

## An Improvement Over Threads Communications on Multi-Core Processors

<sup>1</sup>Reza Fotohi, <sup>2</sup>Mehdi Effatparvar, <sup>3</sup>Fateme Sarkohaki, <sup>4</sup>Shahram Behzad, <sup>5</sup>Jaber Hoseini balov

<sup>1</sup>Department of Computer Engineering, Germei branch, Islamic Azad University, Germei, Iran

<sup>2</sup>ECE Department, Ardabil Branch, Islamic Azad University, Ardabil, Iran

<sup>3</sup>Department of Computer Engineering, Germei branch, Islamic Azad University, Germei, Iran

<sup>4</sup>Department of Computer Engineering, Germei branch, Islamic Azad University, Germei, Iran

<sup>5</sup>IT Engineering Department, Komvux Skellefteå, Sweden

---

**Abstract:** Multicore is an integrated circuit chip that uses two or more computational engines (cores) places in a single processor. This new approach is used to split the computational work of a threaded application and spread it over multiple execution cores, so that the computer system can benefits from a better performance and better responsiveness of the system. A thread is a unit of execution inside a process that is created and maintained to execute a set of actions/ instructions. Threads can be implemented differently from an operating system to another, but the operating system is in most cases responsible to schedule the execution of different threads. Multi-threading improving efficiency of processor performance with a cost-effective memory system. In this paper, we explore one approach to improve communications for multithreaded. Pre-send 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. we show how we could improve the overall system performance by addition of these architecture optimizations to multi-core processors.

**Key words:** Multi-core processors, Multi-threaded, Pre-send

---

### INTRODUCTION

Multithreading, a new technique to run several tasks simultaneously, aims to increase utilization of a single core. However, using single core will not fully exploit parallelism used in multithreading. Parallel computing could dramatically increase the speed, Efficient and performance of computers by simply putting 2 or more CPUs in only one chip: Multicore.

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. 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 approach 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. 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 [Varoglu, 2011].

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-

---

**Corresponding Author:** Reza Fotohi, Department of Computer Engineering, Islamic Azad University, Germei branch, Ardabil, Iran,  
E-mail: Fotohi.Reza@Gmail.com

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.

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 Pre-Send, which is a software controlled data forwarding technique to provide communications support in multi-core processors. The basic idea in Pre-Send 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.

Due to diminishing returns in single-core design in recent years, all major competitors in the semiconductor industry have all announced a multi-core designs. For instance, Niagara is a Chip-Multithreading (CMT) processor from SUN that features eight cores, each able to simultaneously executing four threads. 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 [Fide, 2008].

Implementation However, ongoing research efforts attempt to change RTSJ to incorporate features that will make it capable of taking advantage of the multi-core technology. Therefore, we believe that software packages and tools like those presented in this paper will become of interest for the embedded systems world as well.

In this paper, we focus on the interaction of software to improve the performance of multithreaded applications running on multi-core processors. So, we present One approach to improve thread communications in multi-core processors. 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 establishing as follows: Section II presents related studies by other researchers that aim to improve the application performance on multi-core processors. Section III describes Thread Communications Approach. Section IV describes Performance Evaluation. Section V describes our architectural techniques to improve communications support on multi-core processors. Section VI the Conclusion.

#### *Related Work:*

Helper threading has been studied to reduce miss rates and improve performance on multi-core processors [Ibrahim, 2003].

CMP-NuRapid [Chishti, 2005] 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. Cooperative Caching [Chang, 2006] 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. Cachier [Chilimbi, 1994] 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. There are only directives to inform the memory system when it should invalidate cache blocks.

The aim is to accelerate single-threaded applications running on multi-core processors by spawning helper threads to perform preexecution. The helper threads run ahead of the main thread and execute parts of the code that are likely to incur cache misses. These approach s cause extra work and may result in low resource utilization. Rather than improving a single thread's performance, the pre-pushing and SCE approach s 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 [Wang, 2006]. When the number of threads running on a multi-core platform

increases, the contention problem for shared resources arises. The Stanford Dash Multiprocessor [Lenoski, 1992] 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 [Frank, 1993] provides programmers with a post store operation that causes a copy of an updated variable to be sent to all caches. The experimental study of the post store is presented in [Rosti, 1993].

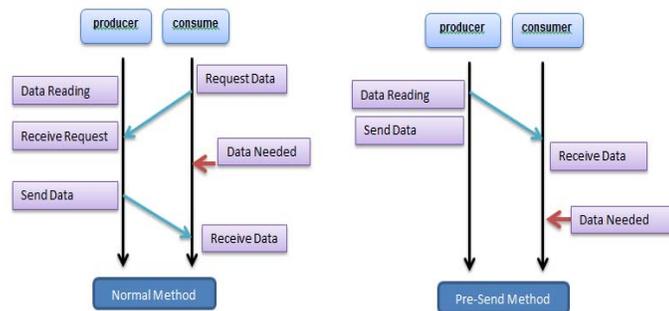
**Thread Communications Approach:**

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 Normal parallel programming, data parallelism is exploited by partitioning data and associated computation across multiple threads. This programming model is called Data Parallelism Model. 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 One approach to overcome these problems.

