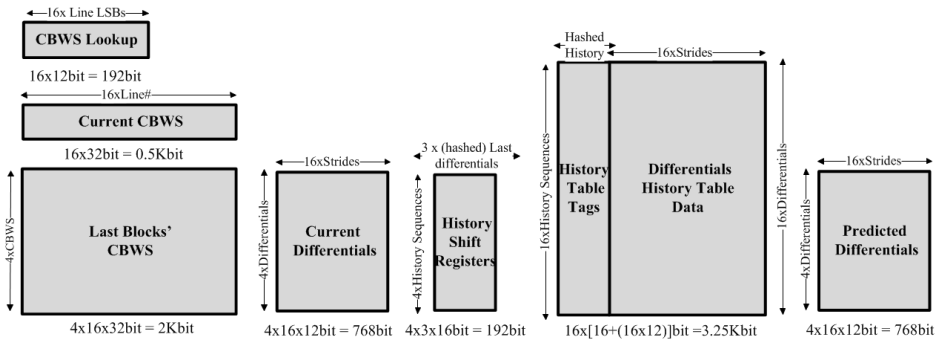

Figure 8. The hardware structures comprising the CBWS prefetcher.

Algorithm 1 Differentials prediction.

```
function BLOCK_BEGIN(block_id)
    lines_set ← ∅
    idx ← 0
    curr_cbws ← []
    for i ∈ {1..max_step} do
        curr_diff[i] ← []
    end for
end function

function MEMORY_ACCESS(line_#)
    if (line_# ∉ lines_set) then
                                  ▷ Push address to the current CBWS
        lines_set ∪ {line_#}
        curr_cbws[idx] ← line_#
                        ▷ Generate an entry for the differential vectors
        for i ∈ {1..max_step} do
            stride ← line_# − last_cbws[i][idx]
            curr_diff[i][idx] ← stride
        end for
        idx ← idx + 1
    end if
end function

function BLOCK_END(block_id)
                                  ▷ update tracing + prediction DB
    for i ∈ {1..max_step} do
        diff_table[hash(diff_history[i])] ← curr_diff[i]
        diff_history[i].enqueue(curr_diff[i])
    end for
    for i ∈ {1..max_step − 1} do
        last_cbws[i] ← last_cbws[i + 1]
    end for
    last_cbws[max_step] ← curr_cbws
                                  ▷ prefetch using prediction DB
    for i ∈ {1..max_step} − 1 do
        pred_diff ← diff_table[hash(diff_history[i])]
        for j ∈ {1..size(pred_diff)} do
            prefetch{last_cbws[0][j] + predict_diff[j]}
        end for
    end for
end function
```

In summary, the CBWS predictor uses a simple prediction logic to prefetch complete CBWSs. The predictor is agnostic to static loop optimizations through the use of compiler-generated annotations of tight loops.

V. Architecture

In this section we describe the hardware components of the prefetcher and their operational flow during memory accesses, code block completion and prediction.

A. Hardware Structures

The hardware components are categorized according to the phases in the CBWS prediction algorithm: (1) Constructing the working set for the currently executing block and computing its CBWS differential; (2) Storing the computed differentials in the history table; and (3) Predicting the next CBWSs. The components and their sizes are shown in Figure 8 and require, in total, less than 1KB of storage.

The *current CBWS* buffer is a FIFO that stores the lower 32 bits of the line addresses accessed by the currently executing code block and constructs the current CBWS.

Four predecessor CBWSs are stored in the *last blocks CBWS* buffer. These CBWSs are used to compute the differentials for the current CBWS and its four predecessors. Maintaining a history of a few CBWSs has two major benefits: (1) they can be used to uncover CBWS correlations across non-consecutive code blocks (e.g., instances #1 and #3 might be correlated, while instances #2 and #3 are not); and (2) they can be used to predict CBWSs into the more distant future (as described in Figure 7).

The differential vectors for the current CBWS and its four predecessors are stored in the *current differentials* buffer. Since address strides are typically small, 16 bits are sufficient to represent each element in a differential vector. The computation of the differentials is incremental, as discussed in Section V-B and shown in Figure 10.

History shift registers record the differentials computed using the four predecessor CBWSs. The shift registers implement a functionality similar to a branch history register (BHR) [34], except that they shift CBWS differentials rather than branch outcomes. Each register stores a 3-deep differential history, and differentials are represented using 12 bits extracted from the original differential (bit-select hashing).

The *differential history table* has 16 entries. It is fully-associative and uses a random eviction policy. The table is indexed by the history shift registers, whose 48 bits are xored to provide a 16-bit tag.

Finally, every prediction extracts four candidate differentials into the *predicted differentials* buffer. The differentials are added to the current CBWS to predict future CBWSs.

