PRODUCER-ORIENTED VERSUS CONSUMER-ORIENTED PREFETCHING:

A COMPARISON AND ANALYSIS OF PARALLEL APPLICATION PROGRAMS

Moriyoshi Ohara

TECHNICAL REPORT: CSL-TR-96-695

June 1996

This research has been supported by ARPA under contracts N00039-91-C-0138 and DABT63-94-C-0054, and IBM Japan, Ltd.

PRODUCER-ORIENTED VERSUS CONSUMER-ORIENTED PREFETCHING: A COMPARISON AND ANALYSIS OF PARALLEL APPLICATION PROGRAMS

Moriyoshi Ohara

Technical Report: CSL-TR-96-695

June, 1996

Computer Systems Laboratory
Department of Electrical Engineering and Computer Systems
Stanford University
Gates Building, A-408
Stanford, CA 94305-9040
pubs@shasta.stanford.edu

Abstract

Due to large remote-memory latencies, reducing the impact of cache misses is critical for large scale shared-memory multiprocessors. This thesis quantitatively compares two classes of software-controlled prefetch schemes for reducing the impact: consumer-oriented and producer-oriented schemes. Examining the behavior of these schemes leads us to characterize the communication behavior of parallel application programs.

Consumer-oriented prefetch has been shown to be effective for hiding large memory latencies. Producer-oriented prefetch (called deliver), on the other hand, has not been extensively studied. Our implementation of deliver uses a hardware mechanism that tracks the set of potential consumers based on past sharing patterns. Qualitatively, deliver has an advantage since the producer sends the datum as soon as, but not before, it is ready for use. In contrast, prefetch may fetch the datum too early so that it is invalidated before use, or may fetch it too late so that the datum is not yet available when it is needed by the consumer. Our simulation results indeed show that the qualitative advantage of deliver can yield a slight performance advantage when the cache size and the memory latency are very large. Overall, however, deliver turns out to be less effective than prefetch for two reasons. First, prefetch benefits from a "filtering effect," and thus generates less traffic than deliver. Second, deliver suffers more from cache interference than prefetch. The sharing and temporal characteristics of a set of parallel applications are shown to account for the different behavior of the two prefetch schemes. This analysis shows the inherent difficulties in predicting future communication behavior of parallel applications from recent history of the application behavior. This suggests that cache accesses involved with coherency in general are much less predictable based on past behavior than other types of cache behavior.

Key Words and Phrases: prefetching, latency hiding, data sharing, cache coherency, shared-memory multiprocessors.

Acknowledgments

Without assistance of many people, this dissertation could not have been realized. Especially, I thank my principal advisor, John Hennessy, for his wisdom and guidance throughout my graduate study at Stanford. I also thank Anoop Gupta, Monica Lam, and Arogyaswami Paulraj for their insights on my dissertation work.

I thank all of my colleagues for making my student life at the farm stimulating. I also thank faculty and students of the DASH/FLASH project. J.P. Singh and Todd Mowry gave me valuable insights for my research. Steve Goldschmidt made tango simulator available to me. Truman Joe gave me a part of his simulation program. I thank Dave Ofelt and Jeff Kuskin as a great officemate. I also thank Steve Woo and Evan Torrie who are insightful and fun to work with.

My thanks also go to support stuffs at Stanford: Charlie Orgish and Thoi Nguyen for keeping the computers running, and Margaret Rowland and Darlene Hadding for making my school life easier. My graduate study was supported by ARPA under contracts N00039-91-C-0138 and DABT63-94-C-0054, and IBM Japan, Ltd. Finally I thank my family, especially my wife, Kyoko, who has shared a graduate student life with me from the beginning.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reducing the impact of memory latency is a key issue for shared-memory multiprocessor systems. In shared-memory multiprocessors, inter-processor communication implicitly occurs by memory accesses. Since memory modules are typically distributed among processor nodes in large-scale systems, processors need to access remote memory to communicate with other processors. Thus, communication can stall processors for long-latency memory accesses and offset the performance gain due to parallel processing. As the number of processors increases, moreover, the communication latency generally increases.

Several techniques to reduce the impact of memory latency have been proposed and evaluated. We can categorize those techniques in two types: latency reducing and latency hiding. Latency reducing techniques include coherent caches and page allocation techniques, which reduce the memory latency by storing data at a location closer to the processor that accesses the data. Latency hiding techniques, on the other hand, include multi-threading, prefetching, and relaxing the consistency model, which let the processor perform useful tasks during long-latency memory accesses. Prefetching by the consumer with coherency caches has been shown to be one of the most useful techniques [12, 27, 29]. This thesis examines consumer-based and producer-based prefetching strategies. The producer-based approach, which we call *deliver* in this thesis, could maximally hide memory latencies because the producer sends the shared datum to prospective consumers at the earliest possible time after the datum is produced. The consumer-based approach, which we simply call *prefetch* in this thesis, on the other hand, might not hide memory latencies as effectively because the consumer may fetch the datum before it is produced, or may not fetch it early enough to hide the latency completely. Qualitative arguments can be made for both capabilities, and the DASH multiprocessor [39] and the KSR1 system [23] actually support both features.[1]

We quantitatively evaluate these two alternatives using a software-controlled mechanism for both; the programmer or a language system explicitly inserts data transfer operations (prefetch or deliver) in the program. The consumer-based prefetch needs to perform prefetch operations early enough so that the prefetched datum comes to the processor by the time the processor needs the datum. Mowry and Gupta [44] have shown that the consumer-based prefetch is not very effective for applications using pointers extensively because addresses to prefetch are often not available early enough. The producer-based deliver, on the other hand, needs to perform a deliver operation when processors write a shared datum for the last time before other processors read the datum. Thus, the last write to a shared datum before a synchronization operation is a good candidate to insert the deliver operation. It is relatively easy to find such places. Even if the program uses pointers, the address for deliver operations is available where these operations should be inserted. Our implementation of a deliver uses a hardware mechanism in a cache directory that keeps track of potential consumers based on past sharing patterns. The deliver operation sends a specified cache line to those potential consumers.

We have chosen to simulate a directory-based cache-coherent NUMA machine as the base in which we evaluate these two schemes. Our simulation results indeed show that the qualitative advantage of the deliver can yield a slight performance advantage when the cache size and the memory latency are very large. For realistic architectural parameters, however, the prefetch scheme generally outperforms the deliver scheme for two reasons. First, the deliver scheme often sends cache lines to consumers so early that the consumers replace the delivered cache lines before they are used. Second, the prefetch scheme can use the local cache as a filter to eliminate most of the unnecessary cache transfers, while the deliver scheme does not have such an effective filtering mechanism. Examining the behavior of these schemes leads us to new insights into communication behavior in parallel applications.

1. The producer-based approach is called *poststore* in KSR1.

This chapter is organized as follows. In Section 1.1, we discuss other research work that are related this thesis. In Section 1.2, we define the research goal. In Section 1.3, we summarize the main contributions. In Section 1.4, we discuss the organization of the thesis.

1.1 Related Work

We can view the deliver and prefetch schemes as a technique for optimizing cache-coherency protocols or as a technique for combining message-passing and shared-memory paradigms. This section discusses related work from these two perspectives.

1.1.1 Optimizing Cache Protocols

There are two traditional protocols for cache coherency: the invalidate protocol (e.g., Illinois Protocol [48] and Berkeley Ownership Protocol [37]) and the update protocol (e.g., Firefly Protocol [56] and Dragon Protocol [4]). In the invalidate protocol, once a processor obtains the ownership of a cache line, the processor can write to the entire cache line without generating network traffic. Communication between processors occurs through cache misses; when another processor reads any word in the line, a cache miss occurs, and the reading processor stalls until the line is received. In the update protocol, on the other hand, the writing processor updates all the copies on every write to any words in the cache line. Thus, when another processor reads any word in the line, no cache miss occurs. Moreover, no false sharing misses [19] occur in the update protocol. A major drawback of the update protocol is that it often generates much more write traffic than the invalidate protocol. This write traffic can increase the number of processor stalls for write operations because of write-buffer overflows, as well as increasing the latency for read-miss operations because of network congestion. Large-scale machines typically use an invalidate protocol because of the traffic overhead due to the update protocol.

Other researchers have proposed to optimize the update protocol for reducing the write traffic and to combine the two protocols for obtaining the benefit of the two. The deliver scheme can be viewed as an optimized update protocol or as a combination of the two protocols. In this subsection, we first discuss optimization techniques for update protocols. Second, we discuss hybrid protocols that use a back-off technique. Third, we discuss other

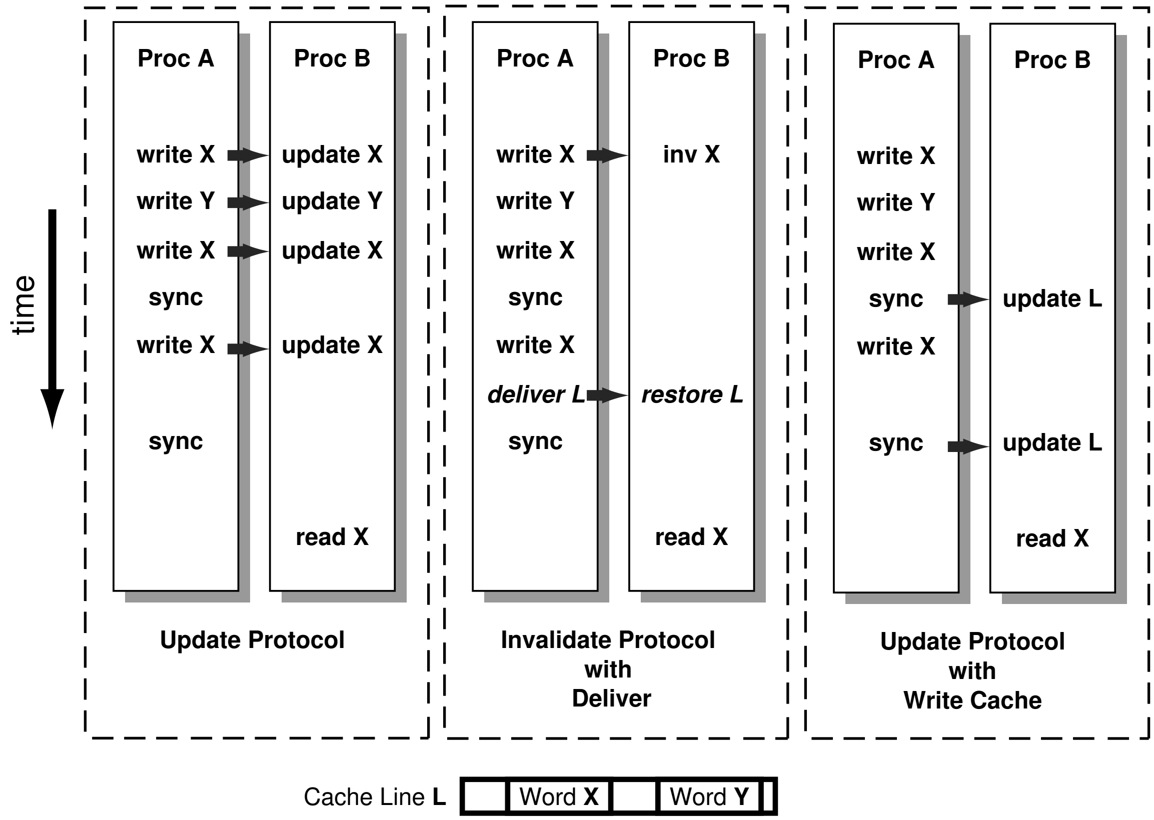

Figure 1-1: Optimizations for an Update Protocol.

techniques for optimizing cache protocols. Fourth, we discuss software-controlled primitives that allow us to exploit software knowledge for cache-coherency protocols.

1.1.1.1 Coalesced Update Protocols

The deliver operation is similar to the update operation in a sense that both operations push a produced value into a consumer's cache based on past sharing patterns. The deliver operation, however, sends a whole cache line to processors that were sharing the line, while the update operation sends a word to processors that are sharing the line. Thus, the deliver operation can be more effective than the update operation because the deliver operation aggregates several updates for the same cache line in a single message. Figure 1-1 illustrates interactions between caches for three cache protocols using the same sharing pattern. Let's focus on the left-most (an update protocol) and the center (an invalidate protocol with deliver operations). A producer (Proc A) and a consumer (Proc B) share the same cache line L, which contains Word X and Word Y. In the update protocol, every write updates the shared copy in the consumer's cache even if the consumer does not use the updated value. In the deliver scheme, on other hand, every write does not update the shared copy; a deliver operation is inserted after the last write before the consumer uses the produced value. The deliver operation aggregates produced values for the same cache line and sends them to the consumer in a single message.[2] Thus, the deliver scheme has two advantages over the update protocols. First, the deliver scheme sends several updates in a single message rather than individually. Thus, the deliver scheme can use the bandwidth of the interconnection network more effectively than the update protocol, since each send operation has an associated fixed overhead (e.g., message header). Second, when the producer writes the same word multiple times before the consumer accesses it, the deliver scheme sends a single updated value rather than multiple copies. Thus, the deliver scheme generally causes less data traffic than the update protocol.

Several schemes have been proposed to aggregate update operations for update protocols. Glasco et al. [24] have proposed a coalescing write buffer that is placed between the processor and its first-level cache and combines write requests for the same cache line into a single write request. The combining is limited to a sequence of write requests for the same cache line that are not interleaved by writes for another cache line. Thus, this scheme exploits only limited locality.

Dahlgren and Stenström [16] have studied write caches for several cache protocols. The write cache is placed after the second-level cache in each processor node and stores write requests that are not satisfied in the second-level cache (e.g., write and write-upgrade misses). The write cache evicts its cache line when the write cache needs to make a room to store another cache line or when the processor issues a release request. The right most chart in Figure 1-1 illustrates the effect of the write cache for an update protocol. Since write requests do not usually propagate to other processors until a release operation, the program has to be synchronized for the correctness of the computation. Dubois et al. [20] have studied a similar mechanism, an Invalidation Send Buffer, which delays sending write requests for an invalidate protocol so that the number of false sharing misses is reduced. The write cache exploits more locality than the coalescing write buffer that

2. Section 2.1 defines more details about the deliver operation.

Glasco proposed [24] because the write cache can combine all write requests between release operations provided the write cache size is sufficient.