**Pre-Send (Our Proposed Method):**

This approach reduces or eliminates the Consumer’s need to fetch the data from the main memory, resulting in better performance. Pre-send , which is a software Normal controlled data forwarding technique, facilitates data transfers among threads. 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. Figure. 1 illustrates the execution behavior of a single producer–Consumer pair in Normal demand-based approach and Pre-send . 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. On the other hand, Pre-send 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. Consquinty, the Pre-send approach reduces the number of data requests, the number of misses, and the communication latency seen by the Consumer.



**Fig. 1:** Execution behaviors.

Figure. 2 shows the cache coherence behavior of a single producer–Consumer pair, comparing the Normal approach with Pre-send Approach. 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 Normal 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 Pre-send . 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 Pre-send because the cache line is forwarded exclusively. Thus, the Pre-send approach

reduces the number of data requests, the number of misses, and the communication latency seen by the Consumer.

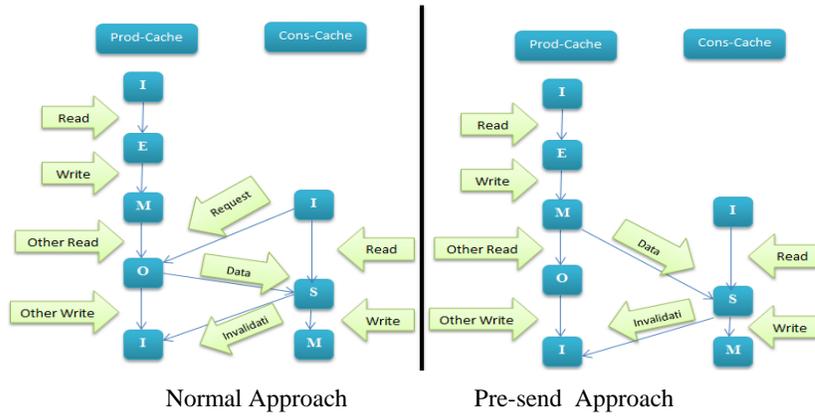


Fig. 2: Cache coherence behaviors.

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 Jobs and the number of Threads for a given set of data sizes. Eq. 3 shows how to calculate the execution Time of per Approach.

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

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

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

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

where I is number of iterations, N access is number of accesses to cache lines, NB is number of blocks, NCLB is number of cache lines per block, SD is size of data, SB is size of blocks, SCL is size of cache lines[Varoglu, 2011]. 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 Normal approach and the Pre-send approach, the number of cache line accesses is calculated.

Table 1: Execution Time Of Normal Approach And Pre-SendApproach.

Prod-Cons	List Job	Normal Approach	Pre-send Approach
1,1	2,4	4.0	3.0
1,2	3,6,8	9.66	5.66
2,2	4,1,9,13	11.75	7.75
2,3	5,6,7,8,9	19.0	11.0
3,3	13,18,20,32,7,1	36.5	29.66
3,4	2,8,10,14,16,20,21	39.57	32.14
4,4	2,7,4,2,8,42,3,56	32.5	26.5

Table 2: Waiting Time Of Normal Approach And Pre-SendApproach.

Prod-Cons	List Job	Normal Approach	Pre-send Approach
1,1	2,4	1.0	0.0
1,2	3,6,8	4.0	0.0
2,2	4,1,9,13	5.0	1.0
2,3	5,6,7,8,9	12.0	4.0
3,3	13,18,20,32,7,1	21.33	14.5
3,4	2,8,10,14,16,20,21	26.57	19.14
4,4	2,7,4,2,8,42,3,56	27.0	11.0

In simulations, Pre-Send cache lines are recorded to see the effectiveness of the Pre-Sender. The ideal represents the maximum number of cache lines that can be pre-Send during simulation, while the others represent the actual number of cache lines that were Pre-Send in each model. The results show that the Pre-Sender works very effectively in all cases, almost achieving the ideal outcome. The results show that the Pre-Send works very effectively in all cases, almost achieving the ideal outcome.

**Performance Evaluation:**

To compare the performance of Normal And Pre-send Approach, we used Eclipse And Java Compiler as our simulation platform, which uses an in-order processor model. We also implemented Pre-send on multi-core processors with three-level caches. The simulated system is a 1 GHz dual-processor machine with private 32 KB L1 caches, private 128 KB L2 caches, and shared 1 MB L3 cache. The Pre-Send and Normal 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 3 summarizes our simulation environment. Normalization was the best choice to present our results. We compare these approach based on execution time.

