XStream: Cross-core Spatial Streaming based MLC Prefetchers for Parallel Applications in CMPs

Biswabandan Panda and Shankar Balachandran,
Department of Computer Science and Engineering,
Indian Institute of Technology Madras,
{biswa, shankar}@cse.iitm.ac.in

Abstract

Hardware prefetchers are commonly used to hide and tolerate off-chip memory latency. Prefetching techniques in the literature are designed for multiple independent sequential applications running on a multicore system. In contrast to multiple independent applications, a single parallel application running on a multicore system exhibits different behavior. In case of a parallel application, cores share and communicate data and code among themselves, and there is commonality in the demand miss streams across multiple cores. This gives an opportunity to predict the demand miss streams and communicate the predicted streams from one core to another, which we refer as cross-core stream communication.

We propose cross-core spatial streaming (XStream), a practical and storage-efficient cross-core prefetching technique. XStream detects and predicts the cross-core spatial streams at the private mid level caches (MLCs) and sends the predicted streams in advance to MLC prefetchers of the predicted cores. We compare the effectiveness of XStream with the ideal cross-core spatial streamer. Experimental results demonstrate that, on an average (geomean), compared to the state-of-the-art spatial memory streaming, storage efficient XStream reduces the execution time by 11.3% (as high as 24%) and 9% (as high as 29.09%) for 4-core and 8-core systems respectively.

1. INTRODUCTION

Hardware Prefetching plays an important role in reducing the execution time of programs. In commercial chip-multiprocessors (CMPs), different types of hardware prefetchers are employed at the mid level caches (MLCs). For example, Intel's Nehalem uses a spatial prefetcher and a stream prefetcher at the private MLCs [2]. Stream based prefetchers fetch data in the form of streams wherein a stream is a *sequence of cache-line-aligned memory addresses*. A demand miss stream corresponds to demand miss addresses only. Temporal (Spatial) streams are streams that recur temporally (spatially). **Streams that spread and recur across multiple cores are known as cross-core streams**.

Figure 1 shows an example of cross-core streams for a 2-threaded application. The X axis denotes the logical time order and the Y axis shows the demand miss addresses (aligned to cache line address) from two cores (core 0 and core 1). It can

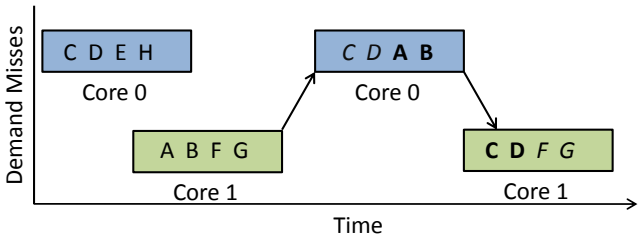

Figure 1: Example of cross-core streaming. A, B, C, and D are part of cross-core streams, and E, F, G, and H are part of intra-core streams.

be seen that, the demand misses observed in one core recur in other cores. For example, demand misses to A and B are first observed in core 1 and later recur in core 0. Similarly, misses to C and D are first observed in core 0 and later recur in core 1. These miss streams are cross-core miss streams. Streams local to a core may result in hits if the stream prefetcher prefetches in-time. In this example, C and D get hits at core 0 and F and G get hits at core 1. But a cross-core spatial stream oblivious prefetcher can not eliminate the misses to cross-core streams A and B (shown in bold) in core 0 and similarly to C and D (shown in bold) in core 1.

Past works related to hardware cross-core stream prefetching, such as temporal memory streaming (TMS) [25] and store-ordered streaming (SORDS) [26] have been explored to minimize the coherent read misses. TMS [25] and SORDS [26] exploit migratory and producer-consumer sharing patterns respectively. These techniques are proposed for distributed shared memory (DSM) systems and when applied to CMPs, require impractical storage (more than 20 MB on a 4-core CMP) to train the streams. Also these techniques store the predicted streams at the DRAM, which results in increased DRAM traffic. In contrast to TMS and SORDS, Somogyi et al.'s spatial memory streaming (SMS) prefetcher [19] is a practical technique that exploits the spatial streams that recur. In general, temporal streaming based techniques get better coverage and accuracy than spatial streaming but at the cost of the complexity of managing the off-chip storage. On the other hand, spatial streaming based techniques are simple, storage efficient and can eliminate the cold-start misses. *In this work, our focus is on on-chip spatial streaming technique and how to make it aware of the cross-core spatial streams*. For the

cross-core spatial stream awareness, we propose cross-core spatial streaming (XStream). XStream predicts and communicates the spatial streams from one MLC prefetcher to the other predicted MLC prefetchers. XStream is based on two observations. First, in case of parallel applications, large fraction of memory space participates in cross-core sharing and communication. Second, there is ample time gap between occurrence of a stream in one core and recurrence of the same in others. *Prefetching techniques oblivious to cross-core streams may potentially reduce the execution time of parallel applications. However by exploiting the cross-core streams, significant reduction in the execution time can be achieved.*

To illustrate the importance of cross-core spatial streams in prefetching, Figure 2 shows the reduction in the execution time that can be achieved with the ideal cross-core MLC prefetchers normalized to the SMS prefetcher. The ideal cross-core prefetcher is an oracle that is aware of all the cross-core spatial streams. It uses local streams along with the cross-core spatial streams for its in-time prefetching. For this experiment, we assume that the ideal cross-core prefetcher has infinite storage to detect and communicate the cross-core streams in-time. Figure 2 shows that for 4 (8) threaded PARSEC [6] applications running on a 4 (8) core CMP, on an average (geomean), more than 23% (19%) reduction in the execution time can be achieved by making the SMS aware of cross-core streams. Given the scope of reduction, we propose XStream. The contributions of this paper are as follows:

- We identify commonality in spatial streams that spread and recur across multiple cores and propose cross-core spatial streaming (XStream), a storage efficient prefetching technique where an MLC prefetcher communicates its spatial streams to the other predicted MLC prefetchers. (Sections 3 and 4)
- We also introduce the notion of cross-core timeliness for the in-time communication of cross-core spatial streams across multiple MLC prefetchers. Cross-core timeliness plays an important role in improving the effectiveness of XStream. (Section 4.4)
- We show the effectiveness of XStream using all the applications from the PARSEC [6] benchmark suite. Compared to the cross-core stream oblivious SMS [19], XStream achieves an average (geomean) speedup of 11.3% and 9% for 4-core and 8-core systems respectively. On a 4-core CMP, XStream gets close to the ideal reduction of 23.3% with the ideal cross-core spatial streamer. (Section 6)

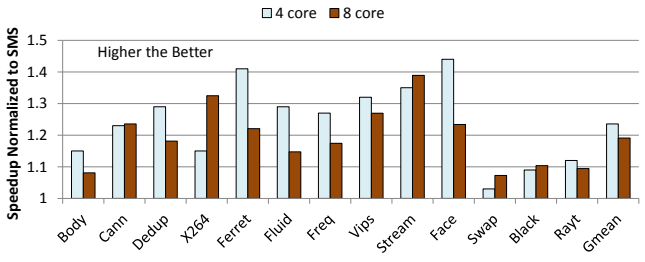

Figure 2: Speedup (Reduction in Execution Time) if all the spatial cross-core streams are communicated in-time.

2. BACKGROUND

In this section, we describe the baseline system design and the sharing patterns observed in the case of shared memory parallel applications. We also describe the design of state-of-the-art SMS prefetcher.

2.1. Baseline CMP Design

We consider a baseline CMP system that consists of multiple cores and multiple DRAM channels. Each core has a private L1 and L2 cache. Last level cache (LLC, which is L3) is shared among all the cores. Each core has prefetchers at the private MLC, which sends prefetch requests to the LLC. In case of a miss at the shared LLC, prefetch request is forwarded to the DRAM. The baseline system uses two hardware prefetchers at the MLC. One for instruction miss streams and the other for data miss streams. In the baseline system, *demand miss streams trigger prefetches, instead of demand access streams as the use of access streams put additional pressure on miss status holding registers (MSHRs) and access streams from multiple cores interfere heavily at the LLC and at the DRAM controllers*. In PARSEC, instruction streams are more predictable compared to data streams. So the system uses a simple stream prefetcher for instructions with 32 stream entries with prefetch degree of 4 and prefetch distance of 64. For data miss streams, the system uses the SMS [19] prefetcher. Both the prefetchers generate prefetch requests which are buffered in a prefetch queue. Prefetch queue (PFQ) is a first-in first-out buffer (FIFO) from which the oldest prefetch request is sent to the MSHRs. The MLC fill buffers are fused with the MSHRs. The baseline system inserts the prefetch responses with non-exclusive cache coherence states.