The deliver scheme differs from an update protocol with the write cache in three ways. First, the deliver scheme is software-controlled so that we can use deliver operations selectively for those data structures or computations for which the deliver operation works efficiently. For example, as shown in Figure 1-1, the deliver scheme can insert a deliver operation only when consumers use produced values with high probability, while the write cache sends produced values at every release operation. Second, application programs do not have to be synchronized for the deliver scheme, while they must be for the write cache. This is because, in the deliver scheme, processors obtain the ownership before writing a cache line so that write operations for the same address are serialized at the home directory. Third, the deliver scheme may generate more false sharing misses than an update protocol with a write cache. This is because a write cache allows multiple processors to update the same cache line at the same time without obtaining ownership. However, as we will discuss in Section 5.2, false sharing does not significantly impact the miss rate of the deliver scheme for line sizes up to 128 bytes for most of our benchmark applications.

Hagersten [31] has proposed a update cache (Ucache), which collects update requests at the receiver's side. The Ucache is placed between the network interface and the local bus to prevent update requests from congesting the local bus in the receiver of update requests. The Ucache, however, does not reduce the network congestion due to update requests.

1.1.1.2 Hybrid Protocols with a Back-off Technique

Another class of techniques for optimizing cache protocols is a hybrid protocol that combines an invalidate and an update protocol. Competitive snooping [22, 36] is a hybrid cache protocol for bus-based multiprocessors that alternates between the two protocols by using a competitive algorithm. Competitive snooping starts with using an update protocol. The cache has a counter for each cache line that tracks the number of updates that the cache has received. The cache invalidates a cache line when the cache receives a certain number of updates for the cache line without any intervening accesses by the local processor. When all shared copies for the cache line are invalidated, the producer processor obtains an exclusive copy of the cache line so that the processor can write the cache line without producing bus traffic. Eggers and Katz [22] have shown that the competitive snooping outperforms the update protocol for applications with so-called sequential sharing (i.e., applications that generate relatively long write runs).[3] This is because, in such cases, the update protocol suffers from a large number of update transactions, which competitive snooping can reduce. Grahn at al. [28] have studied the implementation and the performance of a similar protocol — competitive update — for a directory-base cache-coherent NUMA machine.

Archibald [3] has also proposed a similar adaptive protocol. In his protocol, when a single processor writes a cache line three times without any intervening references by any other processor, the adaptive protocol invalidates all cached copies other than the producer's copy. Dahlgren [15] has shown that Archibald's protocol produces less traffic and fewer misses than the competitive snooping for a bus-based multiprocessor. Archibald's adaptive protocol updates all shared copies of the cache line if any of the sharing processors are actively using the cache line. Competitive snooping, on the other hand, updates a shared copy of the cache line only if the sharing processor that has a copy is actively using the cache line. Thus, Archibald's protocol generally keeps shared copies valid for a longer period than the competitive snooping. Since a single bus transaction can update any number of shared copies in a bus-based multiprocessor, Archibald's protocol generates less traffic and fewer cache misses than competitive snooping for the same competitive threshold.

A write cache can also exploit locality to reduce the traffic in hybrid protocols. Dahlgren has evaluated Archibald's protocol for a snoop-cache multiprocessor in [15] and a competitive-update protocol for a directory-cache multiprocessor in [16] with accompanying write caches.

Hagersten et al. [32] proposed a hybrid protocol for a COMA (Cache-Only Memory Architecture) machine. The hybrid protocol uses an invalidate protocol by default and switches to an update protocol when a processor causes a coherency miss (i.e., the proces-

3. Eggers and Katz [22] defined the write run as a sequence of write operations that the same processor generates without any intervening accesses from other processors.

sor's cache has an invalidated copy of the cache line). The hybrid protocol uses a back-off technique similar to the competitive snooping; each cache has a update counter to keep track of the number of received updates for each cache line and invalidates a cache line when the cache receives a certain number of updates for the cache line without any intervening accesses by the local processor. Competitive snooping resets the update counter only for local processor's accesses. The hybrid protocol, on the other hand, resets the update counter not only for local processor's accesses but also for data migrations (i.e., changes of the producer processor). Therefore, the hybrid protocol tends to eliminate cache misses without eliminating unused update requests for migratory sharing while the protocol tends to eliminate unused update operations without eliminating cache misses for sequential sharing (sharing patterns with relatively long write runs).

Nilsson [47] has studied a hybrid protocol for a NUMA machine. The hybrid protocol combines a competitive-update protocol and a migratory optimization that was studied for an invalidate protocol by Cox and Fowler [13] and Stenström et al. [55]. The migratory optimization is a technique to reduce the write latency and the network traffic by translating a read request to a read-exclusive request when a read cache-miss occurs for migratory objects. This hybrid protocol is similar to the COMA protocol since both protocols include a mechanism that handles migratory sharing. The major difference between the two is that the Nilsson's protocol focuses on reducing the traffic for migratory objects while the COMA protocol focuses on reducing the miss rate for migratory objects.

The hybrid protocols that we have discussed use a back-off technique that basically switches from an update mode to an invalidate mode when the update mode causes more traffic overhead than a certain threshold. The back-off technique provides a trade-off between the number of cache misses and the amount of the traffic. Hybrid protocols reduce the number of misses while generating extra traffic when in the update mode, or reduce the amount of the traffic while generating cache misses when in the invalidate mode. As we will discuss in Section 4.1, we can add a back-off technique to the deliver scheme (called *competitive deliver*) for a similar trade-off. In the competitive deliver scheme, each cache counts the number of received deliver messages for every cache line. When a cache receives a certain number of deliver messages for a cache without any intervening accesses to that line by the local processor, the cache cancels the subscription of the deliver message for that cache line.

1.1.1.3 Other Protocol Optimizations

Read snarfing (also called read broadcast) has been evaluated by Eggers and Katz [22], Anderson and Baer [2], and Dahlgren [15] for bus-based multiprocessors. This technique assumes an invalidate protocol with a snoop mechanism. If a cache finds a cache line transfer for which the cache has an invalidated copy, the cache reads the cache line from the bus and changes the invalidated copy to a valid one. Thus, when the local processor of the cache reads the cache line, a read hit occurs. This technique reduces the number of misses without generating extra traffic. The transfer unit of produced values is a cache line, instead of a word. Thus, this technique exploits more locality than update protocols without a write cache. If the cache directory keeps track of processors that have an invalidated copy for each cache line, a directory-based cache system can use this technique. This read snarfing, however, is effective only if a relatively large number of processors read a produced value. As we will discuss in Section 4.1, the number of processors that read a produced value is relatively small for most of our benchmark applications. Thus, the read snarfing should be less effective than the deliver scheme for most of our applications.

Veenstra and Fowler [57] studied three hybrid protocols of another class. The first one statically assigns an invalidation protocol or an update protocol to each page. The second one statically assigns either protocol to each cache line. The third one dynamically chooses either protocol for each write. While competitive hybrid protocols use a simple on-line algorithm to select the protocol, Veenstra and Fowler [57] used off-line algorithms and found that the per-line static assignment of the protocol can obtain most of the benefit due to the dynamic assignment of the protocol. These off-line algorithms rely on a cost model for selecting the protocol mode. The cost model defines the cost of each cache operation (i.e., cache hit, invalidate, update, and cache block load), which is the summation of a weighted latency cost and a weighted traffic cost of the operation. The cost of a hybrid protocol is defined as the total cost of all operations that an application program generates. This cost model, however, does not necessarily represent the performance of the protocol since the performance depends on the trade-off between the number of misses and the amount of the traffic. In this thesis, therefore, for comparing the deliver and the prefetch schemes, we examine a trade-off curve between the number of misses and the amount of the traffic for each scheme so that we can obtain intuition about the performance effect of the two schemes.

Raynaud et al. [49] have proposed an adaptive update protocol, Distance-Adaptive Update Protocol. This work defines an *update distance* as the number of updates that a processor receives between accesses in an update protocol with a write cache. The distance-adaptive update protocol relies on the fact that there is a correlation between two consecutive instances of the update distance and that the correlation is consistent for all processors and for all data structures of the application. If the update distance changes non-deterministically or if the pattern in the update distance is dramatically different among data structures, this adaptive protocol would perform poorly. The deliver scheme should be able to predict access patterns as accurately as this adaptive protocol does for applications with a deterministic sharing pattern since the deliver scheme exploits software knowledge about the sharing pattern. A quantitative comparison between this adaptive protocol and the deliver scheme, however, requires further experiments and is beyond the scope of this thesis.

1.1.1.4 Software-Controlled Primitives

Hill et al. [35] proposed a programming model, *Check-In / Check-Out (CICO)*, in conjunction with simple hardware, Dir_1SW. In the base CICO model, a check-out annotation is inserted at the expected first use of shared data and a check-in annotation is inserted at the expected last use of shared data, so that simple Dir_1SW hardware can efficiently support the program. Similar to the deliver annotation, the CICO annotation tries to exploit the knowledge of the expected data-access pattern. Check-in annotations are quite similar to deliver annotations in that they are both placed after the last write to shared data. The check-in operation, however, sends the produced data only to the home node, not to consumer's caches. This is because the major focus of the CICO model is simplifying the hardware while the major focus of the deliver scheme is improving performance.

The KSR1 system supports latency hiding mechanisms including prefetch and poststore [23]. The prefetch and poststore are software-controlled and similar to our prefetch and deliver respectively. The KSR1 is a COMA machine, while our study uses a NUMA machine. In the COMA architecture, each node has a very large (32 Mbyte [23]) memory cache which corresponds to the local memory in the NUMA machine. The prefetch and poststore of the KSR1 move the datum only to the memory cache, not to the processor cache. Therefore, the prefetch and poststore cannot hide memory latencies completely, while our prefetch and deliver can. The poststore, however, should not suffer from the cache interference we observed, since the memory cache is very large. The poststore, moreover, can take advantage of the hierarchical ring network in the KSR1 that supports broadcasting, while our deliver scheme only assumes a point-to-point communication mechanism. The KSR1 also supports read snarfing (called *automatic update*), which is a technique that updates invalidated copies in the memory cache by snooping the ring network.

Previous studies have discussed the latency hiding mechanisms of KSR1. Rosti et al. [51] studied the poststore, using synthetic benchmarks and analytical models. Our study uses a set of real parallel applications that provides a more realistic evaluation environment than synthetic benchmarks. Windheiser et al. [58] evaluated the effectiveness of prefetch and poststore for a sparse solver application on KSR1. Our study discusses more detailed statistics than those two studies, such as the cache-size sensitivity, the efficiency of prefetch/deliver schemes, and the intrinsic/contention latency of read operations.

1.1.2 Combining Shared-Memory and Message-Passing Paradigms

Traditionally two distinct paradigms exist for the architecture of multiprocessors: shared memory and message passing. The program does not have to manage communication explicitly in the shared memory, while the program has to do so in the message passing. Thus, the shared memory is considered to be more programmable than the message passing especially for programs with irregular communication patterns. Explicit communication mechanisms in the message passing, however, allow us to optimize the communication so that the communication and the computation are overlapped.

Software-controlled deliver and prefetch operations provide us an ability of explicit communication on the shared memory. Thus we can consider the deliver and the prefetch schemes as a way to combine the shared memory and the message passing so that we can obtain the benefit of the two paradigms.

Other researchers have proposed to integrate block data transfer in shared-memory multiprocessors [1, 14, 33, 61]. Block data transfer can be implemented as a memory copy primitive (memory-to-memory transfer) on cache-coherent shared-memory systems. Byrd [8] has discussed a block transfer scheme (called StreamLine) for cache-to-cache transfers. Block data transfer is similar to deliver since both schemes provide a sender-initiated explicit data-transfer mechanism in a shared-memory system. Both schemes can transfer data in a pipelined fashion and overlap the communication and the computation. The two schemes, however, are different in four points. First, deliver operations do not affect the correctness of the program, while block data transfers usually affect the correctness of the program. This is because deliver operations do not change the value of any memory location while block data transfers change the value in the destination message buffer. Thus, block data transfers need to be inserted in the program so that not only the performance is improved but also the program works correctly. Second, the block data transfer needs message buffers that are explicitly managed by the programmer or the language system, while the deliver scheme does not. Third, block data transfer can be implemented as a transfer operation to the memory or the cache of the destination processor. The deliver scheme, on the other hand, can be implemented as a transfer operation only to the cache of the destination processor. Fourth, block data transfer requires the programmer or the language system to specify that the destination of the message explicitly, while our deliver scheme requires a hardware mechanism to keep track of potential consumers for each cache line.

1.1.3 Summary

We have discussed related work for the deliver and the prefetch schemes from two different perspectives: optimizing cache protocols and merging the shared-memory and the message-passing paradigms.

The optimization techniques of cache protocols can be divided into four groups. First, we discussed a set of techniques that delay sending update messages to exploit locality in updating patterns. Second, we discussed a set of techniques that combine the invalidate and the update protocols by using a back-off technique. Third, we discussed other protocol optimization techniques that include read snarfing and distance-adaptive update protocol. Fourth, we discussed software primitives that exploit software knowledge about sharing patterns for simplifying the hardware or for improving the performance. The deliver and the prefetch schemes can be categorized in the fourth group: software primitives to improve the performance.

We also qualitatively compared the deliver scheme with block data transfer, which combines the message-passing and the shared-memory paradigms. While both schemes provide explicit communication primitives on top of the shared memory paradigm, block data transfer is closer to the message passing paradigm than the deliver scheme. In block data transfer, the communication primitive may affect the correctness of the program and the programmer or the language system needs to mange message buffers explicitly. In the deliver scheme, on other hand, the communication primitive does not affect the correctness of the program and the programmer or the language system does not need to manage message buffers.

1.2 Research Goals

The goal of this thesis is to compare the characteristics of deliver and prefetch operations by analyzing the communication behavior of parallel applications. This goal is broken-down into two sub goals.

- Analyzing sharing patterns to identify ones that produce unnecessary deliver or prefetch messages.
- Understanding the interaction between application characteristics and architectural parameters to identify advantages and disadvantages of the deliver and the prefetch schemes.

For the first goal, we focus on the communication behavior. Through the analysis of sharing patterns, we examine several techniques that eliminate unnecessary messages. For the second goal, we explore a large design space of architectural parameters and compare the characteristics of the deliver and the prefetch schemes. For both sub goals, we focus on underlying application characteristics.

We use explicitly-parallelized scientific applications that represent various algorithms for numerical computations and contain a variety of simple and complex memory-access patterns. Further research is necessary for other types of applications (e.g., compiler-parallelized applications and commercial applications), which may have different characteristics from our benchmark applications. Moreover, since the focus is on the interplay between the prefetching strategy and the application characteristics, we do not discuss detailed design issues of hardware and language systems, which also need further research.

1.3 Contributions

The important contributions of this thesis are:

- Analyses of sharing patterns and program structures of parallel applications that explain the behavior of deliver and prefetch operations.
- A proposal and comparison of several techniques that improve the efficiency of deliver operations by using various types of knowledge about sharing patterns.
- An analysis of working-set and communication characteristics of parallel applications that explains the cache behavior for the deliver and the prefetch schemes.
- A quantitative comparison of the deliver and the prefetch schemes in a large design space of architectural parameters.