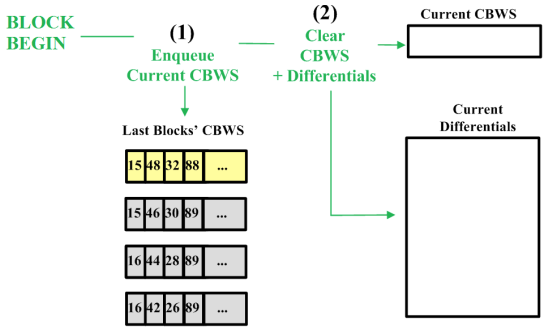

Figure 9. Operation on code block execution initialization.

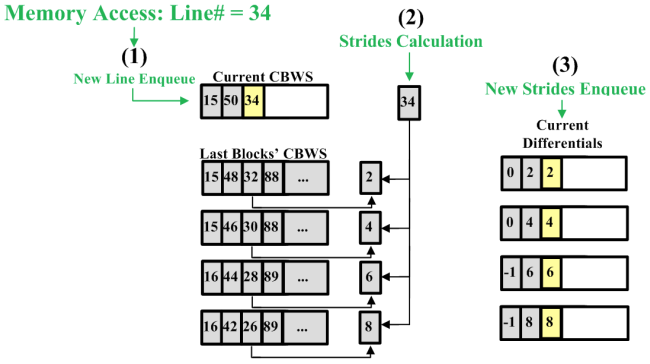

Figure 10. Operation during a memory access.

B. Execution Flow

Figures 9, 10 and 11 outline the operational flow during *code block execution initialization*, *memory accesses* and *code block execution completion*, respectively.

When a code block begins executing (*BLOCK_BEGIN* event), as depicted in Figure 9, the *current CBWS* and *last block CBWSs* buffers are cleared.

Figure 10 depicts the operational flow on *memory accesses*. The address of the accessed cache line is added to the *current CBWS* buffer, if it is not yet present there. In parallel, the address is compared against the corresponding entries in the predecessor CBWSs (*last block CBWSs* buffer), and the resulting strides are pushed to the *current differentials* buffer.

Figure 11 depicts the flow when a block completes its execution (*BLOCK_END* event). The *differentials history table* is first updated with the new differentials using the *history shift registers* as tags (#1 and #2 in the figure). Next, the prefetcher generates a prediction by looking up the *history shift registers* in the *differentials history table* (#3 in the figure). The prefetcher then loads the selected differentials into the predicted differentials buffer (the example depicts 3 hits and a miss) and uses vector addition to predict future CBWSs (#4 in the figure).

The proposed CBWS prefetcher must examine the memory accesses in program order. In an out-of-order pipeline, the prefetcher obtains the address sequence from the in-order commit stage (possibly as an index into the load-store queue, if the reorder buffer does not store memory addresses).

Parameter	*Value*	*Parameter*	*Value*
CPU		Branch Predictor	
clock rate	2.0 GHz	BP Type	Tournament
OoO Width	4	BP Entries	4K
ROB entries	128	BP Tag Size	16-bit
Functional units	6	BP History Size	11-bit
LDQ entries	32		
STQ entries	32		
Physical page size	4KB		
L1 D-Cache		L1 I-Cache	
Total size	32KB	Total size	32KB
Line size	64 Bytes	Line size	64 Bytes
Associativity	4-way LRU	Associativity	2-way LRU
Latency	2 cycles	Latency	2 cycles
Total MSHRs	4	Total MSHRs	4
L2 Cache		Physical Memory	
Inclusion policy	Inclusive	Total size	4GB
Total size	2MB	Latency	300 cycles
Line size	64 Bytes	GHB G/DC Prefetcher	
Associativity	8-way LRU	Table entries	256 fully assoc.
Latency	30 cycles	History Length	3
Total MSHRs	32	Prefetch Degree	3
Stride Prefetcher		Overall Size	2.25KB
Table entries	256 fully assoc.	GHB PC/DC Prefetcher	
Overall Size	2.25KB	Table entries	256 fully assoc.
CBWS Prefetcher		History Length	3
History Table Repl.	Random	Prefetch Degree	3
Max. Vector Members	16	Overall Size	3.75KB
Stride Size	16-bits	SMS Prefetcher	
# Last CWBS Stored	4	AGT Table	32 Entries
Differential Table	16 Entries	Filter Table	32 Entries
CBWS Lookup size	12 Line LSBs	History Table	512 Entries
Overall Size	1KB	Region Size	2KB
		Overall Size	5KB

Table II
SIMULATION PARAMETERS.