2.2. Cross-core Sharing Patterns

In shared memory parallel applications, threads share data and code. Based on the demand requests (reads or writes), and the threads that are making these requests, the sharing patterns for each spatial region can be classified into three categories [4]: (1) Read-only: In a given interval, a spatial region exhibits read-only sharing if it is written or read by one thread and then read by one or more threads later.
(2) Migratory: A spatial region exhibits migratory sharing if the threads get exclusive access to a spatial region back to back.
(3) Producer-consumer: A spatial region exhibits producer-consumer sharing if it is written by one thread and then read by threads other than the writer before the writer writes again.

Idea	Hardware Overhead for a 4-core CMP	Hardware overhead : Total MLC size (256 KB*4)	Supports Cross-core Streaming?
TMS [25]	24 MB	24 : 1	Yes
SORDS [26]	22 MB	22 : 1	Yes
STMS [24]	12 MB	12 : 1	Yes
STEMS [18]	6.4 MB	6.4 : 1	No
SMS [19]	256 KB	0.25 : 1	No
XStream	303.5 KB	0.28 : 1	Yes

Table 1: Overview of streaming techniques for a 4-core CMP.

2.3. Spatial Memory Streaming (SMS)

Table 1 compares various streaming techniques in terms of their hardware overhead and support for cross-core streaming. Table 1 shows that, except SMS and XStream, all the other streaming techniques demand impractical storage[1]. *For a private MLC of 256 KB, it is not practical to use additional on-chip hardware of more than 6 MB for hardware prefetching and all the techniques mentioned in the Table 1, except SMS and XStream, require off-chip hardware support*. Compared to SMS, our XStream supports cross-core streaming with additional on-chip hardware of 47.5 KB.

SMS is a hardware prefetching technique that exploits spatial locality present in demand access streams. SMS logically partitions the system memory space into fixed size spatial regions wherein each spatial region consists of multiple cache lines, represented by bits in a bit vector. The training process for a spatial region kicks off on the occurrence of the very first demand access to any address in that spatial region. *The training is completed when any cache line within that region is evicted or invalidated*. SMS uses three hardware tables: (1) filter table (FT), (2) accumulation table (AT) and (3) pattern history table (PHT). FT and AT are implemented as content addressable memories (CAMs). Both FT and AT are part of a single table called active generation table (AGT). On a demand access, filter table filters out the spatial region tag and the spatial signature, which is PC + *offset* (PC/*offset*). PC is the program counter associated with the access and *offset* is the distance of a cache line from the base address of the spatial region tag, in terms of #cache lines. The base address of a spatial region tag is the higher order bits of the address.

On subsequent demand accesses to the cache line addresses within a spatial region tag, accumulation table records the accessed cache line addresses (in the form of bits in the bit vector). Once a spatial region is trained (upon eviction/invalidation of a cache line), active generation table sends the bit vector to the pattern history table. Pattern history table (which is indexed by the spatial signature) stores the predicted bit vector and the spatial signature (PC/*offset*), and uses it to trigger future prefetch requests. At the end of every training phase, the bit vector in the pattern history table is updated to the *set intersection* of the existing bit vector with the bit vector sent by the active generation table. In future, a demand access to a trained spatial signature triggers prefetch requests for the bits (corresponding cache line addresses), which are set in the bit vector. Figure 3 shows the detailed design of SMS prefetcher where a region consists of 4 cache lines represented by 4 bits in the bit vector. On the very first access to an address A+1, filter table stores the spatial region tag (as A) and PC/*offset* (as PC/1). Accumulation table stores and updates the bit vector for the demand accesses as shown in ❶ and ❷ of Figure 3. ❸ of Figure 3 shows the completion of the training process and transfer of the spatial signature (PC/1) along with its trained bit vector (0 1 1 1) from active generation table to pattern history table. *Even-though SMS is an efficient prefetching technique, it falls short for parallel applications because of its cross-core stream obliviousness. In this work we address this.*

[1]We do not use simple stream and stride prefetcher as both of them are not effective for PARSEC benchmark suite.

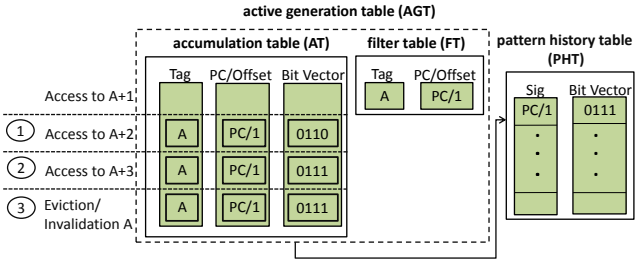

Figure 3: Hardware structure of SMS prefetcher. Sig stands for spatial signature (PC/*offset*).

3. MOTIVATION FOR XSTREAM

Motivation 1: Figure 4 shows the fraction of spatial signatures that recur across multiple cores. On an average, for 4-threaded PARSEC [6] applications, on a 4-core CMP, **80% of the spatial signatures recur in 2 or more cores, which we call cross-core spatial signatures (XCore signatures)**. The rest 20% are intra-core spatial signatures. We observe that a minimum of 160 spatial signatures (in swaptions), and a maximum of 4068 spatial signatures (in facesim), recur at-least 4 times and in more than one core. Also, there is a time gap (on an average, more than 32K clock cycles with minimum of 3 clock cycles, and maximum of 1 million clock cycles) between a particular spatial signature being accessed by one core and later recurring at a different core. This provides opportunity to exploit this behavior. Figure 5 shows the fraction of MLC data misses (which belong to the recurred spatial signatures) satisfied at the LLC, DRAM, and at the remote

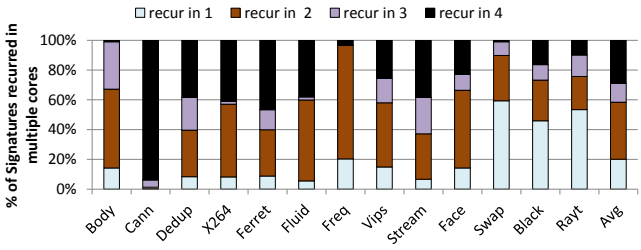

Figure 4: Recurrence of spatial signatures across multiple cores on a 4-core CMP.

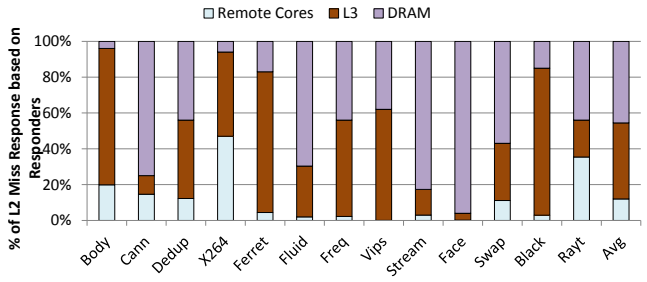

Figure 5: Distribution of MLC demand responses based on the responders on a 4-core CMP.

cores' L1 and L2. From Figure 5, it is clear that 90% of the MLC data misses are satisfied either at the L3 or at the DRAM. **Though 80% of the spatial signatures are cross-core, less than 10% of the data that belongs to these cross-core signatures are present in any of the private MLCs when the spatial signatures recur**.

Motivation 2: Past works related to cross-core streaming target specific kinds of sharing patterns such as migratory [25] and producer-consumer [26], and reduce the read coherent misses. With the current industry trend, more number of cores on a chip share the LLC and the fraction of coherent read misses will be small. **Hence it is important to minimize all types of MLC data misses and not just the coherence misses and it is necessary to handle all kinds of sharing patterns**.

With these motivations, we propose cross-core spatial streaming (XStream), a simple and storage efficient hardware prefetching technique that is aware of cross-core spatial streams.

4. CROSS-CORE SPATIAL STREAMING

In this section, we describe and explain the working of the proposed XStream technique. XStream works in two phases and both the phases work in tandem.

Phase 1: The first phase detects cross-core spatial streams along with the IDs of cores involved in it. In this phase, the MLC prefetchers send their spatial signatures along with their corresponding bit vectors to a shared hardware table named cross-core spatial stream detector (XStream Detector). For every spatial signature, XStream Detector stores these bit vectors along with their core IDs in a temporal order and identifies the commonality in them. In case of a commonality between the bit vectors, it identifies the master prefetcher (*prefetcher of the core in which a spatial signature is trained first before it recurs in other cores*) and the worker prefetcher (*prefetcher of the core that follows the spatial signature of the master core*) associated with a spatial signature.

