2013 IEEE 33rd International Conference on Distributed Computing Systems

# Cache Topology Aware Mapping of Stream Processing Applications onto CMPs

Fang Zheng[1], Chitra Venkatramani[2], Rohit Wagle[2], and Karsten Schwan[1]

[1]College of Computing, Georgia Institute of Technology, Atlanta, Georgia
[2]IBM T. J. Watson Research Center, Yorktown Heights, New York

***Abstract*—Data Stream Processing is an important class of data intensive applications in the "Big Data" era. Chip Multi-Processors (CMPs) are the standard hosting platforms in modern data centers. Gaining high performance for stream processing applications on CMPs is therefore of great interest. Since the performance of stream processing applications largely depends on their effective use of the complex cache structure present on CMPs, this paper proposes the StreamMap approach for tuning streaming applications' use of cache. Our major idea is to map application threads to CPU cores to facilitate data sharing AND mitigate memory resource contention among threads in a holistic manner. Applying StreamMap to the IBM's System S middleware leads to improvements of up to $1.8\times$ in the performance of realistic applications over standard Linux OS scheduler on three different CMP platforms.**

***Keywords*-Data Stream Processing; Cache Topology; Thread Mapping; IBM Infosphere Streams**

## I. Introduction

Data Stream Processing is an important class of data intensive applications in the "Big Data" era. Providing real-time data analytics capabilities to extract insights from live data streams, it has been applied to many application domains, including finance & trading, image processing, network intrusion detection, and environmental monitoring.

An important question to ask about stream processing applications is their performance on common hardware platforms like the modern chip multiprocessors (CMPs) now used across the entire spectrum of portable devices to high end server systems. Today's multicore architectures are equipped with complex cache hierarchies. On one hand, multi-level caches are used to alleviate the two-order-of magnitude gap in speed between CPU and DRAM, making maximizing cache utilization a significant factor for application performance. On the other hand, cores share certain memory resources with each other, including last level cache, hardware prefetch unit, front side bus and memory controller. While sharing those resources between cores can be constructive for data sharing (e.g., to facilitate data reuse in shared cache and reduce cache coherency traffic), such sharing can also cause severe interference due to contention on those resources [18]. Therefore, judiciously managing application's interaction with cache structure is of great importance to achieve high performance on CMPs.

This paper argues that thread-to-core mapping is an effective way to control how a stream processing application interact with CMP's caches. This is because how application threads are placed onto cores largely determines how memory resources are shared between threads. This, in turn, impacts not only the efficiency of data messaging and sharing between threads, but also the intensiveness of resource contention between threads for their private state data. In fact, our experiments show up to $3\times$ difference in performance between different thread-to-core mappings for some streaming applications.

Unfortunately, current operating systems are largely ignorant of data sharing and conflicts in resource demands among application threads, and assign threads to cores based on core idleness, often resulting in sub-optimal and highly varying application performance. Although there has been previous work on mapping applications onto CMPs [14, 24, 34], their effectiveness for stream processing applications is not well understood. Besides, most existing solutions do not consider data sharing and resource contention relationships between threads in a holistic manner and fall short for streaming applications with complicated inter-thread relationships.

This paper proposes *StreamMap*, an approach that makes streaming applications cache topology aware to obtain high performance on CMP architectures. StreamMap assigns application threads to cores so that (i) constructive inter-thread data sharing is respected while (ii) negative resource contention between threads is reduced. It uses offline profiling to collect relevant information about threads' cache behavior and derives high-quality thread-to-core mappings. The mapping is enforced when launching the application onto target machine for production run. StreamMap is transparent to user programs and operates at user level without modifications to operating systems or hardware.

This paper makes the following contributions:

1) *Behavior characterization.* It characterizes the cache behavior of streaming applications and quantifies the impact of cache topology on streaming application performance. Findings include that (1) data sharing and messaging between threads is sensitive to CMP's cache topology, (2) the resource demands of several widely-used streaming operators is quite diverse and resource contention between operators can cause severe performance loss, and (3) the default Linux OS thread scheduler fails to schedule applications in ways that efficiently use CMP memory hierarchies.

2) *Cache topology aware thread mapping.* It proposes a

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.13

holistic mapping policy which improves data reuse for better cache utilization and reduces negative resource contention. Its user-level implementation facilitates its adoption by the stream processing middleware.

3) *Realistic experimental evaluation.* The approach is implemented within IBM's System S middleware and is evaluated with two real-world applications in the financial and scientific domains, respectively. Performance evaluations on three different Intel architectures show consistent, up to $1.8\times$ performance improvements for cache topology aware mapping versus unaware techniques.

The remainder of the paper is organized as follows. Section II presents background information on the System S middleware and CMP cache topology. Section III motivates our work by demonstrating the significance of a CMP's cache topology to the performance of streaming applications. Section IV describes details about the cache topology aware mapping techniques. Section V shows the performance improvements of two realistic applications brought by the StreamMap approach. Section VI reviews related work, and Section VII concludes the paper.

## II. Background

### A. IBM System S Middleware

Our work is based on System S [9] (commercialized as IBM Infosphere Streams). System S is an industry-leading middleware enabling high throughput, low latency stream processing. As shown in Figure 1, it provides a programming language, a compilation framework, and an execution runtime to implement and run streaming application in a distributed environment. The Streams Processing Language (SPL) is the high-level declarative language supporting the operator-stream programming model. Operators can be primitive ones supported by System S, reused from existing toolkits, or implemented by programmers on their own in C++ and/or Java. With SPL, operators are composed into a dataflow graph by defining the streams that connect them. Streams carry a continuous stream of tuples with fixed schema. The SPL compiler compiles the SPL source code to generate C++ code, which is then compiled by native compilers to generate deployable binary executables. The System S runtime provides the execution environment for streaming applications and handles job scheduling, monitoring, and fault-tolerance.

System S provides language support for multi-threading an SPL program. Programmers can specify an input port of an operator as a "Threaded Port", for which the System S runtime will create a thread to handle the incoming tuples from that port and execute the subgraph of operators rooted from that input. In the sample program shown in Figure 1, the operator *FinalData* has its input port configured as a Threaded Port. The SPL compiler will accordingly introduce a separate thread to handle incoming tuples from this input and drive the execution downstream.