In summary, the implementation aggressively reduces the number of address and offset bits to minimize the storage requirements. As a result, the design requires less than 1KB of storage, yet the compact representation of execution history is sufficient to provide accurate predictions.

VI. METHODOLOGY

We have implemented the CBWS prefetcher using the gem5 simulator [3], which was configured with a single out-of-order core connected to a 2-level cache hierarchy. The prefetchers were configured to fetch data to the L2 cache. Table II lists the full architectural parameters.

Table III compares the estimated storage overheads of the evaluated prefetchers (the CBWS overheads are depicted in Figure 8). For the SMS prefetcher [29], we have estimated the table sizes using 12-bit strides, 48-bit PCs, and 5-bit region offset + 36-bit region tag + 16-bit region pattern.

The design was evaluated using 30 benchmarks from the SPEC CPU 2006 [1], PARSEC [2], [9], SPLASH [33], [25], Rodinia [5] and Parboil [31] suites. Benchmarks were selected on the basis of their memory intensity, omitting benchmarks with a low number of misses per kilo-instructions (MPKI) that do not benefit from any prefetching (and a few that did not compile properly). We partition the full set of benchmarks into two groups: the 15 benchmarks with the highest MPKI, referred to as the *memory-intensive* (MI) group, and the 15 low-MPKI group (Table IV lists the benchmarks in the memory-intensive group). For brevity, we

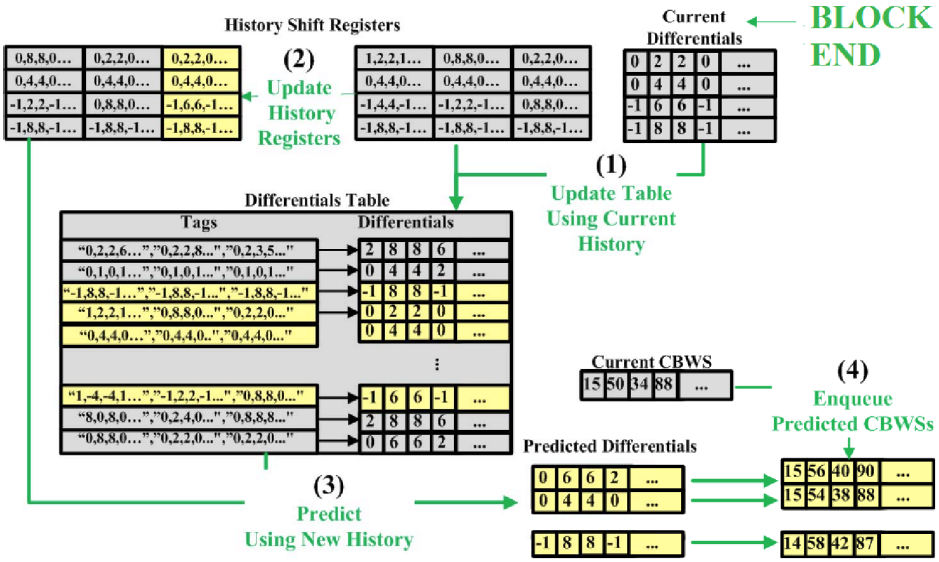

Figure 11. Operation on code block execution termination.

Prefetcher	Storage		
Stride	$18.4Kbit \simeq 2.25KB$	=	$(PC + 2 \times stride) \times 256 = (48 + 2 \times 12) \times 256$
GHB G/DC	$18.4Kbit \simeq 2.25KB$	=	$(3 \times history\_strides + 3 \times prefetch\_strides) \times 256 = (6 \times 12) * 256$
GHB PC/DC	$30.7Kbit \simeq 3.75KB$	=	$(GHB\_G/DC) + PC \times 256 (6 \times 12 + 48) * 256$
SMS	$20.7Kbit \simeq 5KB$	=	$AGT_{size} + Filter_{size} + PHT_{size} = (offset + PC + tag) \times 32 + (offset + PC + tag + pattern) \times 32 + (pattern + PC + offset) \times 512 = (36 + 48 + 5) \times 32 + (36 + 48 + 5 + 16) \times 32 + (16 + 48 + 5) \times 512 = 2848 + 3360 + 35328$

Table III
COMPARISON OF HARDWARE STORAGE REQUIREMENTS FOR THE DIFFERENT PREFETCHERS.