Phase 2: During the second phase, the future demand misses[2] to the trained signature at the master prefetcher trigger the communication of the spatial streams (in terms of a bit vector) to the predicted worker prefetcher in-time. The

[2]As mentioned in Section 2.1, we use demand miss triggered prefetching.

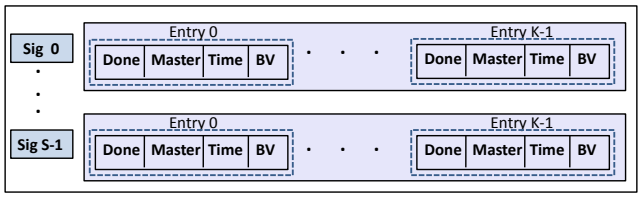

Figure 6: Hardware structure of the XStream Detector. BV stands for bit vector.

worker prefetcher updates its bit vector based on the bit vector communicated by the master prefetcher. Future demand misses at the worker core trigger prefetch requests based on the updated bit vector. *Note that the master prefetcher does not prefetch for the worker prefetcher. The worker prefetcher is solely responsible for sending the prefetch requests for its own private MLC.* Next we describe the XStream Detector.

4.1. XStream Detector

To detect cross-core spatial streams, we use XStream Detector, a hardware table, which is shared by all the MLC prefetchers. We place XStream Detector beside the shared LLC with access latency similar to that of L3 cache. Figure 6 shows the structure of the XStream Detector We connect XStream Detector to all the private MLC prefetchers. XStream Detector keeps track of the trained spatial signatures sent by MLC prefetchers. When a spatial signature is trained and sent by the AGT to the PHT in SMS, it is also communicated to the XStream Detector. XStream Detector maintains S number of signatures wherein each signature stores K entries. Each entry consists of a bit vector, Master field, that stores the core ID of the MLC prefetcher that has sent the trained signature, Time field, that indicates the clock cycle at which a signature is inserted into the XStream Detector and Done field, that indicates whether a bit vector has already participated in the XStream detection process or not. If Done field is set, the bit vector is no more eligible to participate in the XStream detection process. For a given spatial signature, the entries associated with it follow the temporal order (entry 0 is the oldest entry and entry k-1 is the youngest entry). To minimize the noise in the process of detecting cross-core spatial streams, AGT communicates a bit vector to the XStream Detector only if the #1s in that bit vector satisfies a threshold called $numones_{thresh}$.

XStream Detector also uses a threshold called $cross\text{-}core_{thresh}$. For two bit vectors, say BV_0 and BV_1, $cross\text{-}core_{thresh}$ counts the #1s in the set intersection operation be-

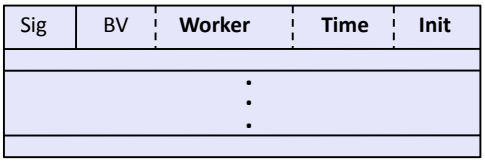

Figure 7: Hardware structure of enhanced PHT. BV stands for bit vector.

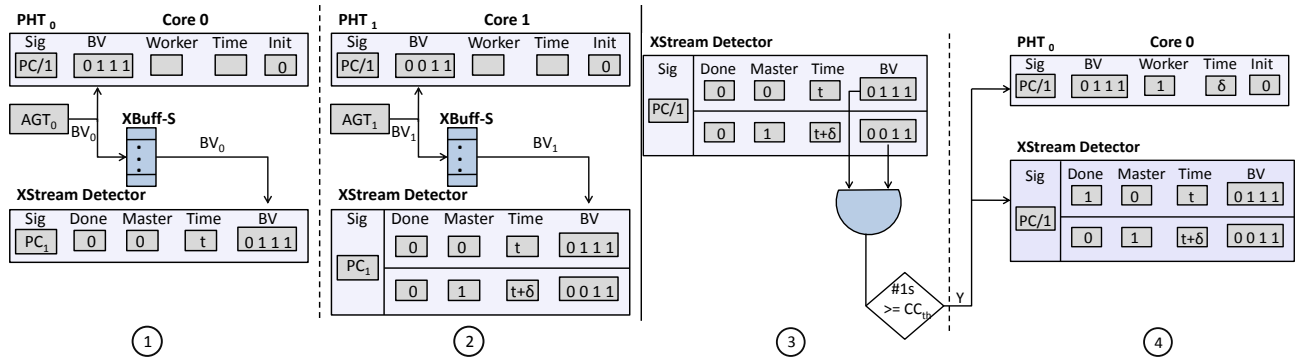

Figure 8: Steps involved in the XStream detection. X_Y denotes X of core Y. CC_{th} is cross-core$_{thresh}$. Each entry of XBuff-P stores the BV along with the predicted worker core ID. Similarly each entry of XBuff-S stores the BV along with the master core ID.

tween BV_0 and BV_1. cross-core$_{thresh}$ quantifies the commonality between two bit vectors. We also modify the PHTs with additional fields. Figure 7 shows the enhanced PHT with three additional fields (Worker, Time and Init fields, shown in bold) along with the baseline PHT. The worker field, added in each entry stores the core ID that follows the spatial signature. Time field indicates the difference in clock cycles between the occurrence of a stream in the master core and the recurrence of the same in the worker core. The third additional field, namely Init field, which indicates whether the master prefetcher has initiated the process of communicating the bit vector to the worker prefetcher. If set, the Init field indicates that the master prefetcher has initiated the process. We now describe the XStream detection phase.

4.2. XStream Detection

Figure 8 shows the steps involved in the XStream detection phase and the contents of the hardware structures at the end of each step. With the help of XStream detector, XStream detects the cross-core spatial streams along with the master and worker prefetchers involved with it. We describe the details of detection phase and communication phase for a dual core CMP (with core 0 and core 1) in the following steps.

Step ❶: In ❶ of Figure 8, for a spatial signature PC/1, AGT of core ID 0 (AGT_0) sends the bit vector to PHT_0 and to the shared XStream Detector through the shared X buffer (XBuff-S). As multiple AGTs can send their bit vectors at the same time, XStream Detector uses a small buffer named XBuff-S, which buffers the bit vectors sent by the AGTs. XBuff-S stores the bit vector along with the sender core ID. XStream Detector stores the bit vector, sender core ID (Master) (as 0) and the Time (t) at which the spatial signature enters XStream Detector.

Step ❷: As shown in ❷ of Figure 8, in the future, say at time t + δ, AGT of core ID 1 (AGT_1) sends its bit vector for the signature PC/1 to PHT_1 and to the shared XStream Detector. It can be observed that, the BVs sent by core ID 0 and core ID 1 are indexed by a common spatial signature PC/1. For every signature, when a bit vector enters XStream Detector, the XStream Detector tries to find commonality between bit vectors and in the process it searches for eligible bit vectors (Done field zero) with the Master field different from the current sender core ID (Master).

Step ❸: In ❸ of Figure 8, for PC/1, XStream Detector searches for the first eligible bit vector from its current entry to entry 0 (for every spatial signature, entry 0 stores oldest entry). In this case, XStream Detector gets a hit for the bit vector sent by the core ID 0 in ❶. Then to find commonality between the bit vectors of different cores (in this example, core ID 0 and core ID 1), it performs the *set-intersection* (bitwise AND) of BV_0 and BV_1 and counts the #1s in it.

Step ❹: If the #1s crosses a threshold called cross-core$_{thresh}$, in ❹ of Figure 8 (b), XStream Detector treats the prefetcher of core ID 0 as the master prefetcher and the prefetcher of core ID 1 as the worker prefetcher of PC/1. XStream Detector communicates the time difference between insertions of bit vectors from two different cores (core ID 0 and core ID 1) to the master core ID (core ID 0). In this case, the difference is δ time units. Along with the time difference, it sends the worker core ID (core ID 1) also. As shown in ❹ of Figure 8 (b), PHT of master core (core ID 0) updates its Worker field with 1 and Time field with δ. At the same time XStream Detector sets the Done bit for the entry maintained by core ID 0.

4.3. XCore Communication

In this section, we describe the next phase of XStream, which is called as the XCore communication phase. The communication phase kicks in when the AGT of a master prefetcher sends its bit vector to a trained cross-core spatial signature (spatial signature with a valid Worker and Time field in the PHT). In case of a valid Worker field, the master prefetcher sends its bit vector to the predicted worker prefetcher (through XBuff-P) based on the Time field[3]. Please note, the master core does not prefetch for the worker core. The worker core is responsible for sending the prefetch requests for its own private MLC. The following steps describe

[3]It helps the master prefetcher gauge how much time it has, to communicate the bit vector to the worker prefetcher before the worker core may start demanding the cache lines.