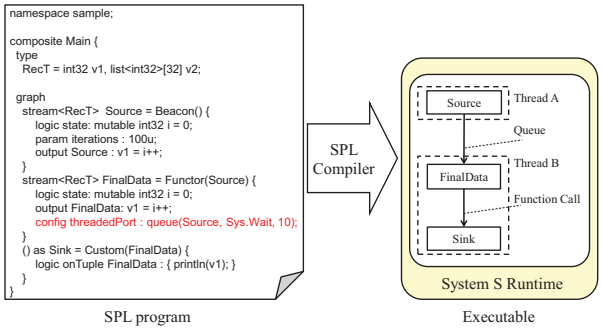
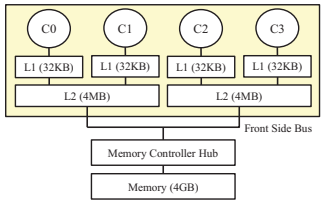
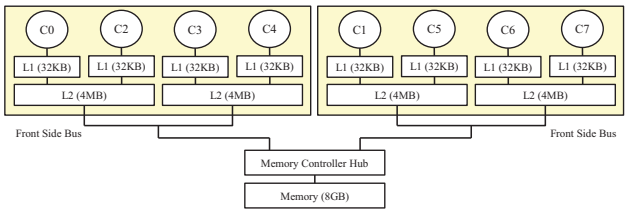


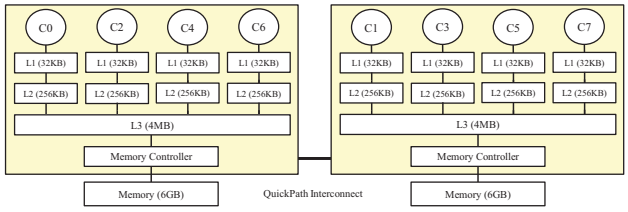

Figure 1. System S Middleware.




(a) 4-core UMA (Xeon X5355 2.66GHz)




(b) 8-core UMA (Xeon E5320 1.86GHz)




(c) 8-core NUMA (Xeon E5506 2.13GHz)

Figure 2. Three Different CMP Cache Topology.

When targeting a multi-core platform, Threaded Ports and in-memory queues provide the necessary mechanism to multi-thread SPL programs and exploit various forms of parallelism inherent in the programs. There are additional ways to introduce threads, however. One common case is that each source operator (those that does not have an input port) has its own thread to drive the execution of operators rooted from the source operator. For example, in Figure 1, there is one thread to drive the execution from operator *FirstSource* to downstream operators until it encounters a threaded port. Other places where additional threads are created and participate in the execution of stream graphs include threads associated with time-based windows and those introduced by the underlying transport layer.

While inter-thread communication is made explicit at the SPL level through the Threaded Port language construct, actual data movement is instantiated through an in-memory

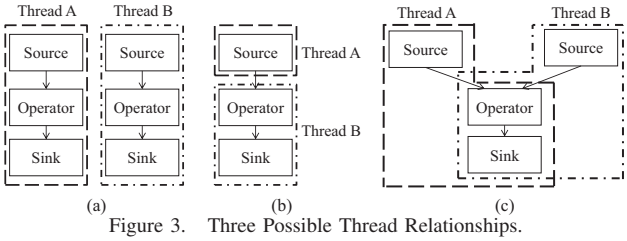



Figure 3. Three Possible Thread Relationships.

queue between producer and consumer threads. The queue is a FIFO, lock-free, circular buffer, with various optimizations to reduce cache coherency traffic (for more details of the queue implementation, we refer readers to [12]).

### B. CMP and its Cache Topology

Chip Multiprocessors (CMPs or simply multi-cores) have been the standard hosting platform for enterprise and scientific computing workloads and are also becoming pervasive in personal and mobile computing environments. Modern CMPs typically feature deep and complex memory hierarchies. Figure 2 shows the cache topology of three machines equipped with different Intel Xeon processors. The first (Figure 2(a)) is a 4-core Xeon X5355 where all cores reside in a single socket. Each core has its own L1 data and instruction cache, and each pair of cores share one L2 cache. All four cores share the Front Side Bus (FSB) and memory controller and have access to DRAM with equal cost (known as UMA, i.e., Uniform Memory Access). The second machine (Figure 2(b)) has two quad-core Xeon E5320 processors. In each socket, each core has its own L1 cache and shares L2 cache with another core. All 8-cores share the FSB and memory controller and have access to DRAM in a UMA fashion. Different from the former two machines, the third machine (see Figure 2(c)) has a NUMA (Non-Uniform Memory Access) architecture, where there are two quad-core processors, each with its own local on-chip memory controller. Accessing data in local memory banks is faster than accessing data in remote memory banks.

## III. Challenges in Mapping Streaming Applications onto CMPs

This section uses experimental measurements to establish the fact that on multicore platforms, thread-to-core mappings can have a significant impact on streaming application performance, motivating the need for carefully determining its best thread-to-core mapping.

### A. Multi-threaded Streaming Applications

There are various forms of parallelism inherent in the operator graphs of streaming applications, typically resulting in a partitioning of operators among threads and a temporal scheduling of how operators are run within each thread.

As mentioned in Section II, with System S, such parallelization is expressed with Threaded Ports introduced into the proper locations in the stream graph. This results in the runtime creation of threads that each execute a group

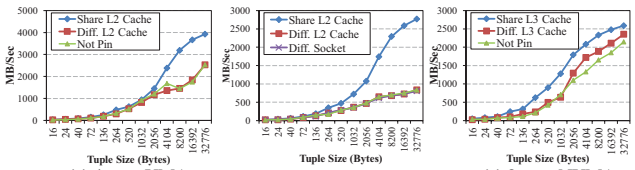



Figure 4. Inter-Thread Communication Performance.

of operators in a per-tuple, depth-first manner. Threads within the same streaming program may have three possible relationships with each other, as shown in Figure 3.

*Independent:* threads progress independently from each other, without communicating or sharing any common operator, as shown in Figure 3 (a).

*Producer-Consumer:* as shown in Figure 3 (b), the producer thread copies tuple into the queue associated with the Thread Port and the consumer thread directly operates on tuples in the queue.

*Operator-Sharing:* two threads share one or multiple operators, as shown in Figure 3 (c); they synchronize through a mutex lock to execute the shared operator(s).