1.4 Organization of Dissertation

Chapter 2 presents hardware and software mechanisms that we assume for the deliver and the prefetch schemes. The two schemes are software-controlled so that we use software knowledge to insert cache transfer operations in applications. The deliver scheme uses an extra hardware mechanism to keep track of past sharing behavior for each cache line. We summarize research issues to improve the accuracy of past behavior as a predictor for future behavior. Furthermore, we qualitatively compare characteristics of the two schemes.

Chapter 3 describes the simulation environment for this thesis. Since a large number of parameters affect the behavior of the deliver and the prefetch schemes, we need to design our experiments carefully to complete a wide range of simulations in a feasible time. We discuss the simulation methodology and the benchmark applications that are used in this thesis.

Chapter 4 analyzes simulation results to examine sharing characteristics of parallel applications and their effects on the behavior of prefetch and deliver operations. For irregular memory access patterns, both schemes may transfer a large number of unnecessary cache lines. For the deliver scheme, we identify sharing patterns that generate unnecessary deliver messages, and we quantitatively examine three techniques that trade off the number of unnecessary deliver messages for the number of eliminated cache misses. For the prefetch scheme, we show that the prefetch scheme has a significant advantage over the deliver scheme since a simple mechanism can eliminate most of unnecessary cache transfers.

Chapter 5 discusses the effect of various architectural parameters for the performance of prefetch and deliver operations. First, we focus on the miss-rate characteristics. One of the most important results in this chapter is that the cache size needs to be about as large as the largest working set of the application before we can obtain most of the benefit due to deliver operations. We examine the working-set and the communication characteristics to understand this behavior of deliver operations. Next, we discuss the traffic characteristics: we compare applications' demand traffic with feasible bandwidth for major system components. Finally, we evaluate the execution-time improvement due to deliver and prefetch operations by simulating the contention delay in the memory system.

Finally, Chapter 6 summarizes our discussions about the behavior of deliver and prefetch operations and the characteristics of parallel applications.

Chapter 2

Producer-oriented and Consumer-oriented Approaches

In this chapter, we discuss hardware and software mechanisms for the deliver and the prefetch schemes and qualitatively compare the two schemes. Both schemes hide the memory latency by using a software-controlled non-binding mechanism. The schemes are software-controlled; we use software knowledge to insert a cache-line transfer operation — deliver or prefetch — for maximum hiding of memory latencies. The schemes are non-binding; the transfer operation does not bind the transferred value to a shared variable, so that the transfer operation is a performance hint and does not affect the correctness of the program. Other researchers have shown that non-binding mechanisms are necessary to achieve good performance [44]. While both schemes need to exploit hardware and software knowledge for maximum performance gain, the two schemes exploit different kinds of knowledge as we will discuss in the chapter.

This chapter is organized as follows. Section 2.1 describes the concept of the deliver scheme, which includes the strategy for inserting deliver operations, the hardware mechanism, and variations on the deliver scheme. Section 2.2 describes the concept of the prefetch scheme and qualitatively compares the prefetch and the deliver schemes.

2.1 Producer-oriented Deliver

Figure 2-1 illustrates an example of deliver and prefetch operations. The producer transfers the produced value to consumer's caches in the deliver scheme, so that the consumer can read the value without causing a cache miss. In the prefetch scheme, on the other hand, the consumer fetches the produced value. The producer-initiated transfer has several advantages over the consumer-initiated transfer: (1) The producer can transfer the pro-

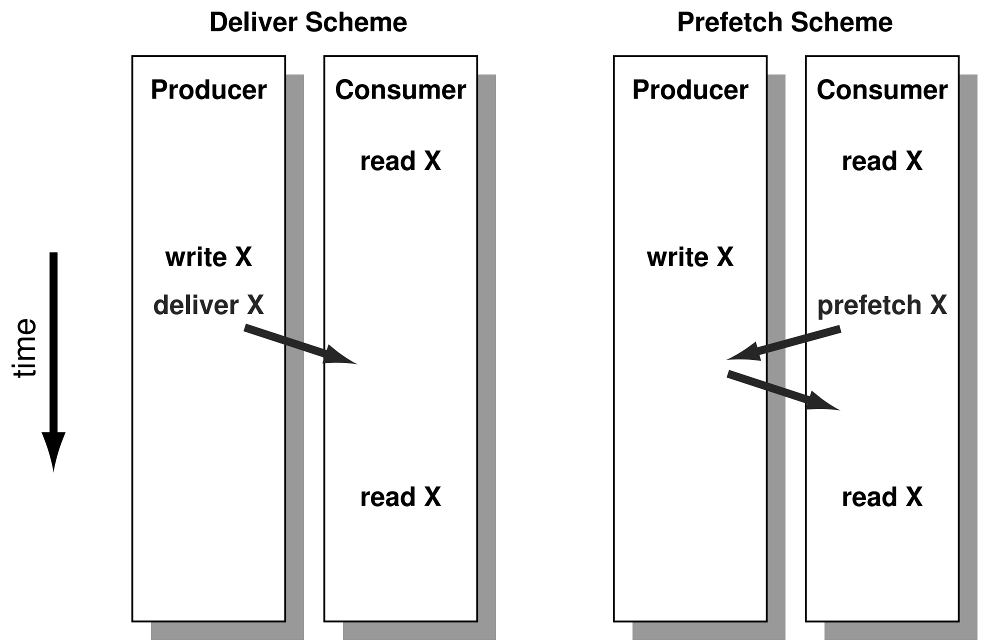

Figure 2-1: An Example of Deliver and Prefetch Operations

duced value when it is produced, which is the earliest possible time. The consumer, on the other hand, may transfer the value too early before it is produced or too late to use it in time; (2) The latency of the producer-initiated transfer is a one-way latency from the producer to the consumer, while the latency of the consumer-initiated transfer is a round-trip latency between the consumer and the producer; (3) The producer can easily find the address of the value to be transferred, while the consumer may not be able to find the address early enough to obtain the value in time. For example, if the application intensively uses a linked list, the consumer cannot prefetch an element in the list until the consumer fetches another element that contains the address. The producer, on the other hand, knows the address of the value when the value is produced.

While the deliver operation is similar to a send operation in the message passing architecture, the deliver operation is a performance hint. That is, the program works correctly, if no deliver operations are inserted, or even if deliver operations are inserted arbitrarily. In the absence of deliver operations, the cache protocol performs exactly as an invalidate protocol. Conversely, if every write is followed by a deliver operation, the cache protocol performs in a way similar to an update protocol.

A key issue for the producer-initiated transfer is how to predict the consumer that will use the produced value. Our deliver scheme uses hardware knowledge for finding potential consumers. We use a hardware mechanism that keeps track of past sharing patterns. The deliver operation sends a cache line to processors that were sharing the line in a similar manner that the update operation does in the update protocol. The deliver scheme, however, differs from the update protocol in two ways. First, the update protocol does not send update information to processors once their cache replaces the cache line that are being updated. The deliver scheme, on the other hand, can send a cache line to processors even if their cache does not have a copy of the cache line. Second, more importantly, the deliver scheme is much more efficient than the update protocol. The deliver operation aggregates updates for the same cache lines into a single message. The deliver scheme, moreover, uses software knowledge to reduce the amount of the traffic. The programmer or language systems use software knowledge about the program structure to insert deliver operations only where communication is likely to occur.

2.1.1 Strategy for Inserting Deliver Operations

A deliver operation for a shared variable should ideally be inserted where processor-to-processor communication occurs through the variable. For compiler-parallelized code, the compiler should be able to insert deliver operations in such places since the compiler knows where communication occurs in the code. For explicitly-parallelized code, on the other hand, it is not obvious to find where communication occurs in the code unless the computation is regular.

Our strategy is to use synchronization operations as a hint to find the timing of communication. We insert a deliver operation after the last write of a shared variable before a synchronization operation, so that the written value is sent to prospective readers before they start reading the variable. For synchronized programs, it should be relatively straightforward to find such places for deliver operations. For non-synchronized programs, the ideal location is where a processor finishes a series of writes for shared variables and other processors are likely to read the variables. For the non-synchronized programs that we studied, it was straightforward to find such places.[1]

1. MP3D and LocusRoute. See Section 3.2.

```
for loop of i {                       for loop of i {
    sharedArray[i] = ...                  sharedArray[i] = ...
}                                         sharedArray[i+1] = ...
                 loop                     deliver(sharedArray + i)
                 unrolling            }

*ptr1 =  ...                          *ptr1 =  ...
*ptr2 =  ...                          *ptr2 =  ...
                                      deliver(ptr1)
                 conditional          if (ptr1 and ptr2 are not in the same block)
                 deliver                  deliver(ptr2)
```

Figure 2-2: Deliver Annotation Techniques.

Because the unit of communication is a cache line, we actually insert the deliver operation after the last write for a cache line containing shared variables instead of the last write for each shared variable. For some applications, this requires transformations to the program if the deliver is to obtain the best performance. Figure 2-2 shows two typical cases. In the first case, the loop is unrolled so that the same line is delivered only once. In the second case, the test statement checks if two writes are to the same line. These modifications increase the probability that a single deliver operation updates all the modified words in the same line, thus reducing the number of deliver messages. These sorts of optimizations are direct analogies to transformations that a compiler or programmer might perform to improve the performance of invalidate protocols by increasing the spatial clustering of shared data in multi-word lines.

In spite of these annotation techniques, the deliver scheme generates some unnecessary deliver messages for applications with irregular computations because communication does not occur after every synchronization operation and because not all of the deliver destinations use the delivered cache line. We use more software knowledge to improve the efficiency of the deliver scheme. Since the performance gain due to the deliver scheme depends on the trade-off between the cache-miss reduction and the traffic overhead, we should insert the deliver operation where the ratio between the number of eliminated cache-misses and the number of deliver-messages is relatively high. We call this technique *Selective Deliver*. Our simulator obtains the ratio for all locations where a deliver operation is potentially inserted. We will examine the simulation results in Section 4.1 to investigate the effect of program structures for the efficiency of deliver operations.

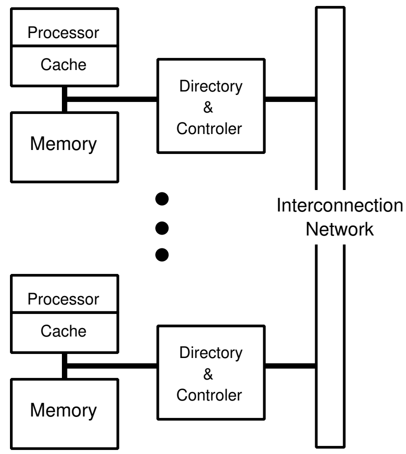

Figure 2-3: NUMA Model.

2.1.2 Hardware Mechanisms for the Deliver Operation

We assume a directory-based cache-coherent NUMA machine (e.g. Alewife [10] and DASH [40]), as illustrated in Figure 2-3. The main memory is distributed among processor nodes. Each node has a directory that keeps track of which nodes are caching each memory block of the node. We assume that each processor cache is kept coherent using an invalidate protocol.[2] When a processor writes a shared variable, the processor sends a write request to the home node, which corresponds to the memory address of the variable. Then, the home node looks up the processors sharing the address in the directory and sends an invalidation request to the sharing processors (except the node of the original writer). Those nodes return an acknowledgment message to the home node, which collects all acknowledgment messages and notifies the original writer of the completion of the invalidation.

We assume each entry of the cache directory has two bit-vectors as shown in Figure 2-4. One bit-vector, the sharing vector, holds the identity of processors that may be currently sharing the copy of the cache line. The second one, the deliver vector, holds the identity of processors that the deliver operation sends the cache line to. The directory controller manages the sharing vector just as in a conventional cache directory with replacement hints. Each bit of the sharing vector is set when a copy of the line is sent to the corresponding processor, and cleared when the processor's copy is invalidated or replaced. The deliver

2. We will discuss more details about the cache protocol in Subsection 5.3.1.

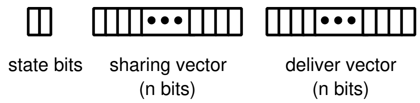

Figure 2-4: Cache Directory Entry for Each Cache Line (*n* processors).

vector, on the other hand, is managed in a different way from the sharing vector. Each bit of the deliver vector is set when a copy of the line is sent to the corresponding processor, but not cleared when the processor's copy is invalidated. If the deliver vector is never cleared, the deliver vector holds the identity of all processors that have touched the line, so that deliver operations probably send a cache line to processors that will not access the line anymore. In the next subsection, we will discuss techniques (e.g., the use of replacement hints) to manage the deliver vector so that it holds only the identity of processors that will probably use the line.

Figure 2-5 illustrates an example of how this cache directory operates for the deliver operation. In this example, three processors (Proc A, Proc B, and Proc C) share the same cache line. A producer (Proc A) writes the cache line, which is read by two consumers (Proc B and Proc C). First, the three processors read the cache line. As the cache directory receives a read request from a processor, the directory turns on the corresponding bit in the sharing

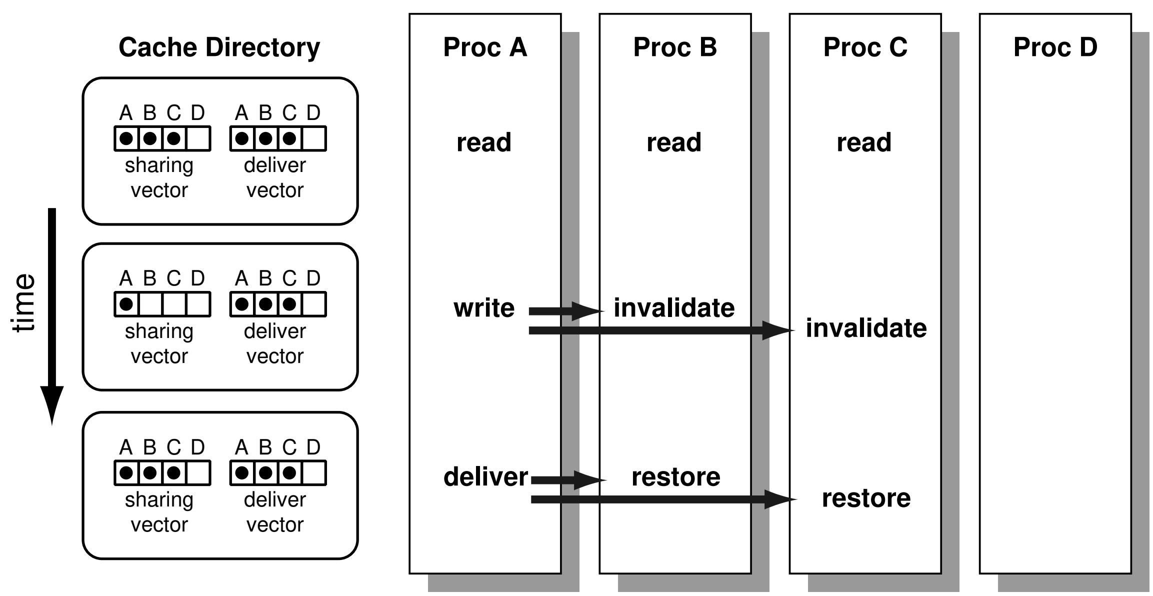

Figure 2-5: An Example of Cache Directory Operations.

and the deliver vectors. Second, Proc A writes the cache line. When the cache directory receives the write request, the directory sends an invalidate request to Proc B and Proc C and turns off the bit for the two processors in the sharing vector. The cache directory does not change the deliver vector, which keeps track of past sharing processors. On receiving the invalidate request, the two processors send back an acknowledgment to the directory. Third, Proc A delivers the cache line to the cache directory. The directory updates the main memory for the cache line, looks up the deliver vector, and forwards the cache line to Proc B and Proc C. On receiving the cache line, the two processors send back an acknowledgment to the directory. Thus, the consumers obtain the new copy of the cache line without causing a cache miss.

