Data Marshaling for Multi-core Architectures

M. Aater Suleman† Onur Mutlu§ José A. Joao† Khubaib† Yale N. Patt†

†The University of Texas at Austin
{suleman, joao, khubaib, patt}@hps.utexas.edu

§Carnegie Mellon University
onur@cmu.edu

ABSTRACT

Previous research has shown that Staged Execution (SE), i.e., dividing a program into segments and executing each segment at the core that has the data and/or functionality to best run that segment, can improve performance and save power. However, SE's benefit is limited because most segments access *inter-segment data*, i.e., data generated by the previous segment. When consecutive segments run on different cores, accesses to inter-segment data incur cache misses, thereby reducing performance. This paper proposes *Data Marshaling (DM)*, a new technique to eliminate cache misses to inter-segment data. DM uses profiling to identify instructions that generate inter-segment data, and adds only 96 bytes/core of storage overhead. We show that DM significantly improves the performance of two promising Staged Execution models, Accelerated Critical Sections and producer-consumer pipeline parallelism, on both homogeneous and heterogeneous multi-core systems. In both models, DM can achieve almost all of the potential of ideally eliminating cache misses to inter-segment data. DM's performance benefit increases with the number of cores.

Categories and Subject Descriptors: C.0 [General]: System architectures;
General Terms: Design, Performance.
Keywords: Staged Execution, Critical Sections, Pipelining, CMP.

1. INTRODUCTION

Previous research has shown that execution models that *segment* a program and run each segment on its best suited core (the core with the data and/or the functional units needed for the segment) unveil new performance opportunities. Examples of such models include Accelerated Critical Sections [41], producer-consumer pipelines, computation spreading [13], Cilk [10], and Apple's Grand Central Dispatch [5]. We collectively refer to these models as *Staged Execution (SE)*.[1] In SE, when a core encounters a new segment, it ships the segment to the segment's *home core* (i.e., the core most suited for the segment's execution). The home core buffers the execution request from the *requesting core* and processes it in turn.

While SE can improve locality [41, 11, 18], increase parallelism [43], and leverage heterogeneous cores [4], its performance benefit is limited when a segment accesses *inter-segment* data, i.e., data generated by the previous segment. Since each segment runs on a different core, accesses to inter-segment data generate cache misses, which reduces performance. Unfortunately, hardware prefetching is ineffective for inter-segment data because the data access patterns are irregular and fewer cache lines need to be transferred than required to train a hardware prefetcher.

To reduce cache misses for inter-segment data, we propose a mechanism to identify and send the inter-segment data required by a segment to its home core. We call it *Data Marshaling (DM)*, a term borrowed from the network programming community where data required by a remote procedure call is sent to the remote core executing that procedure. However, the implementations of the two mechanisms are fundamentally different, as would be expected due to their dissimilar environments, as shown in Section 7.3.

We use a key observation to design DM: *the generator set, i.e., the set of instructions that generate inter-segment data stays constant throughout the execution.* The compiler uses profiling to identify the generator set and the hardware marshals the data written by these instructions to the home core of the next segment. This approach has three advantages. First, since the generator instructions are statically known, DM does not require any training. Second, the requesting core starts marshaling the inter-segment cache lines as soon as the execution request is sent to the home core, which makes data transfer timely. Third, since DM identifies inter-segment data as *any* data written by a generator instruction, DM can marshal any arbitrary sequence of cache lines. DM requires only a modest 96 bytes/core storage, one new ISA instruction, and compiler support.

DM is a general framework and can be applied to reduce inter-segment data misses in *any* SE paradigm. In this paper, we show how DM can improve the performance of the promising Accelerated Critical Section (ACS) paradigm [41] and the traditional producer-consumer pipeline paradigm. ACS accelerates the critical sections by running them on the large core of an Asymmetric Chip Multiprocessor (ACMP). DM can marshal the inter-segment data needed by the critical section to the large core. Our evaluation across 12 critical-section-intensive applications shows that DM eliminates almost all inter-segment data misses in ACS. Overall, DM improves performance by 8.5% over an aggressive baseline with ACS and a state-of-the-art prefetcher, at an area budget equivalent to 16 small cores. In a producer-consumer pipeline, cores are allocated to pipeline stages and each work-quantum moves from core to core as it is processed. In this case, DM can marshal the data produced at one stage to the next stage. Our evaluation across 9 pipelined workloads shows that DM captures almost all performance potential possible from eliminating inter-segment data misses and improves performance by 14% at an area budget of 16 cores. Section 6 discusses other uses of DM, such as task-parallel workloads [21, 10, 5] and systems with specialized accelerators.

Contributions: This paper makes two main contributions:

- We propose Data Marshaling (DM) to overcome the performance degradation due to misses for inter-segment data by identifying and marshaling the required data to the home core. DM is a flexible and general framework that can be useful in many different execution paradigms. DM uses profiling and requires only 96-byte/core storage.

[1]We borrow the term *Staged Execution* from the operating systems [27] and database [18] communities.

- We design and evaluate two applications of Data Marshaling: Accelerated Critical Sections and producer-consumer pipeline parallelism. We show that in both cases DM can eliminate almost all performance loss due to inter-segment data misses.

2. BACKGROUND

2.1 Staged Execution

The central idea of Staged Execution (SE) is to split a program into code *segments* and execute each segment where it runs the best, i.e., its **home core**. The criteria commonly used to choose a segment's home core include performance criticality, functionality, and data requirements, e.g. critical sections which are on the critical path of the program are best run at the fastest core [41]. Mechanisms to choose the home core are beyond the scope of this paper and are discussed in other work on SE [41, 11, 13, 17].

Implementation: Each core is assigned a work-queue which stores the segments to be processed by the core. Every time the core completes a segment, it dequeues and processes the next entry from the work-queue. If the work-queue is empty, the core waits until a request is enqueued or the program finishes.

The program is divided into *segments*, where each segment is best suited for a different core. Each segment, except the first, has a *previous-segment* (the segment that executed before it). Similarly, each segment, except the last, has a *next-segment* (the segment that will execute after it). At the end of each segment is an *initiation routine* for its next-segment. The **initiation routine** enqueues a request for execution of the next-segment at the next-segment's home core. The core that runs the initiation routine to request a segment's execution is the segment's **requesting core**.

Example: Figure 1(a) shows a code with three segments: S0, S1, and S2. S0 is S1's previous-segment and S2 is S1's next-segment. At the end of S0 is S1's initiation routine and at the end of S1 is S2's initiation routine. Figure 1(b) shows the execution of this code on a CMP with three cores (P0-P2) with the assumption that the home cores of S0, S1, and S2 are P0, P1, and P2 respectively. After completing S0, P0 runs S1's initiation routine, which inserts a request for the execution of S1 in P1's work-queue. Thus, P0 is S1's requesting core. P1 dequeues the entry, executes segment S1, and enqueues a request for S2 in P2's work-queue. Processing at P2 is similar to the processing at P0 and P1.

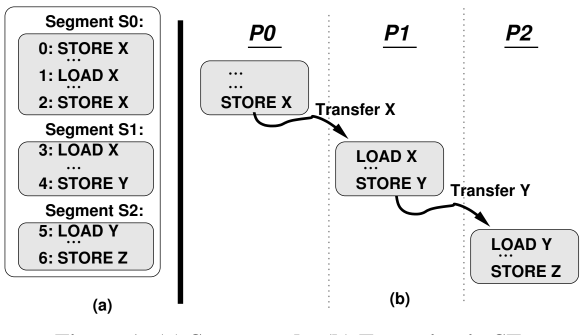


Figure 1: (a) Source code, (b) Execution in SE.