This paper assumes that a streaming application is multi-threaded by programmers or with automated mechanisms like graph partitioning-based operator fusion [15]. Given such an application, our goal is to determine the thread-to-core mapping that maximizes overall application throughput.

### B. Inter-Thread Data Movement

Data movement performance between producer and consumer threads in CMPs is sensitive to the relative distance of source and destination cores along the cache topology. Consider a pair of threads shown in Figure 3 (b). On a machine like that shown in Figure 2(a), if the two threads reside on two cores that share L2 cache (e.g., on core 0 and 1), then the consumer thread may directly read the data from the L2 cache; on the other hand, if the two threads are on two cores that are 'far away' from each other (e.g., core 0 and core 2), then the sender's updates to shared data will cause invalidation of copies in the other L2 cache, and the consumer thread will experience L2 cache misses and wait for data to be moved through cache coherency protocol.

We demonstrate this fact with a sender-receiver benchmark that measures data movement throughout via a queue associated with Thread Port. Figure 4 shows up to a 3 times throughput difference between sharing vs. not sharing Last Level Cache. This suggests that threads with producer-consumer relationship can benefit from sharing cache. It also shows that the OS scheduler does not respect the data movement between threads and leads to performance loss.

### C. Shared Resource Contention

When running a multi-threaded streaming application on a CMP, threads share certain resources in the CMP's memory hierarchy, including the last-level cache, prefetching hardware, the Front Side Bus (FSB), and Memory Controllers

Table I
TUPLE CONSUMPTION RATE OF THE LEFT THREAD WHEN SHARING L2 CACHE WITH THE RIGHT THREAD (NORMALIZED TO THE LEFT THREAD'S SOLO-RUN TUPLE CONSUMPTION RATE.)

| Left \ Right | Filt. | Func. | Aggr. | Sort | HashJ. |
|---|---|---|---|---|---|
| | Filt. | Func. | Aggr. | Sort | HashJ. |
| Filter | 99.9% | 99.9% | 101.8% | 99.9% | 100.4% |
| Functor | 100.4% | 100.4% | 100.2% | 98.0% | 100.6% |
| Aggregator | 99.9% | 99.9% | 103.5% | 112.8% | 114.7% |
| Sort | 100.2% | 99.4% | 148.8% | 192.9% | 205.8% |
| HashJoin | 100.1% | 100.2% | 122.6% | 134.4% | 136.6% |

Table II
TUPLE CONSUMPTION RATE OF THE LEFT THREAD WHEN NOT SHARING L2 CACHE WITH THE RIGHT THREAD (NORMALIZED TO THE LEFT THREAD'S SOLO-RUN TUPLE CONSUMPTION RATE.)

| Left \ Right | Filt. | Func. | Aggr. | Sort | HashJ. |
|---|---|---|---|---|---|
| Filter | 99.3% | 99.8% | 100.3% | 99.8% | 99.8% |
| Functor | 97.9% | 100.5% | 97.5% | 97.5% | 100.8% |
| Aggregator | 100.0% | 100.0% | 100.0% | 101.3% | 104.3% |
| Sort | 96.3% | 96.4% | 95.0% | 100.0% | 103.4% |
| HashJoin | 96.7% | 97.3% | 99.00% | 99.7% | 102.1% |

(as shown in Figure 2). While such resource sharing can be constructive for data movement between producer and consumer threads, it can also cause destructive contention on shared resources demanded by multiple threads and slow down overall performance. This is particularly the case for threads with Independent relationship (Figure 3 (a)): since those threads do not share data with each other, each thread's accesses to its own working set compete for resources (cache space, memory bandwidth, etc.) against other threads.

To assess such contention effects, we run a benchmark program structured as Figure 3 (a). Two independent threads each execute a chain of three operators. The two operators in the middle of the chains are chosen among five commonly-used operators: Filter, Functor, Sort, Aggregator and Hash Join, in order of their working set size. There are $5 \times 5$ combinations. We run the benchmark on the 4-core UMA machine (Figure 2(a)). For each combination, the two threads are mapped to either share L2 cache or use separate L2 caches. Under each of the two thread-to-core mappings, we measure the tuple consumption rate of the left thread and normalize the rate to the rate when the left thread runs in solo and the right thread does not exist. When the two threads share L2 cache, they contend for shared cache space, resulting in performance degradation. The larger the working sets, the worse the performance (e.g., the left thread executing Sort operator is slowed down by 105.8% when the right thread executing Hash Join, as shown in Table I). On the other hand, when each thread is given a separate L2 cache, the contention is greatly reduced and the left thread experiences no more than 4.4% slowdown compared to running solo (shown in Table II).

The benchmark results imply that threads with contending demands on memory resources should be mapped far away from each other in cache topology to reduce contention on shared resources in the memory hierarchy.

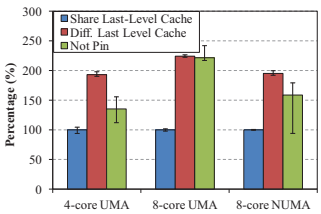



Figure 5. Runtime of Operator Sharing Benchmark (Normalized to the Sharing Last Level Cache Case).

### D. Operator Sharing

When threads share a set of operators protected by a mutex lock, they synchronize with each other to execute the shared operators, as shown in Figure 3 (c). The lock and the shared operator(s)' instructions and data are thus shared by threads. Similar to the producer-consumer case, threads with shared operators may benefit from sharing cache since one thread's access to the lock and shared operators loads data into cache which can then be re-used by other threads.

We run a benchmark to show how sharing operators may affect application performance. The benchmark measures the time of two threads synchronizing on a shared barrier operator 10 million times. The benchmark is run on three machines, each with three different thread-to-core mappings: sharing Last Level Cache (LLC) vs. different LLC vs. OS default thread scheduling. Figure 5 shows that on all three CMPs, sharing LLC between threads improves barrier performance by up to $2.2\times$ over forcing threads use separate LLC. The mapping by OS scheduler ("Not Pin" in Figure 5) causes sub-optimal performance with large variation.

### E. Opportunities and Challenges

Benchmark results indicate that when running multi-threaded streaming applications on CMPs, the thread-to-core mapping can significantly affect application performance due to: (1) inter-thread data movement, (2) contention on shared resources, and (3) inter-thread data sharing. Judicious thread mapping can improve benchmark performance by up to $3\times$ over the default OS scheduler.