Conventional invalidate and update protocols need only one vector, the sharing vector, per cache line. Our protocol assumes two vectors per cache line; the sharing vector is used to identify the processors to which an invalidate request is sent, while the deliver vector is used to identify the processors to which a deliver message is sent. This additional deliver vector allows us to reduce the number of unnecessary deliver messages. We discuss the benefit of the deliver vector by examining a sharing pattern example shown in Figure 2-6. The four processors share the same cache line; Proc A is a producer and the rest of the pro-

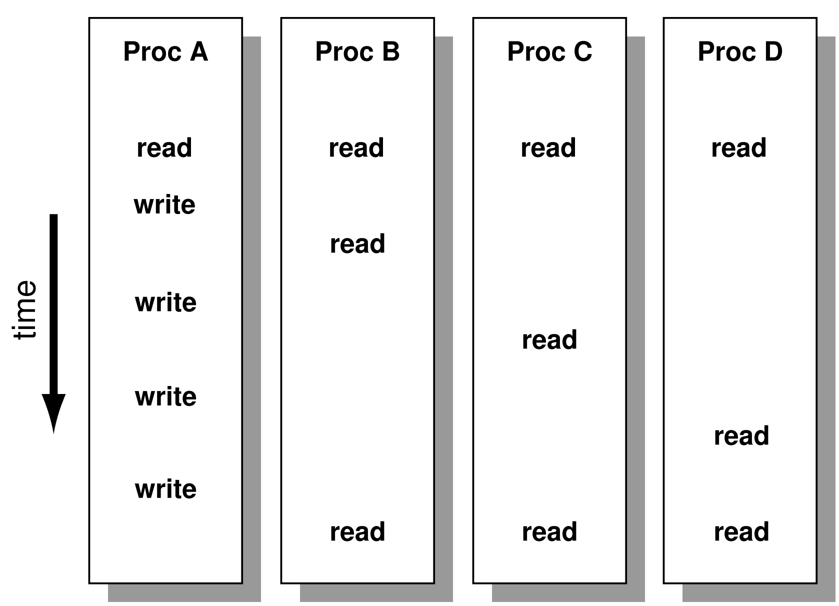

Figure 2-6: An Example of Sharing Patterns. Proc A is a producer and the rest of the processors are a consumer.

cessors are a consumer. First, the four processors read the cache line, so that the sharing vector holds the identify of those processors. Then, Proc A writes the cache line four times. Only one consumer does read the produced value for the first three writes, while all consumers read the produced value for the fourth write. Since the deliver vector remembers past sharing information regardless the current sharing status, we can use deliver operations only when the past sharing pattern is a good predictor of the future access pattern. For the example shown in Figure 2-6, the producer-consumer relation for the first three writes is not ideal for the deliver scheme; only one of the consumers use the produced value. If we insert deliver operations after those writes, we generate unnecessary messages. The producer-consumer relation for the fourth write, on the other hand, is ideal for the deliver scheme; all the consumers use the produced value. Since the deliver vector remembers the past sharing pattern, a deliver operation sends the cache line to all the consumers if we insert a deliver operation immediately after the fourth write. If we do not have the deliver vector, however, the cache directory remembers only the last consumer (Proc D), so that the deliver operation sends the cache line only to Proc D.

Furthermore, the deliver vector allows consumers to reduce the number of unnecessary messages. That is, a consumer can enable or disable the subscription of deliver messages, whether the cache has a copy of the cache line or not. If a consumer knows that it will not access a cache line for a long time, the consumer can cancel the subscription of deliver messages for the cache line by sending a special message to the directory, which turns off the corresponding bit in the deliver vector. We will discuss more details about techniques that manage the subscription of deliver messages in Subsection 2.1.3.

We can reduce the directory space by using several techniques including limited pointer schemes [1]. If the cache directory is controlled by software on a special processor, further reduction is possible. Optimizing the cache-directory space, however, is beyond our scope, and we do not discuss the issue in this thesis.

2.1.3 Variations on the Deliver Operation

In this subsection, we qualitatively discuss variations on the deliver operation for maximizing the performance benefit, which will be quantitative examined by using simulation results in Chapter 4 and Chapter 5. As discussed previously, our deliver mechanism relies

Trade-off Technique	Initiator	Mechanism	Information for Trade-off
Selective Deliver	Producer	Software	The number of sent messages per eliminated miss.
Competitive Back-off	Consumer	Hardware	The number of received messages per eliminated miss.
Subscription Control	Consumer	Software	The number of received messages per eliminated miss.

Table 2-1: Three Trade-offs between the Number of Deliver Messages and the Number of Eliminated Misses.

on the past sharing behavior that is stored in the deliver vector. In reality, the past sharing behavior does not always accurately predict the future behavior. Thus, a deliver operation may send an unnecessary message that the destination processor will not use, hence deliver operations may waste some network bandwidth and some space in the cache memory of the deliver destination. The difficulty of predicting future sharing patterns raises two important issues. First, how can we trade off the number of unnecessary deliver messages versus the number of eliminated cache misses? Second, how can we design a replacement policy to minimize the number of evictions of necessary cache lines due to unnecessary deliver messages?

In Section 4.1, we will evaluate three techniques, shown in Table 2-1, to trade off the number of unnecessary deliver messages versus the number of eliminated cache misses. The three techniques are divided into two types: producer-initiated and consumer-initiated. *Selective Deliver* has been discussed in the preceding subsection. This technique is producer-initiated because the producer controls the timing of deliver operations. This technique is also software-controlled because it relies on a software system that analyzes the sharing pattern and controls the insertion of deliver operations. In this study, we select the insertion place for the deliver operation by using a profiling system that counts the number of sent messages per eliminated miss for each candidate of the insertion place.

The other two techniques are consumer-initiated; a consumer manages the deliver vector in the cache directory to trade off the number of unnecessary deliver messages versus the number of eliminated cache misses. *Subscription Control* is software-controlled; it relies on a software system that controls the insertion of a special operation that removes the identify of the consumer from the deliver vector so that the consumer will no longer receive deliver messages for the cache line. We select the insertion place for the subscription control operation by using a profiling system that counts the number of received messages per eliminated miss for each candidate of the insertion place. *Competitive Back-off* is hardware-controlled and based on the idea of the competitive snooping that alternates the invalidate and update protocols [22, 36]. In our back-off technique, each line in the cache has a counter that tracks the number of deliver messages received. The counter is cleared when the local processor accesses the associated line. When the counter exceeds a certain threshold, we assume the processor is no longer using the line. When an invalidate request comes to the cache (due to a write by another processor), the cache controller invalidates the local copy of the line. At that time, if the counter of the line is equal to the threshold, the invalidate acknowledgment message piggybacks a command that clears the corresponding bit on the deliver vector at the home directory, so that the cache will no longer receive deliver messages for the line. Note that we do not need extra messages for this back-off technique because of piggybacking.

Another important issue for the deliver scheme is the replacement policy for delivered cache lines. For on-demand memory fetching, the processor will use the fetched line so that it is reasonable to evict another cache line to store the fetched line. For producer-oriented delivers, on the other hand, the processor does not necessarily use the delivered line. If the processor evicts a necessary cache line to store an unnecessary delivered line, the deliver operation increases the number of cache misses. We will examine the effect of replacement policies for various cache sizes in Section 5.1. One of the replacement policies that we examine is called the pessimistic policy; a consumer receives a deliver message only if the delivered line does not conflict with other lines. This can be implemented by using replacement hints to *refresh* the deliver vector. Each cache sends a replacement hint to the home directory when the cache replaces a valid or invalid copy of a cache line, so that the cache will no longer receive a deliver message for the line. Delivered lines never conflict with other lines since the cache receives a deliver message only if the cache has an invalidated copy of that line. Our simulation results show that the replacement policy that minimizes the miss rate depends on the cache size.

2.2 Consumer-oriented Prefetch

Since consumer-based prefetching has been discussed extensively [9, 44, 46], this section simply clarifies our approach briefly. We assume the same NUMA machine as the one shown in Figure 2-3. The cache directory is the same as the generic one with a full-bit vector; the directory entry needs only one vector (sharing vector) for each cache line. We assume that the processor cache is lock-up free so that the cache can have multiple outstanding memory requests.

We use two types of prefetch operations: prefetch and prefetch-exclusive. The prefetch operation brings a copy of the cache line into the processor cache, while the prefetch-exclusive operation brings an exclusive copy of the cache line into the processor cache. The cache protocol for prefetch and prefetch-exclusive operations is the same as the one for read and write operations of the invalidate protocol, respectively. If a processor writes a shared variable after reading it, the prefetch-exclusive operation can not only hide the memory latency for the read but also reduce the number of network transactions by combining the read and the write requests. For the deliver scheme, we do not consider an operation that corresponds to the prefetch-exclusive; the deliver operation transfers a copy of the cache line but not the ownership of the line. This is because a deliver operation generally sends a cache line to multiple processors that are specified by the deliver vector.[3]

Unlike the deliver scheme, the prefetch scheme does not need a hardware mechanism that keeps track of consumers' access patterns. The prefetch scheme relies on software knowledge about consumers' future access patterns. As simulation results will show in Section 4.2, consumers can predict their access patterns accurately enough to avoid prefetching cache lines that will not be used. Consumers, however, do not always know the sharing behavior of producers; consumers do not know whether or not some producer has updated the cache line that is being prefetched. Thus, consumers may prefetch cache lines that are still valid in the cache. We can prevent such prefetch operations from generating network traffic by simply checking the local processor cache before sending the prefetch request to the home directory. That is, the processor cache can filter out unnecessary prefetch opera-

3. It is possible to implement a deliver scheme that transfers an exclusive copy of the cache line to a processor. For a deliver scheme in which the producer explicitly specifies the destination, it is reasonable to have two types of deliver operations; deliver and deliver-exclusive.

tions. As we will discuss in Section 4.2, this hardware mechanism is very important for the prefetch scheme to reduce the number of unnecessary prefetch requests significantly. We moreover use a profiling system to trade off the number of unused prefetch operations versus the number of eliminated cache misses.

Chapter 3

Experimental Environment

In this chapter, we discuss the experimental environment for analyzing the characteristics of application programs and the behavior of prefetch and deliver operations. We use an execution-driven simulator to gather various statistics about the memory access behavior of parallel applications. We use ten parallel applications to investigate the effect of various application algorithms and structures for the prefetch and the deliver schemes. This chapter is organized in two sections; the first section discusses our simulation methodology, and the second section discusses the application programs that we use in this study.

3.1 Simulation Methodology

An advantage of simulation-based experiments is flexibility; we can easily obtain detailed statistics about the memory access behavior when various architectural parameters change. A disadvantage, on the other hand, is slowdown; simulators are much slower than real machines, so that we can examine only a small part of the entire parameter space to finish our experiments within a feasible time. Therefore, it is necessary to design the simulator as well as the experiment to obtain enough insights.

In this section, we discuss our simulation methodology; in Subsection 3.1.1, we discuss the methodology for simulating multiprocessor systems, and in Subsection 3.1.2, we discuss the methodology for designing experiments.

3.1.1 Execution-Driven Simulation

Our execution-driven simulator models a multiprocessor system on a uniprocessor by mapping each process of a parallel application to a thread in the simulation environment. The simulator consists of two tightly-coupled components: a TangoLite reference generator [26] and a memory-system simulator. The TangoLite reference generator takes a parallel application and manages the execution of the threads of the application so that memory accesses are interleaved correctly in a uniprocessor environment. TangoLite passes each shared memory reference to the memory system simulator and suspends the execution of the application thread for the memory stall time that the memory system simulator calculates.

The memory system simulator maintains the state of various components in the memory system (e.g., the cache tag memory and the cache directory) and gathers various statistics on memory activities. We model only shared memory references for two reasons. First, our focus is on the behavior of shared memory references. Second, we can speed up our experiments without simulating the rest of memory references (e.g., instruction and stack-frame references), which are assumed to take one clock and are not counted in our statistics. Moreover, we speed up our experiments by optimizing the execution path of the memory system simulator when the simulated memory reference causes a cache hit.

The simulator uses a similar mechanism to MemSpy [41] to collect detailed memory-access information for each variable type. Our simulator, moreover, keeps track of various statistics for each source-code line that contains a memory operation for shared data. We use those statistics to identify the place to insert deliver or prefetch operations. To identify the place for deliver operations, for example, we use a version of the simulator that automatically inserts a deliver operation after each write operation for shared data and counts the number of sent messages by the deliver operation, the number of reduced read misses by the deliver operation, and the number of remaining read misses. Thus, we can find places at which deliver operations can eliminate read misses without generating a large number of unnecessary deliver messages.

Our simulation model has two limitations; we do not model the I/O system or the operating system. While the effect of the I/O system and the operating system is believed small for numerically intensive applications, the evaluation of the effect is beyond the scope of this thesis.

3.1.2 Designing Experiments

We need to consider three major issues for designing experiments: the architectural parameter space to be explored, the accuracy of the simulation model, and the scaling of the problem size. Let's focus on the first two issues.

Since there is a number of architectural parameters that affect the behavior of memory accesses, it is not practical to explore the entire parameter space by using simulation techniques. Therefore, we need to select a subspace carefully from the entire parameter space for our experiments. Another issue is the degree of the accuracy for our simulations. An accurate model is necessary to evaluate the performance of the deliver and the prefetch schemes but it is not necessary to examine the application-intrinsic characteristics. Moreover, the more accurate the simulation model is, generally, the slower the simulator is [26]. Thus, an accurate model cannot be used for exploring a large parameter space.