2.2 Problem: Locality of Inter-Segment Data

We define the **Inter-Segment Data** of a segment as *the data the segment requires from its previous segment*. In Figure 1, X is S1's inter-segment data as X is produced by S0 and consumed by S1. Locality of inter-segment data is high in models where consecutive segments run on the same core: the data generated by the first segment remains in the local cache until the next segment can use it. However, in SE, accesses to inter-segment data incur cache misses and the data is transferred from the requesting core to the home core via cache coherence. In Figure 1, P1 incurs a cache miss to transfer X from P0's cache and P2 incurs a cache miss to transfer Y from P1's cache. These cache misses limit SE's performance.

3. DATA MARSHALING

Data Marshaling (DM) aims to reduce cache misses to inter-segment data. In DM, the requesting core identifies the inter-segment data and marshals it to the home core when it ships a segment to the home core.

3.1 Key Insight

We first define new terminology. We call the last instruction that modifies the inter-segment data in a segment a *generator*. For example, in Figure 1(a), the *STORE* on line 2 is X's generator since there are no other stores to X between this STORE and the initiation of S1. The LOAD on line 1 is *not* a generator because it does not modify X and the STORE on line 0 is *not* a generator since there is a later STORE (the one on line 2) to X. We generalize our definition of generators to cache lines by redefining a generator as *the last instruction that modifies a cache line containing inter-segment data before the next segment begins.*

We call the set of all generator instructions, i.e. generators of all the cache lines containing inter-segment data, the *generator-set*. We observed that the generator-set of a program is often small and does not vary during the execution of the program and across input sets (see Section 4.5.1 for details). This implies that any instruction that has once been identified as a generator stays a generator. Thus, a cache line written by a generator is very likely to be inter-segment data required by the following segment, hence, a good candidate for data marshaling. Based on this observation, we assume that *every cache line written by an instruction in the generator-set is inter-segment data and will be needed for the execution of the next segment*. We call the set of inter-segment data cache lines generated by a segment its *marshal-set*. DM adds all cache lines written by generators to the marshal-set. When a core finishes the initiation routine of the next segment, all cache lines that belong to the marshal-set are transferred to the next-segment's home core.

We only marshal data between consecutive segments for two reasons: 1) it achieves most of the potential benefit since 68% of the data required by a segment is written by the immediately preceding segment, and 2) in most execution paradigms, the requesting core only knows the home core of the next segment, but not the home cores of the subsequent segments.

3.2 Overview

The functionality of DM can be partitioned into three parts:

Identifying the generator-set: DM identifies the generator-set at compile-time using profiling.[2] We define the *last-writer* of a cache line to be the last instruction that modified a cache line. Thus, *a line is inter-segment data if it is accessed inside a segment but its last-writer is a previous segment instruction*. Since the generator-set is stable, we assume that last-writers of all inter-segment data are generators. Thus, every time DM detects an inter-segment cache line, it adds the cache line's last-writer to the generator-set (unless it is already in the generator-set). The compiler conveys the identified generator-set to the hardware using new ISA semantics.

Recording of inter-segment data: Every time an instruction in the generator-set is executed, its destination cache line is predicted to be inter-segment data and added to the marshal-set.

[2]We use profiling because a static analysis may not be able to identify all generators without accurate interprocedural pointer alias analysis. However, if an efficient static analysis is developed, the proposed ISA and hardware will be able to fully leverage it.

Marshaling of inter-segment data: All elements of the marshal-set are transferred, i.e. marshaled, to the home core of the next-segment when the next segment is initiated.

For example, DM for the code shown in Figure 1 works as follows. DM detects X to be inter-segment data, identifies the STORE on line 2 to be X's last-writer, and adds it to the generator-set. When P0 executes the STORE on line 2 again, DM realizes that it is a generator and adds the cache line it modifies to the marshal-set. When P0 runs the initiation routine for S1, DM marshals all cache lines in the marshal-set to P1. Consequently, when S1 executes at P1, it (very likely) will incur a cache hit for X.

DM requires support from the compiler, ISA, and hardware.

3.3 Profiling Algorithm

The profiling algorithm runs the application as a single thread and instruments all memory instructions.[3] The instrumentation code takes as input the PC-address of the instrumented instruction and the address of the cache line accessed by that instruction. Figure 2 shows the profiling algorithm. The algorithm requires three data structures: 1) a *generator-set* that stores the identified generators, 2) a *current-last-writer* table that stores the last-writer of each cache line modified in the current segment, and 3) a *previous-last-writer* table that stores the last-writer of each cache line modified in the previous segment.

```
On every memory access:
    If cache-line not present in segment's accessed lines and
      last-writer is from previous segment
            Add last-writer to generator-set
On every store:
    Save address of store in current-last-writer
At the end of segment:
    Deallocate previous-last-writer
    current-last-writer becomes previous-last-writer
    Allocate and initialize current-last-writer
```

Figure 2: The profiling algorithm.

For every memory access, the algorithm checks whether or not the line was modified in the previous segment by querying the previous-last-writer table. If the line was not modified in the previous segment, the line is ignored. If the cache line was modified in the previous segment, the last-writer of the line (an instruction in the previous segment) is added to the generator-set. When an instruction modifies a cache line, the profiling algorithm records the instruction as the last-writer of the destination cache line in the current-last-writer table. At the end of each segment, the lookup table of the previous segment is discarded, the current segment lookup table becomes the previous segment lookup table, and a new current segment lookup table is initialized. After the program finishes, the generator-set data structure contains all generators.

3.4 ISA Support

DM adds two features to the ISA: a GENERATOR prefix and a MARSHAL instruction. The compiler marks all generator instructions by prepending them with the GENERATOR prefix. The MARSHAL instruction is used to inform the hardware that a new segment is being initiated. The instruction takes the home core ID of the next-segment as its only argument. When the initiation routine initiates a segment at a core, it executes the MARSHAL instruction with that core's ID.[4] When the MARSHAL instruction executes, the hardware begins to marshal all cache lines in the marshal-set to the core specified by HOME-ID.

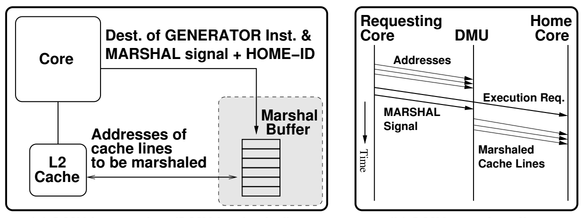
[3]We also evaluated a thread-aware version of our profiling mechanism but its results did not differ from the single-threaded version.

[4]The runtime library stores a segment-to-core mapping which is populated by executing a CPUID instruction on each core.

3.5 Data Marshaling Unit

Each core is augmented with a Data Marshaling Unit (DMU), which is in-charge of tracking and marshaling the inter-segment data to the home core. Figure 3(a) shows the integration of the DMU with the core. Its main component is the *Marshal Buffer*, which stores the addresses of the cache lines to be marshaled. The Marshal Buffer is combining, i.e. multiple accesses to the same cache line are combined into a single entry, and circular, i.e. if it becomes full its oldest entry is replaced with the incoming entry.

Figure 3(b) shows the operation of the DMU. When the core retires an instruction with the GENERATOR prefix, it sends the physical address of the cache line written by the instruction to the DMU. The DMU enqueues the address in the *Marshal Buffer*. When the core executes the MARSHAL instruction, it asserts the MARSHAL signal and sends the HOME-ID to the DMU. The DMU starts marshaling data to the home core. For each line address in the Marshal Buffer, the DMU accesses the local L2 cache to read the coherence state and data. If the line is in shared state or if a cache miss occurs, the DMU skips that line. If the line is in exclusive or modified state, the DMU puts the line in a temporary state that makes it inaccessible (similar to [30]) and sends a *DM Transaction* (see Section 3.6) containing the cache line to the home core. The home core installs the marshaled line in its fill buffer and responds with an ACK, signaling the requesting core to invalidate the cache line. If the fill buffer is full, the home core responds with a NACK, signaling the requesting core to restore the original state of the cache line.