However, complexities exist in determining the thread-to-core mapping that leads to the optimal application performance. Real-world streaming applications may have a large number of threads that exhibit sophisticated relationships. Figure 6 shows a three-thread streaming program in which Threads 1 and 2 share an operator and both move data to Thread 3 through a queue. Each thread's data working set consists of (1) *private data* (including state data of operators only executed by this thread and tuples passed between private operators) and (2) *shared data* (including the state data of operators shared with other threads and tuples passed through queues). As a result, obtaining the optimal mapping requires non-trivial knowledge of threads' cache behavior and intelligently considering how mapping would impact all threads' accesses to their private and shared data (e.g., assess

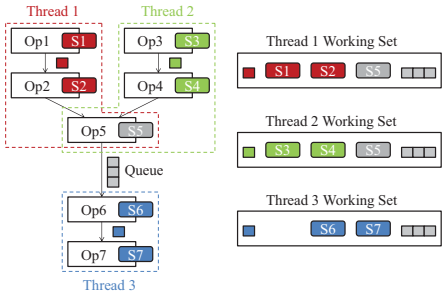



Figure 6. Cache Behavior of Streaming Programs.

whether the benefit of data sharing between two threads outweighs their contention for accessing private data).

## IV. StreamMap: Algorithms & Implementation

### A. Overview

Motivated by the potential performance gains and complexity of thread mappings, we implement *StreamMap* as an offline optimization step in the System S compilation process to decide and enforce thread mapping onto a target multicore platform. As shown in Figure 7, StreamMap is an optional step in application compilation and build process. StreamMap takes as inputs a description of the target machine and the executable file generated by the SPL compiler. Depending on the mapping algorithm used, it may perform one or multiple trial runs of the executable on target machine and record various information. StreamMap then invokes the thread mapping algorithm to calculate the best thread-to-core mapping. The mapping is enforced during application initialization by setting threads' CPU affinity.

StreamMap has the following advantages:

(1) *Generality.* It targets arbitrary SPL-programmed streaming applications for multicore nodes and does not require knowledge about operators' internal implementation.

(2) *Portability.* It works on diverse homogeneous CMPs. Its user-level implementation makes it easy to change its mapping methods and its realization for different OSes.

(3) *Transparency.* Optimizations are transparent to user programs and require minimal programmer involvement.

Although StreamMap is currently implemented with System S middleware, the techniques are applicable to any stream processing applications.

### B. Thread Mapping Algorithms

A good thread-to-core mapping for a streaming application should place intensively communicating and data sharing threads close to each other, and meanwhile isolate threads with conflicting demands on resources. Below we describe four thread mapping algorithms.

*1)* ***Exhaustive Search****:* The "*Exhaustive Search*" algorithm takes as inputs the set of threads within the streaming application and the core ids of the target machine. It runs the application on the target machine with all possible thread-to-core mapping combinations, and after completion, chooses the mapping with highest application throughput.

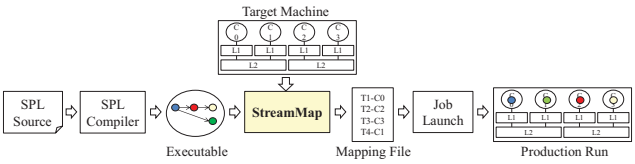



Figure 7. Optimization Workflow of StreamMap.

The algorithm also takes advantage of the symmetry of cache topology to eliminate obviously redundant mappings. Exhaustive Search is guaranteed to find the optimal thread-to-core mapping, and does not require any additional profiling information about the application or target machine. This algorithm, however, suffers from its poor scalability with numbers of threads and CPU cores.

*2)* ***Communication Aware Mapping – TreeMatch****:* The "*TreeMatch*" algorithm [14] aims to minimize data movement cost for mapping a group of MPI processes onto a CMP machine. It takes as input the machine's cache topology and an inter-process communication matrix. The cache topology is modeled as a tree with cores as leaves. The inter-process communication matrix describes the data transfer volumes between each pair of MPI processes. TreeMatch incrementally divides processes into non-overlapping groups whose sizes are equal to the arity of each level of the topology tree, starting from the leaf level and up to the root. At each level, the grouping uses a greedy heuristic that minimizes inter-group communication volume. After process groupings at all levels of the cache topology tree are determined, the mapping of processes to cores can be identified.

For System S applications, since inter-thread communications are explicitly specified at the SPL level (via Threaded Ports), it is straightforward to determine the communication relationship between threads. System S also has profiling support to record the total data volumes passing through Threaded Ports, based on which the inter-thread communication matrix can be constructed. Greedy partitioning is then applied to determine the thread mapping.

*3)* ***Sharing Aware Mapping – TreeMatch-S****:* We have developed the "*TreeMatch-S*" algorithm (shown in Figure 8), which extends TreeMatch by additionally considering operator sharing between threads. Since both inter-thread communication and operator sharing can be viewed as data sharing between threads, TreeMatch-S quantifies these two relationships with a uniform metric that measures the 'intensity' of data sharing. This intensity depends on: (i) the amount of data shared by threads, which depends on the size of shared operators' internal states, and (ii) on how threads access such data, i.e., thread cache access behavior.

The metric is obtained with the DynamoRIO/Umbra tool, which measures threads' cache access at cache line level. DynamoRIO [1] uses dynamic instrumentation of binary executables to obtain various program characteristics at instruction granularity. Umbra [31] adds a set of plug-ins to DynamoRIO for tracking program's memory references.

We measure thread cache behavior using Umbra's cache

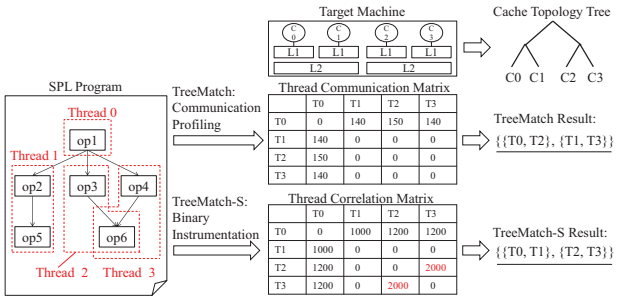



Figure 8. Illustration of TreeMatch and TreeMatch-S Algorithms.

line ownership tracking capability [32]. During program execution, Umbra assigns an ownership bitmap in shadow memory for each application-accessed cache line. Each bit in the bitmap represents one thread, and setting a bit to 1 means the corresponding thread owns a copy of that cache line in its private cache. Umbra dynamically inserts instructions before every memory access instruction to maintain the ownership bitmaps. When a thread reads a cache line, it installs a copy of that cache line in its private cache; accordingly, the instructions inserted by Umbra set thread's bit in that cache line's ownership bitmap. If a thread updates a cache line, it invalidates all other threads' copies of that cache line; Umbra-inserted instructions accordingly set the writing thread's bit and clear all other bits in the bitmap.

