SCP: Synergistic Cache Compression and Prefetching

Bhargavraj Patel
Qualcomm
Raleigh-Durham, NC, USA
bhargavrajpatel2012@u.northwestern.edu

Nikos Hardavellas, Gokhan Memik
Northwestern University, EECS
Evanston, IL, USA
{nikos, g-memik}@northwestern.edu

***Abstract*—While processor caches cannot grow arbitrarily large due to area, power, and latency considerations, dataset sizes grow faster than Moore's Law and pressure caches to grow to accommodate the increasing working sets. Cache compression partially mitigates this problem by providing an effective cache capacity larger than the physical capacity of the cache, but the prevalent rule of thumb dictates that the miss rate lowers by only the square root of the additional cache capacity. Data prefetching and streaming engines can offer a better utilization of the cache space, but sophisticated schemes typically require significant on-chip space, and some even save part of the history they track in main memory (e.g., Spatio-Temporal Memory Streaming—STEMS) and oversubscribe the already limited off-chip bandwidth. In this paper we present synergistic cache compression and prefetching (SCP), a technique that utilizes the cache space saved by cache compression to implement the storage arrays required by data prefetching and streaming engines. SCP outperforms cache-compression-only and data-streaming-only schemes, and approximates the performance of a combined scheme that employs both cache compression and data streaming in hardware, but without the overhead of the additional history and storage arrays for the streaming engine. Utilizing the cache compression hardware to compress the storage arrays for a STEMS streaming engine, in addition to the data cache, allows the streaming engine to operate entirely on-chip using space saved by compressing the cache, obviating the need to offload parts of the history to main memory and further increasing performance.**

***Keywords*—processor cache, cache compression, prefetching, spatio-temporal data streaming**

I. Introduction

The exponential growth of dataset sizes and the proliferation of multicore and manycore computing put an immense pressure on the already oversubscribed off-chip memory bandwidth. The bandwidth wall is one of the most pressing problems of modern processors [4], and the ITRS [3] roadmap indicates that the performance gap between core and memory will keep increasing. This realization has prompted research to mitigate the memory-processor performance gap. The most commonly-employed techniques today are prefetching and cache compression [1, 7, 12, 14].

The cache hierarchy itself is one of the many techniques that hide the main memory latency. However, its effectiveness is constrained by the limited capacity and the core's sensitivity to the access latency of on-chip cache memory. Cache compression is often proposed as a remedy to relax the trade-off between cache size and cache access latency [1] as well as to reduce the off-chip bandwidth requirement [2]. Compressing cache data increases the effective cache size. This results in reduced miss rates and fewer accesses to off-chip memory. Recent research indicates that as the baseline cache size increases, the additional cache capacity offers relatively smaller benefit, thus cache compression provides diminishing returns [2]. At the same time, however, hand-in-hand with the increased cache capacity of newer processors comes an increasing core count on chip. These cores tend to share the last-level cache, which now again becomes a limited resource as the capacity per core remains relatively stable or falls, while data set sizes continue to grow.

Another architectural solution to hide memory latency is hardware prefetching. Hardware prefetchers are mainly categorized into two categories: stride prefetchers and stream prefetchers. Stride prefetchers are simple to implement but less accurate [16, 17]. Stream prefetchers are more accurate and render high coverage, but they follow complex algorithms that require a lot of meta-data storage and maintenance [6, 7]. These meta-data are typically stored on chip and stress the hardware resources with their demand for storage. In some cases, to realize the high storage requirement, a significant fraction of meta-data is stored in off-chip DRAM [5, 8]. While successful prefetching improves performance, storing bookkeeping information in off-chip memory results in additional demand for the already oversubscribed bandwidth to main memory and increases energy consumption.

Thus, on one hand we have cache compression which makes more efficient use of the on-chip storage but provides diminishing returns for larger caches. On the other hand we have stream prefetchers which are highly successful in hiding memory latency, but require significant amounts of on-chip storage to operate. Combining the two techniques seems an appealing proposition, as cache compression can free up on-chip resources for the stream prefetcher, who may then make better use of the additional capacity than just naively extending the effective capacity of the cache.

In this paper we propose Synergistic cache Compression and Prefetching (SCP), an architecture that leverages cache compression to implement the storage components of streaming prefetchers on the compressed cache. By doing so, SCP eliminates the need to access off-chip memory. Without loss of generality, we assume Spatio-Temporal Memory Streaming (STeMS) [5] as the underlying stream prefetcher and Base Delta Immediate (BDI) [2] to compress the cache data.

We demonstrate that SCP outperforms the compressed cache or the streaming prefetcher alone, as well as their combination when the two techniques are employed as discrete components that are separate from each other, rather than synergistic. We further improve the performance of SCP by compressing the prefetcher's meta-data in addition to the cache data. This compression of STeMS meta-data allows SCP to reduce the size of the various STeMS components and gradually eliminate the need for off-chip meta-data storage and reclaim the wasted bandwidth. Overall, our contributions are:

- We propose Synergistic cache Compression and Prefetching (SCP), an architecture that leverages cache compression to implement the storage components of streaming prefetchers on the compressed cache.

TABLE I. STeMS Storage Requirements

STeMS Component	Storage	Location
AGT	2.5 KB	On chip
PST	640 KB	Off chip
RMOB	1024 KB	Off chip
SVB	4 KB	On chip
Reconstruction / streaming buffers	2.5 KB	On chip
Lookup into MOB	4096 KB	Off chip (shared)

- Using applications from SPLASH-2 and PARSEC we show that SCP outperforms standalone cache compression (BDI), standalone memory streaming prefetchers (STeMS), and their combination.
- We show that SCP can be further enhanced by compressing the prefetcher's meta-data in addition to the cache data, which gradually obviates the need for off-chip storage of the prefetcher's meta-data.