Benchmark name	*Suite*	*Dataset*
MCF	SPEC2006	ref
Soplex	SPEC2006	ref
Libquantum	SPEC2006	ref
MILC	SPEC2006	ref
Bzip2	SPEC2006	ref
MRI-Q	Parboil	large
Histo	Parboil	large
Stencil	Parboil	default
SGEMM	Parboil	medium
NW	Rodinia	default
LBM	Parboil	large
LU_NCB	PARSEC-SPLASH	simlarge
FFT	PARSEC-SPLASH	simlarge
Radix	PARSEC-SPLASH	simlarge
Streamcluster	PARSEC	simlarge

Table IV
LIST OF MEMORY-INTENSIVE BENCHMARKS.

only show detailed results for the memory-intensive benchmarks, alongside the average results for all benchmarks. Only Figure 14, which depicts the ultimate IPC speedups achieved by the evaluated prefetchers, details the results for all benchmarks.

We have extended LLVM [16] to automatically annotate code blocks in innermost tight loops. The benchmarks were executed for 1 billion (10^9) instructions, beginning in each benchmark's region-of-interest (ROI) to omit any initialization phases. For the SPEC2006 benchmarks, we fast forwarded our simulation beyond the initialization phases. The number of forwarded instructions was based on running memory statistics [12].

VII. EVALUATION

We compare the CBWS prefetcher with the *Stride* [8], [14], *GHB G/DC* (global delta-correlation) [22], *GHB PC/DC* (PC directed delta-correlation) and spatial memory streaming (SMS) [29] prefetchers, as well as a system without prefetching. The SMS prefetcher serves as a baseline for our performance comparison, as it was the best performing existing prefetcher. Importantly, to demonstrate the benefits of CBWS over a stride prefetcher, the latter was configured to support an unrealistic number of 256 streams [18].

We evaluate the CBWS prefetcher both as a standalone prefetcher and as an add-on to an existing prefetcher:

- *CBWS*: Standalone CBWS prefetcher. In this mode, prefetch operations are issued only if there is a hit in the CBWS history table. On a miss, no prefetch is issued.
- *CBWS+SMS*: Using CBWS as an add-on for the SMS prefetcher (*integrated policy*) to optimize performance of tight loops. The *CBWS* prefetcher issues a prefetch only if the current access pattern hits in the history table. Otherwise, the SMS prefetcher issues the prefetch.

A. Misses Per 1K Instructions

Figure 12 shows the number of misses per kilo-instructions (MPKI) among all prefetchers (using a 2MB L2 cache, as described in Section VI). The figure shows

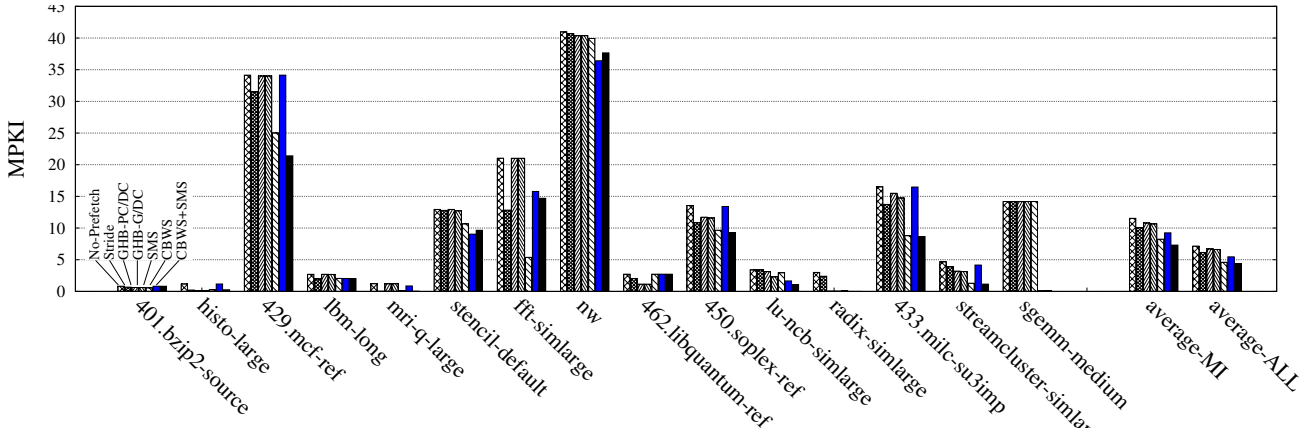

Figure 12. Misses per kilo-instructions (MPKI) in the last-level cache (lower is better).