(a) DMU integration (DMU is shaded) (b) DMU functionality

Figure 3: Data Marshaling Unit.

Note that DM marshals lines into L2 caches (not L1). This reduces contention for cache ports between the core and DMU. Moreover, L2, being bigger, is less prone to cache pollution. However, this also implies that DM does not save L1 misses. We find that the performance benefit of marshaling into the L2 outweighs its cost.

3.6 On-Chip Interconnect Support

The interconnect supports one new *DM transaction*, which is sent by the DMU to a remote core via the directory/ordering-point. Each DM transaction is a point-to-point message containing the address and data of a marshaled cache line, and requires the same interconnect support (wires, routing, etc.) as a cache-to-cache transfer, which is available in existing CMPs.

3.7 Handling Interrupts and Exceptions

An interrupt or exception can modify the virtual-to-physical address mapping, thereby making the contents of the Marshal Buffer invalid. Since DM does not impact correctness, we simply flush the contents of the Marshal Buffer at an interrupt or exception.

3.8 Overhead

DM can significantly improve performance by reducing inter-segment cache misses. However, it also incurs some overhead:

- DM adds the GENERATOR prefix to all generator instructions. This only marginally increases the I-cache footprint, since the average size of the generator-set is only 55 instructions.
- DM adds a MARSHAL instruction to each segment. This overhead is low because the MARSHAL instruction: (1) does not

read/write any data, and (2) executes only once per segment, which often consists of thousands of instructions.

- The DMU can contend with the core for cache ports. This overhead is no different from the baseline, where the inter-segment data cache-to-cache transfer requests contend for the cache.
- DM augments each core with a Marshal Buffer. The storage overhead of this buffer is only 96 bytes/core (16 6-byte entries, each storing the physical address of a cache line). Section 4.5.8 shows the sensitivity of performance to Marshal Buffer size.

In summary, the overhead of DM is low. We now discuss the use of DM in different execution paradigms.

4. ACCELERATED CRITICAL SECTIONS

Data Marshaling is beneficial in any execution paradigm that uses the basic principle of Staged Execution. In this section we describe the application of DM to improve a recently proposed SE paradigm: Accelerated Critical Sections (ACS) [41].

4.1 Background

Critical Sections: Accesses to shared data are encapsulated inside critical sections to preserve mutual exclusion. Only one thread can execute a critical section at any given time, while other threads wanting to execute the same critical section must wait, thereby reducing performance. Furthermore, contention for critical sections increases with more threads, thereby reducing application scalability. This contention can be reduced if critical sections execute faster. When threads are waiting for the completion of a critical section, reducing the execution time of the critical section significantly affects not only the thread that is executing it but also the threads that are waiting for it.

Suleman et al. [41] proposed *Accelerated Critical Sections (ACS)* to reduce execution time of critical sections. ACS leverages the Asymmetric Chip Multiprocessor (ACMP) architecture [39, 19, 31], which provides at least one large core and many small cores on the same chip. ACS executes the parallel threads on the small cores and leverages the large core(s) to speedily execute Amdahl's serial bottleneck as well as critical sections.

ACS introduces two new instructions, CSCALL and CSRET, which are inserted at the beginning and end of a critical section respectively. When a small core executes a CSCALL instruction, it sends a *critical section execution request (CSCALL)* to the large core P0 and waits until it receives a response. P0 buffers the CSCALL request in the *Critical Section Request Buffer (CSRB)* and starts executing the requested critical section at the first opportunity. When P0 executes a CSRET, it sends a *critical section done (CSDONE)* signal to the requesting small core, which can then resume normal execution.

ACS is an instance of Staged Execution (SE) where code is split into two types of segments: critical-section segments that run at the large core, and non-critical-section segments that run at the small cores. Every critical-section segment is followed by a non-critical-section segment and vice-versa. Every non-critical-section segment terminates with a CSCALL and every critical-section segment terminates with a CSRET instruction. A CSCALL instruction initiates a critical-section segment and enqueues it at the large core's work-queue (the CSRB). A CSRET instruction initiates a non-critical-section segment at the small core.

4.2 Private Data in ACS

By executing all critical sections at the same core, ACS keeps the shared data (protected by the critical sections) and the lock variables (protecting the critical section) at the large core, thereby improving locality of shared data. However, every time the critical section accesses data generated outside the critical section (i.e. thread-private data), the large core incurs cache misses to transfer this data from the small core. This data is inter-segment data: it is generated in the non-critical-section segment and accessed in the critical-section segment. Marshaling this data to the large core can reduce cache misses at the large core, thereby accelerating the critical-section even more. Note that marshaling the inter-segment data generated by the critical-section segment and used by the non-critical-section segment only speeds up the non-critical section code, which is not critical for overall performance. We thus study marshaling private data from the small core to the large core. Since private data and inter-segment data are synonymous in ACS, we use them interchangeably.

4.3 Data Marshaling in ACS

Employing DM in ACS requires two changes. First, the compiler identifies the generator instructions by running the profiling algorithm in Section 3.3, treating critical sections and the code outside critical sections as two segments. Second, the library inserts the MARSHAL instruction right before every CSCALL instruction. The argument to the MARSHAL instruction, the core to marshal the data to, is set to the ID of the large core.

4.4 Evaluation Methodology

Table 1 shows the configuration of the simulated CMPs, using our in-house cycle-level x86 simulator. We use simulation parameters similar to [41]. All cores, both large and small, include a state-of-the-art hardware prefetcher, similar to the one in [42].

Table 1: Configuration of the simulated machines.

Small core	2-wide, 2GHz, 5-stage, in-order
Large core	4-wide, 2GHz, 2-way SMT, 128-entry ROB, 12-stage, out-of-order; 4x the area of small core
Interconnect	64-byte wide bi-directional ring, all queuing delays modeled, minimum cache-to-cache latency of 5 cycles
Coherence	MESI on-chip distributed directory similar to SGI Origin [28], cache-to-cache transfers, # of banks = # of cores, 8K entries/bank
Prefetcher	Aggressive stream prefetcher [42] with 32 stream buffers, can stay 16-lines ahead, prefetches into cores' L2 caches
Caches	Private L1I and L1D: 32KB, write-through, 1-cycle, 4-way. Private L2: 256KB, write-back, 6-cycle, 8-way (1MB, 8-cycle, 16-way for large core). Shared L3: 8MB, 20-cycle, 16-way
Memory	32 banks, bank conflicts and queuing delays modeled. Precharge, activate, column access latencies are 25ns each
Memory bus	4:1 CPU/bus ratio, 64-bit wide, split-transaction
Area-equivalent CMPs. Area is equal to N small cores. We vary N from 1 to 64.	
ACMP	1 large core and N-4 small cores; large core runs serial part, 2-way SMT on large core and small cores run parallel part, conventional locking (Maximum number of concurrent threads = N-2)
ACS	1 large core and N-4 small cores; (N-4)-entry CSRB at the large core, large core runs the serial part, small cores run the parallel part, 2-way SMT on large core runs critical sections using ACS (Max. concurrent threads = N-4)
IdealACS	Same as ACS except all cache misses to private data on the large core are *ideally* turned into cache hits. Note that this is an *unrealistic* upper bound on DM.
DM	Same as ACS with support for Data Marshaling

Unless specified otherwise: 1) all comparisons are done at equal area budget, equivalent to 16 small cores, 2) the number of threads for each application is set to the number of available contexts.