The remainder of this paper is organized as follows: Section II provides details on STeMS and the general idea of how it can work synergistically with cache compression. We present the compressed cache architecture in Section III and SCP in Section IV. Sections V and VI detail our experimental methodology and results. Section VII presents additional sensitivity analysis and extensions to SCP. We present related work in Section VIII and conclude in Section IX.

II. Spatio-Temporal Memory Streaming (STeMS)

STeMS exploits both spatial and temporal locality to provide highly accurate data streaming [5]. Its spatial and temporal predictors continuously train on the execution of applications, and store the discovered data streams and their predictors in meta-data. During the reconstruction phase, streams are formed based on the meta-data and finally, in the streaming phase, by following the appropriate stream, data are fetched from memory just ahead of their use [5]. STeMS consists of the Active Generation Table (AGT), Pattern Sequence Table (PST), Region Miss Order Buffer (RMOB), Lookup into MOB, Streamed Value Buffer (SVB), and the reconstruction buffer.

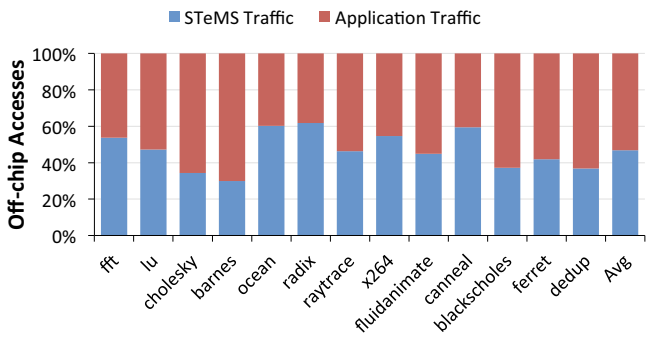
A. Storage Requirements for STeMS

Table I shows the components of STeMS, their storage requirement, and their location. Please note this list does not include area estimates for computation hardware like adders or shifters. Overall, STeMS requires a total of 1673 KB per core (most of it off chip). In addition, it also requires a 4 MB lookup into MOB table that all cores share. STeMS' storage requirement is 9 KB/core on chip, 1664 KB/core off chip, plus a 4 MB off-chip table shared across all cores.

B. Off-chip Bandwidth Requirements for STeMS

The RMOB and Lookup into MOB tables require MBs, hence they are implemented off chip. As these arrays are accessed frequently, STeMS implements an ingenuous method that slowly streams on chip only the necessary parts while hiding their latency. Even with an optimized access pattern to these tables, though, STeMS may create considerable off-chip bandwidth pressure. Figure 1 shows the breakdown of the off-chip accesses for STeMS meta-data vs. the ones initiated by the application. STeMS is responsible for up to 62% of the off-chip data traffic (47% average). Hence, STeMS meta-data accesses consume a significant fraction of the off-chip bandwidth and may elicit detrimental performance consequences.

Fig. 1. Distribution of off-chip traffic.



C. Cache Compression and STeMS Limitations

Overall, although STeMS is a highly accurate streaming prefetcher, its performance is affected by (i) large storage requirements for all its buffers and tables that force the designers to push the majority of them outside the chip, and (ii) high off-chip bandwidth utilization to access all this meta-data. Our proposed architecture, SCP, utilizes the cache capacity saved by the compressed cache to implement the on-chip storage for STeMS without the need for additional on-chip memory, and also brings the off-chip STeMS meta-data on chip and can even compress them to allow maximum effective capacity. Hence, SCP not only reduces STeMS' hardware requirement, but also completely eliminates the additional DRAM memory requirement for meta-data, and relieves the additional off-chip bandwidth pressure. By bringing the meta-data on chip, it also lowers their access latency. The reduced access latency allows SCP to resize the arrays without performance loss and use the available on-chip capacity even more efficiently.

III. Cache Compression

Cache compression increases the effective size of the baseline cache. It is a heavily studied technique to reduce the cache misses and the off-chip bandwidth requirement. Various compression techniques have been presented in literature [1, 2, 9]. To find a feasible underlying compression technique for SCP and to understand the correlation between cache size and compression effectiveness, we analyze Base Delta Immediate (BDI) compression [2].

To analyze BDI, we compress the last level cache (LLC), which is the L3 in our system. We model a conventional inclusive cache read/write policy. The L1 is a private split instruction/data cache, and the L2 is a unified private L2 cache. The L2 cache accesses the data from the L3 cache via compression and decompression units that use BDI. All writes to L3 go through the compression unit and all reads from L3 go through the decompression unit. Furthermore, similar to prior work [2], we constrain every cache block to contain a maximum of two compressed cache lines to limit the tag space.

A. Cache Size and Compression

Recent architectures employ large caches as the multicores grow in core counts and the last-level cache is typically shared. We analyze the performance of BDI on large caches by imple-

TABLE II. ARCHITECTURAL PARAMETERS

L1 cache	Split I/D, 64 KB, 2-way set associative, LRU replacement, 64-byte cache line, 2-cycle access time
L2 cache	Unified, 4 MB (unless otherwise specified), 8-way set associative, LRU replacement, 64-byte cache line, 2.54 ns access time (12 cycles), core-private
L3 cache	Unified, 32 MB (unless otherwise specified), 16-way set associative, LRU replacement, 64-byte cache line, 5.26 ns access (25 cycles), shared
Main Memory	DDR3, 1600MHz, 51.2 GB/s, 4 GB (128MB x 4 x 8 banks), single rank, 180-cycle access time, timing: CL-tRCD-tRP-tRAS = 9-9-9-25
Processor	4 cores, each core 4-wide x86 OoO superscalar, 11-stage pipeline: fetch (3), decode (3), schedule (1), execute (1+), retire (3), 64-entry ROB, 2.5GHz
Compression / Decompression	BDI Compression: 1 cycle for L2, 2 cycles for L3, Decompression: 1 cycle for L2 and L3

menting BDI in Gem5 [10]. Table II details the simulated architecture on which we run the SPLASH-2 [13] applications described in Table III. The latencies for all caches are derived using CACTI [11].

