Parallel Computing 37 (2011) 26–41

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Architectural support for thread communications in multi-core processors

Sevin Varoglu *, Stephen Jenks

Department of Electrical Engineering and Computer Science, University of California, Irvine, USA

ARTICLE INFO

Article history:
Received 11 July 2008
Received in revised form 28 April 2010
Accepted 12 August 2010
Available online 17 September 2010

Keywords:
Multi-core processors
Parallelism and concurrency
Shared memory

ABSTRACT

In the ongoing quest for greater computational power, efficiently exploiting parallelism is of paramount importance. Architectural trends have shifted from improving single-threaded application performance, often achieved through instruction level parallelism (ILP), to improving multithreaded application performance by supporting thread level parallelism (TLP). Thus, multi-core processors incorporating two or more cores on a single die have become ubiquitous. To achieve concurrent execution on multi-core processors, applications must be explicitly restructured to exploit parallelism, either by programmers or compilers. However, multithreaded parallel programming may introduce overhead due to communications among threads. Though some resources are shared among processor cores, current multi-core processors provide no explicit communications support for multithreaded applications that takes advantage of the proximity between cores. Currently, inter-core communications depend on cache coherence, resulting in demand-based cache line transfers with their inherent latency and overhead. In this paper, we explore two approaches to improve communications support for multithreaded applications. *Prepushing* is a software controlled data forwarding technique that sends data to destination's cache before it is needed, eliminating cache misses in the destination's cache as well as reducing the coherence traffic on the bus. *Software Controlled Eviction (SCE)* improves thread communications by placing shared data in shared caches so that it can be found in a much closer location than remote caches or main memory. Simulation results show significant performance improvement with the addition of these architecture optimizations to multi-core processors.

1. Introduction

Multi-core processors have become prevalent to address the growing demand for better performance. Recently, multi-core processors with three-level caches have been introduced to the market [1]. However, we cannot fully exploit the potential of these powerful processors, and there is an increasing gap between new processor architectures and mechanisms to exploit the proximity and possible connectivity between cores in these architectures. Even though the number of cores and cache levels per chip is increasing, the software and hardware techniques to exploit the potential of these powerful processors are falling behind. To achieve concurrent execution of multiple threads, applications must be explicitly restructured to exploit thread level parallelism, either by programmers or compilers. Conventional parallel programming approaches do not efficiently use shared resources, like caches and the memory interface on multi-core processors. Thus arises the challenge to fully exploit the performance offered by today's multi-core processors, especially when running applications with multiple threads that frequently need to communicate with each other. Even though there are shared resources among the cores, there is no explicit communications support for multithreaded applications to take advantage of the proximity between these cores.

* Corresponding author. Tel./fax: +1 949 735 6036.
E-mail addresses: sevin.varoglu@gmail.com (S. Varoglu), sjenks@uci.edu (S. Jenks).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2010.08.006

Data parallel programming is a common approach in which data and associated computation are partitioned across multiple threads. While this approach is easy to program, it raises the following issues in multi-core processors:

- *Simultaneous accesses may overwhelm the shared memory interface.* As each thread works independently on its own data, the shared memory interface may be overwhelmed due to simultaneous accesses to different data partitions. The memory controller may not be adequate to handle multiple memory request streams simultaneously.
- *Multiple threads may cause cache pollution.* As each thread works concurrently on different data partitions, they may evict each other's data from any shared cache when bringing data from the main memory. This behavior may result in increased miss rates, and consequently execution time may significantly increase.
- *Thread communications depend on cache coherence mechanisms.* As threads communicate via a shared cache or main memory, thread communications depend on cache coherence mechanisms resulting in demand-based data transfers among threads. Cache coherence mechanisms observe coherence state transitions to transfer data among caches. Therefore, the requesting thread has to stall until the data is delivered, as there is likely insufficient out-of-order or speculative work available. This behavior results in increased communication latency and miss rates.

A parallel execution approach called *Synchronized Pipelined Parallelism Model (SPPM)* [2] was introduced to reduce the demand on the memory bus by restructuring applications into producer–consumer pairs that communicate through shared caches rather than main memory. The producer fetches data from the main memory and modifies it. While the data is still in the cache, the consumer accesses it. The producer and consumer need to be synchronized to prevent the producer from getting far ahead of the consumer or to prevent the consumer from getting too close to the producer. Therefore, SPPM enforces a tight synchronization window between the producer and consumer to make producer's data available in the cache for consumer's access. This approach reduces or eliminates the consumer's need to fetch the data from the main memory, resulting in better performance. However, there are some hardware weaknesses such as demand-based data transfers that limit SPPM's performance. Our work is inspired by these hardware weaknesses and then evolved into architectural support for thread communications in multi-core processors. This paper addresses the following problems: *increased communication latency* due to demand-based data transfers and *increased data access latency* due to capacity-based evictions.

In multi-core processors, a mechanism to exploit the cores' proximity and allow fast communications between cores is needed. At the hardware level, thread communications depend on cache coherence mechanisms, resulting in demand-based data transfers. This may degrade performance for data-dependent threads due to the *communication latency* of requesting data from a remote location, sending an invalidation request (if the data is to be modified), and delivering the data to the destination. Previous research has shown the benefits of data forwarding to reduce communication latency in distributed shared memory multiprocessors. Employing a similar data movement concept within the chip on multi-core processors will extend the communications support offered by today's multi-core processors, and result in reduced communication latency and miss rates. Therefore, we present *prepushing*, which is a software controlled data forwarding technique to provide communications support in multi-core processors [3]. The basic idea in prepushing is to send data to destination's cache before it is demanded, eliminating cache misses in the destination's cache as well as reducing the coherence traffic on the bus.

Furthermore, the conventional capacity-based cache eviction approach that evicts cache lines from higher level caches to lower levels depends on the capacity of the caches. When higher level caches are full, conflicting cache lines are evicted from higher levels to lower levels based on a replacement algorithm. Thus, it is likely to find shared data in remote caches or main memory because the data gets evicted to lower cache levels when higher levels are full. The location of the data has a crucial effect on performance because it may vary depending on the cache sizes and application behavior. If the data requested by a thread is found in the shared cache rather than a remote cache or main memory, the *data access latency* will be less and hence it will take less time to deliver the data to the requesting thread resulting in performance improvement. To reduce data access latency in multi-core processors with shared caches, we present a novel software controlled cache eviction technique in hardware, called *Software Controlled Eviction (SCE)*. The basic idea in SCE is to put shared data in shared caches so that it can be found in a much closer location than remote caches or main memory.

In this paper, we focus on the interaction of software and hardware to improve the performance of multithreaded applications running on multi-core processors. So, we present two approaches to improve thread communications in multi-core processors by eliminating demand-based data transfers. Then, we compare their performance behaviors to find out their similarities and differences, and understand which approach should be used under what circumstances. The rest of the paper is organized as follows: Section 2 describes our architectural techniques to improve communications support on multi-core processors. Section 3 describes our benchmark applications and presents our performance estimation models. Section 4 presents the simulation results with a detailed analysis. Section 5 presents related studies by other researchers that aim to improve the application performance on multi-core processors. Finally, Section 6 summarizes our contributions and outlines future work to be done to further improve communications support on multi-core processors.

2. Thread communications approaches

Software and hardware techniques to exploit parallelism in multi-core processors are falling behind, even though the number of cores per chip is increasing very rapidly. In conventional parallel programming, data parallelism is exploited

by partitioning data and associated computation across multiple threads. This programming model is called *Data Parallelism Model (DPM)*. Even though this approach is easy to program, multiple threads may cause cache pollution as each thread works on different data partitions. When each thread brings its data from memory to shared cache, conflict misses may occur. In addition, data communications among concurrent threads depend on cache coherence mechanisms because threads communicate via shared caches or main memory. This approach results in demand-based data transfers, and hence the requesting thread has to stall until the data is delivered, as there is likely insufficient out-of-order or speculative work available. Consequently, performance degradation occurs due to increased communication latency, data access latency, and miss rates. In this section, we present two approaches to overcome these problems.