Workloads: Our evaluation focuses on 12 critical-section-intensive workloads shown in Table 2. We define a workload to be critical-section-intensive if at least 1% of the instructions in the parallel portion are executed within critical sections. The benchmark `maze` solves a 3-D puzzle using a branch-and-bound algorithm. Threads take different routes through the maze, insert new possible routes in a shared queue and update a global structure to indicate which routes have already been visited. `iplookup`, `puzzle`, `webcache` and `pagemine` are similar to the benchmarks with the same names used in [40].

Table 2: Simulated workloads.

Workload	Description	Source	Profile Input	Evaluation Input	# of disjoint critical sections	What is Protected by CS?
is	Integer sort	NAS suite [6]	n = 16K	n = 64K	1	buffer of keys to sort
pagemine	Data mining kernel	MineBench [32]	2K pages	10Kpages	1	global histogram
puzzle	15-Puzzle game	[40]	3x3-easy	3x3	2	work-heap, memoization table
qsort	Quicksort	OpenMP SCR [15]	4K elements	20K elements	1	global work stack
sqlite	sqlite3 [2] database engine	SysBench [3]	insert-test	OLTP-simple	5	database tables
tsp	Traveling salesman problem	[26]	7 cities	11 cities	2	termination condition, solution
maze	3D-maze solver		128x128 maze	512x512 maze	2	visited nodes
nqueen	N-queens problem	[22]	20x20 board	40x40 board	534	task queue
iplookup	IP packet routing	[45]	500 queries	2.5K queries	# of threads	routing tables
mysql-1	MySQL server [1]	SysBench [3]	insert-test	OLTP-simple	20	meta data, tables
mysql-2	MySQL server [1]	SysBench [3]	insert-txn	OLTP-complex	29	meta data, tables
webcache	Cooperative web cache	[40]	10K queries	100K queries	33	replacement policy

4.5 Evaluation

We evaluate DM on four metrics. First, we show that the generator-set stays stable throughout execution. Second, we show the coverage, accuracy, and timeliness of DM followed by an analysis of DM's effect on L2 cache miss rate inside critical sections. Third, we show the effect of DM on the IPC of the critical program paths. Fourth, we compare the performance of DM to that of the baseline and ideal ACS at different number of cores.

4.5.1 *Stability of the Generator-Set*

DM assumes that the generator-set, the set of instructions which generate private data, is small and stays stable throughout execution. To test this assumption, we measure the stability and size of the generator set. Table 3 shows the size and variance of the generator-set in 12 workloads. Variance is the average number of differences between intermediate generator-sets (computed every 5M instructions) and the overall generator-set, divided by the generator-set's size. In all cases, variance is less than 7% indicating that the generator-set is stable during execution. We also evaluated the stability of the generator-set on different input sets and found that the generator-set is constant across input sets.

Table 3: Size and variance of the generator-set.

Workload	is	pagemine	puzzle	qsort	tsp	maze	nqueen	sqlite	iplookup	mysql-1	mysql-2	webcache	amean
Size	3	10	24	23	34	49	111	23	27	114	277	7	58.5
Variance (%)	0.1	0.1	0.1	0.9	3.0	2.2	4.9	4.2	4.3	6.4	6.3	1.2	-

4.5.2 *Coverage, Accuracy, and Timeliness of DM*

We measure DM's effectiveness in reducing private data misses using three metrics: coverage, accuracy, and timeliness. **Coverage** is the fraction of private data cache lines identified by DM. **Accuracy** is the fraction of marshaled lines that are actually used at the large core. **Timeliness** is the fraction of useful marshaled cache lines that reach the large core before they are needed. Note that a marshaled cache line that is in transit when it is requested by the large core is not considered timely according to this definition, but it can provide performance benefit by reducing L2 miss latency.

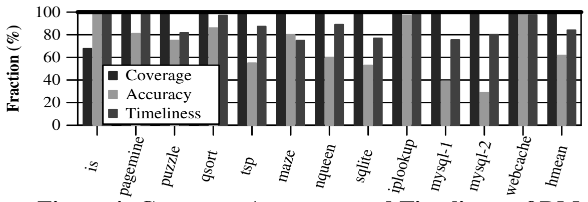


Figure 4: Coverage, Accuracy, and Timeliness of DM.

Figure 4 shows the coverage of DM. DM is likely to detect all private lines because it optimistically assumes that every instruction that once generates private data always generates private data. We find that DM covers 99% of L2 cache misses to private data in all workloads except `is`. The private data in `is` is 117 cache lines, which fills up the Marshal Buffer, and thus several private lines are not marshaled (see Section 4.5.8).

Figure 4 also shows the accuracy of DM. Recall our assumption that every cache line written by any of the generator instructions is private data. This assumption is optimistic since a generator's destination cache line may or may not be used by a critical section depending on control-flow inside the critical section. For example, the critical sections in the irregular database-workloads `mysql-1` and `mysql-2` have a larger number of data-dependent control flow instructions, which leads to a lower accuracy of DM. Despite our optimistic assumption, we find that a majority (on average 62%) of the cache lines marshaled are useful. Moreover, note that marshaling non-useful lines can cause cache pollution and/or interconnect contention, but only if the number of marshaled cache lines is high. We find this not to be the case. Table 4 shows the number of cache lines marshaled per critical section for every workload. In general, we found that transferring *only* an average of 5 cache lines, 62% of which are useful on average, causes a minimal amount of cache pollution and/or interconnect contention.

Table 4: Number of cache lines marshaled per critical section.

Workload	is	pagemine	puzzle	qsort	tsp	maze	nqueen	sqlite	iplookup	mysql-1	mysql-2	webcache	amean
Lines	**16**	9.0	5.6	1.8	3.6	1.8	2.8	8.5	2.9	9.8	15.4	1.8	5.0

Figure 4 also shows DM's timeliness. We find that 84% of the useful cache lines marshaled by DM are timely. Since coverage is close to 100%, timeliness directly corresponds to the reduction in private data cache misses. DM reduces 99% of the cache misses for private data in `pagemine` where timeliness is the highest. In `pagemine`, the main critical section performs reduction of a temporary local histogram (private data) into a persistent global histogram (shared data). `pagemine`'s private data is 8 cache lines: 128 buckets of 4-bytes each. Since the critical section is long (212 cycles on the large core) and contention at the large core is high, DM gets enough time to marshal all the needed cache lines before they are needed by the large core. DM's timeliness is more than 75% in all workloads.

4.5.3 *Cache Miss Reduction Inside Critical Sections*

Table 5 shows the L2 cache misses per kilo-instruction inside critical sections (CS-MPKI) for ACS with no prefetcher (ACS-NP), ACS with prefetcher (ACS) and DM. ACS is only marginally better (11%) than ACS-NP, showing that an aggressive hardware prefetcher is ineffective for inter-segment data misses. When DM is employed, the arithmetic mean of CS-MPKI reduces by **92%** (from 8.92 to 0.78). The reduction is only 52% in `is` because DM has low coverage. We conclude that DM largely reduces L2 cache misses inside critical sections.

Table 5: MPKI inside critical sections.