As we mentioned in Section I, cache compression yields diminishing returns as the cache size increases. Figure 2 shows the miss rate for cache sizes between 0.5-4MB with compression (solid bars) and without compression (patterned bars). A BDI-compressed cache offers lower miss rate than a baseline cache of the same size. Actually, it approximates the miss rate of a cache with double the size. For example, a 512KB cache with compression achieves almost the same miss rate as a 1MB cache without compression. For some applications (e.g., LU) a 1MB compressed cache renders lower miss rate than a 2MB uncompressed cache. The reason behind this surprising result is that cache compression alters the eviction patterns of the compressed cache by implementing a slightly different LRU [2] policy. Unlike conventional LRU, the algorithm used in compressed caches considers the position of the blocks in the LRU chain as well as the compressed size of the incoming cache line and the cache lines already present in the set. The different eviction patterns may randomly help or hurt performance; in the case of LU, they help overall. These observations corroborate previous studies [2].

As expected, we also observe that cache compression yields diminishing returns as the cache size increases (Figure 2). This happens because for small cache sizes (e.g., 512KB) compression improves the effective cache size by 97% (Figure 3), while for larger caches (e.g., 32MB) the improvement is only 32%. This difference readily translates to performance improvement as shown in Figure 4: compressing a 512KB cache increases IPC by 25% on average (up to 30%), while compressing a 32MB cache yields only 12% speedup on average (up to 19%).

Fig. 2. Effect of cache compression on miss rate.

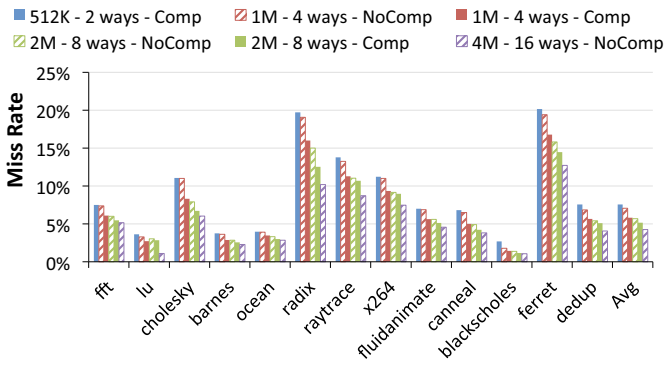


TABLE III. SPLASH-2 AND PARSEC APPLICATIONS

Application	Domain	Input Parameters
FFT	Signal Processing	4,194,304 data points
LU	HPC	1024×1024 matrix, 64×64 blocks
Ocean	HPC	514×514 grid
Cholesky	HPC	tk29.O
Barnes	HPC	65,536 bodies
Radix	General	8,388,608 integers
Raytrace	Graphics	Car
x264	Media Processing	256 frames, 1280x800 pixels
Fluidanimate	Animation	10 Frames, 600,000 particles
Canneal	Engineering	800,000 elements
Blackschole	Financial Analysis	128 swaptions, 20,000 simulations
Ferret	Similarity Search	512 queries, 34,973 images
Dedup	Enterprise Storage	256MB data

IV. SYNERGISTIC COMPRESSION AND PREFETCHING (SCP)

The advent of multicores has led to a significant increase in cache sizes, and as Section II demonstrates, compression offers diminishing returns with increasing cache size. In addition, accurate prefetchers like STeMS are constrained by their extensive memory requirement for meta-data storage and may suffer from increased off-chip memory traffic [5]. These observations motivate our proposed architecture: the STeMS limitations can be mitigated if it is possible to keep the majority of the meta-data on chip without adding extra hardware. A compressed cache is a natural fit for this goal. Compression provides extra space, but using that extra space as a conventional cache does not render performance improvement equivalent to STeMS. Hence, SCP implements STeMS memory components on the extra cache space and eliminates any off-chip communication for meta-data. SCP not only achieves the best of both techniques, but also eliminates two of their major limitations.

Fig. 3. Increase in effective cache size due to compression.

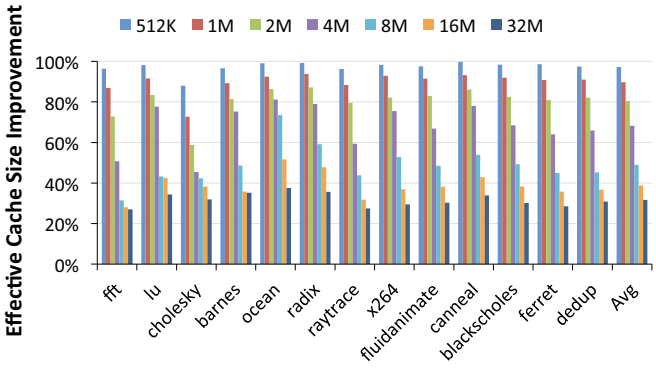
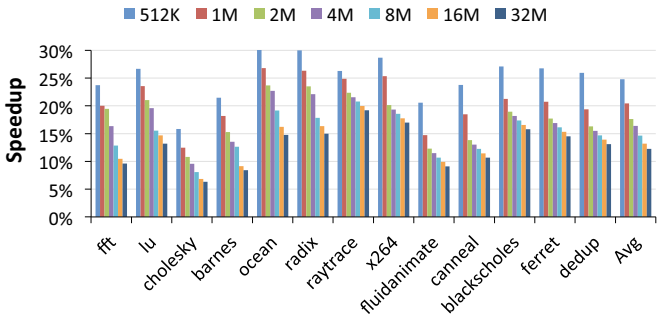


Fig. 4. Performance improvement of compressed caches over uncompressed.