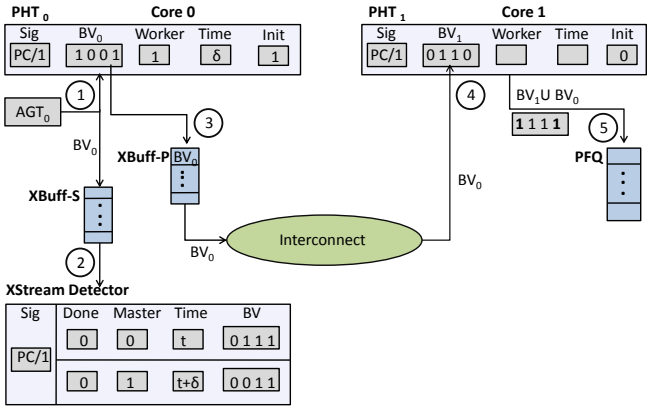

Figure 9: Steps involved in the XCore communication.

the cross-core communication.

A spatial signature is eligible for XCore communication if its Worker field is different from the core ID it belongs to. For example, a spatial signature of core ID 0 is eligible for XCore communication if its Worker field contains any valid core ID except 0. Figure 9 illustrates the steps involved in this phase.

Steps ❶-❸: In ❶ of Figure 9, AGT of core ID 0 communicates the bit vector along with the core ID to the shared XBuff-S. In ❷ of Figure 9, XBuff-S transfers the bit vector along with the core ID to the XStream Detector. ❶ and ❷ of Figure 9 correspond to ❶ of Figure 8 (a). In ❶ of Figure 9, when PHT_0 receives the trained bit vector, it counts the #1s in BV_0. If the #1s crosses $numones_{thresh}$ and the corresponding Worker field is different from its own core ID then it sets the Init bit. In this example, the bit is set for PC/1 in PHT_0. Once the Init bit is set, the master prefetcher can communicate the bit vector to the worker prefetcher. In ❸ of Figure 9, PHT_0 communicates BV_0 to the private cross-core buffer (XBuff-P) of core ID 0 after δ units of time, where δ is predicted time initiated during the detection phase. At a given moment, multiple PHTs can send their bit vectors and their predicted Worker fields for the cross-core communication. To buffer them, we use XBuff-P. Kindly note, we subtract the average communication latency (65 clock cycles, with minimum of 25 clock cycles, and maximum of 83 clock cycles) between PHT_0 and PHT_1 from the δ, at the time of XStream detection. This helps XStream to communicate the streams in-time.

Steps ❹-❺: Based on the interconnect scheduling, PHT_1 gets BV_0 as shown in ❹ of Figure 9. PHT_1 updates its bit vector BV_1 to the *set union*[4] (bitwise OR) of BV_0 and BV_1. In the future, when a demand miss to the trained entry in PHT_1 occurs, it triggers the prefetch requests based on the updated bit vector (set union of BV_1 and BV_0) as shown in ❺ of Figure 9. In this way, the spatial locality, both within and across the cores, are exploited at the worker prefetcher.

4.4. XCore Timeliness

The effectiveness of XStream depends on two factors: 1) how timely a bit vector is communicated from the master prefetcher to the worker prefetcher (XCore timeliness) and 2) the accuracy of XStream detector.

XStream maintains timeliness using the clock domain that drives the LLC, to clock all the per core PHTs, the interconnect as well as the XStream Detector. XStream maintains a set of local registers (local to each PHT), which stores the current clock cycle. It also maintains a shared global register in the XStream Detector. During the XStream detection phase, when an entry is inserted into the XStream Detector, the Time field is set to the current clock cycle based on the global register. In future, when a bit vector from a different core satisfies $crosscore_{thresh}$ as shown in ❹ of Figure 8, the XStream Detector sends the time difference (in terms of the difference in clock cycles) between the insertions of the bit vectors by two different cores to the master prefetcher. In future, when AGT of the master prefetcher sends its bit vector to PHT, it checks for the $numones_{thresh}$. If it satisfies $numones_{thresh}$, it checks the Worker field and the Time field. Based on the current clock cycle, PHT of the master core schedules the communication of the bit vector after Time clock cycles from the current clock cycle and sends it to XBuff-P. For example if the present clock cycle is x, the master core schedules the communication at $x+\delta$, where δ is the content of the Time field. All the registers are reset at fixed intervals. To reduce the hardware overhead, we store the Time field in an encoded format.

Our experiments show that the on an average, the estimates of δ are accurate 67% of the time. A slight difference ($+/-$ 3) between the actual and the predicted encoded δ values leads to an accuracy of 83.2%. Incorrect delta estimates can affect the timeliness and the overall accuracy and a fine grain feedback mechanism (feedback on delta estimates per spatial signature) can be used to further improve the accuracy of delta estimates. It is important to note that the clock domain that is used at the LLC is typically different from the clock domain that drives the processor cores (similar to intel's haswell [1]). Also, this domain is unaffected by power modes of the cores and the C-states at which the cores may be in. This choice of clock domain source for XStream ensures that all the steps in Sections 4.2 and 4.3 are executed with respect to the same reference clock, thus ensuring correct operation no matter what states the cores are in. In our experimental setup, the clock domain for LLC is clocked at the same rate as that of the cores.

4.5. Special Cases

In this section, we highlight some of the special cases that can happen in a CMP with more than 2 cores.

[4]Set union operation between two bit vectors preserve both the local and cross-core bits.

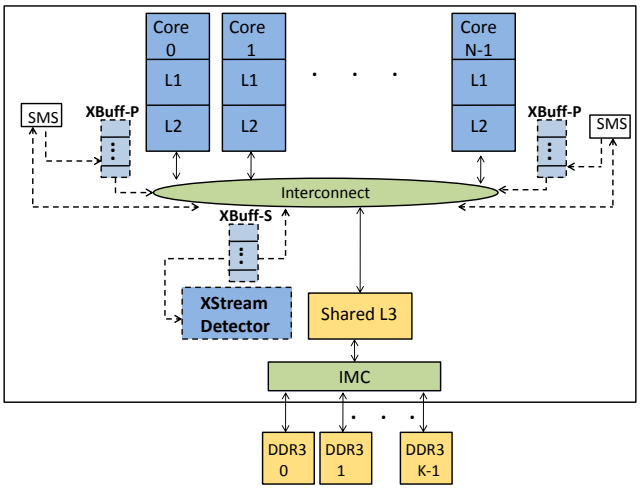

Figure 10: Enhanced CMP with XStream. Dashed blocks and dashed lines are from XStream.

System Address Space	32 bits
Hashed Spatial Signature (PC/*offset*)	16 bits
Size of the Spatial Region	2 KB
Width of the Bit Vector	32 bits
Private Cross Core Buffer(XBuff-P)	8 entries
Shared Cross Core Buffer(XBuff-S)	8 entries
numones$_{thresh}$, cross-core$_{thresh}$	8
Local and Global Clock Registers	16 bits

Table 2: Parameters specific to XStream.

Case 1: In the enhanced PHT, we store the core ID of only one worker core but within a small window of time, a spatial signature can have more than one worker core. XStream resolves this issue implicitly. For example, in a 4-core CMP, for a given spatial signature, with core i as the master core if core j, k and l follow core i then PHTs of core i, core j and core k store core j, k and l in their respective Worker fields. In this way, XStream does not incur extra hardware to store more than one worker core.

Case 2: Section 4.4 describes the mechanism by which the master prefetcher sends its bit vector to the worker prefetcher. But there are scenarios where a spatial signature recurs multiple times at the master core before it recurs at the worker core. In this case, XStream communicates the latest bit vector from the master prefetcher to the worker prefetcher.

Figure 10 shows the new hardware added to the baseline CMP system to make it cross-core streams aware. The additional hardware and the interconnect transactions are shown as dashed blocks and dashed lines.