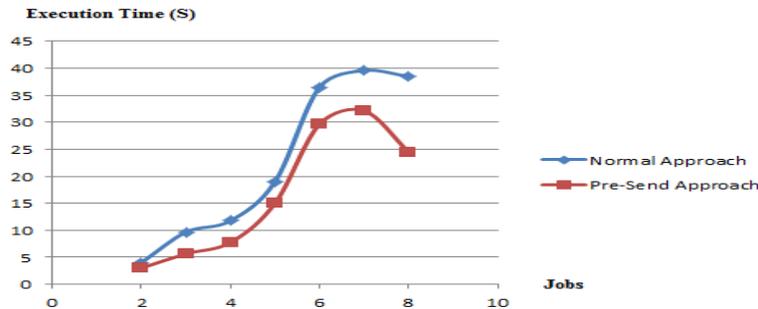
**Table 3:** Simulation Environment.

CPU	1 GHz
L1 Cache	32 KB
L2 Cache	128 KB
L3 Cache	1 MB
Cache Line Size	32 B
Main Memory	2 GB
Operating System	Windows 7

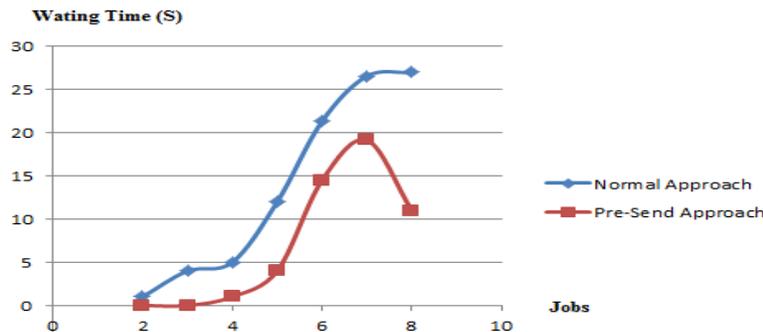
**Our Architecture:**

Pre-Send, which is a software controlled data forwarding technique, facilitates data transfers among threads. It is flexible because the data transfers are specified using software hints.

Figure. 3 and Figure. 4 shows the normalized Execution Time And Waiting Time for Normal Approach and Pre-Send Approach. The Execution Time And Waiting Time is reduced by 4–8% in Pre-Send Approach Ratio Normal Approach . Pre-Send forwards data in shared state resulting in explicit requests from the Consumer to make its data exclusive. 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.



**Fig. 3:** Normalized execution time.



**Fig. 4:** Waiting time.

**Conclusion:**

In this paper, Our technique use software hints for proactively move data to the cache where it will be needed before it is accessed. Our thread communications approach employ Pre-send to improve the performance of multithreaded applications running on multi-core processors. Simulation results show that Pre-send offer the Better improvement in terms of Execution Time and Waiting Time. Our evaluation exposes the differences of these approach and allows system designers to select an approach for specific benefits. Since Pre-send offer the Better improvement in terms of Execution Time And Waiting Time, it is important to understand their similarities and differences in order to obtain the best performance.

Pre-send 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. Pre-send should be used for applications that have data access patterns with writes followed by remote reads. 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.

**ACKNOWLEDGMENT**

The Author Would Like To Thank Dr. Mehdi Effatparvar For Constructive Comments And Suggestions.

**REFERENCES**

- A.M.D., Phenom Processor, <[http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/44109.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/44109.pdf)>.
- Chang, J., G.S. Sohi, 2006. Cooperative caching for chip multiprocessors, in: International Symposium on Computer Architecture.
- Chilimbi, T.M., J.R. Larus, 1994. Cachier: A tool for automatically inserting CICO annotations, in: International Normalference on Parallel Processing.
- Chishti, Z., M.D. Powell, T.N. Vijaykumar, 2005. Optimizing replication, communication, and capacity allocation in CMPs, in: International Symposium on Computer Architecture.
- Chu, M., R. Ravindran, S. Mahlke, 2007. Data access partitioning for fine-grain parallelism on multicore architectures, in: International Symposium on Microarchitecture.
- Fide, S., S. Jenks, 2008. Architecture optimizations for synchronization and communication on chip multiprocessors, in: International Parallel and Distributed Processing Symposium: Workshop on Multithreaded Architectures and Applications.
- Frank, S., H.B. III, J. Rothnie, 1993. The KSR1: Bridging the gap between shared memory and MPPs, in: IEEE Computer Society Computer Conference.
- Ibrahim, K.Z., G.T. Byrd, E. Rotenberg, 2003. Slipstream execution mode for CMP-based multiprocessors, in: International Normalference on Architectural Support for Programming Languages and Operating Systems.
- Lenoski, D., J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M.S. Lam, 1992. The stanford DASH multiprocessor, IEEE Computer, 25(3): 63-79.
- Rosti, E., E. Smirni, T.D. Wagner, A.W. Apon, L.W. Dowdy, 1993. The KSR1: experimentation and modeling of poststore, in: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.
- Varoglu, S., J. Jenks, 2011. Architectural support for thread communications in multi-core processors, in: Parallel Computing.
- Wang, S., L. Wang, 2006. Thread-associative memory for multicore and multithreaded computing, in: International Symposium on Low Power Electronics and Design.