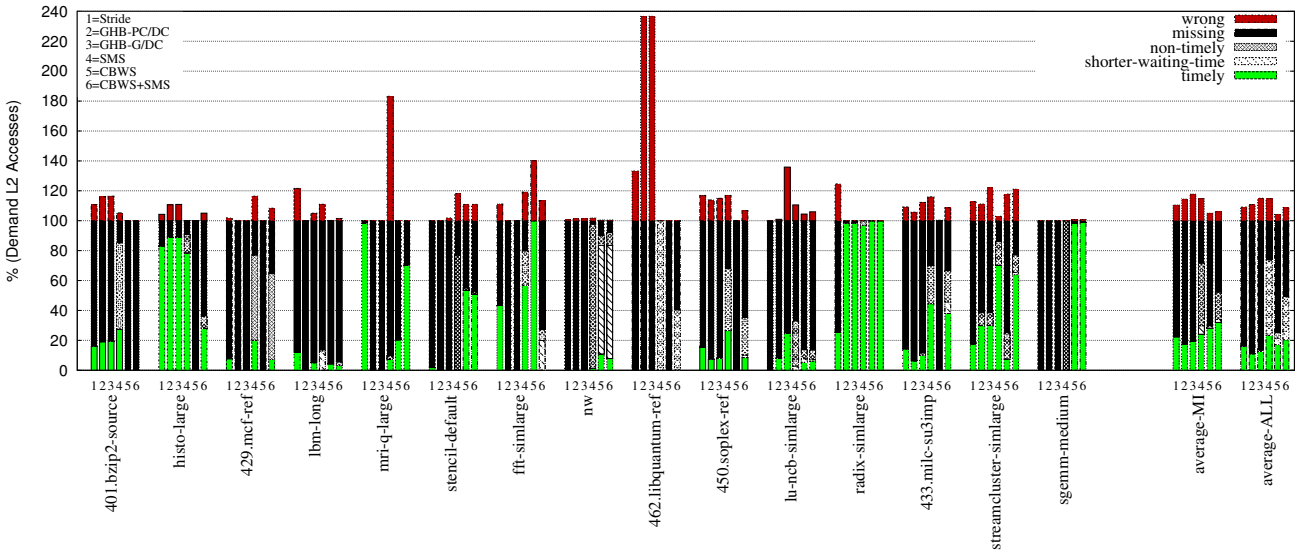

Figure 13. Timeliness and Accuracy for the competing prefetchers.

that the integrated *CBWS+SMS* policy delivers better (lower) MPKI than all other policies for all benchmarks, except for *libquantum* and *fft* (and ties with *SMS* for *bzip2*). The figure also shows that the standalone *CBWS* scheme achieves low MPKI for benchmarks with regular access patterns, but is outperformed by *SMS* in vectorized stream benchmarks such as *fft* and *streamcluster*. We have found that several segments in *fft* and *streamcluster* have a large number of distinct differential vectors. As a result, the history table is too small to represent a meaningful CBWS differential history and the fraction of hits in the table is low. This behavior highlights the benefit of the *CBWS+SMS* scheme, which falls back to *SMS* when no CBWS prediction can be made.

Furthermore, the figure shows that the CBWS schemes effectively eliminate misses in block structured benchmarks such as *sgemm* and *radix*. This results from a combination of highly skewed differential distributions and a timely pattern that fits into the history recorded by the history table.

The failure to reduce MPKI in *soplex* demonstrates that a skewed distribution of differentials, as shown in Figure 5, is not always sufficient for successful CBWS prediction. Specifically, the code blocks in *soplex* consist of loops that include many branches. The branch divergence in loop iterations results in access patterns that are hard to predict.

Finally, we see that, on average, the CBWS+SMS prefetcher delivers the lowest MPKI. In addition, we see that the standalone CBWS prefetcher delivers higher MPKI than the SMS prefetcher. As discussed above, This is due to the limited size of the history table in the standalone CBWS. We conclude that hybrid CBWS prefetcher that falls back to the SMS policy (when it cannot issue a prediction) combines the best of both worlds.

B. Timeliness and Accuracy

The timeliness and accuracy metrics [30] provide a deeper understanding of prefetcher behavior. The timeliness metric