Because of these conflicting issues, we use two machine models: a unit-delay model and a more realistic model. In the unit-delay model, all memory accesses are assumed to take one clock cycle even if the access does not hit in the cache. Thus, the simulator generates the same memory access pattern for different cache configurations even if the application is non-deterministic. This makes it easy for us to compare the simulation results for different cache configurations. The unit-delay model, moreover, is so simple that we can minimize the simulation time. Therefore, using this model, we explore a relatively large parameter space to investigate the interplay between application characteristics and architectural parameters in the deliver and the prefetch schemes. A drawback of the unit-delay model is that we cannot accurately evaluate the performance effect of deliver and prefetch operations. For evaluating the performance effect, we use a more complex and realistic model than the unit-delay model.

In Chapter 4, we use a unit-delay model with an infinite cache per processor to examine sharing patterns (e.g., producer-consumer relations) and the effects for deliver and prefetch operations. We also examine the effectiveness of deliver and prefetch operations when we vary the number of deliver and prefetch operations that are inserted in the application.

In Section 5.1 through 5.3, we use a unit-delay model with various cache configurations. We vary the cache size, the associativity, the line size, and the number of processors to examine the effect of those parameters on the benefits of the deliver and the prefetch operations. For this examination, we use a reasonable deliver/prefetch annotation that we base on the preceding analyses in Chapter 4.

In Section 5.4, we use a more realistic model than the unit-delay model to examine the performance effect due to deliver and prefetch operations. We assume a realistic latency for each of major system components (e.g., the main memory, the cache memory, and the network switch) and we model delays due to contention for each of the components. Since this simulation model is slower than the unit-delay model, we use a few interesting points in the parameter space that we pick up through the preceding analyses in Section 5.1 through 5.3.

Since the simulation requires a larger amount of memory for collecting detailed statistics, we assume that the number of processors is 16 for most of the analyses and we vary the number of processors where necessary.

Another major issue for our experiment design is how to scale the problem size. Because the simulator has considerably slower speed and smaller memory than real machines, we typically need to scale down the problem size. To simulate realistic memory-access behavior, we need to scale down the machine size, particularly the cache size, as well. For example, if we scale down the problem size without changing the cache size, the miss rate may become unrealistically small. An earlier study [59] proposed a method to choose a cache size for a scaled problem. The proposed method uses characteristics of an important working set; fitting or not fitting an important working set significantly changes the miss rate. Thus, if the important working set is expected to fit in the cache in realistic problems and machines, one should choose a cache size that is larger than the corresponding working set for a scaled problem. If the important working set is not expected to fit in the cache, on the other hand, one should choose a cache size that is smaller than the corresponding working set for a scaled problem. This method is useful if the miss rate behavior for a variation of the cache size is about the same between architectural techniques that we compare.

Deliver and prefetch operations, however, significantly change the miss-rate behavior for a variation of the cache size. Therefore, we chose to simulate various cache sizes.

3.2 Benchmark Programs

Table 3-1 summarizes our ten benchmark applications and their inputs. These programs represent various algorithms and program/data structures in numerical-intensive applications. FFT, LU, and Ocean are from SPLASH-2 parallel application suite [59]. Barnes-Hut, LocusRoute, MP3D, Pthor, and Water are from the SPLASH-1 parallel application suite [54]. The applications from SPLASH-2 are more architecturally aware than those from SPLASH. For example, the three applications from SPLASH-2 optimize data allocation among processor nodes to reduce the number of remote memory accesses, while the applications from SPLASH-1 do not. Maxflow and Mincut have fewer than 1000 lines in

Application	Description	Input
Barnes-Hut (Barnes)	Hierarchical N-body gravitation simulation	8192 bodies 6 steps
FFT	Blocked 1-D FFT	65536 complex numbers
LocusRoute (Locus)	VLSI wire routing	Primary2 (25.8K cells, 3817 wires)
LU	Blocked LU decomposition	256×256 matrix
Maxflow	Maximum flow determination in a directed graph	full75 (a fully connected 75-node graph)
Mincut	Graph partitioning using simulated annealing	Graph500 (a 500-node graph)
MP3D	Rarefied hypersonic flow simulation	3000 molecules 6 steps
Ocean	Eddy current simulation in an ocean basin	130×130 grid
Pthor	Gate-level logic circuit simulation	RISC (5060 elements) 100 clocks (20000 ticks)
Water	Water molecule simulation	288 molecules 6 steps

Table 3-1: Benchmark Applications.

the source code and are much simpler programs than the rest of our applications. While these two applications do not represent realistic work loads, we use them to evaluate the effect of various application structures for the deliver and the prefetch schemes. In the rest of this subsection, we briefly discuss the algorithm and the data structure of each application. We will discuss more details about the memory access behavior in Subsection 4.1.4 and 4.2.3.

Barnes-Hut (Barnes for short) calculates gravitational attraction of galaxies, an N-body problem using the $O(n \log n)$ Barnes-Hut algorithm [53]. The main data structure is a tree representing hierarchically decomposed galaxies (bodies). Each leaf node represents a body, and each internal node represents a physical cell containing all child cells and bodies. Processors traverse the tree and calculate the gravitational potential of each body. If a processor finds the body is far enough away from a cell, the processor approximates the gravity of bodies in the cell by their center of mass and does not go down the tree further. In this way, the algorithm reduces the computation to $O(n \log n)$. For each time-step, processors build the tree, partition bodies, and calculate the gravity of each assigned body. We simulate 8192 bodies and use about 4000 cells. The memory size for bodies and cells is about 1.1 Mbytes. We collected statistics from the last 5 time-steps out of a 6 time-step simulation.

FFT uses a complex 1-D version of the radix $\sqrt{n}$ six-step FFT algorithm described in [5]. The main data structure is about 3 Mbytes, which includes three 256×256 complex matrices. Transpose operations generate communication between processors and are blocked to exploit the spatial locality. The matrices are statically partitioned and assigned to each processor. The partition for each processor is allocated in the local memory of the processor to minimize the number of remote memory accesses. We collected statistics after the initialization of all data structures.

LocusRoute (Locus for short) is a VLSI standard-cell router [50]. The main data structure is CostArray, each element of which maintains the number of wires passing through the corresponding cell and represents the cost when a new wire runs through the cell. Locus chooses the route with the minimum cost for each wire. Our benchmark circuit, Primary2.grin has 1290 routing cells in each of 20 routing channels (about 200 Kbytes in total). Another actively-shared data structure is DensityArray (about 100 Kbytes), which keeps track of the wire density for each routing cell. Locus is a non-synchronized program; processors update elements of the CostArray and the DensityArray without locking them. The program iterates the main loop twice to reduce the overall wiring cost. We collected statistics from both of the two iterations and used a geographical partitioning method [54] to improve the locality.

LU performs a blocked LU decomposition for a 256×256 matrix. The main data structure is the 256×256 matrix (512 Kbytes), which is blocked into 16×16 submatrices. A set of submatrices are assigned to processors by using a 2-D scatter decomposition [61]. Submatrices are allocated in the local memory of the assigned processor to minimize the number of remote memory accesses. Most of the communication occurs due to first accesses of submatrices that another processor has updated. Since the deliver operation does not send a cache line to processors that have never accessed it, the deliver operation does not eliminate cache misses due to communication. Thus, we add simple code so that each processor subscribes submatrices for which communication will occur. Since the computation is regular in LU, it is easy for a compiler to add such subscribing code automatically. We collected statistics only for the main computation, not for the initialization and the subscription of submatrices.

Maxflow finds the maximum flow in a directed graph with a single source and a single sink node [25]. The main data structure is a fully-connected 75-node graph, which consists of about 220 Kbytes of node and edge records. The computation is divided into relatively small-grain tasks, which are dynamically distributed to processors. We collected statistics after the initialization of all data structures.

Mincut tries to split a graph into two partitions with a minimum number of cuts by using a simulated annealing algorithm. The main data structure is a 500-node graph (82 Kbytes). This program does not attempt to divide computations for reducing the communication-to-computation ratio or to allocate data records for reducing the number of remote memory accesses. We collected statistics from the last 22 iterations out of 31 iterations.

MP3D is a rarefied hypersonic flow simulator that uses a Monte Carlo method for modeling particle collisions [43]. The main data structures are two arrays: one for particles and one for cells. Each record of the particle and the cell arrays represents properties of each particle (molecule) and each cell (unit-sized physical space), respectively. Particle records are statically partitioned and assigned to each processor. Cell records, on the other hand, are actively accesses by multiple processors, so that most of the sharing misses occur for the cell records. MP3D is a non-synchronized program; processors update cell records without locking them. Our simulation uses 3000 molecules (about 105 Kbytes) and 2352 cells (about 92 Kbytes). We collected statistics from the last 5 time-steps out of a 6 time-step simulation.

Ocean simulates eddy currents in an ocean basin [60]. The main data structure is a set of grids (about 4 Mbytes) that represent several physical values for a 130×130 grid. The rest of input parameters are the same as the default. Each grid is divided into a set of square subgrids and assigned to each processor for minimizing the communication-to-computation ratio. Subgrids are allocated in the local memory of the assigned processor for reducing the number of remote memory accesses. We collected statistics after the first step of the iterations.

Pthor is a parallel logic simulator that uses a variant of the Chandy-Misra [11] distributed-time algorithm. The main data structures include element and node structures that correspond to logic circuit elements and wires respectively. Each node has an event list that keeps track of value changes at that node. Our input circuit is a small RISC processor that consists of 5060 elements. The data-set size for this input is about 1.5 Mbytes. We simulated the circuit for 100 clocks (20,000 ticks) and collected statistics after all elements and nodes are initialized.

Water is adapted from the Perfect Club Benchmarks [7] and simulates water-molecule system in a liquid state by using an $O\left(n^2\right)$ N-body algorithm with a spherical cut-off radius, which eliminates computations for unimportant molecule interactions. The main data structure is an array of water-molecule records, which are statically allocated to processors. Our simulations use 288 molecules (187 Kbytes), and we collected statistics from the last 5 steps out of 6 step.

Chapter 4

Sharing Characteristics of Parallel Applications

As we discussed in Chapter 2, deliver operations can significantly decrease the number of sharing misses but can significantly increase the amount of the traffic. Our simulation results show that both the deliver and the prefetch schemes can generate a large number of unnecessary data transfers for applications with irregular computations. In this chapter, we discuss the sharing characteristics and the application structures that determine the behavior of deliver and prefetch operations, and we compare the efficiency of the deliver and the prefetch schemes. We show that the prefetch scheme has a significant advantage in reducing the number of unnecessary data transfers due to irregular computations.

Since our focus is on the true sharing characteristics of application programs, we assume that each processor has an infinite cache with 16-byte lines. The infinite cache eliminates memory accesses due to finite cache capacity, and the 16-byte line size sufficiently reduces the effect of false sharing. As we discussed our simulation strategy in Section 3.1, we use a unit-delay memory model in this chapter since our focus is on application characteristics instead of the performance.

In this chapter, we begin with examining the effect of the sharing characteristics for the deliver scheme in Section 4.1. We identify the sharing patterns that cause unnecessary data transfers in the deliver scheme. Detailed discussions of sharing characteristics presented in the section will enable us not only to understand the behavior of the deliver operation, but also to examine techniques that improve the efficiency of the deliver operation. In Section 4.2, we examine the effect of the sharing characteristics for the prefetch scheme. Finally, we summarize our discussions about the sharing characteristics of applications and their impact for the deliver and the prefetch operations.

4.1 Producer-oriented Deliver

By using the deliver operation only where it works efficiently, we can decrease the number of cache misses with a minimum increase in the amount of the traffic. We examine several techniques that trade off the number of misses versus the amount of the traffic. While the deliver scheme is a producer-oriented approach, the memory-access behavior of both producers and consumers affects the efficiency of the deliver operation; producers directly initiate the data transfer operation, and consumers control the destination of the data transfer operation. We examine the sharing pattern from both the producer and the consumer perspectives to understand the application characteristics that affect the behavior of deliver operations and to evaluate trade-off techniques that use sharing information from different perspectives.

In this section, we first discuss our assumptions and define several terms to be used in this chapter. Second, we discuss the sharing characteristics from the consumer's perspective. Third, we discuss the sharing characteristics from the producer's perspective. Fourth, we discuss details about application algorithms and data structures to understand the difference in the effectiveness of trade-off techniques that improve the efficiency of deliver operations.

4.1.1 Background

Our deliver scheme assumes that programmers or language systems explicitly insert deliver operations into the application. In this chapter, however, we use a simulator that automatically inserts deliver operations for all shared variables to examine sharing patterns thoroughly and to avoid limitations of existing language systems. At each synchronization operation, the simulator inserts a deliver operation so that the processor delivers all cache lines that the processor has recently written. The deliver message is sent to all processors on the deliver vector of the cache line.

For non-synchronized programs (i.e., Locus and MP3D), I manually insert pseudo-synchronization operations where deliver operations should be executed after a series of shared data updates.[1] The pseudo-synchronization operation does not cause actual synchronization but only specifies the place where deliver operations should be inserted. The pseudo-synchronization operation does not, therefore, affect the memory access pattern or the execution path of application programs. Adding pseudo-synchronization operations does not change a non-synchronized program to a synchronized one.

Our simulation model assumes that deliver operations have no instruction overhead and that deliver messages are sent without any delay. We simply measure the benefit of the deliver operation by the decrease of the number of read misses. This metric is valid because we assume aggressive write buffering techniques that can hide write access latencies; hence read misses have a primary effect on the performance. We measure the overhead of the deliver operation by the increase of the number of deliver messages. These simplifications allow us to focus only on the interplay between the sharing characteristics of applications and the behavior of deliver operations. We use a more realistic machine model to evaluate the effect of deliver operations for the execution time in Section 5.4.

Since we assume an infinite cache, deliver operations can eliminate most of the read misses but not all of them. We classify read misses to identify those that deliver operations can eliminate.

Cold miss: A cache miss due to an initial read for the cache line. Since deliver messages are not sent to processors that have never accessed the cache line, deliver operations do not eliminate these read misses.

Sharing miss: A cache miss due to an invalidation for a write operation of another processor. These misses are divided into two types by the timing of the write and the following read operations:

Synchronous sharing miss: A sharing miss that occurs after the last writer executes a synchronization or a pseudo-synchronization operation. For synchronized programs, communication occurs only through this type of miss. For non-synchronized programs, communication may occur through another type of miss, namely, asynchronous sharing misses. Deliver opera-

1. For Locus, a pseudo-synchronization operation is inserted immediately after each loop that updates *CostArray* elements. For MP3D, a pseudo-synchronization operation is inserted at the end of *move_single()*, which updates one particle.

tions can completely eliminate synchronous read misses because we assume no delay in sending deliver messages.