A. SCP Implementation

All on-chip tables for STeMS are in SRAM memory so it can be easily placed into the cache memory. As mentioned in Table I, STeMS requires 1673 KB per core in addition to the fixed 4 MB shared memory requirement. The components AGT, PST, RMOB, SVT, and reconstruction buffer are per-core components, while the Lookup into MOB table is shared between all cores. In our architecture, L2 and L3 are both compressed using BDI compression. L2 is core-private and implements all per-core STeMS components except the reconstruction buffer. As the reconstruction buffer is accessed most frequently throughout the reconstruction process, it is not implemented on the cache but as a separate buffer similar to the original STeMS proposal. Hence, 1673 KB out of total 4MB of L2 space are dedicated to STeMS components, and the rest of the L2 cache works as a BDI-compressed cache described in Section II. Similarly, the Lookup into MOB array, which is shared between four cores, is implemented on the shared L3 cache. Thus, 4 MB out of a total 32 MB of L3 cache memory are allocated to the Lookup into MOB table, and the rest of the L3 cache is implemented as a BDI-compressed cache described in Section II. Table IV summarizes the location of each component for the STeMS and the proposed SCP implementation.

Figure 5 depicts the implementation of SCP for a single core. The L1 instruction, L1 data, and L2 caches are core-private and the L3 cache is shared among the cores. As mentioned earlier, all core-private STeMS components except the Reconstruction and Streaming buffers are implemented on the compressed L2 cache and all shared STeMS components (i.e., the Lookup into MOB table) are implemented on the shared L3 compressed cache. As both caches are compressed using the same BDI algorithm [2], no compression and decompression modules are required between L2 and L3. For SCP we assume that the STeMS data stored on the caches are not compressed; we will evaluate a version of SCP that compresses the STeMS data as well at a later section.

Fig. 5. STeMS on compressed cache.

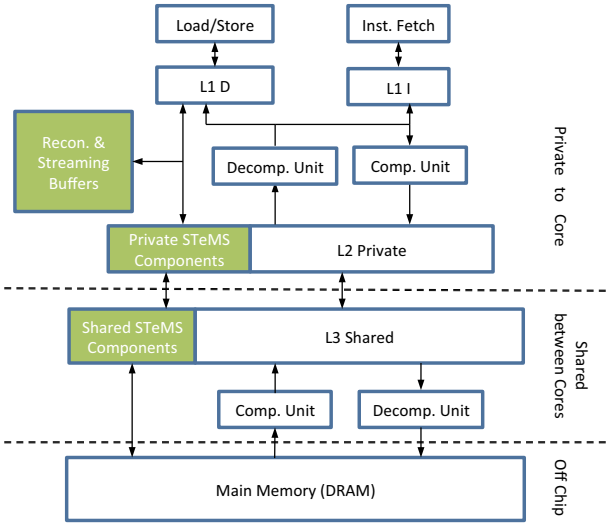


TABLE IV. STeMS Components Locations

STeMS Component	STeMS Implementation	SCP Implementation
AGT	On Chip	On Chip (L2 Cache)
PST	Off Chip (DRAM)	On Chip (L2 Cache)
RMOB	Off Chip (DRAM)	On Chip (L2 Cache)
SVB	On Chip	On Chip (L2 Cache)
Lookup into MOB	Off Chip (DRAM) shared by n cores	On Chip (L3 Cache) shared by n cores
Reconstruction and streaming buffers	On Chip	On Chip

B. STeMS Components on Cache

All the STeMS components that were originally on chip are still on chip in SCP and stored on the caches, so they can be easily implemented. However, in the base STeMS implementation, RMOB and Lookup into MOB were originally off chip. In SCP, they are stored in the caches too. Each STeMS component is allocated a specific address range on the cache according to the component's size. A state machine to mimic the same behavior of the STeMS component out of cache memory is also implemented. Moreover, any access to any entry in a STeMS table is implemented as a conventional cache access. Each component implementation is explained below.

1) Hash Table

The Hash table is used to retrieve the location of the first occurrence of the off-chip miss address in the RMOB. We use the same bucketized probabilistic hash table as STeMS [8] and used by Wenisch et al. [6]. The difference here is that a bucket points to multiple entries of a particular set in the cache instead of one in main memory. All the entries available in 16 ways are linearly compared to the miss address in order to retrieve the RMOB pointer for that address. Furthermore, unlike previous work [6], SCP does not require a dedicated implementation of LRU as support for the same is already available in a conventional cache which follows LRU.

2) Region Miss Order Buffer (RMOB):

The RMOB is a circular buffer and records off-chip miss addresses (5-bytes physical address) along with 16 bits for the PC and 8 bits for the delta, totaling of 8B per entry, having

Fig. 6. RMOB entry access mechanism in SCP

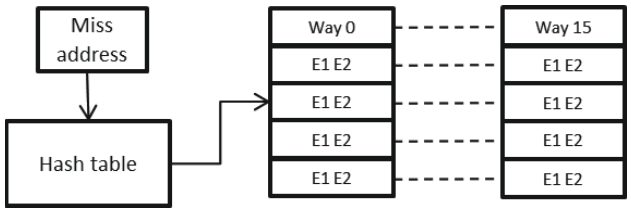


Fig. 7. Miss rate reduction for cache compression, STeMS, C+S, and SCP.