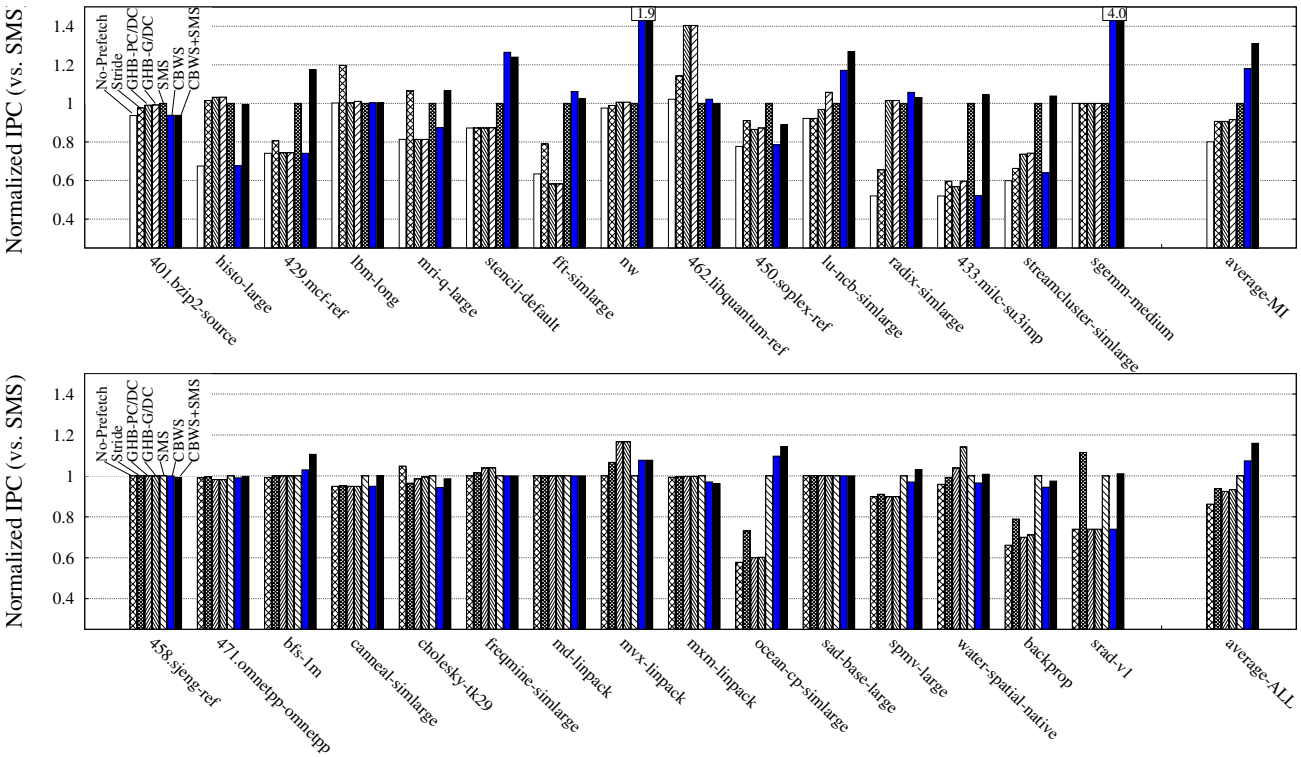

Figure 14. Performance comparison of the different prefetchers, presented as IPC normalized to SMS (higher is better).

quantifies the ability of a prefetcher to fetch addresses ahead of time. The accuracy metric quantifies the correctness of predicted addresses, as wrongly predicted addresses consume excessive bandwidth and waste cache space.

We distinguish between 5 timeliness/accuracy scenarios:

- *wrong*: A line was prefetched but was never accessed (i.e., wrong prediction).
- *missing*: The line is missing from the cache, e.g., either because a prefetch operation was never issued for the line or because the prefetch operation was premature and line was evicted before it was accessed.
- *non-timely*: The line was identified by the prefetcher, but a prefetch operation was not yet issued.
- *shorter-waiting-time*: A prefetch operation was issued but did not complete before the demand access. In this case, the prefetch operation reduced the effective memory latency for the memory operation (but did not eliminate it completely).
- *timely*: The prefetch operation completed before the demand access and a cache miss was avoided.

Figure 13 presents the timeliness and accuracy analysis. It is scaled to the relative percentile of demand fetches, such that *wrong* accesses are accounted as beyond 100%.

The figure shows that, on average, the *CBWS* scheme achieves the best accuracy, as *wrong* accesses average to 5% of all demand accesses in the group of memory-intensive benchmarks (marked *average-MI*) and to 4% of demand accesses in all benchmarks (*average-ALL*). Moreover, the *CBWS* prefetcher delivers *timely* prefetches for 28% of demand accesses in the memory-intensive benchmarks (18% over all benchmarks).

When comparing the *SMS* scheme to the *CBWS+SMS* scheme, we see that timeliness improves for the memory-intensive benchmarks as the fraction of timely accesses increases from 24% to 31%, and the fraction of *shorter-waiting-time* accesses decreases from 6% to 2% (the remainder 4% accesses may be accounted as either *timely* or *non-timely*). Moreover, the integrated prefetcher improves accuracy significantly as the number of *wrong* predictions drops from 14% to 6% (an average reduction ratio of 56%).

The results show that the integration of differential CBWS prefetching significantly improves prefetching accuracy and timeliness, regardless of benchmarks' memory intensity.

C. Speedup

Figure 14 shows the performance speedup achieved by the different prefetching policies for *all* benchmarks. The SMS prefetcher serves as baseline, since it is the best performing non-CBWS prefetcher,

The figure demonstrates the benefit of the *CBWS+SMS* prefetcher over *SMS*. Specifically, *CBWS+SMS* outperforms *SMS* by 1.31× for the memory-intensive benchmarks and by 1.16× for all benchmarks.