With cache line ownership tracking, thread sharing intensity is measured as follows. (1) It records the last thread that updates each cache line, using an array of counters for each thread to record its interaction intensity with other threads. (2) When a thread reads a cache line of which it does not own a copy (a cache miss) or updates a cache line which it does not exclusively own (a cache invalidation), its counter corresponding to the last updating thread of that cache line is increased by 1. (3) When the program finishes, a thread correlation matrix is constructed from threads' counters. This matrix records the sharing intensity between thread which uniformly captures both inter-thread communication and operator sharing relationships. TreeMatch-S uses this thread correlation matrix and applies thread grouping and mapping in the same way as TreeMatch.

*4) **Holistic Mapping**:* Neither TreeMatch nor TreeMatch-S considers shared resource contention between threads. The *Holistic Mapping* algorithm takes into account both sharing AND contention intensity between threads and strikes a balance between them to determine an appropriate mapping.

To mitigate contention on shared memory resources (e.g., last level cache, FSB), we need to quantify each thread's demand on those resources and distribute those demands in a balanced way. Previous work [34, 26] suggests that the Last Level Cache Miss Rate (measured as number of last level cache misses per thousand instructions) is a good measure of a thread's demand on those resources. This is because the LLC miss rate not only indirectly measures a thread's working set size (in terms of how much of its working set cannot fit into last level cache), but also measures how much traffic it imposes on the Front Side Bus. We adopt this approach and obtain each thread's LLC miss rate values as follows. We first apply TreeMatch-S to get an initial thread mapping. We then run the application on the target machine with this mapping and measure the relevant hardware performance counter events. We use the Likwid tool [2] to collect performance counter values and calculate the LLC miss rate for each thread.

With the generated measurements, the Holistic Mapping method groups threads in accordance with the cache topology tree. It treats the grouping of threads at each level as a graph partitioning problem. Each thread is assigned a weight that is its LLC miss rate, and each pair of threads is assigned a weight that represents the data sharing intensity obtained with the TreeMatch-S measurements. The goal of graph partitioning is to reduce cross-group data sharing and in addition, to maintain a reasonable balance of aggregate LLC miss rate values among thread groups. Partitioning is performed with the SCOTCH graph partitioning tool [3].

*5) Additional Implementation Details:*

*NUMA Effect*: When running on a NUMA machine, each thread initialize its operators AFTER binding to target cores so that its data is placed in local NUMA domain.

*Measurement Cost.* Obtaining a machine's cache configuration is a one-time cost. For any application/machine combination, Exhaustive Search requires a complete run. For TreeMatch, information about inter-thread communication can be measured once and used across machines. For TreeMatch-S, data sharing intensity needs to be measured once for any given cache line size. Holistic Mapping needs to obtain the same data sharing intensity information as TreeMatch-S; it must additionally collect hardware performance counter values with one run on target machine.

*Sensitivity to Input Data.* All four algorithms make mapping decisions based on profiling information using sample input data. If threads' runtime behavior diverges dramatically from the profiling runs, the offline mapping generated by those algorithms may lead to unsatisfactory performance. One possible solution to this problem is to continuously monitor and adjust thread mappings at runtime. Since most streaming applications with which we have worked have steady behavior (the same observation is made for StreamIt applications [28]), we leave this topic for future work.

## V. Performance Evaluation

We apply StreamMap to two real-world streaming applications (VWAP and LOIS) on three CMP machines whose architectures are shown in Figure 2. All machines runs Red Hat Enterprise Linux Server release 5.8. The diversity of machines' cache structures helps assessing the effectiveness of StreamMap across different CMP platforms.

For comparison, we also measure the application performance achieved without StreamMap, i.e., we run application

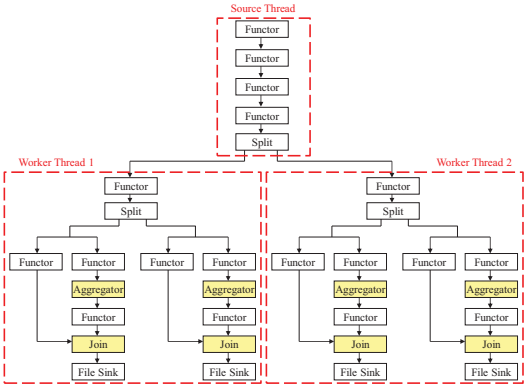



Figure 9. VWAP Application.

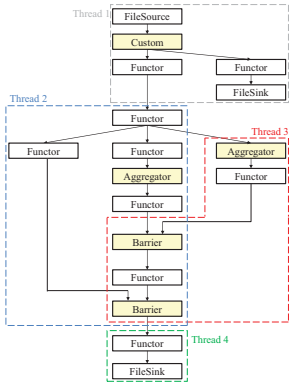



Figure 10. LOIS Application.

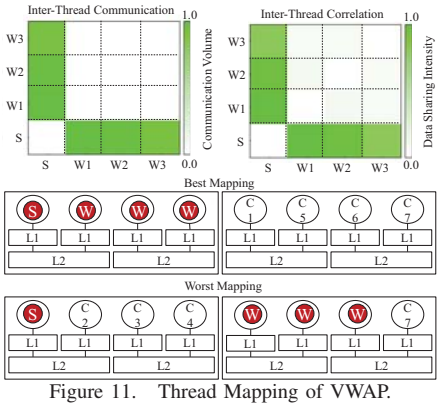
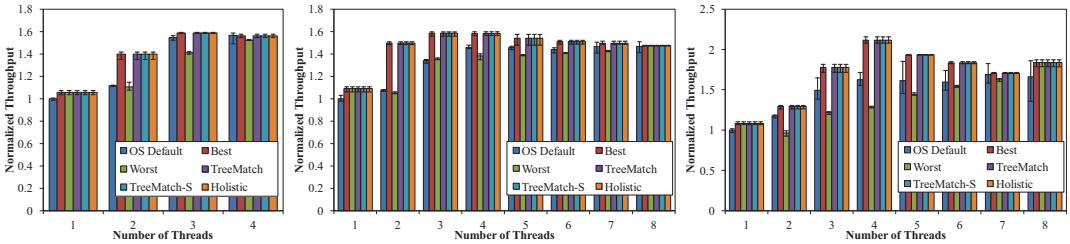



Figure 11. Thread Mapping of VWAP.