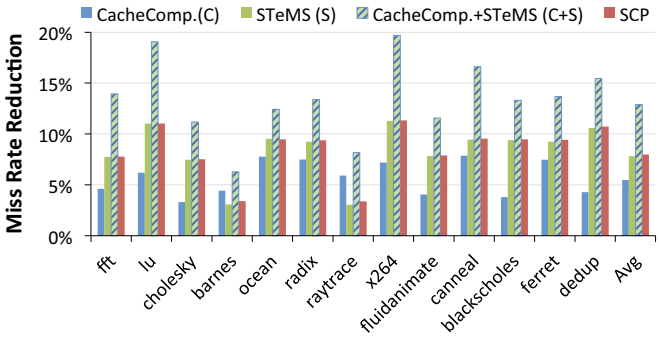
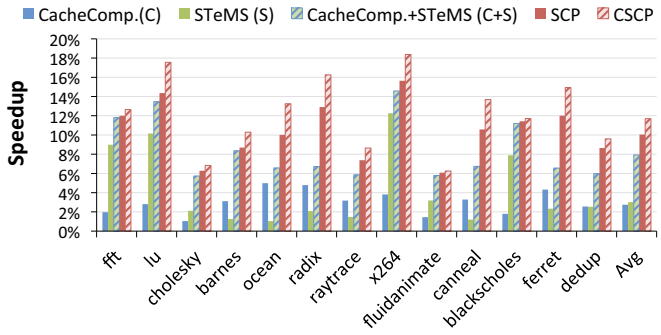


Fig. 8. Performance improvement over baseline uncompressed cache.



128K entries. A Hash table (explained previously) is used to find the recent location of the address in the RMOB and using that pointer an entry related to the specific miss address is retrieved from the RMOB. To implement RMOB on the cache (Figure 6) each cache block contains 2 RMOB entries and a contiguous 128K-entry region is allocated for RMOB, which can be accessed by the RMOB pointer retrieved from the Hash table for a particular miss address. As Figure 6 depicts, each off-chip miss address goes to the Hash table to find the RMOB entry for the first occurrence of that address. Once an address for the same is generated from the table, it is given back to the RMOB to retrieve the data for the reconstruction.

3) Active Generation Table (AGT):

The Spatial Streaming engine records the blocks accessed over the course of a spatial region generation in the active generation table (AGT). When a spatial region generation begins, an entry is allocated in the AGT and continuously updated until the spatial block ends and is sent to PST [5]. An end-of-spatial-block is detected by eviction or invalidation of any block accessed during the generation. Each entry in AGT is stored using a spatial region tag, the high-order bits of the region base address, and also accessed using the same tag. The same table is implemented on the SRAM by storing its spatial region tag into a conventional cache tag in the region allocated for AGT.

4) Pattern Sequence Table (PST):

PST is a set-associative structure similar to a cache. The PST is accessed using a prediction index constructed from the PC and a spatial region offset of the trigger access for a generation. Each entry in the PST stores the spatial pattern that was accumulated in the AGT [6]. As PST itself is a set associative structure like a cache, it can be directly mapped on the cache using prediction index formed by PC and offset from the cache. Upon a trigger access, the Spatial streaming engine consults the PST to predict which blocks will be accessed during the generation. If an entry in the PST is found, the spatial region's base address and the spatial pattern are copied to one of several prediction registers [6].

5) Prefetch Buffer (SVB):

This buffer stores the prefetched data and avoids cache pollution due to mispredictions. Once the processor accesses the data from SVB, the data is moved to the cache. As SVB is a buffer consisting of chunks of data, it can be directly mapped to the cache. We allocate a linear region on the last level cache to store 64 entries of SVB. However, this implementation suffers from the higher latency of L2 cache compared to the STeMS dedicated SVB implementation. This is not on the critical path of the processor, though, and does not affect the performance.

V. Experimental Methodology

We evaluate SCP and compare it to other techniques using the cycle accurate full system simulator Gem5 [10]. The Gem5 models a full system x86 architecture and can execute unmodified workloads and different operating systems. We model a processor with four x86 out-of-order-cores and three levels of coherent inclusive cache memory. A detailed system configuration is described in Table II. In our system, the L1 cache is split into instruction and data cache, the L2 cache is a private cache, and L3 is shared between four cores. All compression and decompression units follow the BDI [2] algorithm. DRAM is implemented by integrating Gem5 with DRAMSim. The latency and timing parameters for different caches are derived using Cacti 5.3 [11]. We model a MESI [15] protocol for cache coherence.

To achieve a good balance when evaluating SCP we use a mixture of SPLASH-2 [13] and PARSEC [12] benchmarks. Table III details the applications we use in our experiments. We collect our results by taking checkpoints at various points. To avoid cold-start effects, a measurement is not collected for the first 2M memory requests. After that, a simulation is allowed to run for 4B memory requests. To measure performance we calculate IPC as the aggregate number of instructions committed per cycle by all cores.

VI. Experimental Results

For simplicity, in the remainder of the paper we will use the following notation: C = cache compression only, S = baseline STeMS only, C+S = compressed cache and STeMS together, SCP = synergistic cache compression and prefetching using BDI for compression and STeMS for memory streaming, and CSCP = SCP where the STeMS meta-data are also compressed

A. Miss Rate Improvement

Figure 7 presents the miss rate reduction achieved by C, S, C+S, and SCP for the SPLASH-2 and PARSEC applications in Table III. The miss rate reduction is calculated over a baseline architecture that does not employ compression or STeMS. The misses represent accesses to off-chip main memory (i.e., L3 misses). They do not include accesses related to STeMS meta-data. Figure 7 indicates that the miss rate improvement for C+S is the highest as it leverages the advantages of both techniques without the downside of accessing main memory for STeMS meta-data. SCP is outperformed by C+S (8% vs. 13% miss rate reduction respectively), but performs better than compressed cache alone, and perform as well as STeMS. This result is expected due to two factors. First, C+S compresses the cache and utilizes the extra space as a cache. In our SCP implementa-

Fig. 9. Performance sensitivity of resizing STeMS components with BDI.