2.1. Prepushing

2.1.1. Motivation

In DPM, the number of conflict misses increases on multi-core processors with shared caches as each thread works on different data partitions. Consequently, the memory interface becomes overburdened and performance degrades due to large numbers of cache misses. A parallel execution approach, called *Synchronized Pipelined Parallelism Model (SPPM)*, was introduced to reduce the demand on the memory bus by restructuring applications into producer–consumer pairs that communicate through caches rather than main memory. First, the producer fetches data from the main memory and modifies it. While the data is still in the cache, the consumer accesses it. This approach reduces or eliminates the consumer's need to fetch the data from the main memory, resulting in better performance. However, there are some hardware weaknesses that limit SPPM's performance and demand-based thread communications is one of them.

Data forwarding has been shown to reduce communication latency and miss rates, as discussed in Section 5. Previous studies mostly involve distributed shared memory multiprocessors and directory cache coherence protocols. However, we bring the data forwarding idea (i.e. sending data before it is requested) to today's multi-core processors as a flexible and portable communications support for multithreaded applications. *Prepushing*, which is a software controlled data forwarding technique, facilitates data transfers among threads [3]. It is flexible because the data transfers are specified using software hints, and portable because it can be applied to any cache coherence protocol with minor changes.

Fig. 1 illustrates the execution behavior of a single producer–consumer pair in conventional demand-based approach and prepushing. Conventionally, data is pulled by the consumer rather than pushed by the producer. Each data block consists of several cache lines. As the consumer accesses a data block, it issues cache line requests and must wait or stall while each block is retrieved from the remote cache. If each cache line is consumed in less time than it takes to get the next one, prefetching will not be fully effective at masking the remote fetch latency. Furthermore, prior prefetched cache lines may still have been in use by the producer, so a prefetcher may add to the coherence traffic overhead rather than reduce it. On the other hand, prepushing allows the producer to send the data as soon as it is done with it. Therefore, the consumer receives the data by the time it is needed. Consequently, the prepushing approach reduces the number of data requests, the number of misses, and the communication latency seen by the consumer.

The attributes that define data transfers are categorized as *prepush policy* and *placement policy*. The prepush policy defines the coherence state of prepushed cache lines, while the placement policy defines the location of prepushed cache lines to be written. According to the prepush policy, data transfers can be done either in *shared* or in *exclusive* state. The placement policy states that a prepushed cache line can be written to either the L1 cache or the L2 cache. We consider MOESI cache coherence protocol. In *shared prepushing*, the prepush policy is to send each cache line to the consumer in *shared* state as soon as it is produced, while the producer's cache line assumes *owned* state. This sharing behavior allows the producer to reuse the data, if necessary. However, in *exclusive prepushing*, the prepush policy is to send each cache line to the consumer in *exclusive* state as soon as the producer no longer needs it, thus invalidating it in the producer's cache. This behavior is similar to migratory sharing [4]. It improves performance when the consumer needs to write to the cache line, as it eliminates the invalidate request to make the cache line exclusive.

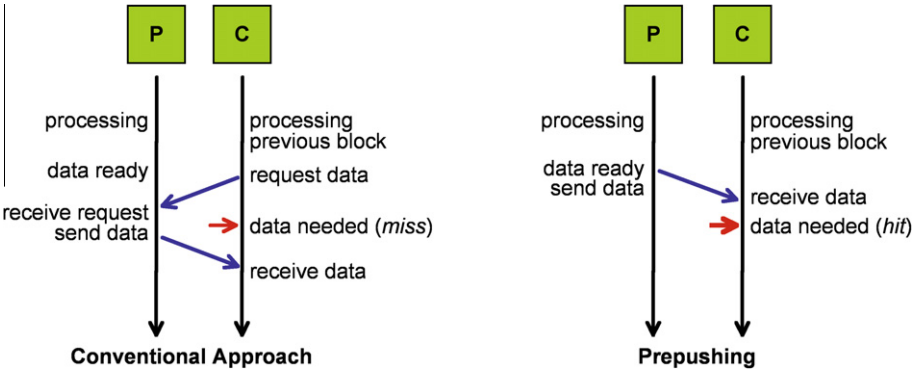


Fig. 1. Execution behaviors.

Fig. 2 shows the cache coherence behavior of a single producer–consumer pair, comparing the conventional approach (CON) with shared prepushing (PUSH-S) and exclusive prepushing (PUSH-X). Initially, the producer's read request changes the coherence state of the cache line from *invalid* to *exclusive*. After the producer modifies the cache line, the coherence state changes to *modified*. In the conventional approach, the consumer issues an explicit request to retrieve the cache line from the producer's cache. As the cache line will be read by other requesters, the producer's copy becomes *owned* and the cache line is sent to the consumer in *shared* state. When the consumer needs to modify the cache line, it has to send an invalidation message to the producer so that the producer's copy becomes invalid. The consumer's copy then becomes *modified*. This execution behavior results in fewer requests in shared prepushing and exclusive prepushing. First, the consumer doesn't incur any cache misses because the cache line is found in its cache by the time it is needed. So, the consumer doesn't need to issue any explicit cache line requests. Second, there is no explicit invalidation message in exclusive prepushing because the cache line is forwarded exclusively. Thus, the prepushing approach reduces the number of data requests, the number of misses, and the communication latency seen by the consumer.

2.1.2. Implementation

To evaluate the prepushing approach, we extended GEMS [5] Ruby memory model, which is based on the Simics [6] full system simulation platform. Each thread is assigned to a specific processor to prevent necessary data from being evicted from the cache during context switches. Since simulations take place on a Solaris 9 platform, *processor_bind* function is used to assign each thread to a specific processor. The implementation of the prepushing approach involves inserting software hints into applications to assist the underlying hardware mechanism with data transfers. The underlying hardware mechanism consists of a component, called *prepusher*, integrated with the cache controller. The prepusher's algorithm is shown in Algorithm 1. When the producer is done with a block of data, it signals the prepusher to initiate data movement by providing information such as data block address, data block size, and destination. The following steps describe how prepushing takes place in Fig. 3.

Algorithm 1. Prepusher's algorithm

translate *virtualAddress* to *physicalAddress*
while *remainingBytes* $\geqslant$ *cacheLineSize* or *remainingBytes* > 0 **do**
  if cache line request not issued by destination **then**
    add *cacheLineAddress* to prepush queue
  end if
  remainingBytes$\leftarrow$*remainingBytes* $-$ *cacheLineSize*
  compute next *cacheLineAddress*
end while

(1) When the producer is done with a block of data, it sends a signal to the prepusher so that the prepusher begins data transfers.
(2) The prepusher gathers information about the request such as data block address, data block size, destination, and prepush type. Then, it determines which cache lines comprise the data block and calculates the physical address of each cache line. If a request has not been issued for a cache line, the prepusher adds the cache line address to the tail of the prepush queue.

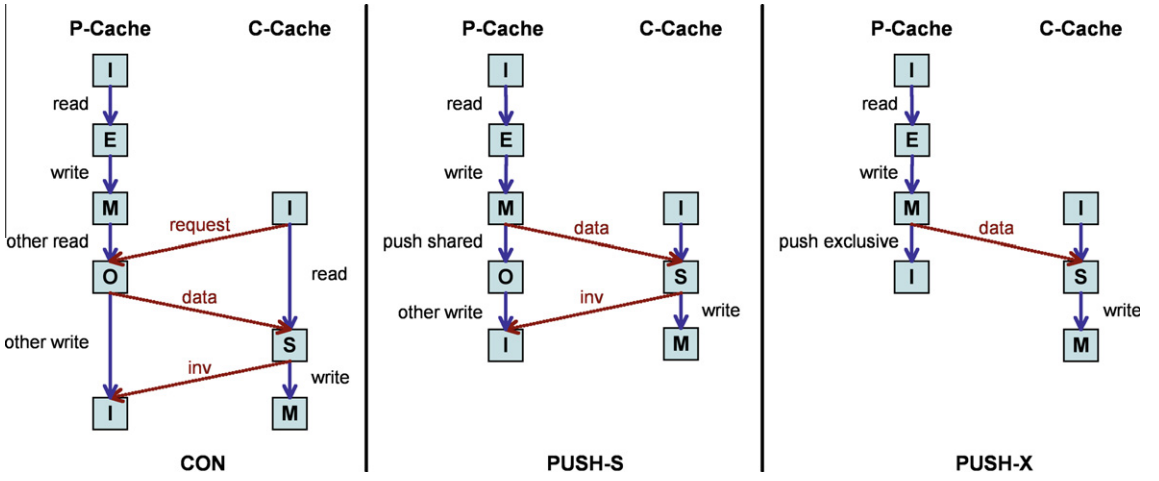


Fig. 2. Cache coherence behaviors.