Workload	.is	pagemine	puzzle	qsort	tsp	maze	nqueen	sqlite	iplookup	mysql-1	mysql-2	webcache	amean	hmean
ACS-NP	13.4	15.6	4.7	25.6	0.8	22.6	18.5	0.8	1.0	5.3	11.5	0.9	10.0	1.8
ACS	3.0	15.6	4.7	25.5	0.8	22.5	16.9	0.8	1.0	5.0	10.9	0.9	8.9	1.6
DM	1.4	0.2	0.5	0.2	0.9	2.2	1.3	0.4	0.4	0.5	1.2	0.1	0.8	0.4

4.5.4 Speedup in Critical Sections

DM's goal is to accelerate critical section execution by reducing cache misses to private data. Figure 5 shows the retired critical-section-instructions per-cycle (CS-IPC) of DM normalized to CS-IPC of baseline ACS. In workloads where CS-MPKI is low or where L2 misses can be serviced in parallel, DM's improvement in CS-IPC is not proportional to the reduction in CS-MPKI. For example, in `webcache`, DM reduces CS-MPKI by almost 6x but the increase in CS-IPC is only 5%. This is because CS-MPKI of `webcache` is only 0.85, which has a small effect on performance since these misses are serviced by cache-to-cache transfers.

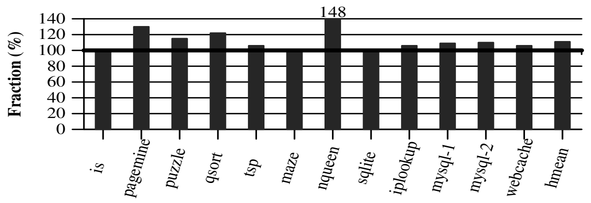


Figure 5: Increase in CS-IPC with DM.

In workloads where CS-MPKI is higher, such as in `pagemine`, `puzzle`, `qsort`, and `nqueen`, DM speeds up critical section execution by more than 10%. Most notably, `nqueen's` critical sections execute 48% faster with DM. Note that in none of the workloads do critical sections execute slower with DM than in ACS. On average, critical sections execute 11% faster with DM. Thus, in benchmarks where there is high contention for critical sections, DM will provide a high overall speedup, as we show next.

4.5.5 Performance

DM increases the IPC of critical sections, thereby reducing the time spent inside critical sections. For highly-contended critical sections, reducing the time spent inside the critical section substantially increases overall performance. Moreover, as the number of available cores on a chip increases (which can increase the number of concurrent threads), contention for critical sections further increases and DM is expected to become more beneficial. We compare ACMP, ACS, and DM at area budgets of 16, 32 and 64.

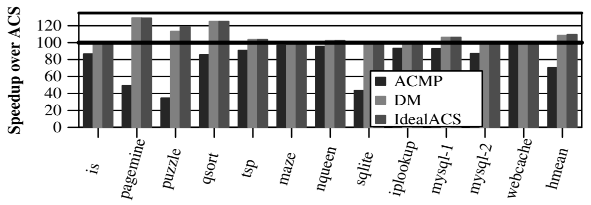


Figure 6: Speedup of DM with an area-budget of 16.

Area budget of 16: Figure 6 shows the speedup of ACMP, DM, and IdealACS normalized to the baseline ACS. DM outperforms ACS on all workloads, by 8.5% on average. Most prominently, in `pagemine`, `puzzle`, and `qsort`, DM outperforms ACS by more than 10% due to large increases in CS-IPC. In other benchmarks such as `tsp` and `nqueen`, DM performs 1-5% better than ACS. Note that DM's performance improvement strongly tracks the increase in critical section IPC shown in Figure 5. There is one exception, `nqueen`, where the main critical section updates a FIFO work-queue that uses very fine-grain locking. Thus, contention for the critical sections is low and, even though DM speeds up the critical sections by 48%, overall performance improvement is only 2%. In all other workloads, faster critical sections lead to higher overall performance. DM's performance is within 1% of the IdealACS for all workloads. Thus, *DM achieves almost all the performance benefit available from eliminating cache misses to private data.*

Note that DM is able to provide an overall speedup of 8.5% by accelerating the execution of *only* the large core by 11% (as shown in Figure 5). This is because when critical sections are on the critical path of the program, accelerating *just* the critical sections by any amount provides an almost-proportional overall speedup without requiring acceleration of all threads.

Larger area budgets (32 and 64): Figure 7 shows that DM's average performance improvement over ACS increases to 9.8% at area budget 32. Most prominently, in `pagemine` DM's improvement over ACS increases from 30% to 68%. This is because `pagemine` is completely critical-section-limited and any acceleration of critical sections greatly improves overall speedup. DM's performance is again within 1% of that of IdealACS, showing that DM achieves almost all potential benefit.

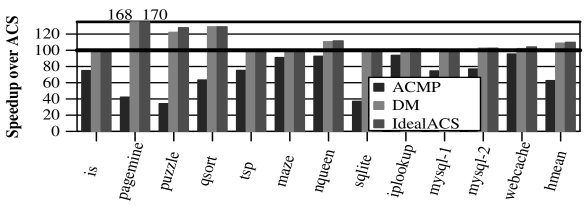


Figure 7: Speedup of DM with an area-budget of 32.

As the chip area further increases to 64, DM's improvement over ACS continues to increase (Figure 8). On average, DM provides 13.4% performance improvement over ACS and is within 2% of its upper bound (IdealACS). We conclude that DM's benefits are likely to increase as systems scale to larger number of cores.

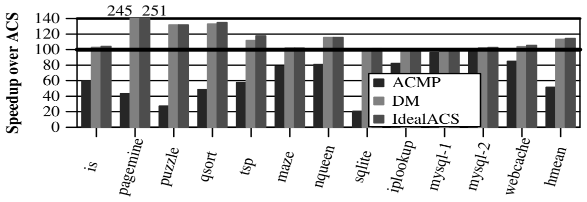


Figure 8: Speedup of DM with an area-budget of 64.

At best-threads: The best-threads of a workload is the minimum number of threads required to minimize its execution time. We also evaluated DM's speedup with best-threads normalized to ACS with best-threads for an area budget of 16. Our results show that even if we use oracle information to pick the best number of threads, which penalizes DM since DM performs better at a higher thread count, DM improves performance by 5.3% over ACS.

4.5.6 Sensitivity to Interconnect Hop Latency

DM's performance improvement over ACS can change with the interconnect hop latency between caches for two reasons. First, increasing the hop latency increases the cost of each on-chip cache miss, increasing the performance impact of misses to private data and making DM more beneficial. Second, increasing the hop latency increases the time to marshal a cache line, which can reduce DM's timeliness, reducing its benefit. We evaluate ACS and DM using hop latencies of 2, 5, and 10 cycles. On average, the speed up of DM over ACS increases from 5.2% to 8.5% to 12.7% as hop latency increases from 2 to 5 to 10. We conclude that DM is even more effective in systems with longer hop latencies, e.g. higher frequency CMPs or SMPs.

4.5.7 *Sensitivity to L2 Cache Size*

Private data misses are communication misses that cannot be avoided by increasing cache capacity. Thus, DM, which reduces communication misses, stays equally beneficial when L2 cache size increases. In fact, DM's benefit might increase with larger caches due to three reasons: 1) enlarging the cache reduces capacity and conflict misses, increasing the relative performance impact of communication misses and techniques that reduce such misses; 2) increasing the L2 size of the large core increases the likelihood that a marshaled cache line will not be evicted before it is used, which increases DM's coverage and timeliness; 3) increasing the small core's L2 capacity increases the amount of private data that stays resident at the small cores' L2 caches and thus can be marshaled, which can increase DM's coverage.

Table 6: Sensitivity of DM to L2 Cache Size.