Both CBWS prefetchers (*CBWS* and *CBWS+SMS*) outperform all others for *nw*, *sgemm*, *radix*, *stencil* *lu_ncb*, *fft*. Furthermore, the integrated *CBWS+SMS* prefetcher delivers

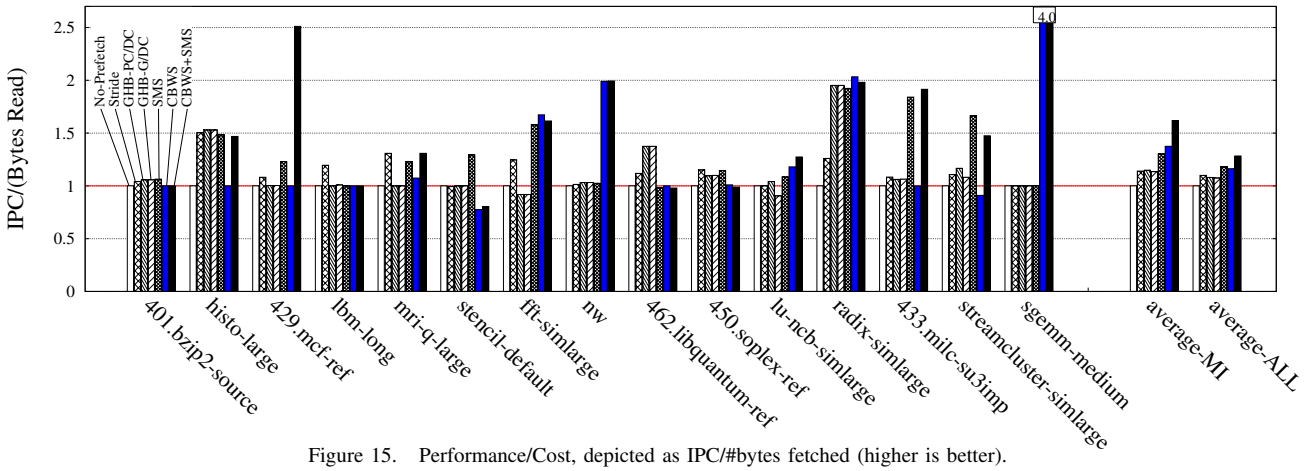

Figure 15. Performance/Cost, depicted as IPC/#bytes fetched (higher is better).

the best performance for *mcf*, *nw*, and *milc*.

Interestingly, for *fft* the *CBWS* prefetcher prevails even though it only yields better timeliness (Figure 13) but not the lowest MPKI (Figure 12). This is an example where MPKI, timeliness and accuracy interact in a conflicting manner.

Another unique example is *bzip2*, in which the performance of both CBWS prefetchers is ∼5% lower than that of *SMS*. This is because *bzip2* uses loops that perform large buffer reads from a file (hundreds of cache lines), whereas the CBWS prefetcher only traces working sets that consist of up to 16 cache lines. Nevertheless, since 16 lines are sufficient for the CBWS mappings of most benchmarks tested, increasing the number of differentials is not justified.

The CBWS-based schemes were also outperformed in *lbm*, *soplex*, *histo*. In these benchmarks, the data accessed by the tight, innermost loops is highly data-dependent.

```
for (i = 0; i < img_width*img_height; ++i) {
  BLOCK_BEGIN(0);
  const unsigned int value = img[i];
  if (histo[value] < UINT8_MAX) {
    ++histo[value];
  }
  BLOCK_END(0);
}
```

Figure 16. The main execution loop of the *histo* benchmark.

Figure 16 illustrates the main loop of *histo*, showing how its address calculation depends on input data. Therefore, the resulting access pattern cannot be detected using CBWS differential representation.

On the whole, the proposed *CBWS+SMS* prefetcher outperforms all others. Specifically, it outperforms the best non-CBWS prefetchers, namely *SMS*, by 31% for memory-intensive benchmarks and by 16% for all benchmarks.

D. Performance/Cost

Our last experiment compares the performance/cost ratio of the different prefetchers using the *no-prefetcher* configuration as baseline. We contrast the performance gained (ΔIPC) with the overhead incurred (in excess memory traffic) by fetching wrong memory addresses.

Figure 15 depicts the performance/cost compared to the baseline scheme, which has, by definition, a performance/-cost ratio of 1 ($baseline\ IPC/demand\ fetched\ bytes$). The figure shows that, on average, the *CBWS+SMS* policy provides the best performance/cost, with an average of 1.64 $IPC/bytes\ fetched$ compared to 1.39 $IPC/bytes\ fetched$ for the best non-CBWS prefetcher (SMS).