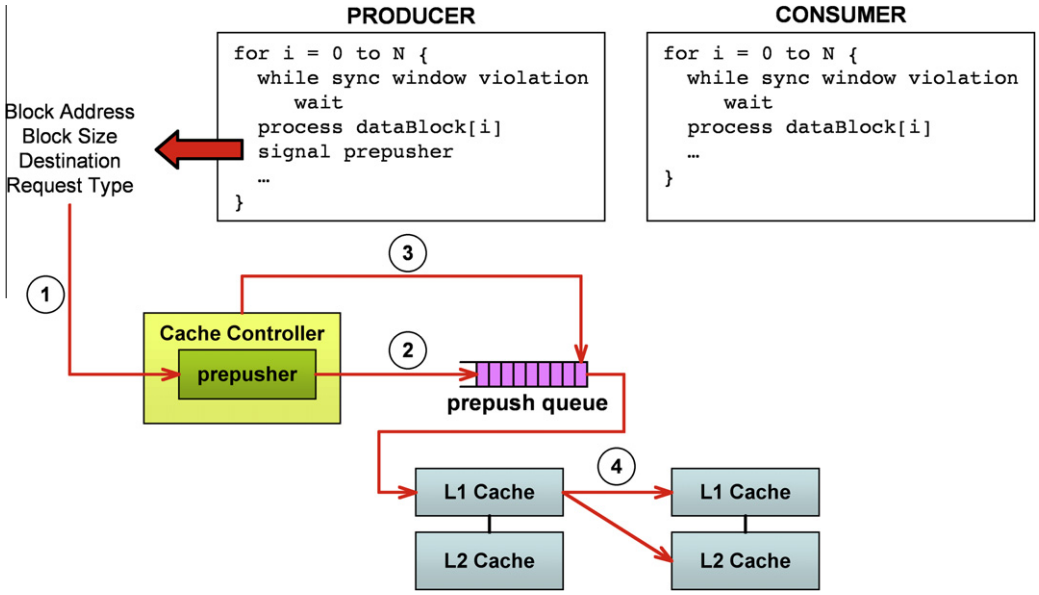


Fig. 3. Prepushing steps.

(3) The cache controller reads the request at the head of the prepush queue and then accesses the L1 cache to retrieve the cache line. The data transfers are done either in *shared* or in *exclusive* mode, based on the prepush request type.
(4) The cache line is prepushed to the L1 cache or the L2 cache of the consumer, depending on the prepushing model. On the consumer side, if a tag exists for a particular prepushed cache line, it is directly written to the existing location. Otherwise, it is written to a newly allocated location, which may require a cache line replacement.

We added new states, events, and actions to the cache controller to define the prepushing behavior. On the sender side, the cache controller is responsible for forwarding cache lines according to the prepush policy. If there is a request in the prepush queue, the cache controller checks whether the cache line exists in the L1 cache or not. Then, it checks if the existing cache line is in modified state, and triggers an event based on the request type. The cache line being in modified state means that it has been recently written, in other words the sender thread is done with that cache line. If the cache line does not exist in the L1 cache or exists in another state, the prepush operation is cancelled.

On the receiver side, the cache controller is responsible for retrieving cache lines according to the placement policy. If there is any incoming cache line, the cache controller checks whether the cache line exists in the L1 cache or not. If it does, it triggers an event based on the request type. Otherwise, the cache controller checks if there is any available space for the cache line in the L1 cache. Then, a space is allocated for the prepushed cache line and an appropriate event is triggered. If there is no space in the L1 cache, then the L2 cache is tried. If there is available space in the L2 cache, that means the conflicting cache line in the L1 cache must be moved to the L2 cache so that there is space for the prepushed cache line in the L1 cache. Otherwise, the conflicting cache line in the L2 cache must be written back to the main memory so that there is space in the L2 cache.

The cache controller behaves similarly when it retrieves data prepushed to the L2 cache. Prepushed cache lines are written to the L1 cache as long as there is available space. If there is no available space in the L1 cache, the cache controller checks whether the cache line exists in the L2 cache or not. If it does, it triggers an event based on the request type. Otherwise, the cache controller checks if there is any available space for the cache line in the L2 cache. Then, a space is allocated for the prepushed cache line and an appropriate event is triggered. If there is no space in the L2 cache, then the conflicting cache line in the L2 cache must be replaced and written back to the main memory.

2.2. Software controlled eviction

2.2.1. Motivation

Multi-core processors with three-level caches such as AMD's Phenom processor have been recently introduced. In such processors, the shared L3 cache acts as a buffer that keeps data and instructions other cores may have requested. This allows other cores to access data and instructions more quickly than going out to main memory or other cores' private caches. Thus, the location of shared data has a crucial effect on the performance of multithreaded applications when running on multi-core processors with many cache levels. The optimum approach is to find the data in the local L1 cache, which is provided by the prepushing approach. Prepushing addresses both communication latency and L1

cache miss rates to improve performance. However, it may introduce drawbacks such as increased pressure on the L1 cache. To further improve thread communications on multi-core processors, we present a novel software controlled hardware technique called *Software Controlled Eviction (SCE)* [7]. SCE puts shared data in the L3 cache; therefore, the data needed by other threads is found in a much closer location rather than remote caches or main memory, resulting in reduced data access latency.

Fig. 4 illustrates inter-core communications with the conventional approach and the SCE approach. The conventional eviction approach is based on the capacity of the caches and evicts conflicting cache lines from higher level caches to lower levels when higher level caches become full. This approach may degrade performance because it introduces the overhead of fetching the data from remote caches or main memory. As illustrated in Fig. 4(a), inter-core communications take place via cache-to-cache transfers or through main memory. The SCE approach, however, aims to improve performance by evicting cache lines that will be needed by other threads to the L3 cache. Since the L3 cache is shared, the data can be found in a closer location rather than the remote caches or main memory, resulting in reduced access latency. This approach also turns the L3 cache misses into L3 cache hits. Fig. 4(b) shows that inter-core communications do not involve fetching data from remote caches or main memory as the shared data resides in the shared L3 cache.

2.2.2. Implementation

Implementing SCE approach involves inserting software hints into applications to assist the underlying hardware mechanism with cache line evictions. In the applications, each thread is assigned to a specific processor to prevent necessary data from being evicted from the cache during context switches. SCE is software controlled in that the sender thread notifies the eviction manager so that it starts evicting cache lines to the L3 cache. To explore the behavior of the SCE approach, we modified GEMS Ruby memory model and added an *eviction manager* as a hardware component integrated with the cache controller. The eviction manager's algorithm is shown in Algorithm 2.

Algorithm 2. Eviction Manager's algorithm

```
translate virtualAddress to physicalAddress
while remainingBytes ⩾ cacheLineSize or remainingBytes > 0 do
  if cache line not requested then
    add cacheLineAddress to eviction queue
  end if
  remainingBytes←remainingBytes − cacheLineSize
  compute next cacheLineAddress
end while
```

Similar to the prepushing approach, when the producer is done with a block of data, it signals the eviction manager to initiate cache line evictions by providing information such as data block address and data block size. Upon receiving a signal from the application, the eviction manager determines the physical address of the base cache line in the requested data block. There is a queue, called *eviction queue*, associated with the eviction manager that contains information about data that will be evicted from the local L1 cache to the L3 cache. As long as there is data to be transferred, the eviction manager adds the cache line address to the eviction queue, if it has not been requested. The cache controller is then responsible for cache line evictions. We added new states, events, and actions to the cache controller to define the prepushing behavior. Algorithm 3 presents the cache controller's algorithm that evicts cache lines from the L1 cache. If there is a request in the eviction queue, the cache controller checks whether the cache line exists in the L1 cache or not. If it does, the cache controller checks if the existing cache line is in modified state, and then triggers an eviction event. If the cache line does not exist in the L1 cache or exists in another state, the eviction process is cancelled.