(a) 4-Core UMA (b) 8-Core UMA (c) 8-Core NUMA

Figure 12. Throughput of VWAP with Small Sliding Window (Normalized to the Throughput of Single-Threaded VWAP with Default OS Scheduler).

without setting thread affinity and let the OS scheduler decide the core on which each thread runs. On a typical Linux OS, the scheduler schedules threads based on core idleness and may migrate threads across cores during execution.

### A. VWAP Application Performance

The VWAP application ingests financial ticket streams containing trade and quota data from a stock exchange and detects bargains from the data. As shown in Figure 9, input data streams flow into VWAP via a source operator. A split operator uses a hash function to route each tuple to one of the downstream branches. Each branch has an aggregator operator to maintain a running average of trades and a join operator with a sliding window where each new quota tuple will be matched with trades to determine bargains. There is one *source thread* executing the subgraph from the source to the split operator. Each downstream branch is executed by a *worker thread*. The source thread passes data to each worker thread via a separate Threaded Port queue. Worker threads do not communicate or share operator with each other.

We run VWAP with two different configurations: the first (referred to as "Small Window") sets the sliding window size to 5 tuples, and the second (referred to as "Large Window") sets the sliding window size to 75000 tuples. We use a sample input dataset consisting of a day of traces. We use a subset of this sample dataset for StreamMap to quickly calculate offline thread mappings and use the whole dataset to measure resulting application performance.

Figure 12 shows the normalized throughput of VWAP with "Small Window" configuration. We make three observations from the results. First, the maximum difference

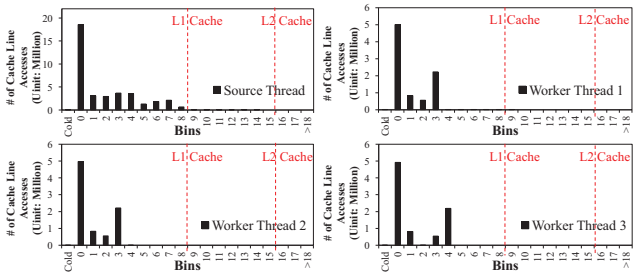
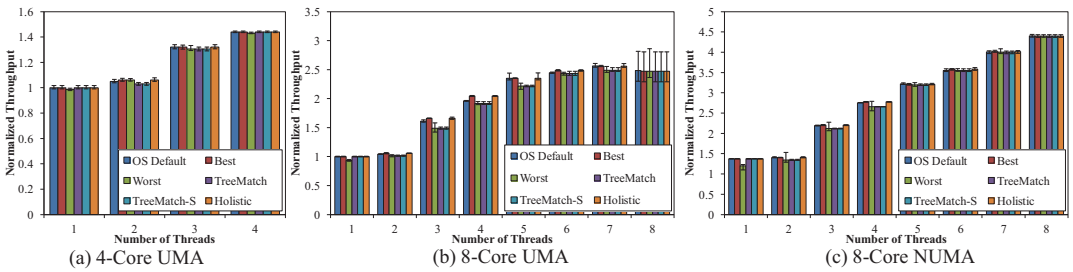



Figure 13. Memory Reuse Distance of VWAP with 4 Threads.

between the best and worst performing mappings is 27% on 4-Core UMA, 43% on 8-Core UMA, and 68% on 8-Core NUMA. On both 8-Core UMA and 8-Core NUMA machines, the differences are the most evident with 3, 4 and 5 threads. Figure 11 shows the inter-thread communication and correlation matrices for VWAP with 4 threads. Both matrices reveal that the source thread has balanced data sharing relationships with each of the three worker threads, which is expected from VWAP's dataflow structure. Figure 11 also shows the best and worst performing mappings on 8-Core UMA. Placing intensively communicating threads close to each other is beneficial to application performance. As VWAP scales beyond 6 threads, the performance differences of different mappings diminish. This is due to the symmetry of machines' cache topologies and the balanced data and work distribution among worker threads in VWAP.

Second, TreeMatch, TreeMatch-S, and Holistic Mapping all generate the best thread mapping in all cases. Since threads in VWAP do not share operators, thread correlation matrix only reflects inter-thread communication intensity (Figure 11), based on which TreeMatch-S derives the same mappings as TreeMatch. We further measure threads' cache




(a) 4-Core UMA (b) 8-Core UMA (c) 8-Core NUMA

Figure 14. Throughput of VWAP with Large Sliding Window (Normalized to the Throughput of Single-Threaded VWAP with Default OS Scheduler).

behavior to explain why Holistic Mapping gives the same mappings as TreeMatch-S. We use Reuse Distance [33] to measure cache locality: when a thread access a cache line, this access' Reuse Distance is the number of distinct cache lines referenced between current access and the previous access to that cache line; a cache line access with a long reuse distance has a high probability of being a cache miss. Figure 13 shows the reuse distance histograms of VWAP with 4 threads.We see that all 4 threads has good temporal locality since the majority of cache line accesses can fit into L1 cache. We also measure the working set size (the number of distinct cache lines touched) of each thread over one million instructions, and find that the working set size of the Source thread is 465 cache lines, and that of each Worker threads is 9 or 10 cache lines. Due to the small working set sizes, contention on last level cache and FSB is not severe among VWAP threads. Therefore, Holistic Mapping has the same results as TreeMatch and TreeMatch-S.

Third, Figure 12 shows that the mappings by the default Linux OS scheduler is close to the worst cases for most tests. This is because the Linux scheduler tends to spread threads across the system, and this hurts the performance of VWAP which is communication dominant.