Asynchronous sharing miss: A sharing miss that occurs before the last writer executes a synchronization or a pseudo-synchronization operation. In real machines, deliver operations may or may not reduce these read misses. In our memory model, however, deliver operations cannot reduce these read misses since we assume the last writer performs a deliver operation at the synchronization point following the write operation.

Figure 4-1 shows the ratio of the number of read misses for each miss type when no deliver operations are used. We assume an infinite cache with 16-byte lines. Figure 4-1 shows that the number of read misses is dominated by synchronous sharing misses, which deliver operations can eliminate. The ratio of asynchronous read misses is small (< 7.2%) for synchronized applications and also small (< 11%) for non-synchronized applications. This is because the cache line size is so small (16 bytes) that false sharing does not have a significant effect on the number of cache misses and also because pseudo-synchronization operations are properly inserted in non-synchronized applications so that processors do not frequently read shared data before the previous writer executes a pseudo-synchronization operation. The ratio of cold misses is relatively large for Locus (25%), Barnes (21%), and Maxflow (22%) particularly because of limited data reuse. For example, Maxflow is

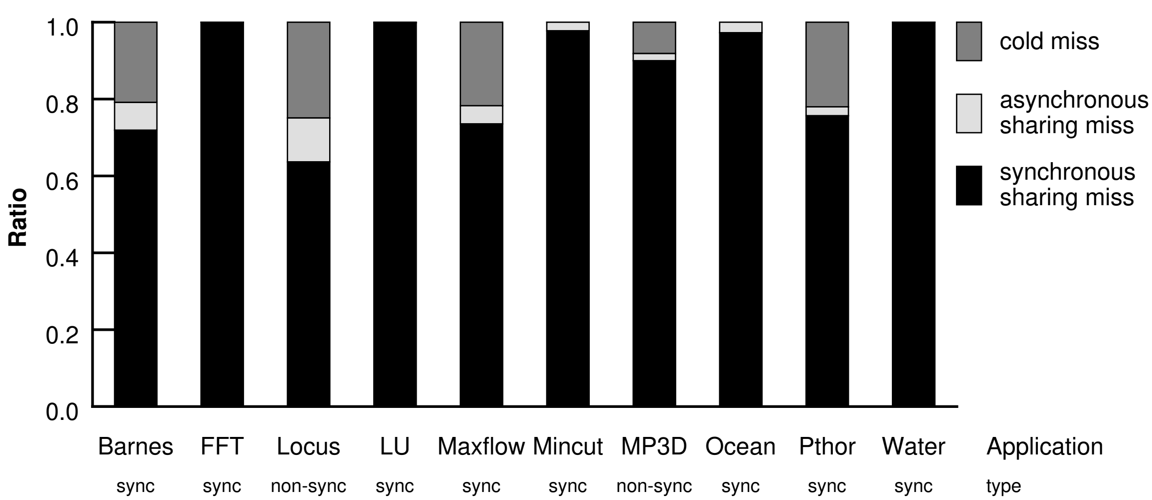

Figure 4-1: Ratio of Read Miss Types. No deliver operations are used.

not an iterative program, and Locus is iterative but does not require a large number of iterations; hence data reuse is limited. Barnes, on the other hand, is an iterative program, and each processor generally reuses the same data record in the logical data structure. The same logical data record, however, is not always allocated in the same physical memory location because of dynamic data allocations. Thus, Barnes generates a relatively large number of cold misses. Pthor also has a relatively large ratio of cold misses (22%). This results from the artifact of our simulation that takes statistics from a relatively short simulation. The ratio should be smaller in actual uses.

The benefit of the deliver operation appears as some reduction of synchronous sharing misses, which is defined as a *covered ratio*:

$$\text{covered ratio} = 1 - \frac{\text{the number of synchronous sharing misses with delivers}}{\text{the number of synchronous sharing misses without delivers}} \qquad \text{EQ 4-1}$$

The covered ratio varies as the threshold of the competitive back-off technique or the number of inserted deliver operations varies. If competitive back-off is not used, for example, the covered ratio is zero when no deliver operations are executed, and the covered ratio is one when a deliver operation is executed at all synchronous or pseudo-synchronous operations for all written shared data.

We measure the overhead of the deliver operation as the number of deliver messages relative to the number of synchronous sharing misses without deliver operations, which is defined as a *deliver message overhead*:

$$\text{deliver message overhead} = \frac{\text{the number of deliver messages sent}}{\text{the number of synchronous shring misses without delivers}} \qquad \text{EQ 4-2}$$

In this chapter, we count only deliver messages that are sent to processors and do not count acknowledgment messages or messages between processors and directories. This allows us to ignore implementation details of the deliver mechanism. Since we need at least one deliver message to eliminate one miss, the deliver message overhead cannot be smaller than the covered ratio. If no deliver message is wasted, the deliver message overhead is the same as the covered ratio.

4.1.2 Sharing Characteristics from the Consumer's Perspective

It would be ideal if a processor used every received deliver message. In reality, as shown in Figure 4-2, a processor may use only one message out of multiple deliver messages that the processor receives between two consecutive accesses for the same cache line. As illustrated in Figure 4-2, we define a *deliver-run* as a sequence of received deliver messages for the same cache line without any intervening accesses by the local processor. The length of the deliver-run, measured by the number of deliver messages received, shows how long it has been since the processor has not used the cache line. We can see the deliver-run as a consumer's access pattern for deliver operations.

The deliver-run is similar to the write-run that was proposed by Eggers and Katz [21]. The write-run is defined as the number of writes that a producer performs for the same address without intervening accesses by any other processors. Thus, the write-run is an architecture-independent metric of applications and useful to determine whether the write-invalidate or the write-update policy produces less coherency traffic for snoop cache systems with one-word lines. The deliver-run is different from the write-run in two ways. First, the deliver-run takes into account the fact that a deliver operation aggregates all updates that the producer made for the same cache line. Thus, the deliver-run length depends on the line size, while the write-run length does not. Second, the deliver-run length corresponds to the number of messages due to deliver operations in a point-to-point interconnection network, while the write-run length corresponds to the number of bus transactions due to update operations in a bus-based interconnection network.

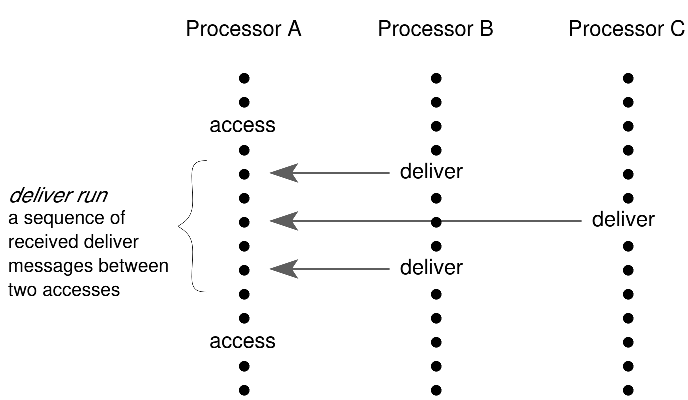

Figure 4-2: Deliver-run.

In this section, we analyze characteristics of deliver-runs to discuss consumer's access patterns as well as consumer-oriented techniques that reduce the deliver message overhead. First, we categorize deliver-runs to identify ones that eliminate a cache miss. Second, we examine the distribution of the length of deliver-runs to understand consumer's access behavior. Finally, we discuss two techniques in which consumers control the subscription of deliver messages by using information about deliver-runs; in one technique, consumers dynamically control the subscription at run-time, and in the other technique, consumers statically control the subscription at compile-time.

4.1.2.1 Classification of Deliver-runs

Not all deliver-runs eliminate a cache miss. As shown in Table 4-1, we classify deliver-runs into four types by the operation type that the consumer performs at the end of the deliver-run. A cache miss can be eliminated only for sync-read deliver-runs: deliver-runs that are followed by a consumer's synchronous read operation. Figure 4-3 shows the ratio of the number of deliver messages for each deliver-run type. The ratio of async-read deliver-runs is small (< 4.7%) for all applications, and the ratio of write deliver-runs is also small (< 5.5%), except Pthor (21%). Most of the write deliver-runs occur when the program overwrites data structures without reading them. In Pthor, about 70% of write deliver-runs occur at overwriting operations when the program appends or deletes an element of linked lists or when the program sets a flag associated with each task queue.

The ratio of open deliver-runs is relatively large in Barnes and Locus because of dynamic data assignment to processors. As the data assignment changes, deliver vectors begin to hold processors that are not using the cache line anymore. Thus, deliver messages to those processors cause open deliver-runs. The ratio of open deliver-runs is particularly large in Barnes. Barnes reconstructs the main data structure (a Barnes-Hut tree) for each iteration

deliver-run type	consumer's operation that ends the deliver-run
sync-read deliver-run	synchronous read operation
async-read deliver-run	asynchronous read operation
write deliver-run	write operation
open deliver-run	no operation (the end of the program)

Table 4-1: Deliver-run Types.

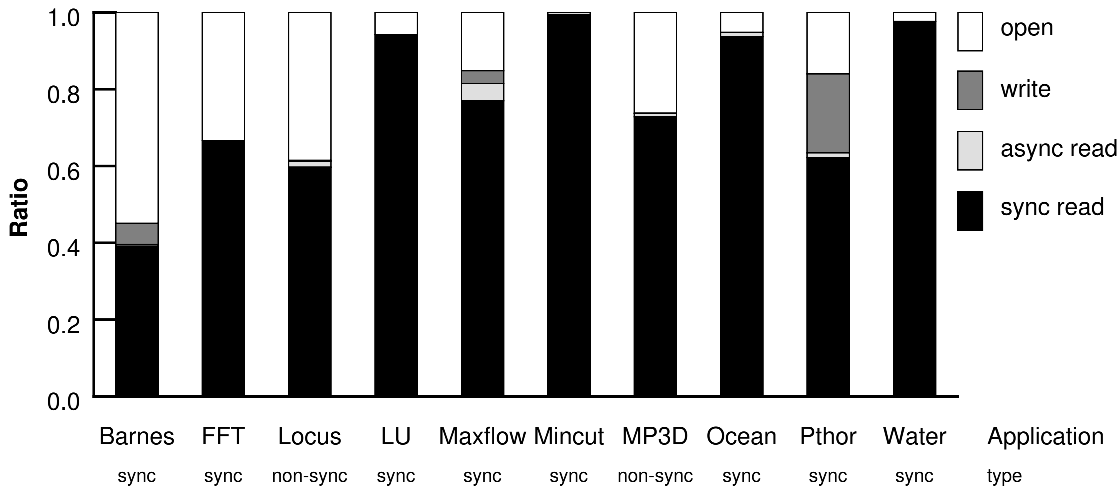

Figure 4-3: Ratio of Deliver-run Types. Deliver-runs are classified by the consumer's operation that terminates the deliver-run.

by dynamically allocating non-leaf nodes, so between iterations the same logical node of the tree might be mapped to different physical memory addresses. Thus, although the same logical node is used by the same or a similar set of processors between iterations, the deliver vector for non-leaf nodes does not represent the algorithmic access pattern. In Barnes, therefore, the dynamic memory allocation causes the large ratio of open deliver-runs.

4.1.2.2 Distribution of Deliver-runs

Now, we discuss the distribution of the length of deliver-runs. Figure 4-4 shows a cumulative distribution of the length of sync-read deliver-runs for our applications. The length of a sync-read deliver-run indicates the efficiency of deliver messages that the deliver-run generates; if a deliver-run length is l, one deliver message is used and $l-1$ deliver messages are not used. FFT, Mincut, and Ocean have an advantageous characteristic for the deliver scheme. The length of all sync-read deliver-runs is one for FFT, and the length of more than 90% of sync-read deliver-runs is one or two for Mincut and Ocean. Thus, these applications use most of the transmitted deliver messages. For the rest of applications, however, more than 50% of the deliver-runs are longer than two. That is, processors receive more than two deliver messages for more than half of the synchronous read operations. In particular, Water and Pthor have a large ratio of long deliver-runs: more than 20% of their deliver-runs are longer than ten.

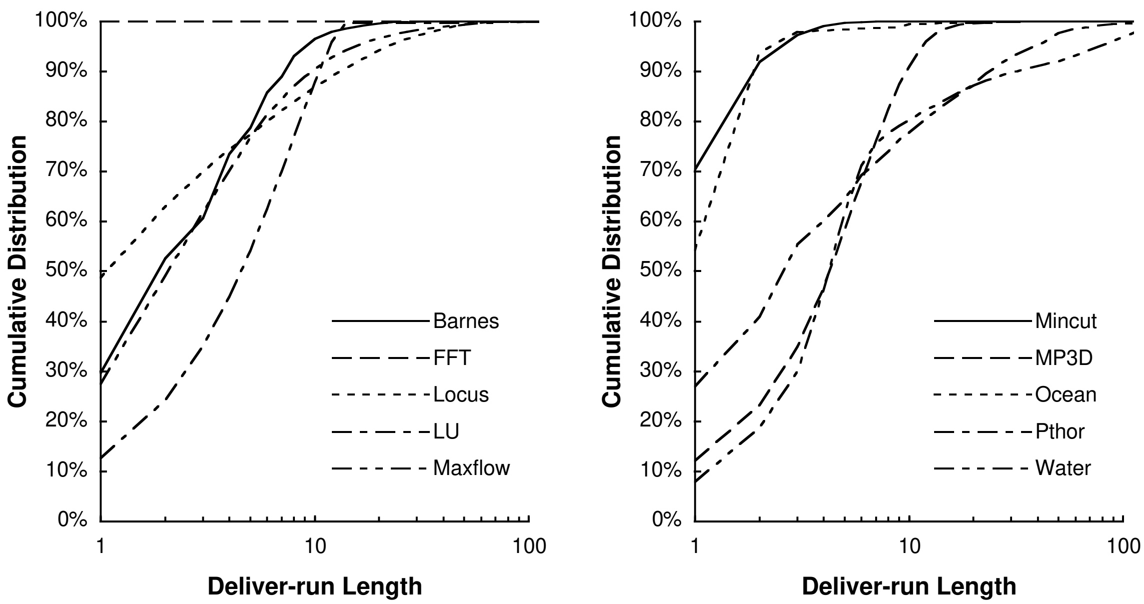

Figure 4-4: Cumulative Distribution of Sync-read Deliver-runs.

A simple sharing pattern between producers and consumers causes short deliver-runs in FFT. Each processor performs transpose operations between two statically-partitioned matrices, and each processor writes the results to other processor's partitions. Synchronous sharing misses occur when a processor reads a part of its partition that was written by other processors. Because processors write shared data in other processor's partitions only once, the length of sync-read deliver-runs is always one.