Algorithm 3. Cache Controller - Evict from L1 Cache

```
while eviction-queue not empty do
  if evictedCacheLine exists in L1 cache then
    if state (evictedCacheLine) = modified then
      trigger eviction event
    else
      cancel eviction {invalid state}
    end if
  else
    cancel eviction {cache line not present}
  end if
end while
```

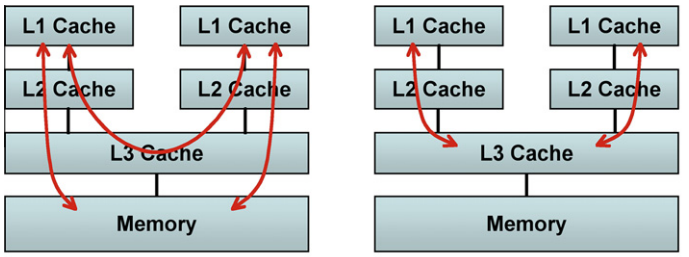


Fig. 4. Comparison of inter-core communications.

Furthermore, the cache controller is responsible for retrieving evicted cache lines in the L3 cache. If there is any incoming cache line, the cache controller checks if it exists in the L3 cache. If so, it triggers an event to write the cache line to the L3 cache. Otherwise, the cache controller checks if there is any available space for the cache line in the L3 cache. Then, a space is allocated for the evicted cache line and an event is triggered to write the cache line. If there is no space in the L3 cache, the conflicting cache line in the L3 cache must be written back to the main memory to make room.

2.3. Race conditions

Since both approaches forward data to the destination before it is demanded, it is likely to run into race conditions. Therefore, the prepushing and SCE implementations consider the following race conditions and take actions accordingly.

(1) *An explicit cache line is requested before it is forwarded to the destination.* After the cache line address is calculated, the request table is searched to see if the corresponding cache line has been explicitly requested. If so, prepushing or eviction of that cache line gets cancelled. If this condition is not handled, the cache line will be sent to the remote cache, incurring extra network traffic.
(2) *An explicit cache line is requested after it is forwarded to the destination.* On the destination side, a cache line is accepted if it has not already been requested. When the destination issues a cache line request, the cache controller changes the coherence state of the cache line to an intermediate state. Therefore, it is possible to find out whether the cache line has been requested or not. If this race condition is not handled, the cache line will be written to the cache twice, incurring extra overhead.
(3) *A dirty cache line is explicitly requested after it is forwarded to the destination.* If the source has the only copy of a dirty cache line, exclusive prepushing or eviction of such a cache line invalidates the only copy in the system. To avoid such race conditions, the synchronization window between the producer and consumer is adjusted to be larger in exclusive prepushing.

2.4. Hardware realization

The two proposed architecture optimizations can be implemented via a memory mapped interface with a dedicated memory region for communications support among cores on the chip. This space only exists on the processor and the addresses will never be seen on an external bus. The implementations of *Prepushing* and *Software Controlled Eviction (SCE)* involve modifications to cache controllers. The prepushing and SCE behaviors should be defined in the cache controllers by adding new states and events. Since the applications will be modified to notify the underlying hardware mechanism (i.e. prepusher and eviction manager) to begin data transfers, a mechanism to inform the processor core of these notifications will be necessary. Rather than adding new instructions to the ISA, this can be done by using the memory mapped interface. So, the applications should write to dedicated memory locations to notify each processor core of data transfers. This memory mapped interface can be implemented to support one-to-one, one-to-many, and one-to-all mappings, such that a single processor core, many processor cores, or all processor cores are notified when a particular memory location is written. For example, when sending a signal to only one processor core, the memory location associated with it should be accessed. To notify many processor cores, the memory location associated with all destination cores should be accessed.

3. Performance estimation

3.1. Benchmarks

This section describes several applications used as benchmarks for the performance analysis of prepushing and SCE. Each application has different characteristics that make it possible to isolate the memory behavior in a variety of situations on multi-core processors. All of the benchmarks use SPPM programming model, such that the producer fetches data it needs from the main memory, generates its results, and updates the data in the cache. While the data is still in the cache, the

consumer fetches and uses the data. The producer and consumer are synchronized to prevent the producer from getting far ahead of the consumer. As discussed earlier, prepushing and SCE implementations include inserting software hints into applications.

3.1.1. ARC4 stream cipher

ARC4 stream cipher (ARC4) is the *Alleged RC4*, a stream cipher commonly used in protocols such as Secure Sockets Layer (SSL) for data security on the Internet and Wired Equivalent Privacy (WEP) in wireless networks. The encryption process uses a secret user key in a key scheduling algorithm to initialize internal state, which is used by a pseudo-random number generator to generate a keystream of pseudo-random bits. Because the generation of each byte of the keystream is dependent on the previous internal state, the process is inherently sequential, and thus non-parallelizable using conventional data parallelism models. By treating the keystream generation and the production of ciphertext as producer and consumer, we exploit the inherent concurrency. The data access pattern is a write followed by a remote read as the producer writes to the shared data and the consumer reads from the shared data.

3.1.2. Finite-difference time-domain method

The Finite-Difference Time-Domain (FDTD) method is an extremely memory-intensive electromagnetic simulation [8]. It uses six three-dimensional arrays, three of which constitute the electric field and the other three constitute the magnetic field. During each time step, the magnetic field components are updated using values of the electric field components from the previous time step. This is followed by an update of the electric field components using values of the magnetic field components computed in the current time step. The FDTD application is restructured into a single producer–consumer pair such that the producer does the magnetic field update while the consumer does the electric field update. The data access pattern is a write followed by a remote read for magnetic field components, and a read followed by a remote write for electric field components.

3.1.3. Ping pong benchmark

The Ping Pong benchmark (PPG) is a simple benchmark that pingpongs data between two threads back and forth in a number of iterations. The threads can be assigned to read from or write to the shared data, which is partitioned into blocks. The PPG benchmark is primarily used in evaluating the prepushing and SCE approaches because it is simple and hence estimating the data accesses is straightforward. The producer processes a data block and issues a command to notify the prepusher or eviction manager to start data transfers. The consumer can be configured to either read from or write to the data.

3.1.4. Red black equation solver

The Red Black equation solver (RB) solves a partial differential equation using a finite differencing method [9]. It works on a two-dimensional grid, where each point is updated by combining its present value with those of its four neighbors. To avoid dependences, alternate points are labeled *red* or *black*, and only one type is updated at a time. All the points are updated until convergence is reached. The application is restructured into a single producer–consumer pair such that the producer does the red computation while the consumer does the black computation. The data access pattern is a write followed by a remote write, as both the producer and consumer write to the data.

3.2. Prepushing

This section discusses a performance estimation model for the prepushing approach and focuses on the PPG benchmark as an example. In the PPG benchmark, the data size and block size are given by the user. The block size determines the number of blocks processed during execution. Eqs. 1 and 2 show how the number of blocks and the number of cache lines are calculated, respectively. Given a block size of 32 KB and a cache line size of 64 B, Table 1 shows the number of blocks and the number of cache lines for a given set of data sizes. Eq. 3 shows how to calculate the number of cache lines per block. The number of cache lines per block is 512 in all cases.

$$N_B = \frac{S_D}{S_B}, \tag{1}$$

$$N_{CL} = \frac{S_D}{S_{CL}}, \tag{2}$$

$$N_{CLB} = \frac{S_B}{S_{CL}}, \tag{3}$$

$$N_{access} = I * N_B * N_{CLB}, \tag{4}$$

where I is number of iterations, N_{access} is number of accesses to cache lines, N_B is number of blocks, N_{CLB} is number of cache lines per block, S_D is size of data, S_B is size of blocks, S_{CL} is size of cache lines.

Using this information, the number of cache line accesses for iterations I can be calculated as shown in Eq. 4. To analyze the execution behavior of the conventional approach and the prepushing approach, the number of cache line accesses is calculated. Then, the execution behavior of the PPG benchmark is considered to estimate the number of cache misses for each

Table 1
PPG blocks and cache lines.

Data size (K)	Blocks	Cache lines
32	1	512
256	8	4,096
512	16	8,192
1024	32	16,384

approach. The PPG benchmark is evaluated using shared prepushing when the producer writes to and the consumer reads from the shared data because shared prepushing leaves the prepushed data in shared state, allowing the consumer to re-use the data, as necessary. However, when both the producer and consumer write to the shared data, the PPG benchmark is evaluated using exclusive prepushing. The conventional data transfers are demand-based, so if a cache line does not exist in local caches it will be requested from remote caches or main memory.