5. HARDWARE IMPLEMENTATION

5.1. Storage, Energy, and Power Requirements

Table 2 shows the details of the various parameters used in XStream. For all the parameters mentioned in Table 2, we sweep through values in powers of two and we find that the combination of values mentioned in Table 2 achieves the maximum coverage. We use 4K entry PHT, which is organized as an associative cache with tag and data arrays. XStream Detector, the heart of XStream keeps track of 4096 signatures. Both PHT and XStream Detector are indexed using the spatial signature (PC/*offset*). We vary the number of spatial signatures (in powers of two) and compared the coverage achieved by XStream and we find that the table with 4096 signatures gives the best coverage. Table 3 shows the storage, energy, and power requirements of XStream on a 4-core CMP. We use CACTI 6.0 [15] with 45nm technology to obtain the access energy, static power, access time, and the area overhead. As XStream does not augment additional bits in the AGT, the storage requirement of AGT is same as that of the baseline SMS. In PHT, XStream augments a few additional fields. The major chunk of storage overhead comes from the XStream Detector. The ideal cross-core streamer requires 64 MB of additional storage for in-time cross-core communication. The storage requirements of each table are as follows:

	PHT (Per Core)	XStream Detector (Shared)
Size	19.5 KB	33.5 KB
Access Energy	0.024nJ	0.008nJ
Static Power	24.77mW	59.04mW
Access Time	0.61ns	0.24ns
Area (Data Array)	0.13 mm^2	0.19mm^2
Area (Tag Array)	0.07 mm^2	-

Table 3: Storage, Energy, and Power requirements.

- Per core AGT: 16 way, 64 entries (4 sets * 16 entries).
  Tag Store: 64 * (32[5] - 11[6] - log_2 (4)) bits = **0.14 KB**
  Data Store: 64 * (32 + 21[7] + 16) bits = **0.53 KB**
- Per core PHT: 8 way, 4096 entries (512 sets * 8 entries).
  Tag Store: 4096 * (16 - log_2 (512)) bits = **3.5 KB**
  Data Store: 4096 * (32 + 4 + 2 + 1) bits = **19.5 KB**
- Shared XStream Detector: 4096 entries
  Data Store: 4096 * (32 + 16 + 16 + 2 + 1) bits = **33.5 KB**

Data store of the PHT uses bit vector (32 bits), Time (4 bits), Worker (2 (3) bits for 4 (8) cores), and Init (1 bit) fields. The size of PHT in the baseline SMS is (4096 * 32) bits = 16 KB. So XStream incurs additional 3.5 KB per core (14 KB for a 4-core CMP) for PHT. *The total hardware overhead of XStream for a 4-core CMP is* 14 KB + 33.5 KB = 47.5 KB, *which is a little more than* $\frac{1}{6}$*th of a single MLC (256 KB).* XStream Detector table is implemented as an SRAM (not as SRAM cache) with 4096 signatures and each signature consists of 8 entries. Each entry consists of bit vector (32 bits), Signature (16 bits), Time (16 bits[8]), Master (2 (3) bits for 4 (8) cores) and Done (1 bit) fields. XStream Detector is also power efficient with power consumption of less than 60mW. In terms of area, the overhead because of XStream Detector is a little more than $\frac{1}{14}$th of a single MLC (2.74 mm^2). XStream also uses encoders (16 : 4) for encoding the 16 bit clock cycle into

[5] 32 bits because of address space and not because of the bit vector.
[6] log_2 (2 KB), where 2 KB is the size of the spatial region.
[7] spatial region tag.
[8] This unencoded field is different from the encoded 4 bit Time field of PHT.

4 bit Time field and combinational circuits to count the #1s (popcount) present in bit vectors. The encoding logic takes less than a cycle to encode and the popcount circuit takes less than 4 clock cycles[9] to count the #1s in a bit vector. XStream also uses 32 bit registers (one per core at the private SMS and one at the shared XStream Detector) to store the popcounts.

Processor	4-core and 8-core CMP, 3.7 GHz Out of Order
Fetch/Decode/Commit width	8
ROB/LQ/SQ/Issue Queue	192/96/64/64 entries
L1 D/I Cache	32 KB,4 way
L2 Unified Cache	256 KB,8 way
L3 Shared Unified Cache	2(4) MB for 4(8) cores, 16 way
MSHRs	16, 32, 64(128) MSHRs at L1, L2, L3 with 4(8) cores, Targets per MSHR = 4
Cache Line Size	64B in L1, L2 and L3
Replacement Policy	LRU at L1 and L2, EAF [17] at the shared L3
Prefetchers at L2	Data - SMS [19], Inst - Stream Prefetcher, with Prefetch Degree = 4 and Prefetch Distance = 16
Coherence Protocol	MOESI
On-chip Interconnect	Crossbar Topology, Transfer/Arbitration Latency - 4/5 clock cycles, Channel Width - 64B Peak Bandwidth/link - 29.9 GB/s
DRAM Controller	On-chip, Open Row, 64 read & write queues, FR-FCFS scheduler, drain-when-full
DRAM Bus	split-transaction, 800 MHz, BL=8
DRAM	DDR3 1600 MHz (11-11-11) 1(2) channels for 4(8) core CMPs, 2 Ranks/channel and 8 Banks/Rank Peak Bandwidth = 12.8 GB/s

Table 4: Parameters of the simulated system.

5.2. On-chip Interconnect Support

To support XStream, we introduce three new interconnect transactions, which we name as XStream transactions. The first transaction named X-request, is a transaction between MLC prefetchers and the shared XStream Detector. The transaction consists of the bit vector along with the core ID of the sender core. The second transaction named X-response is a response transaction between XStream Detector and the master prefetcher. The transaction consists of the Worker and the Time fields. The third transaction named X-comm is between the MLC prefetchers, which carries the bit vector from the master prefetcher to the worker prefetcher.

5.3. Timeliness

To support in-time transfer of bit vectors, we use multiple local registers at the MLCs and a global register at the XStream Detector. Both local and global registers are 16 bits wide. We reset these registers after every 64K clock cycles. We use encoding logic, which converts a 16 bit clock cycle into a 4 bit Time field of PHT. Time= $\left\lfloor \frac{clockcycle}{4096} \right\rfloor$, where the Time field varies from 0 to 15. When XStream Detector sends the Worker, it also sends the encoded Time. All the PHT entries store these 4 bit Time fields. The Time field of a spatial signature is decremented every 4K clock cycles. Once a Time field becomes zero, PHT of the master core sends the bit vector along with the Worker core ID to its private XBuff-P. We fine tune the width of the local and global registers based on the average interconnect latency and average residence time of cache lines at the MLC to capture the XCore timeliness.

[9] We verify it using Synopsys design compiler for 45nm technology.

6. EXPERIMENTAL EVALUATION

6.1. Methodology

We use gem5 Full System (FS) [7] simulator to simulate 4-core and 8-core CMP. We run PARSEC [6] benchmarks on 4-core and 8-core simulated systems and take a detailed look at the cross-core spatial streams. For our evaluation, we consider the parallel region of each program as our region-of-interest (ROI) with sim-large as the input size. We use the execution time of an application to measure the performance. To minimize the variability in execution time, we pin the software threads to the hardware threads (cores) and we run all the simulations five times and take the average of the execution time as the final execution time. Table 4 shows the parameters of our simulated system. We compare XStream with the baseline SMS prefetcher (which is oblivious to the cross-core streaming) and with the ideal cross-core spatial streamer that has the knowledge of all the cross-core streams.

We also compare XStream with shared SMS (SSMS), wherein the demand miss streams from all the cores are trained by a common AGT (of 128 entries) and the predicted streams are stored at a common PHT (of 8192 entries). Both the AGT and PHT steal a fraction of LLC space to predict and store the streams. SSMS is similar to SHIFT [10]. SHIFT is proposed for the lean core servers in which the instruction streams are common across all the cores. SHIFT does not store the instruction streams of each core locally, SHIFT stores the instruction streams of only one core at the LLC and all the other cores use it for prefetching. SHIFT can also be extended for data streams provided a significant fraction of the data streams are common across all the cores. But in the case of shared memory parallel applications, the degree (#cores involved in XCore communication) of XCore communication varies throughout the parallel phase of an application. The major difference between SHIFT and SSMS is that SHIFT uses the stream of a particular core and all other cores use it for prefetching, but SSMS uses a single AGT and PHT to train and store the streams coming from all the cores. Table 5 shows the programming models, sharing patterns and various metrics related to MLC, LLC, and MLC prefetchers. Kindly note that these values are for a 4-core CMP with SMS as the MLC hardware prefetcher. In Table 5, prefetcher sensitivity refers to the degree of reduction in execution time with SMS over a system without prefetching.