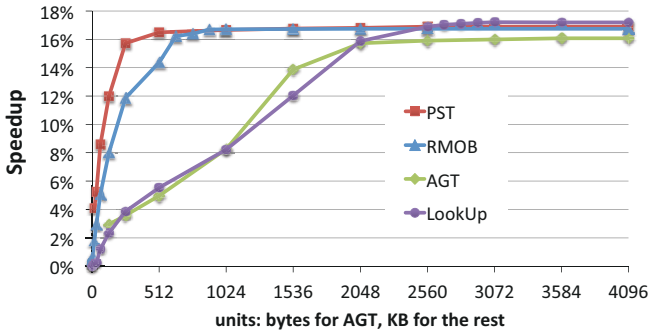
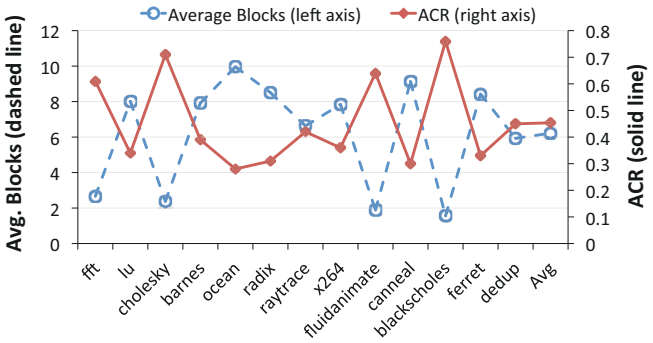


Fig. 10. Average blocks per set and avg. compression ratio (ACR) for CSCP.



tion, although the cache is compressed, the saved space in both L2 and L3 caches is mostly utilized to store STeMS meta-data. Thus, SCP does not benefit much from miss reduction due to compression. Second, we do not change the STeMS algorithm and its design for SCP. Thus, the miss rate improvement of SCP is almost the same as STeMS' miss rate improvement.

B. Performance Improvement

As shown in Figure 8, SCP achieves on average 10% higher IPC (and up to 16%) over a baseline cache with no compression and no STeMS support. Furthermore, for all the applications, compressed cache and STeMS together (C+S) results in higher improvement than C or S alone, because it leverages the advantages of both techniques. The IPC improvement is mostly additive, except for a few applications.

We have further analyzed C+S and SCP to make sure the gain in SCP is not just because of using these two techniques together. These two implementations (C+S and SCP) have major differences. In C+S the cache is compressed and STeMS is in its original implementation so the meta-data is stored off chip or on chip using extra hardware. In SCP the cache is compressed and STeMS meta-data is stored on the compressed cache. Consequently, C+S creates extra demand of off-chip bandwidth while SCP suffers no extra off-chip access at all. That is the main reason why SCP renders higher IPC performance than C+S. As indicated in Figure 1, on average, STeMS meta-data accesses constitute 47% of off-chip bandwidth utilization. Our implementation eliminates all of STeMS-related off-chip accesses and makes more bandwidth available to the actual application. Consequently SCP achieves high prediction accuracy without oversubscribing the off-chip bandwidth.

Applications that are bandwidth limited like ocean, radix, canneal and ferret have shown less IPC improvements for STeMS than compressed cache although they exhibit higher improvement in miss rates compared to the compressed cache (Figure 7). This indicates that for bandwidth-limited applications, STeMS can predict accurately and improve the miss rate but at the cost of extra bandwidth requirement which constrains IPC improvement. Our implementation eliminates that limitation of STeMS and renders better IPC performance than all other alternatives. For example, for radix, SCP results in 11% higher improvement than STeMS (S) and 6% higher than compressed cache and STeMS (C+S) together. Similarly, for ferret, SCP results in 10% higher improvement than STeMS and 5% higher than compressed cache and STeMS together.

In addition to that, barnes and raytrace are also bandwidth limited applications and show less IPC improvement with STeMS compared to the compressed cache. At the same time, the improvements in miss rate for STeMS is also less compared to the compressed cache. In other words, for these applications, STeMS can not predict the patterns accurately. For these applications, STeMS increases the memory bandwidth demand and does not reduce the miss rates considerably. For these scenarios, SCP performs better by eliminating the extra bandwidth requirement and also leveraging the advantage of compressing the cache. Furthermore, SCP compresses both L2 and L3. Hence, compressed data is communicated between L2 and L3. This reduces the bandwidth requirement between L2 and L3 caches. It is important to note here that our technique has this advantage only over STeMS.

Overall, SCP achieves average speedups of 7% over a compressed cache, 6% over STeMS, and 2% over C+S. However, on the applications that SCP really targets (i.e., bandwidth-limited workloads like ocean, radix, canneal and ferret) SCP achieves speedups of 7% over compressed cache, 10% over STeMS and 5% over C+S, respectively.

VII. Compressing SCP and Resizing STeMS Tables

STeMS has different buffers and tables to store meta-data information and predict future accesses. The latency of accessing these buffers and tables, particularly the off-chip ones, is one of the very important parameters that define the sizes of both on-chip and off-chip tables. In general, the higher the off-chip latency, the larger the size of tables required to implement STeMS. In our implementation, all the off-chip tables are on chip, so the access latency to the RMOB and PST tables is reduced. This affects both on-chip and off-chip table sizes.

Further, even though we are using compression for application data, up to now we refrained from compressing the STeMS data in order to make fair comparisons with the original STeMS. Using compression for STeMS data adds compression and decompression latency to access STeMS data, but it also reduces the required space for STeMS tables and renders more space for application data. Hence, in this section, we present results of a system that compresses STeMS meta-data as well using the BDI compression algorithm. We compress all the STeMS meta-data stored in the L2 and L3 caches. With this new implementation, Compressed SCP (CSCP), we need to resize the STeMS tables as described in the next section.