L2 Cache Size (KB)	128	256	512	1024	2048
ACS vs. ACS 256KB (%)	-5.4	0.0	2.1	2.9	3.1
DM vs. ACS 256KB (%)	-11.4	8.5	10.6	11.3	12.0

Table 6 shows the average speedup across all benchmarks for ACS and DM for small-core L2 cache sizes from 128KB to 2048KB. Note that the cache of the large core is always 4x as large as the cache of a small core. The performance benefit of DM over ACS slightly increases as cache size increases from 256KB to 2048KB. In fact, DM with a 256KB L2 cache outperforms ACS with a 2MB L2 cache. However, with a 128KB L2 cache, DM performs worse than ACS. This is because marshaling private data into a small L2 cache at the large core causes cache pollution, evicting shared data or marshaled data of other critical sections not yet executed, and leading to longer-latency L2 cache misses, serviced by the L3. We conclude that DM's performance benefit either increases or stays constant as L2 cache size increases.

4.5.8 *Sensitivity to Size of the Marshal Buffer*

The number of entries in the Marshal Buffer limits the number of cache lines DM can marshal for a critical section segment. We experimented with different Marshal Buffer sizes and found that 16 entries (which we use in our main evaluation) suffice for all workloads except `is`. Since `is` requires the marshaling of 117 cache lines on average, when we use a 128-entry Marshal Buffer, CS-MPKI in `is` is reduced by 22% and performance increases by 3.8% compared to a 16-entry Marshal Buffer.

5. PRODUCER-CONSUMER PIPELINE

Pipeline parallelism is a commonly used approach to split the work in a loop among threads. In pipeline parallelism, each iteration of a loop is split into multiple work-quanta where each work-quantum executes in a different pipeline stage. Each stage is allocated one or more cores. Pipeline parallelism is another instance of Staged Execution: each iteration of a loop is split into code segments (pipeline stages) which run on different cores.

Figure 9(a) shows a code example of two pipeline stages: S1 and S2, running on cores P1 and P2, respectively. S1 computes and stores the value of a variable X (lines 1-2) and then enqueues a request to run S2 at core P2 (line 3). Note that X is used by S2 (line 4). P2 may process the computation in S2 immediately or later, depending on the entries in its work-queue.

5.1 DM in Pipelines

Processing of a pipeline stage often requires data that was generated in the previous pipeline stage. Since each stage executes

Pipeline Stage S1:	*Pipeline Stage S1:*
1:	1: .... ;Compute X
2: store X	2: **GENERATOR store X**
3: Enqueue a request at S2's home core	3: Enqueue a request at S2's home core ;S2's initiation
	4: **MARSHAL <S2's home core>**
Pipeline Stage S2:	*Pipeline Stage S2:*
4: Y = ... X ...	5: Y = ... X ... ;Compute Y using X
....	....
(a) Code of a pipeline.	(b) Modified code with DM.

Figure 9: Code example of a pipeline.

at a different core, such inter-segment or inter-stage data must be transferred from core to core as the work-quantum is processed by successive pipeline stages. For example, in the pipeline code in Figure 9(a), variable X is inter-segment data as it is generated in S1 (line 2) and used by S2 (line 4). When S2 runs on P2, P2 incurs a cache miss to fetch X from P1.

DM requires two code changes. First, the compiler must identify the generator instructions and prepend them with a GENERATOR prefix. Second, the compiler/library must insert a MARSHAL instruction in the initiation routine. Figure 9(b) shows the code in Figure 9(a) with the modifications required by DM. Since X is inter-segment data, the compiler identifies via profiling the store instruction on line 2 as a generator and prepends it with the GENERATOR prefix. Furthermore, the MARSHAL instruction is inserted in the initiation routine (line 4).

When P1 (the core assigned to S1) runs the store on line 2, the hardware inserts the physical address of the cache line being modified into P1's Marshal Buffer. When the MARSHAL instruction on line 4 executes, the Data Marshaling Unit (DMU) marshals the cache line containing X to P2's L2 cache. When S2 runs on P2, it incurs a cache hit for X, which likely reduces execution time.

5.2 Evaluation Methodology

We simulate a symmetric CMP with all small cores with the parameters shown in Table 1. We simulate three different configurations: Baseline (a baseline CMP without DM), Ideal (an idealistic but impractical CMP where all inter-segment misses are turned into hits), and DM (a CMP with support for DM). The Ideal scheme, which unrealistically eliminates all inter-segment misses, is an upper bound of DM's performance. We evaluate DM on 16-core and 32-core CMPs, but not on a 64-core CMP because a majority of our workloads do not scale to 64 threads. Table 7 shows the simulated workloads. A MARSHAL instruction was inserted in the initiation routine of each workload. The core-to-stage allocation is proportional to execution times, e.g. if two stages execute in 500 and 1000 cycles (on a single core), we assign the latter twice as many cores as the former. All comparisons are at equal-area of 16 small cores unless otherwise stated.

5.3 Evaluation

We evaluate coverage, accuracy, timeliness, inter-segment data MPKI, and overall performance of DM. We also show DM's sensitivity to relevant architectural parameters.[5]

5.3.1 *Coverage, Accuracy, and Timeliness*

Figure 10 shows DM's coverage, i.e., the fraction of inter-segment data cache lines identified by DM. Coverage is over 90% in all workloads except `dedupD`, `image`, and `mtwist` where the inter-segment data per segment exceeds the size of the Marshal Buffer (16) and not all inter-segment data is marshaled.

Figure 10 also shows the accuracy of DM, i.e., the fraction of

[5]We validated that the generator-sets are stable during execution and across input sets for pipeline workloads, but we do not show the results due to space constraints.

Table 7: Workload characteristics.

Workload	Description (No. of pipeline stages)	Major steps of computation	Profile Input	Evaluation Input
black	BlackScholes Financial Kernel [33] **(6)**	Compute each option's call/put value	0.25M opts	1M opts
compress	File compression using bzip2 algorithm **(5)**	Read file, compress, re-order, write	0.5MB image	4MB text file
dedupE	De-duplication (Encoding) [8] **(7)**	Read, find anchors, chunk, compress, write	simsmall	simlarge
dedupD	De-duplication (Decoding) [8] **(7)**	Read, decompress, check-cache, write	simsmall	simlarge
ferret	Content based search [8] **(8)**	Load, segment, extract, vector, rank, out	simsmall	simlarge
image	Image conversion from RGB to gray-scale **(5)**	Read file, convert, re-order, write	10M pixels	100M pixels
mtwist	Mersenne-Twister PRNG [33] **(5)**	Read seeds, generate PRNs, box-muller	path= 20M	path= 200M
rank	Rank string similarity with an input string **(3)**	Read string, compare, rank	100K strings	800K strings
sign	Compute the signature of a page of text **(7)**	Read page and compute signature	100K pages	1M pages

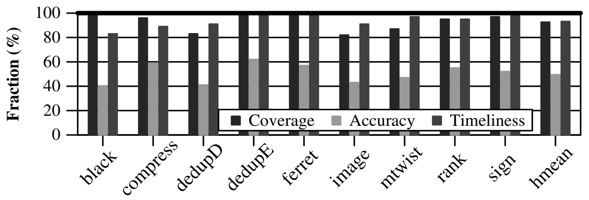


Figure 10: Coverage, Accuracy, and Timeliness of DM.

marshaled lines that are actually used by the home core. DM's accuracy is low, between 40% and 50%, because stages contain control flow. However, the increase in interconnect transactions/cache pollution for DM is negligible because the number of cache lines marshaled per segment is small: the average is 6.8 and the maximum is 16 (the size of the Marshal Buffer).

Figure 10 also shows DM's timeliness, i.e., the percentage of useful cache lines identified by DM that reach the remote home core before their use. Timeliness is high, more than 80% in all cases, for two reasons: (1) segments often wait in the home core's work-queue before their processing, giving DM enough time to marshal the lines, (2) transferring the few lines that are marshaled per segment requires a small number of cycles.