In the PPG benchmark, when the producer writes to and the consumer reads from the shared data, the producer will miss the data in its L1 cache only in the first iteration (cold misses) as long as the data fits in the L1 cache. When the data does not fit in the L1 cache, the producer will miss in every iteration. Regardless of the data size, the consumer will miss the data in every iteration because the producer will request the data exclusively, invalidating the consumer's copy. However, the shared prepushing approach sends data in shared state to the destination. So, it is estimated that the producer's L1 cache misses will be the same as in the conventional approach, but the consumer will only incur cold misses when data is prepushed to the L1 cache. There will not be any difference when the data is prepushed to the L2 cache. In the conventional approach, when both the producer and consumer write to the shared data, the only difference is that the producer will miss the data in every iteration because the consumer will request the data exclusively in every iteration, invalidating the producer's copy. There is no decrease in L1 cache misses when the data is prepushed to the L2 cache.

When the producer does not find the data in its L1 cache, it will look for it in the L2 cache. In the prepushing approach, the producer's L2 misses will be the same as in the conventional approach. As for the consumer, it is estimated that there will not be any L2 cache misses in shared prepushing and exclusive prepushing. When the data is prepushed to the L1 cache, the consumer will not need to look for the data in the L2 cache as it will be readily available in the L1 cache. In addition, when the data is prepushed to the L2 cache, there will not be any L2 cache misses because the data will be found in the L2 cache. On a multi-core system with 64 KB L1 cache and 1 MB L2 cache, where each data size fits in each cache level, our experiment showed that the actual results were very close to the estimated results, stating that the shared prepushing approach worked as expected.

This section continues with estimating and evaluating the effectiveness of the prepusher. To avoid the race conditions discussed earlier, the prepusher checks the request table to see if the consumer has explicitly requested a cache line, before pushing it to the consumer. If so, the prepusher cancels the prepushing of that particular cache line. It is important to find out the effectiveness of the prepushing approach by comparing the ideal prepushed cache lines to the actual prepushed cache lines. To estimate the ideal prepushed cache lines, Eqs. 1 and 3 are used to calculate the number of blocks and the number of cache lines per block, respectively. Then, the number of prepushed cache lines, N_{PCL}, which is the number of cache line accesses is calculated using Eq. 4.

In simulations, prepushed cache lines are recorded to see the effectiveness of the prepusher. Table 2 and 3 show the effectiveness of the prepusher in multi-core processors with private caches (MPC) and shared caches (MSC), respectively. The ideal represents the maximum number of cache lines that can be prepushed during simulation (N_{PCL}), while the others represent the actual number of cache lines that were prepushed in each model. The results show that the prepusher works very effectively in all cases, almost achieving the ideal outcome.

3.3. Software controlled eviction

This section discusses a performance estimation model for the SCE approach, which is similar to that of the prepushing approach. It uses the PPG benchmark as an example. To analyze the execution behavior of the conventional eviction approach and the SCE approach, Eq. 4 is used to calculate the number of cache line accesses during execution. The conventional

Table 2
Effectiveness of the Prepusher – MPC (ideal vs. actual).

	IDEAL	PUSH-S-L1	PUSH-X-L1	PUSH-S-L2	PUSH-X-L2
ARC4 (10)	163,840	163,524	163,552	163,680	163,680
ARC4 (50)	819,200	818,272	818,272	818,017	818,171
FDTD (20)	64,152	63,500	62,811	63,500	62,351
FDTD (30)	206,976	204,746	203,833	204,746	203,012
FDTD (40)	476,520	473,828	473,814	473,828	473,814
RB (200)	191,900	186,387	191,012	186,387	191,012
RB (400)	864,300	864,299	864,300	864,300	864,300

Table 3
Effectiveness of the Prepusher – MSC (ideal vs. actual).

	IDEAL	PUSH-S-L1	PUSH-X-L1	PUSH-X-L2
ARC4 (10)	163,840	163,760	163,800	163,800
ARC4 (50)	819,200	818,170	818,183	818,183
FDTD (20)	64,152	63,534	64,100	64,100
FDTD (30)	206,976	204,372	204,820	205,324
FDTD (40)	476,520	474,601	474,825	475,800
RB (200)	191,900	191,768	191,900	191,900
RB (400)	864,300	864,235	864,300	864,300

Table 4
Effectiveness of the eviction manager (ideal vs. actual).

	IDEAL	SCE
ARC4 (1)	16,384	16,383
ARC4 (2)	32,768	32,538
ARC4 (4)	65,536	65,508
FDTD (20)	64,152	63,500
FDTD (30)	206,976	203,748
FDTD (40)	476,520	475,952
RB (200)	191,900	191,012
RB (400)	864,300	863,200
RB (600)	2,076,900	2,075,899

eviction approach evicts cache lines to lower cache levels when upper level caches are full. However, the SCE approach invalidates cache lines and sends them exclusively to the destination. Therefore, it is estimated that the producer will miss the data in its L1 cache in every iteration. The consumer will also miss in every iteration because the producer will request the data exclusively in every iteration, invalidating the consumer's copy.

In the conventional approach, when both the producer and consumer write to the shared data, the only difference is that the producer will miss the data in every iteration. The consumer will request the data exclusively in every iteration, invalidating the producer's copy. When the producer does not find the data in its L1 cache, it will look for it in the L3 cache. If the data is not in the L3 cache, then it will be profiled as an L3 miss. In the SCE approach, the producer's L3 misses will be the same as in the conventional approach. However, the consumer will not miss in the L3 cache because it will always find what is looking for. We experimented on a multi-core system with 64 KB L1 cache, 512 KB L2 cache, and 2 MB L3 cache, where each cache level was big enough to fit each data size. Our experiment showed that the actual results were very close to the estimated results, stating that the SCE approach worked as expected.

In order to further see the performance improvement of the SCE approach, the consumer's average memory access time is estimated. In the conventional approach, the consumer always misses in the L1 cache because the data gets invalidated by the producer. Moreover, the consumer never hits the data in the L3 cache. Considering a multi-core system with L1 cache of 3-cycle latency, L2 cache of 12-cycle latency, L3 cache of 30-cycle latency, and main memory of 120-cycle latency, this behavior results in an average memory access time of 165 cycles for the consumer as shown in Eq. 5. The improvement in the SCE approach is that there is no need to fetch the data from the memory as it resides in the L3 cache. Therefore, the average memory access time becomes 45 cycles as shown in Eq. 6.

$$\text{Memory access time} = \text{Hit time} + \text{Miss rate} * \text{Miss penalty} = 3 + 1 * 12 + 1 * 30 + 120 = 165. \quad (5)$$

$$\text{Memory access time} = \text{Hit time} + \text{Miss rate} * \text{Miss penalty} = 3 + 1 * 12 + 1 * 30 = 45. \quad (6)$$

This section continues with estimating and evaluating the effectiveness of the eviction manager. To avoid the race conditions discussed earlier, the eviction manager checks the request table to see if the corresponding cache line has been explicitly requested. If so, eviction of that cache line gets cancelled. To find out the effectiveness of the SCE approach, the ideal evicted cache lines are compared to the actual evicted cache lines. The number of evicted cache lines, N_{ECL}, which is the number of cache line accesses, is calculated using Eq. 4. Actual evicted cache lines are recorded in simulations. Table 4 shows the ideal and actual evicted cache lines. The ideal represents the maximum number of cache lines that can be evicted during simulation (N_{ECL}). The results show that the eviction manager works very effectively in all cases, almost achieving the ideal outcome.

4. Performance evaluation

In order to show the performance of cache management instructions available on today's multi-core processors, we modified PPG benchmark and inserted prefetch instructions to the consumer's code using Intel's PREFETCHT0 instruction. Before accessing cache lines in every iteration, the cache lines are prefetched so that they will be available in the consumer's cache.