Migratory sharing patterns cause long deliver-runs in Pthor and Water although the two applications have a very different structure. In Pthor, each processor picks up a circuit element from a task queue that contains circuit elements to be evaluated and updated, and if the task queue becomes empty, the processor moves to another queue. Each task queue is associated with a different set of circuit elements. Thus, as the program proceeds, the ownership of cache lines for each circuit-element record migrates among processors, and the deliver vector of cache lines for each circuit element holds the identity of all processors that have ever evaluated the element. Deliver operations, therefore, keep sending messages even to those processors that are no longer accessing the circuit element for a long period. Thus, the length of sync-read deliver-runs tends to become long in Pthor.

Water, on the other hand, is very different from Pthor in both program and data structures. Water does not use task queues. The main data structure, a set of molecules, is statically partitioned and assigned to each processor. While each processor most often updates molecules that are assigned to the processor, each processor sometimes updates other molecules that are assigned to another processor. As a result, the ownership of cache lines for each molecule record migrates among processors, and the deliver vector of those cache lines holds the identity of all processors that have updated the molecule. When a processor updates local molecules, therefore, a deliver operation sends a message to processors that access those molecules only infrequently. Although Pthor's and Water's methods of partitioning differ completely, migratory sharing patterns generate a large number of unused deliver messages in both applications.

4.1.2.3 Consumer Control of Deliver Operations

The distribution of the length of deliver-runs indicates the efficiency of deliver operations as well as the degree of potential improvements of deliver operations. While the deliver operation is producer-oriented, consumers determine the destination of deliver operations; consumers subscribe to a cache line for future deliver operations by accessing the cache line. If the consumer knows that a deliver-run will become long, the consumer can notify the cache directory to cancel the subscription.[2] This subscription control can be done by using run-time or compile-time information as illustrated in Figure 4-5.

The *competitive back-off* technique (the left hand side in Figure 4-5) uses run-time information about deliver-runs. The consumer counts the length of the deliver-run for each cache line and cancels the subscription of the line when the length exceeds a certain threshold. The *subscription control* technique (the right hand side in Figure 4-5), on the other hand, uses compile-time information about deliver-runs. This technique assumes that the language system (e.g., compiler and profiler) knows the average length of deliver-runs that follow each source-code line with a memory access. The language system inserts

2. Read and write operations automatically subscribe the accessed cache line in our deliver scheme. If an application does not need to subscribe lines normally, we can choose not to subscribe cache lines automatically and to insert subscribing operations explicitly in the application.

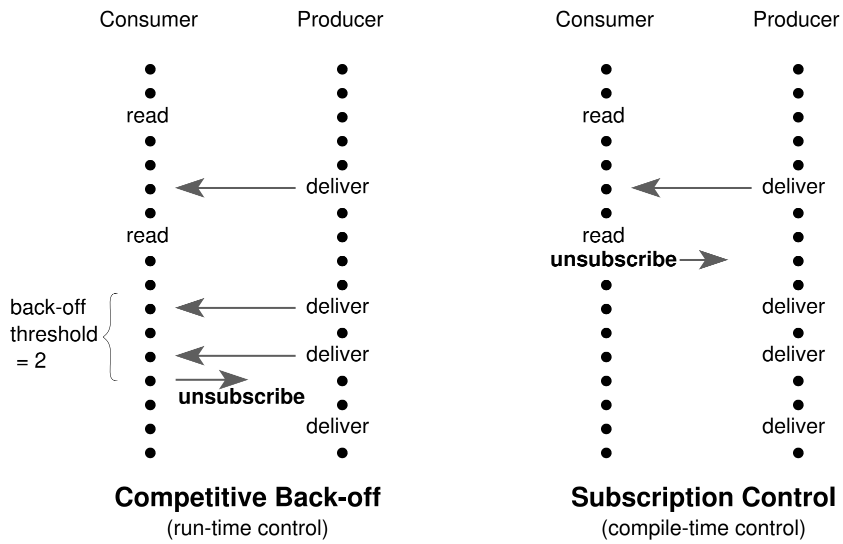

Figure 4-5: Consumer Control of Deliver Operations.

a special operation, an unsubscribe operation, after a source code line if the average length of deliver-runs that follow the source code line is longer than a certain threshold.[3]

Both techniques have an advantage and a disadvantage. While competitive back-off wastes received deliver messages up to the back-off threshold if the deliver-run is longer than the threshold, subscription control does not waste such deliver messages since the subscription can be canceled at the beginning of the deliver-run. The subscription control technique, however, is not adaptable to changing sharing patterns at run-time because this technique applies the same subscription policy to all memory accesses for the same source code line. Thus, if the length of deliver-runs for the same source-code line changes significantly during run-time, the subscription control technique does not effectively improve the efficiency of deliver operations. Competitive back-off, on the other hand, can selectively unsubscribe deliver messages only for the long deliver-runs.

Analyses of deliver-run characteristics can tell us which of the two techniques discussed above is more effective than the other. The cumulative distribution of sync-read deliver-

3. Our subscription control technique performs a subscribe or unsubscribe operation only at the beginning of a deliver run (i.e., immediately after the use of the cache line). A more general technique can perform such an operation at any arbitrary places in the source code. If the subscribe operation is performed near the end of a deliver-run, the operation becomes similar to a prefetch operation.

runs (Figure 4-4) shows the covered ratio for the competitive back-off technique. When the threshold of the back-off is *th*, and if a sync-read deliver-run is equal to or shorter than *th*, the following synchronous read becomes a read hit. Consequently,

$$\text{covered ratio}\,(\text{threshold} = th) = \frac{\sum_{l=1}^{th} D_{\text{sync}}(l)}{\sum_{l=1}^{\infty} D_{\text{sync}}(l)} \qquad \text{EQ 4-3}$$

where $D_{\text{sync}}(l)$ is the number of sync-read deliver-runs whose length is l. The deliver message overhead is also obtained from the distribution of deliver-runs. When a deliver-run is equal to or shorter than the threshold *th*, the processor receives all deliver messages for the deliver-run. When a deliver-run is longer than *th*, on the other hand, the processor receives deliver messages only up to *th*. Thus, the deliver message overhead is

$$\text{deliver message overhead}\,(\text{threshold} = th)$$
$$= \frac{\sum_{l=1}^{th} l \times D_{\text{any}}(l) + \sum_{l=th+1}^{\infty} th \times D_{\text{any}}(l)}{\sum_{l=1}^{\infty} D_{\text{sync}}(l)} \qquad \text{EQ 4-4}$$

where $D_{\text{any}}(l)$ is the number of any deliver-runs whose length is l.

The above two equations give us the trade-off between the covered ratio and the deliver message overhead when the back-off threshold varies. Similar equations show the covered ratio and the deliver message overhead for the subscription control technique. We define an effective deliver-run length (l_{eff}) for each source-code line with a memory operation; l_{eff} is the total of the length of deliver-runs following the line divided by the number of sync-read deliver-runs following the line. Note that l_{eff} indicates the number of received deliver messages per synchronous read. An unsubscribing operation is inserted after a source-code line if the l_{eff} is larger than a threshold, *th*. Therefore,

$$\text{covered ratio}\,(\text{threshold} = th) \;=\; \frac{\sum_{l \le th} \hat{D}_{\text{sync}}(l)}{\sum_{l \le \infty} \hat{D}_{\text{sync}}(l)}$$

$$\text{deliver message overhead}\,(\text{threshold} = th) \;=\; \frac{\sum_{l \le th} l \times \hat{D}_{\text{any}}(l)}{\sum_{l \le \infty} \hat{D}_{\text{sync}}(l)} \qquad \text{EQ 4-5}$$

where $\hat{D}_{\text{sync}}(l)$ and $\hat{D}_{\text{any}}(l)$ are the number of sync-read deliver-runs and any deliver-runs following source-code lines whose l_{eff} is l. Note that the subscription control technique does not generate deliver messages if l_{eff} is larger than th, while competitive back-off generates th deliver messages if the deliver-run length is larger than th.

We will discuss detailed characteristics of the competitive back-off and the subscription control techniques along with a producer-oriented technique in Section 4.1.4. For now, we discuss only some general characteristics of the two consumer-oriented techniques by comparing two examples. Figure 4-6 shows the trade-off between the deliver message overhead versus the covered ratio for Barnes and Locus. For Barnes, subscription control performs significantly better than competitive back-off. The computation of each iteration in Barnes consists of several phases, and the memory access pattern is significantly different among those phases. Thus, subscription control can utilize such knowledge about

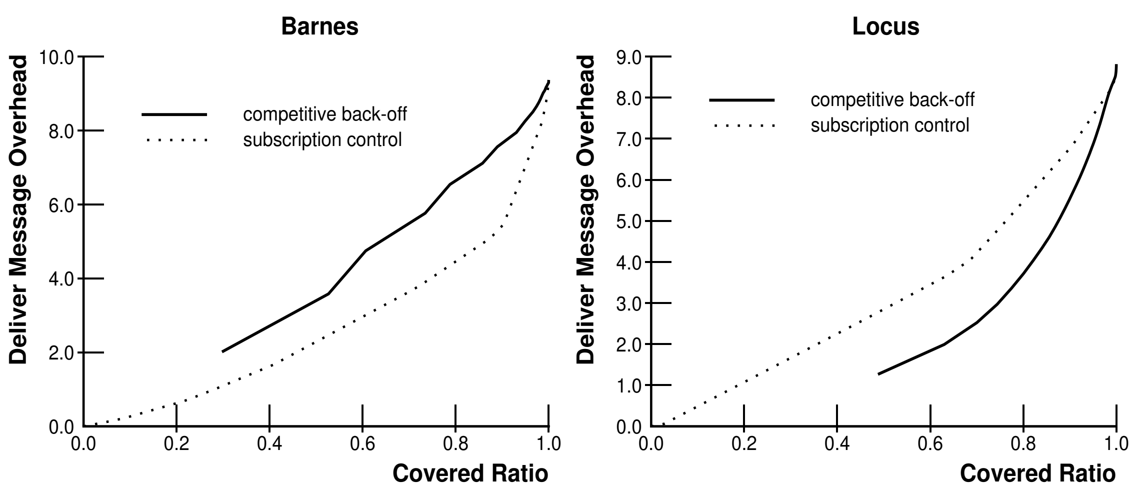

Figure 4-6: Deliver Message Overhead versus Covered Ratio (Consumer-initiated Trade-offs).

access patterns and apply deliver operations only to some of those phases in which deliver operations work efficiently. Competitive back-off, however, cannot utilize such knowledge. Thus, subscription control performs better than competitive back-off. Locus, on the other hand, has an opposite characteristic. Locus accesses the main data structure, *CostArray*, primarily at a few locations in the source code so that the same source-code line is followed by deliver-runs that have a dramatically different length. Thus, subscription control cannot selectively apply deliver operations only for short deliver-runs, while competitive back-off can. The program structure, therefore, strongly affects the effectiveness of these subscription techniques.

4.1.3 Sharing Characteristics from the Producer's Perspective

In the previous subsection, we discussed the consumer's access patterns by examining the distribution of the length of sync-read deliver-runs, which shows the number of received deliver messages per used deliver message. In this subsection, we discuss similar statistics from the producer's perspective. Namely, we analyze the distribution of the number of delivered and used messages as shown in Figure 4-7 and Figure 4-8. As we did for consumer's sharing characteristics, we first examine characteristics of the distribution to understand producer's access behavior. Then, we discuss a technique to improve the efficiency of deliver operations by using the producer's sharing characteristics.

4.1.3.1 Delivered Messages versus Used Messages

Figure 4-7 and Figure 4-8 show three-dimensional histograms of the number of delivered and used messages; each bar shows the ratio of deliver operations that cause the corresponding number of delivered and used messages. Note that several applications have a large ratio of deliver operations that do not actually generate deliver messages (23% for Barnes, 34% for Locus, 42% for MP3D, 88% for Ocean, and 50% for Pthor). This type of deliver operation occurs when data records are allocated in shared memory but used exclusively by a single processor. Examples include non-boundary grid elements in Ocean, most of the particle records in MP3D, and local body and cell tables for each processor (*mybodytab* and *mycelltab*) in Barnes. Deliver operations occur for these data records since our simulator automatically inserts deliver operations for all data records in

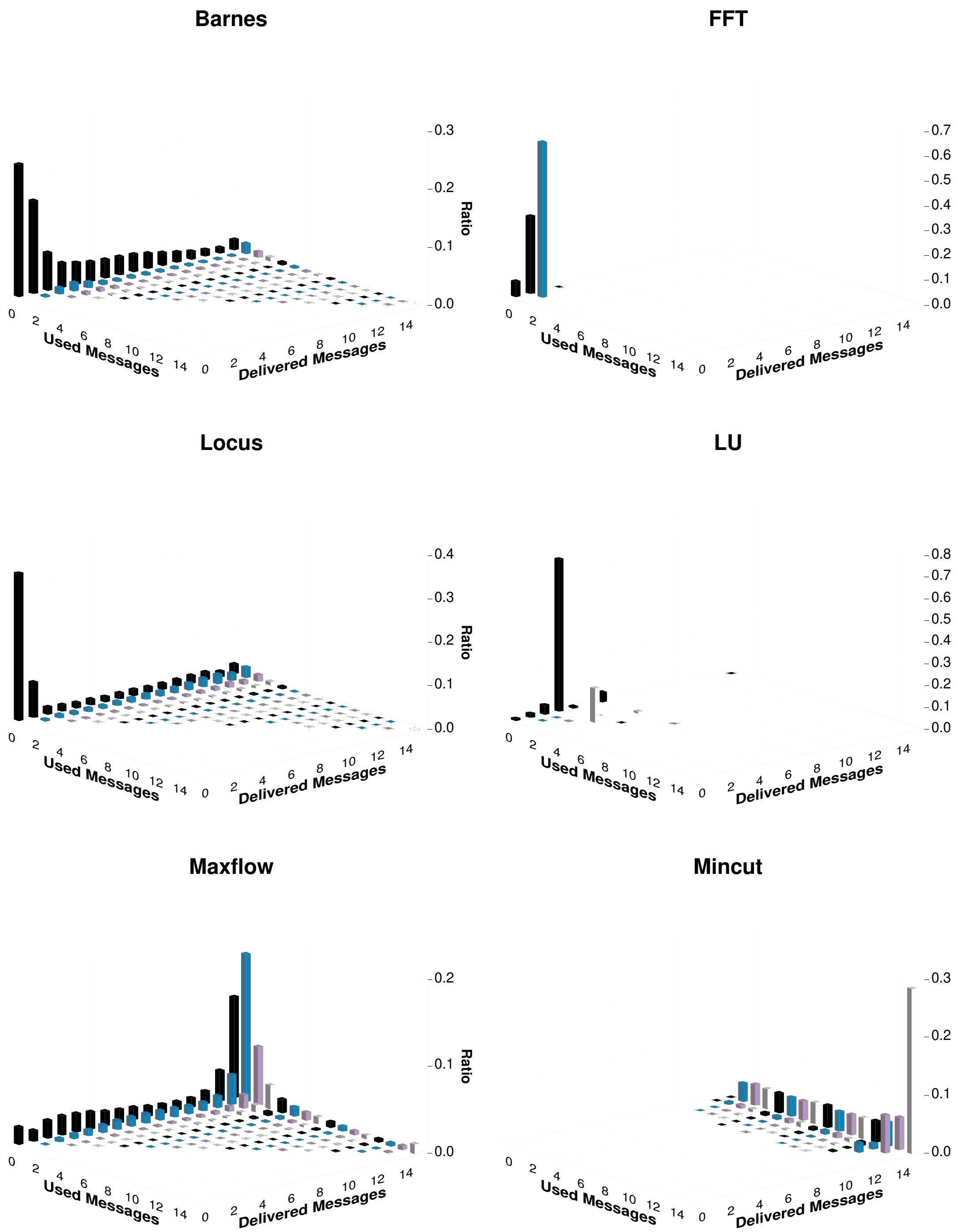

Figure 4-7: Delivered Messages versus Used Messages.