5.3.2 Reduction in Inter-Segment Cache Misses

Table 8 shows the L2 MPKI of inter-segment data in baseline-NP (baseline with no prefetcher), baseline and DM. The prefetcher reduces the average MPKI by only 9%. DM reduces the MPKI significantly more in all cases. Most noticeable is `sign` where DM reduces the MPKI from 30.3 to 0.9. In `sign`, the main inter-segment data is a 256-character page signature array (4 cache lines). Since DM's profiling algorithm marks the instruction that stores the array as a generator, DM saves all cache misses for the array. Similarly, DM almost completely eliminates inter-segment data misses in `ferret` and `dedupE`. DM reduces the harmonic and arithmetic mean of MPKI by 81% and 69% respectively.

Table 8: L2 Misses for Inter-Segment Data (MPKI). We show both amean and hmean because hmean is skewed due to `dedupE`. Note: MPKI of inter-segment data in Ideal is 0.

Workload	black	compress	dedupD	dedupE	ferret	image	mtwist	rank	sign	amean	hmean
baseline-NP	15.2	7.8	49.5	0.7	4.9	67.6	59.4	4.2	31.3	26.6	4.57
baseline	14.2	7.7	47.5	0.4	4.4	55.6	51.4	4.1	30.3	24.0	2.76
DM	2.8	1.7	33.0	0.0	0.1	20.4	7.4	0.3	0.9	7.4	0.53

5.3.3 Performance

Execution time of a pipelined program is always dictated by its slowest stage. Thus, DM's impact on overall performance depends on how much it speeds up the slowest stage. Figure 11 shows the speedup of Ideal and DM over the baseline at 16 cores. On average, DM provides a 14% speedup over the baseline, which is 96% of the potential. DM improves performance in all workloads. DM's improvement is highest in `black` (34% speedup) because DM reduces inter-segment misses by 81% and as a result speeds up the slowest stage significantly. DM's speedup is less (5% lower) than the Ideal speedup in `black` because accesses to inter-segment data are in the first few instructions of each stage and consequently the marshaled cache lines are not always timely. DM's speedup is lower in `dedupE` and `ferret` because these workloads only incur a small number of inter-segment misses and DM's potential is low (Ideal speedup is only 5% for ferret).

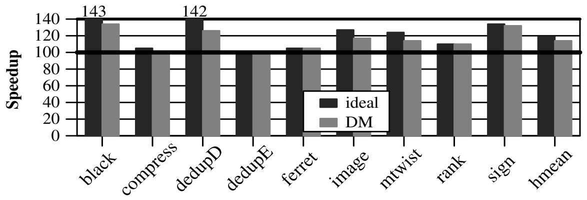


Figure 11: Speedup over baseline at 16 cores.

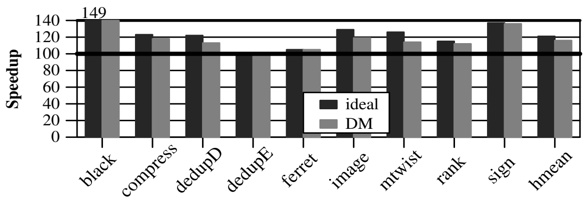


Figure 12: Speedup over baseline at 32 cores.

32-core results: Figure 12 shows the speedup of Ideal and DM over the baseline with 32 cores. DM's speedup increases for all workloads compared to 16 cores. Most significant is the change in `compress`, from 1% to 18%, because the slowest stage in `compress` changes between 16 and 32 cores. At 16 threads, `compress`'s slowest stage is the stage that compresses chunks of input data. This stage is compute-bound and does not offer a high potential for DM. However, the compression stage is scalable, i.e., its throughput increases with more cores. At 32 cores, the compression stage's throughput is more than the non-scalable re-order stage (the stage which re-orders chunks of compressed data before writing them to the output file). Unlike the compression stage which is compute-bound, the re-order stage is bounded by cache misses for inter-segment data, ergo, a higher potential for DM and thus the higher benefit. On average, at 32 cores, DM improves performance by 16%, which is higher than its speedup at 16 cores (14%). In summary, DM is an effective technique that successfully improves performance of pipelined workloads, with increasing benefit as the number of cores increases.

Performance at best-threads: We also evaluate pipelined workloads with the minimum number of threads required to maximize performance. Since a majority of our workloads saturate between 16 and 32 threads, our results at best-threads resemble those at 32-threads, i.e., DM provides a 16% speedup over the baseline.

5.3.4 Sensitivity to Interconnect Hop Latency

We find that the speedup of DM increases with hop latency (refer to Section 4.5.6 for reasons). We evaluated DM with hop latencies of 2, 5, and 10 cycles and find that it provides speedups of 12.4%, 14%, and 15.1% over the baseline, respectively.

5.3.5 Sensitivity to L2 Cache Size

DM's benefit increases with cache size for pipeline workloads as well (see Section 4.5.7 for reasons). DM's speedup over the baseline is 4.6%, 14%, 14.8%, 15.3%, and 15.4% for cache sizes of 128KB, 256KB, 512KB, 1MB, and 2MB, respectively.

5.3.6 Sensitivity to Size of the Marshal Buffer

We find that a 16-entry Marshal Buffer is sufficient for all workloads except `dedupD`, `image`, and `mtwist`. The number of inter-segment cache lines per segment in these workloads is greater than 16. For example, the primary inter-segment data structure in image, an array of 150 32-bit RGB pixels, spans 32-cache lines. Table 9 shows the performance of DM with varying Marshal Buffer sizes in these three workloads. In each case, performance saturates once there are enough entries to fit all inter-segment cache lines (32, 32, and 128 for dedupD, image, and mtwist respectively). In summary, while there are a few workloads that benefit from a larger Marshal Buffer, a 16-entry buffer suffices for most workloads.

Table 9: Speedup (%) for different Marshal Buffer sizes.

# of entries	16	32	64	128	256
dedupD	26	40	40	40	40
image	17	24	24	24	24
mtwist	14	18	21	22	22

6. DM ON OTHER PARADIGMS

We have shown two concrete applications of DM: ACS and pipeline parallelism. DM can also be applied to any paradigm that resembles SE, for example:

- Remote special-purpose cores, e.g., encryption or video encoding engines, which are often used to accelerate code segments. DM can be used to marshal data to such accelerators.
- Task-parallelism models such as Cilk [10], Intel TBB [21] and Apple's Grand Central Dispatch [5]. DM can marshal the input arguments of the task to the core that will execute the task.
- Computation Spreading [13], which improves locality by always running the operating system code on the same set of cores. DM can marshal the data to and from these cores.
- Thread Motion [35], which migrates threads among cores to improve power-performance efficiency. DM can be extended to reduce cache misses due to thread migration.
- The CoreTime OS scheduler [11], which assigns data objects to caches and migrates threads to increase locality. DM can marshal any extra data required by the thread (e.g., portions of its stack).

DM can also enable new execution paradigms. DM lowers the cost of data-migration, the single most important overhead associated with remote execution of code segments. DM can thus enable remote execution in scenarios where it was previously infeasible. Examples include remote execution of fine-grain tasks for increasing parallelism, sharing of rarely-used functional units between cores for saving chip area, and new core specialization opportunities for increased performance and power-efficiency.

In summary, DM is applicable to widely-used current paradigms, and can potentially enable new paradigms.

7. RELATED WORK

The main contribution of this paper is a novel solution to a major limitation of the Staged Execution paradigm: the large increase caused by it in cache misses to inter-segment data. Data Marshaling has related work in the areas of hardware prefetching and OS/compiler techniques to improve locality.