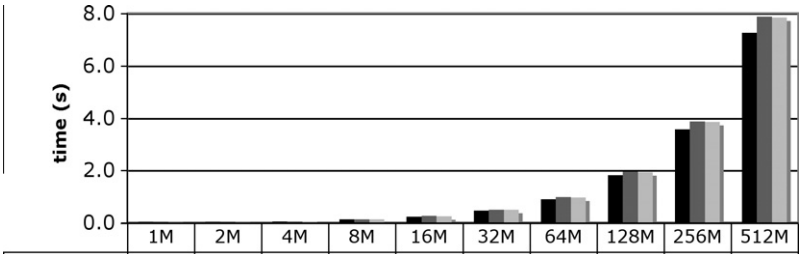
We also inserted cache flush instructions to the producer's code using Intel's CLFLUSH instruction. After the producer is done in every iteration, the cache lines are flushed to the main memory. These experiments were carried out on a system with 2.16 GHz Intel Core 2 Duo processor, 4 MB L2 cache, and 1 GB main memory. Fig. 5 shows PPG user times for various data sizes. The system times are not relevant as the application does not do any I/O. "CON" refers to the base model without any cache flush instructions, while "PREFETCH" refers to the one with prefetch instructions and "FLUSH" refers to the one with cache flush instructions. There is a 7–11% increase in user time when prefetch instructions are used. This is because the producer and consumer both modify cache lines, and prefetching cannot effectively bring cache lines to the consumer's cache as they are still in use in the producer's cache. Ineffectiveness of prefetch instructions clearly shows that prefetching cannot help in multithreaded applications with tightly synchronized threads and there is a need for prepushing to improve performance. Furthermore, there is a 6–10% increase in user time when cache flush instructions are used. Thus, the cache management instructions (e.g. CLFLUSH in Intel architectures) that flush cache lines from caches to main memory are inadequate because flushing to off-chip main memory incurs extra overhead when tightly synchronized threads try to fetch the same data iteratively.

To compare the performance of prepushing and SCE, we used GEMS Ruby memory model as our simulation platform, which uses an in-order processor model. We also implemented prepushing on multi-core processors with three-level caches. The simulated system is a 2 GHz dual-processor SPARC machine with private 64 KB L1 caches, private 512 KB L2 caches, and shared 2 MB L3 cache. The prepush and eviction latencies used in our simulations include the latency to fetch the data from the L1 cache and the latency to transfer the data to the destination cache. Table 5 summarizes our simulation environment. Normalization was the best choice to present our results, since the data size and workload in each benchmark differed remarkably. Thus, all the results shown in the graphs are normalized to the conventional approach. *CON* stands for the conventional approach, *SCE* stands for software controlled eviction, *PUSH-S* stands for shared prepushing, and *PUSH-X* stands for exclusive prepushing. We compare these approaches based on execution time, cache misses, data requests from other caches, and cache pressure.

Fig. 6 shows the normalized execution times of all the benchmarks. The execution time is reduced by 9–12% in ARC4, 18–28% in FDTD, and 13–42% in RB. PUSH-S does not perform as good as SCE and PUSH-X because PUSH-S forwards data in shared state resulting in explicit requests from the consumer to make its data exclusive. The consumer's exclusive data requests can be seen in Fig. 11. In general, the execution time improvement depends on the application behavior, especially on how tight the threads are synchronized and the amount of data accessed per iteration.

The consumer's normalized L1 data cache misses are shown in Fig. 7. As expected, there is no decrease in L1 cache misses with SCE because SCE puts shared data in the shared L3 cache rather than the private L1 cache. Thus, the consumer still incurs L1 cache misses as many as in the conventional approach. PUSH-S and PUSH-X show a significant decrease of 17% in ARC4, 74–80% in FDTD, and 96–98% in RB.

The consumer's normalized L3 cache misses are shown in Fig. 8. ARC4's L3 cache misses are reduced by 17%. There is not much reduction in ARC4's L3 cache misses because each thread works on two streams to allow the producer and consumer to execute concurrently, and hence each thread incurs a lot of capacity misses that cannot be improved. FDTD shows a



	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
CON	0.011	0.020	0.028	0.109	0.223	0.453	0.893	1.803	3.552	7.242
PREFETCH	0.012	0.022	0.031	0.118	0.244	0.484	0.969	1.934	3.853	7.863
FLUSH	0.012	0.022	0.030	0.116	0.238	0.479	0.953	1.922	3.835	7.826

Fig. 5. Pingpong benchmark user time.

Table 5
Simulation environment.

CPU	2 GHz, UltraSPARC III + processors
L1 Cache	64 KB, 2-way associative, 3 cycles
L2 Cache	512 KB, 8-way associative, 12 cycles
L3 Cache	2 MB, 32-way associative, 30 cycles
Cache line size	64 B
Main memory	4 GB, 120 cycles
Operating system	Solaris 9

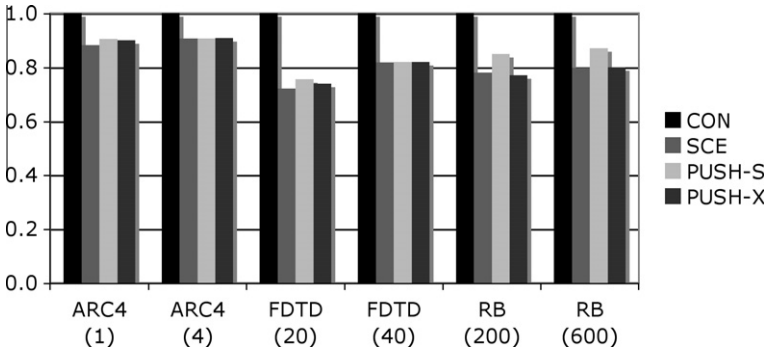


Fig. 6. Normalized execution time.

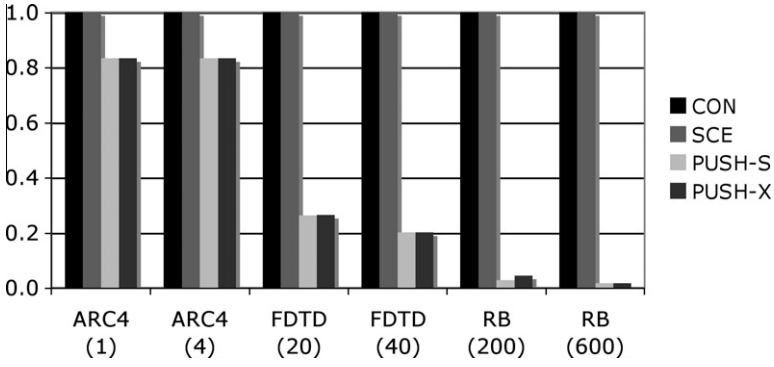


Fig. 7. Normalized L1 data cache misses.

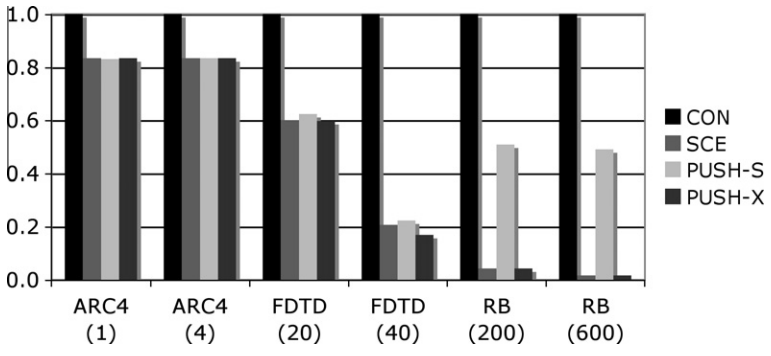


Fig. 8. Normalized L3 cache misses.

reduction of 38–40% with smaller arrays and 79–83% with larger arrays. The difference is because the smaller arrays reside in upper level caches and hence there is not as many L3 cache misses to be eliminated as with the larger arrays. RB's L3 cache misses are reduced by 96–98% in SCE and PUSH-X, while 49–51% in PUSH-S. The difference is due to the explicit requests issued by the consumer to make its data exclusive.

Fig. 9 illustrates the producer's and consumer's combined data requests from the L3 cache. ARC4 shows an improvement of 9–10%, while FDTD and RB show an improvement of 21–46%. The data requests in SCE are slightly more than those in PUSH-S and PUSH-X because SCE puts shared data in the L3 cache and hence the requested data needs to be fetched from the L3 cache rather than upper cache levels. Furthermore, the data requests in PUSH-S are slightly more than those in PUSH-X because PUSH-S forwards data in shared state resulting in explicit data requests to make the data exclusive.