Application	Programming Model	Communication Pattern	Prefetcher Sensitivity	XCore Ratio	MLC MPKI	LLC MPKI	MLC PPKI	Pf Accuracy (in %)
bodytrack	Data	Read Only	Low	High	0.73	0.46	0.15	36
canneal	Irregular	Read Only	Low	High	17.85	9.09	0.8	15
dedup	Pipeline	Migratory, P & C	High	High	1.80	0.90	1.40	59
x264	Pipeline	Read Only	High	High	0.76	0.73	0.22	67
ferret	Pipeline	Migratory, P & C	Low	High	6.52	0.85	4.00	70
fluidanimate	Data	Read Only	High	Medium	1.82	1.78	0.23	47
freqmine	Data	Read Only	High	Medium	4.76	0.04	0.40	50
vips	Data	Read Only	Medium	High	1.73	1.16	0.09	48
streamcluster	Data	Read Only	High	High	11.32	5.55	14.07	46
facesim	Data	Read Only	High	High	1.68	14.69	9.02	95
swaptions	Data	Read Only	Low	Low	0.02	0.09	1.53	86
blackscholes	Data	Read Only	Low	Low	0.22	0.10	0.18	89
raytrace	Data	Read Only	Medium	Low	0.18	0.36	0.14	51

Table 5: Characteristics of PARSEC benchmarks. P & C stands for producer & consumer, MPKI stands for misses per kilo instructions. PPKI stands for prefetch issued per kilo instructions. We define Pf Accuracy in Section 6.5.

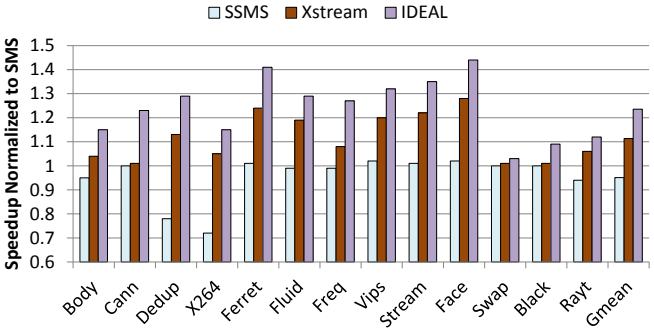

Figure 11: Speedup (Reduction in Execution time) on a 4-core CMP.

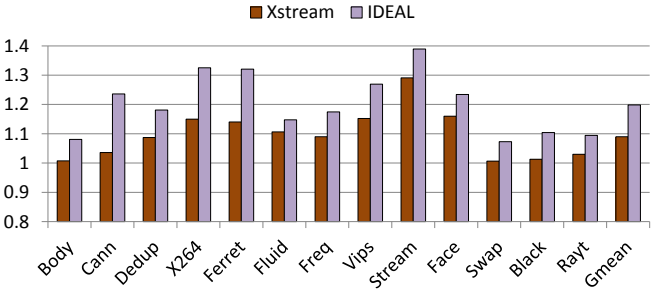

Figure 12: Speedup (Reduction in Execution time) on an 8-core CMP.

The sensitivity of an application is considered to be High if the reduction is more than 10%, Medium if the reduction is in between 2% to 10%, and Low if the reduction is less than 2%. XCore ratio refers to the ratio of XCore signatures to the sum of XCore signatures and intra-core signatures. This ratio is only for the spatial signatures that recur at-least 4 times. XCore ratio is considered to be High if the ratio is greater than 0.85, Medium if the ratio is in between 0.60 and 0.85, and Low if it is less than 0.6.

6.2. Execution Time

We evaluate XStream on both 4-core and 8-core systems. Figure 11 compares the execution time of SSMS, XStream, and the ideal cross-core prefetcher normalized to SMS on a 4-core CMP. On an average (geomean), compared to SMS, XStream reduces the execution time by 11.3%. Applications such as dedup, vips, streamcluster, ferret, and facesim are the major benefactors with reductions of 13%, 20%, 22%, 24%, and 28% respectively. Cores associated with these applications communicate heavily as compared to rest of the PARSEC applications. SSMS does a poor job compared to SMS because of the cross-core noise that comes from non-recurring non-cooperative streams. We use 256 and 8192 entries (provides the best coverage) for the shared AGT and PHT respectively in SSMS. swaptions and blackscholes have negligible cross-core spatial streams and their performance with SSMS is similar to their performance with SMS. The ideal cross-core spatial streamer reduces the execution time by 23.13%. Figure 12 compares the reduction in execution time with XStream and with the ideal cross-core prefetcher normalized to SMS on an 8-core system. *We do not compare XStream with SSMS because SSMS causes increase in the execution time for all the benchmarks except* swaptions and blackscholes. For 8-core CMP, on an average (geomean), XStream reduces the execution time by 9.4%. streamcluster, x264, ferret, and vips are the major benefactors with reduction of 29.09%, 15%, 14% and 15.2% respectively.

6.3. Analysis

Table 5 shows that the applications such as dedup, x264, facesim and streamcluster are highly sensitive to prefetching and their XCore ratio is also high. XStream delivers its best for these applications except for x264. For x264, XStream increases the LLC pollution (eviction of demand lines because of prefetching, that are later used) by 6%.

facesim uses an iterative Newton-Raphson method over a matrix, which is used by multiple cores. Hence, it has high cross-core spatial correlation. dedup and ferret use pipeline programming model where each stage of the pipeline is occupied by one or more threads and each stage communicates heavily to its successor stage. Applications such as canneal also shows high XCore ratio but XStream fails to deliver performance because of irregular and non-deterministic behavior of spatial signatures that recur.

Applications such as swaptions and blackscholes do not

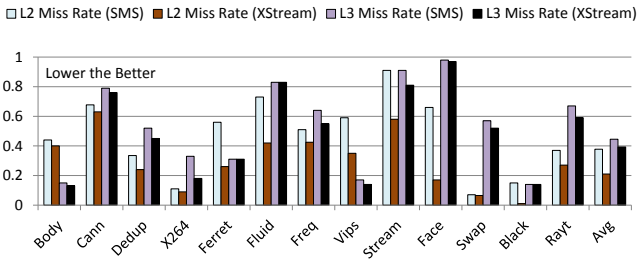

Figure 13: Miss rate on a 4-core CMP.

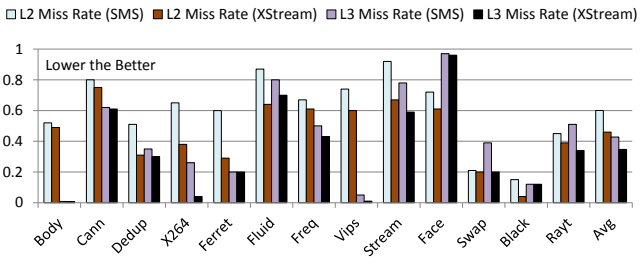

Figure 14: Miss rate on an 8-core CMP.

communicate heavily. blackscholes uses data parallel model and it partitions its data structures in such a way that each thread works in its own partition, which limits the XCore communication. So XStream reduces the execution time of swaptions and blackscholes only by 2% and 3%.

From all the applications mentioned in Table 5, ferret has an interesting trend. ferret is an application which belongs to the Low prefetch sensitive class but with XStream, the execution time reduces significantly. Figure 11 shows, there is a gap between XStream and the ideal cross-core prefetcher. One of the factors that limits XStream as compared to the ideal cross-core prefetcher is the number of harmful prefetches issued by XStream. To bridge the gap between the ideal cross-core prefetcher and XStream, we propose feedback-driven XStream in Section 6.9. On a 4-core system, applications such as dedup, ferret, x264 and facesim suffer from this trend. *For these applications, more than 88% of the harmful prefetches are issued from 37 spatial signatures. Out of these 37 spatial signatures, only 9 of them issue useful prefetches.*

On an 8-core system, streamcluster has an interesting behavior as it communicates consistently with other threads but after a fixed window of 128K clock cycles, the communication becomes intensive. This happens in case of fluidanimate also.

Impact of XCore Timeliness: XCore timeliness plays an important role in improving the effectiveness of XStream. To quantify the effect of timeliness, we compare XStream with and without XCore communication. In case of XStream without in-time communication, the AGT of the master core sends the bit vector to the PHT of the master core and communicates the same to the worker core also. Compared to XStream with in-time communication, on an average (geomean), XStream without in-time communication increases the execution time by 6.5% (7.18%) on 4-core (8-core) system. On a 4-core system, applications such as dedup and facesim show an increase of 35% and 29.2% in their respective execution times as compared to XStream with in-time communication. The XCore timeliness plays a bigger role for the 8-core system. Applications such as facesim, dedup, fluidanimate and x264 are sensitive to the in-time communication of the streams.