A. Resizing

Figure 9 shows the speedup achieved when resizing different STeMS tables. The experiment is run on the same system configuration and same mix of PARSEC and SPLASH-2 appli-

cations as in the previous section. Results collected over all these applications are averaged and presented here. Please note that only the tables available on L2 and L3 caches are compressed, i.e., the reconstruction buffer and SVB are not compressed. According to our results, for AGT at 2176 bytes, PST at 512 KB, RMOB at 640KB and Lookup into MOB at 2688 KB performance improvement saturates. As shown in Table V, in total, the original STeMS design needs 5769 KB of storage while with the new compressed meta-data implementation it needs 3848.125 KB storage. This corresponds to an average of 33.4% storage reduction.

B. Performance Evaluation of CSCP

Figure 8 compares the speedup achieved by the new implementation (CSCP) with the previous one (SCP) and configurations C, S, and C+S. On average, CSCP achieves 2% higher performance than SCP across all applications, and 4% higher performance across bandwidth-limited applications. Overall, CSCP achieves average speedups of 9% over a compressed cache, 9% over STeMS, and 4% over C+S. For bandwidth-limited workloads (ocean, radix, canneal and ferret) CSCP achieves speedups of 10% over compressed cache, 13% over STeMS and 8% over C+S.

This improvement is due to the increase in available cache memory for application data due to the reduction in STeMS tables sizes. In order to measure the effective cache size, we also collected average blocks per set with compression throughout the simulation (Figure 10). Further, to observe the correlation between performance improvement and compression, we also measured the Average Compression Ratio (ACR). For a cache line, the Compression Ratio is the ratio of the cache line data size after compression over the uncompressed size. ACR is calculated by taking the average of the compression ratios of all cache lines compressed throughout the simulation. Hence, if ACR is low it means that compression is high and vice versa. In Figure 10, ACR is represented on right axis. Figure 10 clearly depicts the reason behind the higher performance improvement of CSCP compared to SCP. For low ACR, the improvement in average blocks per set is high as compression is high. Consequently, more cache size is available for application data. As a result, higher performance improvement is achieved with CSCP compared to SCP.

Finally, Figure 11 shows the speedup of SCP and CSCP assuming 1 and 2 ported caches. SCP 1p and CSCP 1p are the configurations we have used in our evaluation so far, and employ a single port in the cache. SCP 2p and CSCP 2p are configurations with two cache ports that we use in this experiment to evaluate the improvement that our technique could

TABLE V. SIZE OF STEMS COMPONENTS WITH BDI COMPRESSION

STeMS component	Original size	Compressed	% Improvement
AGT	2.5 KB	2.125 KB	15%
PST	640 KB	512 KB	20%
RMOB	1 MB	640 KB	37.5%
Lookup into MOB	4 MB	2688 KB	34.375%
SVB	4 KB	4 KB	0%
Reconstruction and Streaming Buffers	2.5 KB	2.5 KB	0%
Total	5769 KB	3848.125 KB	33.4%

achieve if more ports are available in the cache to eliminate contention. As simulation results indicate, SCP and CSCP benefit from the additional port by gaining 2% higher speedup over their single-ported counterparts. Still, compared to the original cache configurations, the majority of the improvements come from the SCP and CSCP techniques themselves, not from the additional ports. While SCP and CSCP may exacerbate contention on the cache ports, their benefits severely outweigh this limitation. If additional ports or other contention-mitigation techniques are employed on the caches, then SCP and CSCP will perform even better that what our results indicate.

C. Sensitivity of CSCP to Cache Size

The size of the L2 and L3 caches affects CSCP in complex ways. The smaller the caches are, the harder it is for STeMS tables to fit in them. At the same time, the miss rate increases and provides more opportunity for cache compression and STeMS to improve performance. In our evaluation so far we modeled the cache sizes indicated in Table II, which are on the high-end of modern processors. Figure 12 shows the performance impact of CSCP on a multicore with 1MB L2 and 16MB L3 caches (CSCP-1-16), providing a data point that is closer to modern systems (the rest of the architectural parameters remain the same). Overall, even with 2-4x smaller caches, the performance advantage of CSCP remains robust. On average across all workloads, CSCP-1-16 provides 2% higher speedup than CSCP on larger caches (CSCP-4-32). On bandwidth-limited workloads, CSCP-1-16 does better and achieves on average 4% higher speedup (up to 6%). The per-application results are mixed, however, as there are workloads showing a small performance degradation (e.g., -2% for raytrace) while others show an improvement (e.g., 6% for radix). While a full-scale sensitivity analysis is beyond the scope of this paper, these results offer a glimpse into the complex nature of this trade-off

Fig. 11. Impact of additional ports on performance.

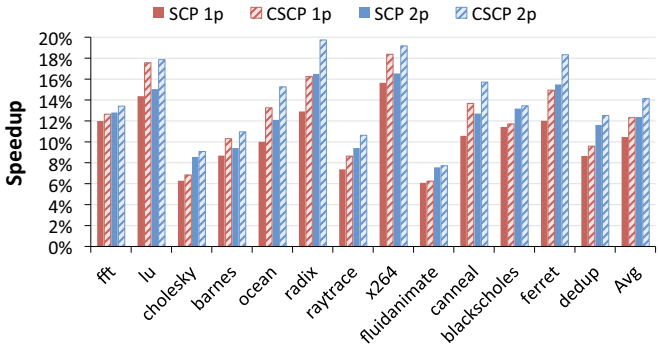
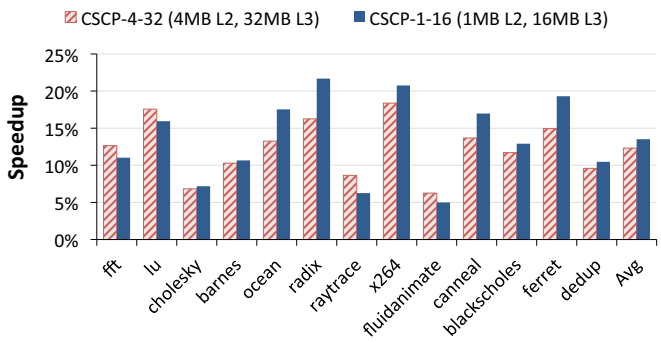


Fig. 12. Sensitivity of CSCP to cache size.



and indicate that the overall performance of CSCP is likely to be preserved even with significant changes in cache size.