The consumer's shared data requests from the producer's cache are illustrated in Fig. 10 . In ARC4, there is a decrease of 25%. However, FDTD and RB show much better improvement due to the data access patterns of the applications. In ARC4, the consumer reads the shared data and in RB the consumer writes to the shared data. The consumer both reads and writes to

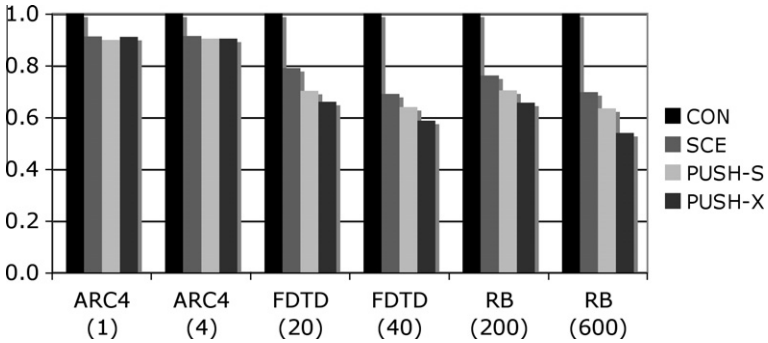


Fig. 9. Data requests from L3 cache.

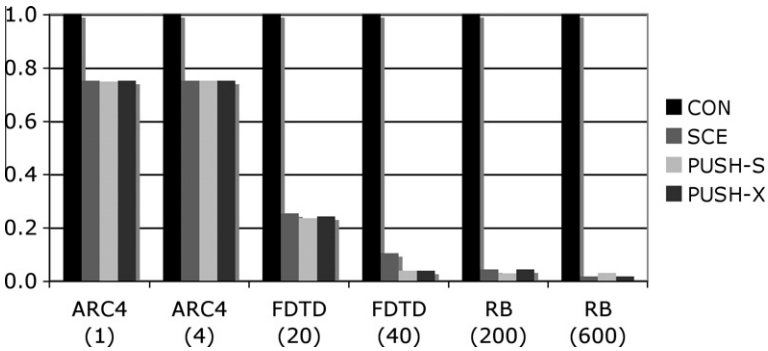


Fig. 10. Consumer's shared data requests from producer's cache.

the shared data in FDTD. So, FDTD shows a reduction of 75–96% and RB shows a reduction of 96–98%. PUSH-S already forwards data in shared state to the consumer's cache and hence the consumer does not need to issue shared data requests to the producer's cache. In SCE, the data that will be needed by the consumer is exclusively available in the L3 cache. Therefore, the consumer does not need to issue shared data requests. When PUSH-X is used, the data is forwarded in exclusive state to the consumer's cache. So, there is no need to request the data in shared state.

Fig. 11 illustrates the consumer's exclusive data requests from the producer's cache. There is a reduction of 2–5% in ARC4, while the improvements in FDTD and RB are much better. FDTD shows a reduction of 2–96% and RB shows a reduction of 28–99%. As expected, PUSH-S is not as effective as SCE and PUSH-X because PUSH-S forwards the data in shared state and when the consumer needs to modify the data, it has to issue explicit data requests to make its copy exclusive.

Fig. 12 shows the pressure on the consumer's cache. In other words, it shows the increase in the number of cache events that move data from the L1 cache to the L2 cache. As expected, there is no change in SCE because the data is put into the shared L3 cache without affecting the consumer's cache. However, PUSH-S and PUSH-X show an increase of 1–6% in the

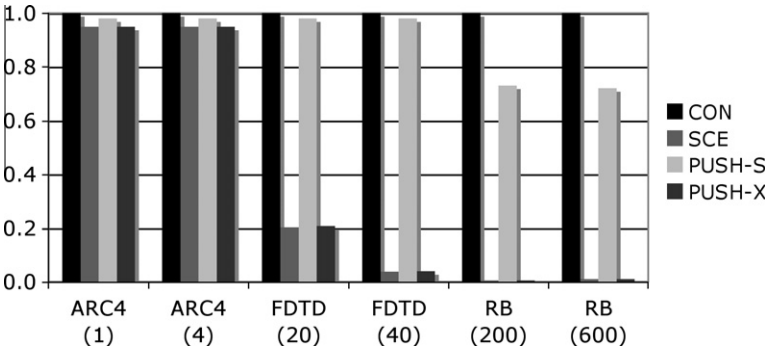


Fig. 11. Consumer's exclusive data requests from producer's cache.

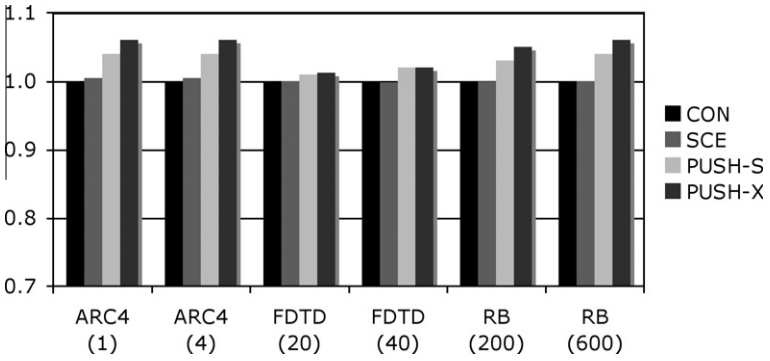


Fig. 12. Pressure on consumer's cache.

number of cache events that move data from the L1 cache to the L2 cache. This behavior does not affect the performance as the data immediately needed by the consumer can be found in its L1 cache.

To sum up, both prepushing and SCE offer the same improvement in terms of execution time. However, SCE does not contribute to a decrease in L1 misses but prepushing significantly reduces L1 misses. So, when running applications with small data sizes that fit in upper level caches (i.e. L1/L2 cache), prepushing will be more effective because the required data will be found in the destination's cache without putting too much pressure on the destination's cache. When running applications with large data sizes that fit in lower level caches (i.e. L3 cache), SCE will be more effective because the shared data will be found in the shared cache without putting any pressure on the destination's cache. If prepushing is used with large data sizes, there will be increased pressure on the destination's cache. In other metrics, SCE and exclusive prepushing behave similarly, while shared prepushing does not perform as good due to exclusive data requests issued from the consumer to the producer. Thus, shared prepushing is more effective for data access patterns with writes followed by remote reads, while SCE and exclusive prepushing are more effective for data access patterns with writes followed by remote writes.

5. Related work

There have been many studies to improve application performance in multiprocessor systems. Data forwarding has been studied to reduce miss rates and communication latencies on distributed shared memory multiprocessors. The prepushing approach was inspired by the effectiveness of data forwarding on large-scale multiprocessors, and designed for small-scale multi-core processors considering flexibility and portability in mind. The Stanford Dash Multiprocessor [10] provides operations *update-write* and *deliver* that allow the producer to send data directly to consumers. The *update-write* operation sends data to all processors that have the data in their caches, while the *deliver* operation sends the data to specified processors. The KSR1 [11] provides programmers with a *poststore* operation that causes a copy of an updated variable to be sent to all caches. The experimental study of the poststore is presented in [12]. A compiler algorithm framework for data forwarding is presented in [13]. *Write-and-Forward* assembly instruction is inserted by the compiler replacing ordinary write instructions. The same data forwarding approach and prefetching are compared in [14]. A study on data forwarding and prefetching shows that prefetching is insufficient for producer–consumer sharing patterns and migratory sharing patterns [15]. A range of consumer- and producer-initiated mechanisms and their performance is studied in [16]. Store-ORDered Streams (SORDS) [17] is a memory streaming approach to send data from producers to consumers in distributed shared memory multiprocessors.

Cachier [18] is a tool that automatically inserts annotations into programs by using both dynamic information obtained from a program execution trace, as well as static information obtained from program analysis. The annotations can be read by a programmer to help reasoning about communication in the program, as well as by a memory system to improve the program's performance. The annotations provided by Cachier do not include flushing data from private caches to shared caches as in the SCE approach. There are only directives to inform the memory system when it should invalidate cache blocks. Helper threading has been studied to reduce miss rates and improve performance on multi-core processors [19–22]. The aim is to accelerate single-threaded applications running on multi-core processors by spawning helper threads to perform pre-execution. The helper threads run ahead of the main thread and execute parts of the code that are likely to incur cache misses. These approaches cause extra work and may result in low resource utilization. Rather than improving a single thread's performance, the prepushing and SCE approaches aim to improve multiple threads' performance by providing an efficient communications mechanism among threads. Furthermore, the memory/cache contention problem among concurrent threads in multi-core processors has been addressed in several studies [23–25]. When the number of threads running on a multi-core platform increases, the contention problem for shared resources arises.