Interestingly, the performance of VWAP with "Large Window" configuration (shown in Figure 14) show trends opposite to the "Small Window" case: spreading threads away frome each other achieves better performance (up to 7%) than placing them close. This is because worker threads with large sliding windows have large working sets, so the contention on shared cache and memory bandwidth outweighs inter-thread communication. As a result, both TreeMatch and TreeMatch-S generate sub-optimal mappings due to failure to consider resource contention. Holistic Mapping, however, still finds the best mapping. This clearly shows the importance of holistically considering both data sharing and resource contention for thread mapping.

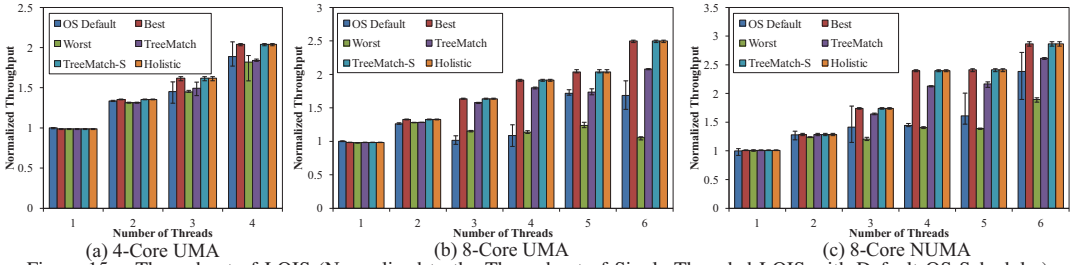
### *B. LOIS Application Performance*

The LOIS application detects outliers in radio data from outer space. The version of LOIS used in this paper reads from a disk file containing a sample dataset collected from a Scandinavian radiotelescope in Europe. Figure 10 shows a 4-thread setup of LOIS. Thread 1 calculates point-wise coordinates for each input data record which are consumed by Thread 2. Thread 2 and 3 maintains aggregate statistics on two sliding windows of past records. Thread 2 produces data to feed into Thread 3, and also synchronizes with Thread 3 on a shared Barrier operator. Thread 4 receives from Thread 2 and 3 the upper and lower bounds of coordinates derived from the sliding windows, and finds all outliers in the incoming data record. 5-thread and 6-thread versions of LOIS are constructed by further introducing Threaded Ports to split the work of Thread 3 and 4.

Figure 15 shows the performance of LOIS. Both TreeMatch-S and Holistic Mapping are able to find the best-performing thread mapping in all cases which outperforms the worst mapping and the default OS mapping by up to 2.4 and 1.8 times, respectively. TreeMatch, on the other hand, sometimes gives sub-optimal mappings. To explain the difference between TreeMatch and TreeMatch-S, we look at the inter-thread communication matrix and thread correlation matrix for LOIS with 4 threads. Figure 16 shows that thread correlation measurements capture data sharing between threads (especially between Thread 2 and 3 due to their sharing of operators), based on which the graph partitioning algorithm makes better decisions than just using inter-thread communication volumes.

Holistic Mapping achieves the same best mapping as TreeMatch-S for LOIS. This is because resource contention is not severe among threads in LOIS. Figure 17 shows the reuse distance histograms of 4 LOIS threads. Although the LOIS threads show worse temporal locality than VWAP (since the histograms are more scattered towards larger distances), the majority of cache line accesses can still fit into L2 cache. Therefore, the benefit brought by placing thread with intensive data sharing close to each other outweighs the penalty of resource contention.

In terms of cost, Exhaustive Search has poor scalability. For example, enumerating all possible mappings of 6-thread LOIS on 8-Core UMA machine takes 600 profiling runs even after eliminating redundant mappings based on symmetry of cache topology. TreeMatch needs one profiling run with System S runtime's communication profiling facility, but the profiling overhead is negligible. TreeMatch-S needs one run with binary instrumentation which can slow down VWAP and LOIS by up to $8\times$. Holistic Mapping requires two profiling runs: the first run is the same as TreeMatch-S with binary instrumentation enabled, and the second one is with low-overhead hardware performance counter monitoring enabled.




(a) 4-Core UMA (b) 8-Core UMA (c) 8-Core NUMA

Figure 15. Throughput of LOIS (Normalized to the Throughput of Single-Threaded LOIS with Default OS Scheduler).

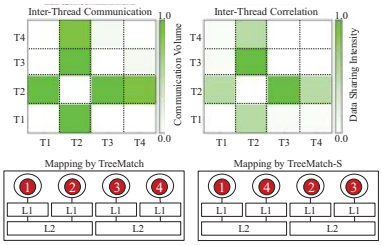



Figure 16. Thread Mapping of LOIS with 4 Threads on 4-Core UMA.

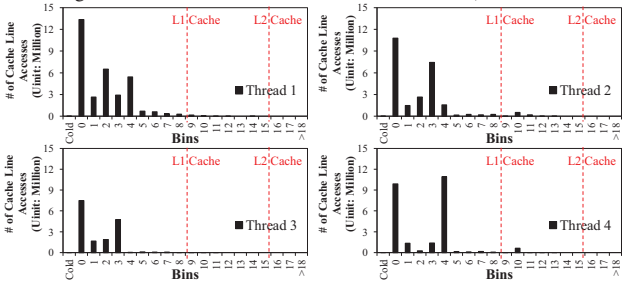



Figure 17. Memory Reuse Distance of LOIS.

### C. Major Observations from Experimental Results

(1) There is up to 2.4x difference between the best and worst thread mappings. The difference is more evident when there is more asymmetry in application structure and/or cache topology. Since streaming applications with complicated dataflows and parallelisms are emerging and current and next generation of CMPs by major vendors are adopting deep and complex cache structures, cache topology aware mapping will show growing importance in the future.

(2) The Linux OS tends to spread threads, which does not always lead to the best performance and sometimes constitutes the worst case (especially for applications whose performance is dominated by inter-thread data sharing). Explicit thread binding also results in more consistent performance than the Linux OS scheduler.

(3) When deciding the optimal thread mapping, it is necessary to consider both constructive data sharing and destructive resource contention between threads. This is demonstrated by our Holistic Mapping algorithm which always finds the best mappings.