VIII. Related Work

Alameldeen and Wood proposed dynamic cache compression based on a decoupled variable-segment cache structure [18] using a frequent pattern compression algorithm [1]. Hallnor and Reinhardt proposed a unified compression scheme based on indirect-indexing cache [19]. Chen et al. proposed C-Pack cache compression based on the PBPM algorithm [20]. Xie and Loh propose thread-aware dynamic cache compression to make better per-thread compression decisions [22]. Mowry et al. proposed Base-Delta-Immediate (BDI) compression [2]. Our technique could use any of these algorithms for compression. Alameldeen and Wood use cache compression's extra tags to help prefetching [21]. Somogyi *et al.* and Wenisch *et al.* [5-8] present spatial-, temporal-, and spatio-temporal streaming. SCP is applicable to any prefetcher and streaming mechanism that requires on-chip storage, but is more beneficial to ones that offload meta-data to off-chip memory (e.g., STeMS).

IX. Conclusion

As the demand for larger caches grows and the increase in core counts continues to oversubscribe the available off-chip memory bandwidth, techniques like cache compression and streaming prefetchers are bound to become increasingly important. However, instead of employing these techniques in isolation, we show that processors can benefit greatly by combining them. In this paper we propose Synergistic Cache Compression and Prefetching (SCP), an architecture that utilizes cache compression to implement efficiently memory streaming prefetchers using the additional space saved by the cache compression. Compressing the prefetcher's meta-data as well (Compressed SCP or CSCP for short) yields even higher benefits. In addition, our proposed architecture removes the extra off-chip memory bandwidth required by the more sophisticated memory streaming prefetchers (e.g., STeMS) and as a result achieves even higher performance than the combination of compression and streaming.

Overall, our results indicate that SCP and CSCP achieve on average 10-13% higher performance (up to 22%) over a baseline architecture that does not employ any of these techniques. SCP and CSCP outperform cache compression and memory streaming (STeMS) alone by 6-9% on average (up to 15%). Even compared to employing cache compression and STeMS memory streaming together as discrete separate techniques with all the necessary additional hardware, SCP and CSCP achieve an additional 2% speedup (up to 6%). Thus, synergistic cache compression and prefetching can match or exceed the performance of implementing these techniques alone. SCP and CSCP, however, allow the implementation of such complex streaming prefetchers with minimal hardware impact, as the majority of the required storage is implemented by harnessing the cache space saved by compressing the cache itself. We believe that this result corresponds to a definitive step forward in making complex prefetchers with large on-chip storage requirements practical to implement.

X. Acknowledgements

This work is partially supported by NSF award CCF-1218768, NSF CAREER award CCF-1453853, DoE award DE-SC0012531, and an Intel URO Energy Smart SoC Program grant. The authors would also like to thank the reviewers for their insightful comments.

References

[1] A. R. Alameldeen and D. A. Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. Technical Report, University of Wisconsin-Madison, 2004.

[2] T. C. Mowry, G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In 21st International Conference on Parallel Architecture and Compilation Techniques, 2012.

[3] ITRS. http://www.itrs.net/Links/2012ITRS/Home2012.htm

[4] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In 36th Annual International Symposium on Computer Architecture, 2009

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

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

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

[8] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for address-correlated prefetching. In 15th Symposium on High Performance Computer Architecture, 2009.

[9] L. Benini, D. Bruni, B. Ricco, A. Macii, and E. Macii. An adaptive data compression scheme for memory traffic minimization in processorbased systems. In IEEE international conference on circuits and systems, 2002.

[10] 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. SIGARCH Computer Architecture News, 39(2), pp. 1-7, August 2011.

[11] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. HP Tech Report HPL-2008-20, April 2, 2008.

[12] C. Bienia, S. Kumar, J. P. Singh and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In 17th International Conf. on Parallel Architectures and Compilation Techniques, 2008.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In 22nd Annual International Symposium on Computer Architecture, 1995.

[14] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In International Conference on Supercomputing, 2009.

[15] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In 11th Annual International Symposium on Computer Architecture, 1984.

[16] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In 17th Annual International Symposium on Computer Architecture, 1990.

[17] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In 33rd International Symposium on Microarchitecture, 2000.

[18] A. R Alameldeen, and D.A Wood. Adaptive Cache Compression for High-Performance Processors. In 31st Annual International Symposium on Computer Architecture, 2004

[19] E. G, Ha1Inor, and S. K. Reinhardt. A fully associative software managed cache design. In 27th Int'l. Symposium on Computer Architecture, 2000

[20] X. Chen, L. Yang, R. P. Dick, L. Shang, H. Lekatsas. C-Pack: a high-performance microprocessor cache compression algorithm. In IEEE Transactions on Very Large Scale Integration Systems, l8(8), 2010

[21] A. R. Alameldeen and D. Wood. Interactions between compression and prefetching in chip multiprocessors, In 13th International Symposium on High Performance Computer Architecture, 2007.

[22] Y. Xie and G. H. Loh. Thread-aware dynamic shared cache compression in multi-core processors. In 29th International Conference on Computer Design, 2011