6.4. MLC and LLC Miss rates

Figure 13 compares the reduction in MLC and LLC miss rates for 4-core system with SMS and XStream. On an average, XStream reduces the MLC miss rate (average across all the private MLCs) from 37% (with SMS) to 20%. Similarly at the LLC, on an average, XStream reduces the miss rate from 44% (with SMS) to 39%. *Please note, miss rate here indicates only the demand miss rate and it does not account for additional prefetch requests and their misses at the LLC.* Figure 14 shows the reduction in MLC and LLC miss rates for an 8-core system. On an average, XStream reduces the MLC miss rate from 60% (with SMS) to 46% and LLC miss rate from 42% (with SMS) to 34%. The reduction in miss rates help XStream in reducing the execution time.

6.5. Accuracy, Coverage, and Pf Traffic

Prefetch accuracy (in %) of XStream is defined as

$$\text{Prefetch Accuracy} = \frac{\text{\#Pf Hits}}{\text{\#Pf Issued}} * 100 \quad (1)$$

wherein Pf Hits is the demand hits to the prefetched cache lines and Pf Issued is the prefetch requests sent to the LLC because of XStream only (we do not add the prefetch requests issued because of the local streams). To find the accuracy, we augment each cache line with an additional bit that is set if the line is brought in to the cache because of cross-core streaming (not because of the intra-core spatial streaming). To achieve this, we change the PHT by augmenting an additional bit for every bit in the bit vector. This additional bit is set whenever a bit is set in the bit vector because of the XCore communication.

Prefetch coverage (in %) of XStream is defined as

$$\text{Prefetch Coverage} = \frac{\text{\#Pf Hits}}{\text{\#Demand Misses} + \text{\#Pf Hits}} * 100 \quad (2)$$

where Demand Misses stands for data misses at the MLC. Additional Pf traffic is the additional prefetch requests issued by the XStream as compared to SMS. *Kindly note that this traffic does not include the communication between PHTs and communication between a PHT and the XStream Detector.* Figure 15 shows that, on an average (not geomean), for 4-core CMP, XStream delivers an accuracy of 63.7% and a coverage of 39.61% with additional prefetch traffic of 19.07%. Applications such as swaptions and blackscholes have high accuracy

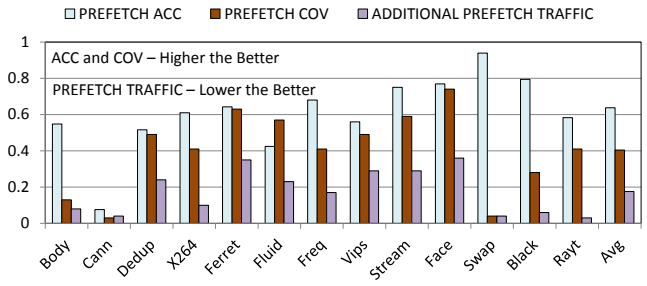

Figure 15: Accuracy, Coverage, and additional Pf Traffic in % of XStream on a 4-core CMP.

with low coverage. The reason behind this behavior is the #prefetches issued by XStream, which is very low (additional 10% compared to SMS). Figure 16 shows the accuracy, coverage, and additional traffic because of XStream on an 8-core CMP. We also quantify the additional interconnect bandwidth consumed because of XStream detection and communication. On an average, for 4-core and 8-core CMP, XStream consume additional interconnect bandwidth of 18% and 15.53% respectively. *This additional bandwidth is because of the communication of streams from PHTs to the shared XStream Detector and from master prefetcher to worker prefetcher.* Applications such as vips, dedup, and facesim consumes an additional bandwidth of more than 40% as compared to the baseline SMS. Note that with XStream, all the applications from PARSEC do not saturate the interconnect bandwidth. Table 6 summarizes the overall results of XStream for 4-core and 8-core systems.

6.6. Comparison with TMS and SORDS

We do not compare XStream with TMS [25] and SORDS [26] because as compared to the baseline SMS, both of them perform poorly with the practical on-chip hardware budget of 256 KB per core. Both [25] and [26] perform better than SMS when we increase the hardware budget to 2 MB per core (with hardware budget of less than 2 MB per core, TMS is unable to beat SMS).

6.7. Scalability

When we move from a 4-core to a 8-core system, the average reduction in the execution time drops from 11.3% to 9.4%. The scalability of XStream depends on the factors which deal with the application behavior such as 1) degree of XCore communication and 2) application scaling.

degree of XCore communication: If more number of threads participate in crosscore streams, we can anticipate an increase in Xcore communication. For applications such as x264 and streamcluster, this is the case. On moving to a 8-core system from a 4-core system, the degree of Xcore communication increases for these applications resulting in reduction of execution time.

application scaling: XStream scales poorly if the ratio of the number of software threads spawned to the number of cores

	Reduction in Execution Time	Additional Pf Traffic	Additional Interconnect Bandwidth
4-core	11.3%(max of 24%)	19.07%	18%
8-core	9% (max of 29.09%)	17.69%	15.53%

Table 6: Summary of results for XStream.

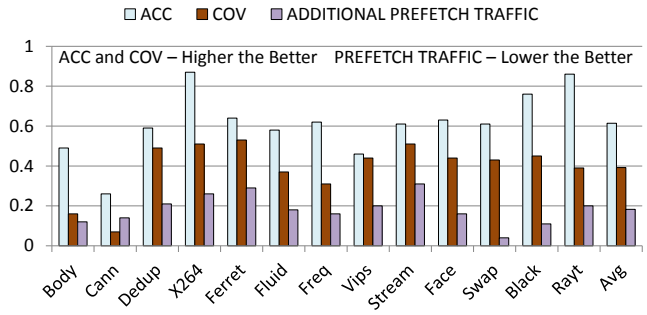

Figure 16: Accuracy, Coverage, and additional Pf Traffic in % of XStream on an 8-core CMP.

is high. For instance, dedup and ferret spawn $2+3n$ threads and $2+4n$ threads respectively in an n core system. The other factor that can limit the true scaling of an application is the number of synchronization primitives (such as locks) used and how frequently synchronization is required. The number of locks acquired by a particular thread per number committed kilo instruction (LPKI) is a clear indicator of application scaling. For instance fluidanimate(1.36), freqmine(0.03) and vips(0.01) have orders of magnitude higher LPKI than others. Naturally, these applications do not scale well with core count and see a drop in the savings of execution time.

organization of XStream framework: Another important factor related to scaling is to understand how scalable the hardware organization of XStream is for 4-core and 8-core systems, the shared XStream detector with an on-chip interconnect such as crossbar does a good job but for many-core systems (16-core and above), shared XStream detector with a crossbar may become a scalability bottleneck. On an average, on each link, XStream uses 57% of the spared interconnect bandwidth in XStream detection and communication but with the increase in the core count, the bandwidth utilization goes down because of more number of cores compete for XStream detector. As part of our future work, we would like to explore a distributed XStream, wherein chunks of XStream will be distributed with the LLC banks and also over a scalable interconnect such as ring.

6.8. Sensitivity Studies and Tradeoffs

We analyze the sensitivity of XStream to the #signatures present in XStream Detector and to the size of MLC and L1 on a 4-core CMP. Our baseline evaluation has 4096 signatures at the XStream Detector with 256 KB of MLC and 32KB of L1 D-cache. With 2048 signatures, the coverage of XStream decreases by 18% for applications such as canneal, fluidanimate, and facesim. These applications have more than 4000 distinct XCore signatures. For MLC size of 64 and 128 KB,

XStream reduces the execution time by 13.27% and 15.24% respectively. In case of MLC size of 512 KB, XStream reduces the execution time by 10.91%. If one is allowed to change only L2's size to increase performance, the size of L2 has to be increased to at-least 1MB/core (512KB/core is enough for applications such as bodytrack and blackscholes) to match the performance of XStream. Our experiments also show that the increase in size of L1 from 32KB to 64KB/128KB provides only a marginal improvement of less than 1.5% over the baseline.