## VI. Related Work

### A. Mapping Multi-Threaded Programs on CMPs

*Communication and Sharing Aware Mapping*. The problem of mapping a multi-process or multi-threaded program to a set of underlying resources to minimize program execution time is NP-hard [8]. Various heuristics have been proposed with the objective of minimizing communication cost, such as graph partitioning [5] and graph matching [19]. [24] implements a OS-level scheduler which places intensively-interacting threads close to each other. As shown in this paper, arranging data movement based on cache topology alone does not capture all important trade-offs and may lead to sub-optimal overall performance.

*Contention Aware Mapping*. [18] shows that contention on shared memory resources can severely degrade application performance on CMPs. Hardware [21] and OS level [17] cache partitioning, user level thread scheduling [34], and compiler-time transformation [22, 25] are proposed to mitigate resource contention among processes and/or threads. However, they only work for independent threads or processes with no communication or sharing relationships.

*Holistic Mapping*. [26] exploits thread mapping for optimizing multi-threaded datacenter applications on CMPs. However, the mapping policies in [26] either use exhaustive search or require a new algorithm for each type of cache topology. Our new contributions include an in-depth study of streaming application's cache behavior and a holistic mapping algorithm which is more general and scalable.

### B. Optimizing Streaming Applications on CMPs

*Algorithmic Optimization of Streaming Algorithms*. Examples include join [27, 11], aggregation [6], sorting [10], and frequency counting [7]. Operators in a streaming program may interact with each other in complicated ways, and such complexity is manifested at thread level at runtime. Managing such complicated interaction is beyond the scope of tuning individual operators, so orchastration provided by StreamMap is necessary to coordinate threads' execution. Besides, StreamMap does not require knowledge of operators' internal implementation, and uses profiling information collected at middleware level to make mapping decisions. Therefore, StreamMap is orthogonal and complementary to code tuning of individual streaming operators.

*Compile-Time Scheduling and Mapping of Streaming Programs on CMPs*. Pioneered by StreamIt [13], work has been done in optimizing static scheduling of streaming programs on multicores [16, 29, 30]. Those work apply compiler analysis and code transformation to exploit various parallelism in program to balance computation load among threads. Since most of those work target Cell B.E. architec-

ture which has explicitly managed memory hierarchy, they commonly try to hide data movement cost by overlapping communication with computation in the execution schedule. In comparison, StreamMap targets cache topologies seen in x86 CMP architecture which impose distinct challenges such as contention on shared resources, and it works at thread level without heroic compiler analysis.

*Cache-Aware Optimization of Streaming Programs*. [23] proposed three techniques to optimize StreamIt programs on a uniprocessor: execution scaling, buffer manager, and register replacement. [20] extends execution scaling to multi-core. Both work assumes synchronous data flow model to calculate static steady state schedule, and [20] does not consider thread mapping to reduce contention or thread synchronization cost. [4] theoretically shows that cache-efficient scheduling of streaming programs on a uniprocessor can be modeled as a partitioning problem and some special cases can be solved in polynomial time. Our work targets multi-core processors, uses mapping algorithms different from the one proposed in [4], implements those algorithms inside System S middleware, and evaluates them with real-world applications on representative architectures.

## VII. Conclusions and Future Work

This paper demonstrates that CMP's cache topology can have a significant impact on the performance of multi-threaded streaming applications. Our StreamMap approach automatically places threads to cores to control the data sharing and resource contention between threads, and can improve real-world streaming applications' performance by up to 1.8 times over the default Linux OS scheduler.

Future directions of this research are twofold. We plan to study how to use StreamMap to provide feedback information to System S' compiler to optimize operator fusion. We also plan to apply StreamMap to a wider range of streaming applications, e.g., dynamic graph analysis.

## References


[1] Dynamorio: Dynamic instrumentation tool platform. http://www.dynamorio.org, October 2012.

[2] Likwid: Lightweight performance tools. http://code.google.com/p/likwid, October 2012.

[3] Scotch: Static mapping, graph, mesh and hypergraph partitioning. http://www.labri.fr/perso/pelegrin/scotch/, October 2012.

[4] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo. Cache-conscious scheduling of streaming applications. In *Proc. of SPAA '12*, 2012.

[5] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In *Proc. of ICS '06*, 2006.

[6] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *Proc. of VLDB '07*, 2007.

[7] S. Das, S. Antony, D. Agrawal, and A. El Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proc. VLDB Endow.*, 2(1):217–228, Aug. 2009.

[8] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss. Evaluating thread placement based on memory access patterns for multi-core processors. In *Proc. of HPCC '10*, 2010.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proc. of SIGMOD Conference*, 2008.

[10] B. Gedik, R. R. Bordawekar, and P. S. Yu. Cellsort: high performance sorting on the cell processor. In *Proc. of VLDB '07*, 2007.

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 18(2):501–519, Apr. 2009.

[12] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proc. of PPoPP '08*, 2008.

[13] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of ASPLOS '06*, 2006.

[14] E. Jeannot and G. Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. In *Proc. of Euro-Par '10*, 2010.

[15] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: optimizing stream processing applications via graph partitioning. In *Proc. of Middleware '09*, 2009.

[16] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08*, 2008.

[17] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of HPCA '08*, 2008.

[18] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Proc. of HiPEAC '11*, 2011.

[19] E. H. Molina da Cruz, M. A. Zanata Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J.-F. Mehaut. Using memory access traces to map threads and data on hierarchical multi-core platforms. In *Proc. of IPDPSW '11*, 2011.

[20] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In *Proc. of DATE '08*, 2008.

[21] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of MICRO 39*, 2006.

[22] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proc. of SC '10*, 2010.

[23] J. Sermulins, W. Thies, R. M. Rabbah, and S. P. Amarasinghe. Cache aware optimization of stream programs. In *Proc. of LCTES*, 2005.

[24] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proc. of EuroSys '07*, 2007.

[25] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for qos in warehouse scale computers. In *Proc. of CGO '12*, 2012.

[26] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proc. of ISCA '11*, 2011.

[27] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proc. of SIGMOD '11*, 2011.

[28] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT '10*, 2010.

[29] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *CGO '09*, 2009.

[30] Z. Wang and M. F. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*, 2010.

[31] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: efficient and scalable memory shadowing. In *Proc. of CGO '10*, 2010.

[32] Q. Zhao, D. Koh, S. Raza, D. Bruening, and W.-F. Wong. Dynamic cache contention detection in multi-threaded applications. In *Proc. of VEE '11*, 2011.

[33] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.*, 31(6), Aug. 2009.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. of ASPLOS '10*, 2010.