CMP-NuRapid [26] is a special cache designed to improve data sharing in multi-core processors. The three main ideas presented are *controlled replication, in situ communication,* and *capacity stealing.* Controlled replication is for read-only sharing and tries to place copies close to requesters, but avoids extra copies in some cases and obtains the data from an

already-existing on-chip copy at the cost of some extra latency. In-situ communication is for read-write sharing and tries to eliminate coherence misses by providing access to the data without making copies. Capacity stealing enables a core to migrate its less frequently accessed data to unused frames in neighboring caches with less capacity demand. Cooperative Caching [27] is a hardware approach to create a globally-managed, shared cache via the cooperation of private caches. It uses remote L2 caches to hold and serve data that would generally not fit in the local L2 cache, if there is space available in remote L2 caches. This approach eliminates off-chip accesses and improves average access latency.

6. Conclusion

In this paper, we presented two architectural optimizations to improve communications support in multi-core processors. Our techniques combine hardware and software approaches to exploit the locality between processor cores to provide high performance communications between threads. Current processors rely on slow, high-latency cache coherence mechanisms, which quickly become bottlenecks. Our techniques use software hints with supporting communications hardware to proactively move data to the cache where it will be needed before it is accessed. This converts high cost demand-based cache line fetches into cache hits, and hence greatly improved performance. Our thread communications approaches employ *prepushing* and *eviction* to improve the performance of multithreaded applications running on multi-core processors. Simulation results show that both approaches offer the same improvement in terms of execution time, however, they perform differently in terms of cache misses and data requests. Our evaluation exposes the differences of these approaches and allows system designers to select an approach for specific benefits.

Since both *Prepushing* and *Software Controlled Eviction (SCE)* offer the same improvement in terms of execution time, it is important to understand their similarities and differences in order to obtain the best performance. SCE does not contribute to a decrease in L1 cache misses, while prepushing significantly reduces L1 cache misses. So, when running applications with small data sizes that fit in upper level caches (i.e. L1/L2 cache), prepushing will be more effective because the required data will be found in the destination's cache without putting too much pressure on the destination's cache. When running applications with large data sizes that fit in lower level caches (i.e. L3 cache), SCE will be more effective because the shared data will be found in the shared cache without putting any pressure on the destination's cache. If prepushing is used with large data sizes, there will be increased pressure on the destination's cache. SCE and exclusive prepushing behave similarly in reducing L3 cache misses, while shared prepushing does not perform as good as the others due to exclusive data requests. So, applications that have data access patterns with writes followed by remote writes should employ SCE or exclusive prepushing. Shared prepushing should be used for applications that have data access patterns with writes followed by remote reads.

The hardware implementations of prepushing and SCE will involve modifications to cache controllers. The prepushing and SCE behaviors should be defined in the cache controllers by adding new states and events. Since the applications will be modified to notify the underlying hardware mechanism to begin data transfers, a mechanism to inform the processor core of these notifications will be necessary. Rather than adding new instructions to the ISA, this can be done by using a memory mapped interface such that there is a dedicated memory region for communications support among processor cores on the chip. This memory region should be non-cacheable to prevent any interference with user applications. Furthermore, the process of parallelizing applications using SPPM should be simplified. We are designing a software infrastructure to support exploiting parallelism and managing synchronization. This will allow programmers to avoid tedious hand coding and concentrate on the functional aspects of their applications. Efforts are also underway to enhance SPPM for scalability when more processor cores are available for use. The eventual goal of this research is to augment parallelizing compilers to identify and exploit SPPM-style parallelism in suitable applications. To further improve synchronization and communications support on multi-core processors, we will investigate architecture optimizations for applications with different data access patterns and execution behaviors. We will also look into a hardware buffer implementation to store prepushed or evicted data that will relieve the pressure on caches.

References

[1] AMD Phenom Processor, <http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/44109.pdf>.
[2] S. Vadlamani, S. Jenks, The synchronized pipelined parallelism model, in: International Conference on Parallel and Distributed Computing and Systems, 2004.
[3] S. Fide, S. Jenks, Architecture optimizations for synchronization and communication on chip multiprocessors, in: International Parallel and Distributed Processing Symposium: Workshop on Multithreaded Architectures and Applications, 2008.
[4] A. Gupta, W.-D. Weber, Cache invalidation patterns in shared-memory multiprocessors, IEEE Transactions on Computers 41 (7) (1992) 794–810.
[5] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, D.A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) Toolset, in: SIGARCH Computer Architecture News, 2005.
[6] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, IEEE Computer 35 (2) (2002) 50–58.
[7] S. Fide, S. Jenks, Proactive use of shared L3 caches to enhance cache communications in multi-core processors, IEEE Computer Architecture Letters 7 (2) (2008) 57–60.
[8] A. Taflove, Computational Electrodynamics: The Finite-Difference Time-Domain Method, Artech House, Boston, 1995.
[9] D.E. Culler, A. Gupta, J.P. Singh, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufman Publishers Inc., 1997.
[10] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M.S. Lam, The stanford DASH multiprocessor, IEEE Computer 25 (3) (1992) 63–79.

[11] S. Frank, H.B. III, J. Rothnie, The KSR1: Bridging the gap between shared memory and MPPs, in: IEEE Computer Society Computer Conference, 1993.
[12] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, L.W. Dowdy, The KSR1: experimentation and modeling of poststore, in: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1993.
[13] D.A. Koufaty, X. Chen, D.K. Poulsen, J. Torrellas, Data forwarding in scalable shared-memory multiprocessors, in: International Conference on Supercomputing, 1995.
[14] D. Koufaty, J. Torrellas, Comparing data forwarding and prefetching for communication-induced misses in shared-memory MPs, in: International Conference on Supercomputing, 1998.
[15] H. Abdel-Shafi, J. Hall, S.V. Adve, V.S. Adve, An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors, in: International Symposium on High Performance Computer Architecture, 1997.
[16] G.T. Byrd, M.J. Flynn, Producer–consumer communication in distributed shared memory multiprocessors, in: Proceedings of the IEEE, 1999.
[17] T.F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, C. Gniady, A. Ailamaki, B. Falsafi, Store-ordered streaming of shared memory, in: International Conference on Parallel Architectures and Compilation Techniques, 2005.
[18] T.M. Chilimbi, J.R. Larus, Cachier: A tool for automatically inserting CICO annotations, in: International Conference on Parallel Processing, 1994.
[19] C.-K. Luk, Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors, in: International Symposium on Computer Architecture, 2001.
[20] K.Z. Ibrahim, G.T. Byrd, E. Rotenberg, Slipstream execution mode for CMP-based multiprocessors, in: International Conference on Architectural Support for Programming Languages and Operating Systems, 2003.
[21] Y. Song, S. Kalogeropulos, P. irumalai, Design and implementation of a compiler framework for helper threading on multi-core processors, in: International Conference on Parallel Architectures and Compilation Techniques, 2005.
[22] H. Zhou, Dual-core execution: building a highly scalable single-thread instruction window, in: International Conference on Parallel Architectures and Compilation Techniques, 2005.
[23] S. Wang, L. Wang, Thread-associative memory for multicore and multithreaded computing, in: International Symposium on Low Power Electronics and Design, 2006.
[24] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, D. Newell, CacheScouts: Fine-grain monitoring of shared caches in CMP platforms, in: International Conference on Parallel Architecture and Compilation Techniques, 2007.
[25] M. Chu, R. Ravindran, S. Mahlke, Data access partitioning for fine-grain parallelism on multicore architectures, in: International Symposium on Microarchitecture, 2007.
[26] Z. Chishti, M.D. Powell, T.N. Vijaykumar, Optimizing replication, communication, and capacity allocation in CMPs, in: International Symposium on Computer Architecture, 2005.
[27] J. Chang, G.S. Sohi, Cooperative caching for chip multiprocessors, in: International Symposium on Computer Architecture, 2006.