6.9. Feedback Driven XStream

The performance of XStream can be further improved with prefetch throttling techniques such as feedback directed prefetching (FDP) [20]. FDP throttles the prefetcher by changing the prefetch degree and prefetch distance based on prefetch metrics such as accuracy, coverage, and cache pollution. As SMS does not use parameters such as prefetch degree and prefetch distance, simple integration of FDP with XStream is not possible. So we use the prefetch accuracy to enable/disable cross-core prefetching. For every prefetch request generated because of the cross-core bits, we add an additional bit with the prefetch requests and the responses. We also add an additional bit for each cache line brought into the cache because of cross-core streaming. After every 64K clock cycles, we enable XStream if the prefetch accuracy is more than 0.45 else we disable it. On an average (geomean), feedback driven XStream reduces the execution time by 14.25% over SMS. Due to space constraints, we do not present the performance results in detail. Feedback driven XStream reduces the Pf traffic by 7.45% over XStream. XStream when integrated with pollution filtering mechanism [28] reduces the execution time by 12.38% and 10.17% on 4-core and 8-core system respectively.

6.10. Effect On Multiprogrammed Workloads

We assess the effectiveness of the proposed XStream by evaluating it in a multiprogrammed scenario also. In multiprogrammed setup, multiple independent applications run independently with minimal or no XCore communication. The XCore communication comes from the operating system code and not from the independent applications. We test the robustness of XStream by simulating 32 4-core and 16 8-core workload mixes from SPEC 2006 benchmark suite [3] and in terms of weighted speedup, XStream performs similar to SMS.

7. RELATED WORK

TMS: Wenisch et al.'s TMS prefetcher [25] exploits temporal correlation among coherent miss streams at the L1 D-cache that spread across the cores. TMS minimizes the coherent read misses in DSMs. The idea targets commercial workloads and scientific applications and it works well for applications with migratory sharing pattern. The major disadvantage of this approach is the large hardware overhead (24 MB on an 4 core CMP) and frequent use of cache directories. Wenisch et al.'s STMS [24] implements TMS by storing the streams at the DRAM Though 48 MB is a small fraction of memory for large scale servers, it is not the case for CMPs. Modern day CMPs use multiple levels of cache with the last level cache (LLC) being shared by all cores. which results in additional DRAM traffic and consumption of additional L2-L3 bandwidth. In an 8-core CMP, on an average, compared to SMS, STMS increases the DRAM traffic by 51%. Wenisch's Ph.D. thesis [23] proposed an extension of TMS to CMPs and showed the effectiveness of TMS on a 4-core CMP. Similar to TMS in DSMs, TMS in CMPs incurs hardware overhead of 9 MB per core.

SORDS: Wenisch et al.'s SORDS prefetcher [26] exploits the producer-consumer sharing pattern. The main observation of SORDS is the correlation between the producer and consumer cores. Based on this observation, SORDS stores the streams generated by the producer that is initiated by a write request. On a future demand miss at the consumer core, SORDS forwards the streams produced by the producer core. Similar to TMS, SORDS incurs additional storage of 22 MB on a 4-core CMP.

STeMS: Somogyi et al.'s spatio temporal memory streaming (STeMS) [18] prefetcher is based on the temporal correlation among streams within a spatial region. STeMS correlates the temporal streams that are local to a particular core and it is oblivious to the cross-core streams. STeMS incurs storage overhead of 6.52 MB on a 4-core CMP. Also, TMS and SORDS consult the cache coherence directory to decide which core's main memory holds the temporal streams.

Cross-core Stride Prefetcher: Panda and Balachandran proposed a cross-core stride prefetcher [16] , which exploits spatial strides (based on the demand misses at L1) present within and across the cores. The idea is similar to Bhattacharjee et al.'s ICC [5], which is an inter-core cooperative prefetching framework proposed for TLBs. The authors evaluated their framework for a 2 level cache hierarchy. Though cross-core strides dominate the demand misses at the L1, MLC misses do not contribute to cross-core strides. In case of PARSEC, on an average, cross-core strides are of more than 128 cache lines. In applications such as blackscholes and facesim, on an average, 42% of strides cross the virtual page boundaries and cross-core strides contribute negligibly in applications such as streamcluster and ferret.

Kamruzzaman et al. proposed inter-core prefetching(ICP) [9], which uses helper thread(s) to prefetch data into a core and then it migrates the execution to the prefetched core. Apart from cross-core streaming techniques, there are data forwarding techniques which forward data from one core to another. Data forwarding method, proposed by Koufaty et al. [11] uses compiler hints to forward data across the cores. The effectiveness of these techniques depend on the cache size because on an average, on a 4-core CMP, with MLC of 256

KB, 82% of the time, data forwarded by one core gets evicted before the same gets consumed by the other core. Recently, Manivanan et al. proposed a consumer initiated forwarding technique [14] for task-based programming models. Koufaty et al. compared data prefetching techniques with the data forwarding techniques in [12] . Other techniques such as Data Marshaling [21] eliminates cross-core write misses with the help of marshal instructions in staged executionmodels. Techniques such as stealth prefetching [8] uses coarse grain coherence tracking to prefetch aggressively in broadcast based shared multiprocessors.

There are proposals such as PACMAN [27] and PADC [13], which provide prefetch awareness at the LLC and at the DRAM controller. We do not evaluate XStream with PACMAN and PADC but we believe prefetch awareness will boost the performance of XStream further as the awareness techniques are orthogonal to the baseline XStream. Recently, Cao at al.'s [22] proposed a framework called ICHAT, which assists inter-cache data transfer for heterogeneous chip multiprocessors. Exploring XStream with ICHAT is a promising future work.

8. CONCLUDING REMARKS

This paper proposed cross-core spatial streaming (XStream), a prefetching technique for the MLCs that exploits cross-core spatial streams. With negligible hardware overhead, XStream results in 11.3% and 9% reduction in execution time for 4-core and 8-core CMPs, respectively. As part of future work, we would like to extend XStream for multi-chip configurations, where each chip will have 4-cores to 8-cores and an XStream framework of one chip can communicate spatial streams to XStream frameworks of other chips. But in multi-chip configurations, all the cores do not share a single monolithic LLC and the placement of the XStream Detector will play an important role.

9. Acknowledgments

The authors thank the anonymous reviewers and the Shepherd of this paper Ravishankar Iyer for their valuable suggestions. Special thanks to Anju Moosad, T V Kalyan, Tripti Warrier, Akanksha Jain, Vivek Seshadri, Madhu Mutyam and Mainak Chaudhuri for their insightful feedback on the initial versions of this paper. Biswabandan is supported by the TCS Ph.D. fellowship. This work is also supported by the IBM India SUR grant. The authors gratefully acknowledge the support of TCS and IBM.

References

[1] http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf.

[2] Intel 64 and ia32 architecture software developer's manuals. http://www.intel.com/products/processor/manuals/.

[3] Spec benchmark suite. http://www.spec.org/cpu2006/.

[4] N. Barrow Williams, C. Fensch, and S. Moore. A communication characterisation of splash-2 and parsec. In *IISWC*, 2009.

[5] A. Bhattacharjee and M. Martonosi. Inter-core cooperative tlb for chip multiprocessors. In *ASPLOS*, 2010.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[7] 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 Comput. Archit. News*, 39(2), Aug. 2011.

[8] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. In *ASPLOS*, 2006.

[9] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *ASPLOS*, 2011.

[10] C. Kaynak, B. Grot, and B. Falsafi. Shift: Shared history instruction fetch for lean-core server processors. In *MICRO*, 2013.

[11] D. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(12), 1996.

[12] D. Koufaty and J. Torrellas. Comparing data forwarding and prefetching for communication-induced misses in shared-memory mps. In *ICS*, 1998.

[13] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware dram controllers. In *MICRO*, 2008.

[14] M. Manivannan, A. Negi, and P. Stenstrom. Efficient forwarding of producer-consumer data in task-based programs. In *ICPP*, 2013.

[15] R. B. N. Muralimanohar and N. P. Jouppi. Cacti 6.0: A tool to understand large caches, technical report, university of utah and hewlett packard laboratories. 2007.

[16] B. Panda and S. Balachandran. Hardware prefetchers for emerging parallel applications. In *PACT*, 2012.

[17] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.

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

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

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

[21] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *ISCA*, 2010.

[22] J. G. T. Cao and B. Beckmann. ichat: Inter-cache hardware-assistant data transfer for heterogeneous chip multiprocessors. 2013.

[23] T. F. Wenisch. Temporal memory streaming. phd thesis, carnegie mellon university. 2007.

[24] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *HPCA*, 2009.

[25] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *ISCA*, 2005.

[26] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, C. Gniady, A. Ailamaki, and B. Falsafi. Store-ordered streaming of shared memory. In *PACT*, 2005.

[27] C. J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. Pacman: Prefetch-aware cache management for high performance caching. In *MICRO*, 2011.

[28] X. Zhuang and H. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP*, 2003.