Interestingly, for *stencil* both differential schemes are less efficient than a system with no prefetching (a performance/-cost ratio lower than 1). This result suggests that the CBWS-based prefetchers are too aggressive for this benchmark.

In summary, the evaluation demonstrates the potential of CBWS-based prefetchers, which outperform all competing prefetchers both in raw performance and performance/cost.

VIII. Conclusions

In this paper, we have presented the code block working set (CBWS) prefetching scheme that predicts complete working sets of tight, innermost loop iterations. The scheme tracks the changes in the complete working sets of loop iterations using a vector of correlated memory strides. Furthermore, the proposed prefetcher uses explicit compiler annotations of tight loops to target code that can benefit from aggressive prefetching.

We have presented two prefetchers that are based on the proposed prediction scheme: a standalone *CBWS* prefetcher and a *CBWS+SMS* prefetcher, which falls back on the *spatial memory streaming* [29] (SMS) prefetcher when the CBWS prefetcher does not have sufficient history information to generate a prediction.

Finally, we evaluated the proposed prefetchers using a simulated out-of-order core on a range of both memory-intensive and regular benchmarks. We show that the integrated *CBWS+SMS* scheme delivers an average speedup of 1.16× over the next best performing prefetcher (SMS), while requiring only a small storage overhead.

ACKNOWLEDGMENT

We thank Micheal Ferdman and the anonymous reviewers for their valuable comments and suggestions. This research was funded by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by the Israel Science Foundation (ISF grant 769/12; equipment grant 1719/12). Y. Etsion was supported by the Center for Computer Engineering at Technion.

REFERENCES

[1] "SPEC CPU2006," http://www.spec.org/cpu2006.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2008.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug 2011.

[4] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2011.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2009.

[6] N. Chidambaram Nachiappan, A. K. Mishra, M. Kademir, A. Sivasubramaniam, O. Mutlu, and C. R. Das, "Application-aware prefetch prioritization in on-chip networks," in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2012.

[7] E. Ebrahimi, O. Mutlu, and Y. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2009.

[8] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Intl. Symp. on Microarchitecture (MICRO)*, 1992.

[9] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S. W. Keckler, "Running PARSEC 2.1 on M5," The University of Texas at Austin, Department of Computer Science, Tech. Rep., Oct 2009.

[10] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *ACM Intl. Conf. on Supercomputing*, 2004.

[11] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[12] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," 2009. [Online]. Available: http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf

[13] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Intl. Symp. on Computer Architecture (ISCA)*, 1997.

[14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Intl. Symp. on Computer Architecture (ISCA)*, 1990.

[15] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, 2011.

[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Intl. Symp. on Code Generation and Optimizations (CGO)*, Mar 2004.

[17] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. on Arch. & Code Optim.*, vol. 9, no. 1, pp. 1–29, Mar 2012.

[18] G. Marin, C. McCurdy, and J. S. Vetter, "Diagnosis and optimization of application prefetching performance," in *ACM Intl. Conf. on Supercomputing*, 2013.

[19] O. Mutlu, H. Kim, and Y. Patt, "Address-value delta (avd) prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *Intl. Symp. on Microarchitecture (MICRO)*, Nov 2005.

[20] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2003.

[21] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: An adaptive data cache prefetcher," in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2004.

[22] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2004.

[23] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Intl. Symp. on Computer Architecture (ISCA)*, 1994.

[24] R. Panda, P. V. Gratz, and D. A. Jimenez, "B-Fetch: Branch prediction directed prefetching for in-order processors," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 41–44, 2012.

[25] P. U. PARSEC Group, "A memo on exploration of SPLASH-2 input sets," Jun 2011.

[26] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 1999.

[27] A. Sharif and H.-H. S. Lee, "Data prefetching by exploiting global and local access patterns," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–17, 2011.

[28] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Intl. Symp. on Computer Architecture (ISCA)*, 2009.

[29] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Intl. Symp. on Computer Architecture (ISCA)*, 2006.

[30] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2007.

[31] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, Mar 2012.

[32] Z. Wang, D. Burger, K. S. Mckinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: A cooperative hardware/software approach," in *Intl. Symp. on Computer Architecture (ISCA)*, 2003.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Intl. Symp. on Computer Architecture (ISCA)*, 1995.

[34] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Intl. Symp. on Microarchitecture (MICRO)*, 1991.

[35] J. Zebchuk, H. W. Cain, X. Tong, V. Srinivasan, and A. Moshovos, "RECAP: A region-based cure for the common cold (cache)," *Symp. on High-Performance Computer Architecture (HPCA)*, 2013.