7.1 Hardware Prefetching

Hardware prefetchers can be broadly classified as prefetchers that target regular (stride/stream) memory access patterns (e.g., [24, 42]) and those that target irregular memory access patterns (e.g., [23, 14, 37, 16]). Prefetchers that handle only regular data cannot capture misses for inter-segment data because inter-segment cache lines do not follow a regular stream/stride pattern and are scattered in memory. Prefetchers that handle irregular data (as well as stride/stream based prefetchers) are also not suited for prefetching inter-segment data because the number of cache misses required for training such prefetchers is often more than all of the inter-segment data (an average of 5 in ACS and 6.8 in pipeline workloads). Thus, by the time prefetching begins, a majority of the cache misses have already been incurred.

DM does not have these disadvantages. DM requires minimal on-chip storage, can marshal any arbitrary sequence of inter-segment cache lines, and starts marshaling as soon as the next code segment is shipped to its home core, without requiring any training. Note that our baseline uses an aggressive stream prefetcher [42] and the reported improvements are on **top** of this aggressive prefetcher.

7.2 Reducing Cache Misses

Yang et al. [29] show that pushing (vs. pulling) cache lines to the core can save off-chip cache misses for linked data structures. In contrast, DM saves on-chip misses for any sequence of inter-segment data. Bhattacharjee et al. [7] schedule tasks to maximize cache locality in TBB [21], another example of SE. DM can further help by eliminating the remaining cache misses. Hossain et al. [20] propose DDCache where the producer pushes a cache line to all the sharers of the line when one of the sharers requests the line. DDCache is orthogonal to DM as it only improves locality of shared data, while DM improves locality of private (inter-segment) data.

Other proposals improve shared data locality by inserting software prefetch instructions before the critical section [44, 36]. Such a scheme cannot work for inter-segment data because prefetch instructions must be executed at the home core as part of the next code segment and very close to the actual use of the data, likely making the prefetches untimely. Recent cache hierarchy designs (e.g., [34]) aim to provide fast access to both shared and private data. They can further benefit from DM to minimize cache-to-cache transfers. Proposals to accelerate thread migration [12, 38] are orthogonal and can be combined with DM. Other mechanisms [46, 25] reduce cache line access latency by migrating cache lines closer to the cores that are accessing them. Such schemes cannot reduce inter-segment data misses because –like prefetching– they do not marshal the data proactively and also require training.

7.3 Other Related Work

Remote Procedure Calls (RPC) [9], used in networking, require the programmers to "marshal" the input data with the RPC request. This is similar to DM, which marshals inter-segment data to the home core of the next code segment. However, unlike RPC, DM is solely for performance (i.e., not required for correct execution), does not require programmer input, and is applicable to instances of Staged Execution that do not resemble procedure calls.

8. CONCLUSION

We propose *Data Marshaling (DM)* to improve the reduced locality of inter-segment data in Staged Execution (SE), thereby overcoming a major limitation of SE. DM identifies and marshals the inter-segment data needed by a segment before or during the segment's execution. We show in detail two applications of DM. First, we use DM to marshal the thread-private data to remotely executing critical sections in the Accelerated Critical Section (ACS) mechanism. Our evaluation with 12 critical-section-intensive workloads shows that DM reduces the average execution time by 8.5% compared to an equal-area (16 small cores) baseline ACMP with support for ACS. Second, we use DM for workloads with pipeline parallelism to marshal the data produced in one pipeline stage to the next stage. Our evaluation with 9 pipelined workloads shows that DM improves performance by 14% on a 16-core CMP and 16% on a 32-core CMP. We find that DM's performance is within 5% of an ideal mechanism that eliminates all inter-segment data misses using oracle information, showing that DM achieves almost all of its performance potential. We describe a variety of other possible applications of DM, and conclude that DM is a promising approach that can successfully recover performance loss due to cache misses incurred for inter-segment data in Staged Execution paradigms.

ACKNOWLEDGMENTS

We thank members of the HPS research group, and the anonymous reviewers for their suggestions. We thank the Cockrell Foundation, Intel, and CMU CyLab for their support and the Texas Advanced Computing Center (TACC) for providing HPC resources. M. Aater Suleman was supported by an Intel PhD Fellowship.

REFERENCES

[1] MySQL database engine 5.0.1. http://www.mysql.com, 2008.

[2] SQLite database engine version 3.5.8. 2008.

[3] SysBench: a system performance benchmark v0.4.8. 2008.

[4] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *ISCA-32*, 2005.

[5] Apple. Grand Central Dispatch. Tech. Brief, 2009.

[6] D. H. Bailey et al. NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, 1994.

[7] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA*, 2009.

[8] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[9] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM TOCS*, 2(1):39–59, 1984.

[10] R. D. Blumofe et al. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[11] S. Boyd-Wickizer et al. Reinventing scheduling for multicore systems. In *HotOS-XII*, 2009.

[12] J. A. Brown and D. M. Tullsen. The shared-thread multiprocessor. In *ICS*, 2008.

[13] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS-XII*, 2006.

[14] R. Cooksey et al. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.

[15] A. J. Dorta et al. The OpenMP source code repository. In *Euromicro*, 2005.

[16] E. Ebrahimi et al. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. *HPCA*, 2009.

[17] M. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[18] S. Harizopoulos and A. Ailamaki. StagedDB: Designing database servers for modern hardware. *IEEE Data Eng. Bull.*, June 2005.

[19] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7), 2008.

[20] H. Hossain et al. DDCache: Decoupled and delegable cache data and metadata. In *PACT*, 2009.

[21] Intel. Source code for Intel threading building blocks.

[22] Intel. Getting Started with Intel Parallel Studio, 2009.

[23] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA*, 1997.

[24] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.

[25] C. Kim et al. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, 2002.

[26] H. Kredel. Source code for traveling salesman problem (tsp). http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.

[27] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *USENIX*, 2002.

[28] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, 1997.

[29] C. lin Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *ICS*, 2000.

[30] M. R. Marty. *Cache coherence techniques for multicore processors*. PhD thesis, 2008.

[31] T. Morad et al. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Comp Arch Letters*, 2006.

[32] R. Narayanan et al. MineBench: A benchmark suite for data mining workloads. In *IISWC*, 2006.

[33] NVIDIA Corporation. CUDA SDK code samples, 2009.

[34] M. K. Qureshi. Adaptive spill-receive for robust high-performance caching in CMPs. *HPCA*, 2009.

[35] K. K. Rangan et al. Thread motion: Fine-grained power management for multi-core systems. In *ISCA*, 2009.

[36] P. Ranganathan et al. The interaction of software prefetching with ILP processors in shared-memory systems. *ISCA*, 1997.

[37] S. Somogyi et al. Spatio-temporal memory streaming. *ISCA*, 2009.

[38] R. Strong et al. Fast switching of threads between cores. *SIGOPS Oper. Syst. Rev.*, 43(2), 2009.

[39] M. A. Suleman et al. ACMP: Balancing hardware efficiency and programmer efficiency. Technical Report TR-HPS-2007-001, Univ. of Texas at Austin, 2007.

[40] M. A. Suleman et al. An asymmetric multi-core architecture for accelerating critical sections. Technical Report TR-HPS-2008-003, Univ. of Texas at Austin, 2008.

[41] M. A. Suleman et al. Accelerating critical section execution with asymmetric multi-core architectures. *ASPLOS*, 2009.

[42] J. M. Tendler et al. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[43] W. Thies et al. Streamit: A language for streaming applications. In *11th Conf. on Compiler Construction*, 2002.

[44] P. Trancoso and J. Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *ICPP*, 1996.

[45] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *SIGCOMM*, 1997.

[46] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, 2005.