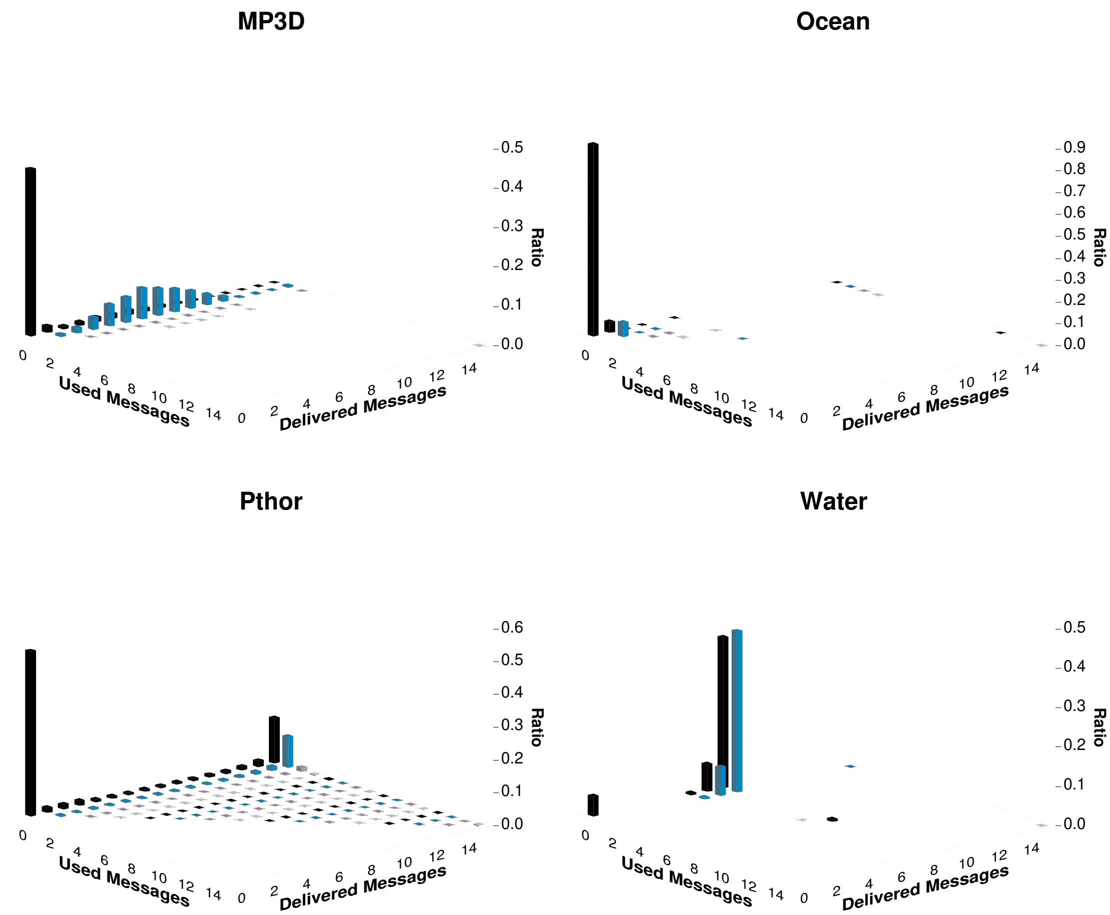

Figure 4-8: Delivered Messages versus Used Messages. (Continued.)

shared memory. Language systems or programmers can easily identify those data records and eliminate deliver operations for them.

Another common pattern in these histograms is that the number of delivered messages is often much larger than that of used messages. For some applications, a large ratio of deliver operations (56% for Barnes, 82% for LU, and 46% for Water) are completely useless; no delivered messages are used. We call such deliver operations *useless deliver operations*. The useless deliver operation occurs when no processors or only the same processor uses the delivered cache line. In particular, when only the same processor uses the delivered cache line, we call the access pattern *single-processor reuse*. Single-processor reuse occurs when one processor frequently uses a datum exclusively but other processors sometimes access it. In Barnes, for example, single-processor reuse occurs during the tree construction phase. Bodies (leaf nodes) are partitioned for each processor so that each processor's portion of the tree has a minimum overlap with another processor's portion. Thus, during the tree construction phase, most of the internal tree nodes (cells) are accessed by the same processor. During the tree traverse phase, however, multiple processors read most of the cells, so that the deliver vector holds those processors. Thus, during the next tree construction phase, deliver messages are sent to all processors that read the cell during the last tree traverse phase even if only one processor is frequently using the cell. Therefore, useless deliver operations occur in Barnes because of the single-processor reuse of cell records. In Water, single-processor reuse occurs during the inter-molecule-force calculation phase. As discussed in Subsection 4.1.2, each molecule is most often accessed by the processor to which the molecule is assigned. Each molecule, however, is also accessed from about half of the processors at some point (from seven to nine processors in our case). Therefore, when a molecule is updated, deliver messages are sent to all of those processors, although the molecule is most frequently accessed by the same processor to which the molecule is assigned. Hence, useless deliver operations occur in Water because of the single-processor reuse of molecule records.

Even if a deliver operation is not completely useless (i.e., not a useless deliver operation), the number of used deliver messages is often small. As shown in Figure 4-7 and Figure 4-8, the number of used messages is often exactly one for deliver operations that generate at least one used message. Table 4-2 shows the average of the number of used and delivered messages for deliver operations that generate at least one used message. The average of the number of used messages is between one and three for nine applications (i.e., all applications except Mincut). The average of the number of delivered messages is, however, much larger than one for six out of the nine applications (i.e., Barnes, Locus, Maxflow, MP3D, Pthor, and Water). Thus, these six applications use only a small portion of delivered messages (< 30%) as shown in Table 4-2. This behavior occurs because between two writes to a cache line, it is read by only a subset of the processors that ever use the cache line. We call this access pattern *reader migration*. A special case of reader migration is called *stale deliver vector* where some of reader processors migrate and never again use the cache line. A stale deliver vector occurs because the deliver vector holds all readers that have ever touched the cache line unless readers clear the subscription of the deliver

Application	used messages (average)	delivered messages (average)	$\frac{\text{used messages}}{\text{delivered messages}}$
Barnes	2.39	9.00	26.6%
FFT	1.00	1.00	100.0%
Locus	1.83	10.45	17.5%
LU	2.97	3.12	95.2%
Maxflow	2.95	13.07	22.6%
Mincut	10.37	14.76	70.3%
MP3D	1.01	6.44	15.7%
Ocean	1.06	1.13	93.8%
Pthor	1.87	12.44	15.0%
Water	1.09	7.84	13.9%

Table 4-2: Deliver Messages Statistics.[a]

a. The number of used and delivered messages are an average in deliver operations that generate at least one used message.

message. A stale deliver vector causes open deliver-runs. Barnes and Locus have a relatively large ratio of open deliver-runs (see Figure 4-3). Another special case of reader migration is called *migratory access*, in which only one processor uses the data record at a time and updates the record at the end of the use.[4] Since the update invalidates previous deliver messages that are sent to other processors, only one deliver message is used for each deliver operation, while each deliver operation sends a deliver message to all processors that have used the record. Migratory access thus generates a large number of unused deliver messages.

Four applications — FFT, LU, Mincut, and Ocean — use a large portion of the delivered messages (> 70%) as shown in Table 4-2. In FFT, LU, and Ocean, a static producer-consumer relation makes deliver operations very efficient. FFT and Ocean have a static one-to-one producer-consumer relation for most of the shared memory accesses, so the producer generates a single deliver message to the consumer. LU, on the other hand, causes useless deliver operations because the producer updates a data structure several times before the consumer reads the data structure. Since our simulator automatically inserts a

4. Gupta and Weber [30] called data objects that cause migratory accesses *migratory data objects*.

deliver operation for each updated structure at each synchronization operation, the producer generates useless deliver operations. We can, however, eliminate such deliver operations simply by inserting deliver operations only after the last update. Mincut has a completely different sharing pattern. As indicated by the large number of delivered messages per deliver operation in Table 4-2, each deliver operation sends a deliver message to virtually all processors. This is because the communication pattern is usually one-to-all (broadcasting) so that these deliver messages are often needed. As a result, Mincut uses a large ratio of delivered messages, even though the producer-consumer relation is dynamically determined.

4.1.3.2 Producer Control of Deliver Operations

If a language system knows the ratio of the number of delivered messages over the number of used messages for each candidate of the deliver annotation, the language system can selectively insert deliver operations only where the ratio is sufficiently small. Just as we examined trade-off techniques between the covered ratio and the deliver message overhead from the consumer's perspective, we shall next examine a similar trade-off technique from the producer's perspective, which is called *selective deliver*.

For each source-code line with a write operation (we call this *write-code*), we assume that a deliver operation is inserted immediately before the synchronization (or pseudo-synchronization) operation that follows the write. We define *deliver ratio* (d_{w}) for each write-code$_{\mathrm{w}}$ as

$$\begin{aligned} D_{\mathrm{w}} &= \text{the number of delivered messages generated for the write-code}_{\mathrm{w}} \\ U_{\mathrm{w}} &= \text{the number of used messages within } D_{\mathrm{w}} \\ d_{\mathrm{w}} &= \frac{D_{\mathrm{w}}}{U_{\mathrm{w}}}\ (U_{\mathrm{w}} > 0) \end{aligned} \qquad \text{EQ 4-6}$$

We consider only write-codes where the number of used messages is not zero. Note that the deliver ratio corresponds to the effective deliver-run length (l_{eff}) in the subscription control technique. The selective deliver technique inserts a deliver operation only if the deliver ratio is equal to or smaller than a threshold, *th*. Thus, the covered ratio and deliver message overhead for threshold (*th*) are

$$\text{covered ratio}\,(\text{threshold} = th) = \frac{\sum_{d_w \le th} U_w}{\sum_{d_w \le \infty} U_w}$$

$$\text{delivered message overhead}\,(\text{threshold} = th) = \frac{\sum_{d_w \le th} D_w}{\sum_{d_w \le \infty} U_w} = \frac{\sum_{d_w \le th} d_w \times U_w}{\sum_{d_w \le \infty} U_w} \qquad \text{EQ 4-7}$$

The above two equations demonstrate the trade-off between the covered ratio and the deliver message overhead when the deliver-ratio threshold varies.

Selective deliver is similar to subscription control (cf. EQ 4-5) since both techniques are a software-based solution that utilizes characteristics of sharing patterns. Both techniques, moreover, use a similar threshold — the number of deliver messages per eliminated cache miss — to improve the efficiency of deliver operations. The two techniques, however, exploit characteristics of sharing patterns from different perspectives. In subscription control, when a consumer is expected to receive more than a certain number of deliver messages until the next use of the cache line, the consumer cancels the subscription. This technique is effective, for example, in the following scenario; if a processor is migrating from one task queue to another, in the near future the processor will not reuse the data records that are associated with the previous task queue. Subscription control can reduce the number of unnecessary deliver messages by canceling the subscription for those data records. In selective deliver, on the other hand, when a producer is expected to send more than a certain number of deliver messages per eliminated miss, the producer cancels the deliver operation. This is technique is effective, for example, in the following scenario; if a processor is updating a set of data records that are assigned to the processor, other processors may not use the updated records every time when the producer performs a release operation. Selective deliver can reduce the number of unnecessary deliver messages by avoiding inserting deliver operations where communication does not occur frequently. The effectiveness of these two techniques, therefore, truly depends on the program structure. For some applications, the best technique is different for different data structures in the same program. We will quantitatively compare these software-based techniques along with hardware-based technique (competitive back-off) by examining simulation results in next subsection.

4.1.4 Application Algorithms and Deliver Operation Behavior

In Section 4.1, we have discussed general characteristics of consumers' and producers' access patterns and introduced three trade-offs between the covered ratio and the deliver message overhead: competitive back-off, subscription control, and selective deliver. In this subsection, we will examine algorithms and data structures of each application to understand the effectiveness of these three techniques.

The three techniques use a similar threshold — the number of deliver messages per eliminated miss — to control the efficiency of deliver operations. The threshold is the deliver-run length for competitive back-off (EQ 4-3), the effective deliver-run length for subscription control (EQ 4-5), and the deliver ratio for selective deliver (EQ 4-7). As we increase the threshold, the covered ratio increases. Figure 4-9 and Figure 4-10 show the covered ratio as a function of the threshold for the three techniques. Each of the three covered-ratio curves exhibits a cumulative distribution of a certain property in the sharing pattern. The covered-ratio curve of competitive back-off illustrates the cumulative distribution of the sync-read deliver-run length (the same as the one shown in Figure 4-4). The covered-ratio curve of subscription control illustrates the cumulative distribution of the effective sync-read deliver-run length, which is the number of deliver messages that a processor receives to eliminate one read miss. The covered-ratio curve of selective deliver illustrates the cumulative distribution of the deliver ratio, which is the number of deliver messages that a processor sends to eliminate one read miss. From these distribution statistics, we can calculate the deliver message overhead of the three techniques by using equations from EQ 4-3 to EQ 4-7.[5] Figure 4-11 and Figure 4-12 show the three trade-offs between the deliver message overhead and the covered ratio. As the threshold parameter (*th*) increases, the covered ratio and the deliver message overhead increase. The slope of the trade-off curve represents the ratio of the number of deliver messages that are generated to eliminate one read miss.

5. For calculating the deliver message overhead of competitive back-off and subscription control, we also need the statistics about non-sync-read deliver-runs. (See EQ 4-4 and EQ 4-5.)

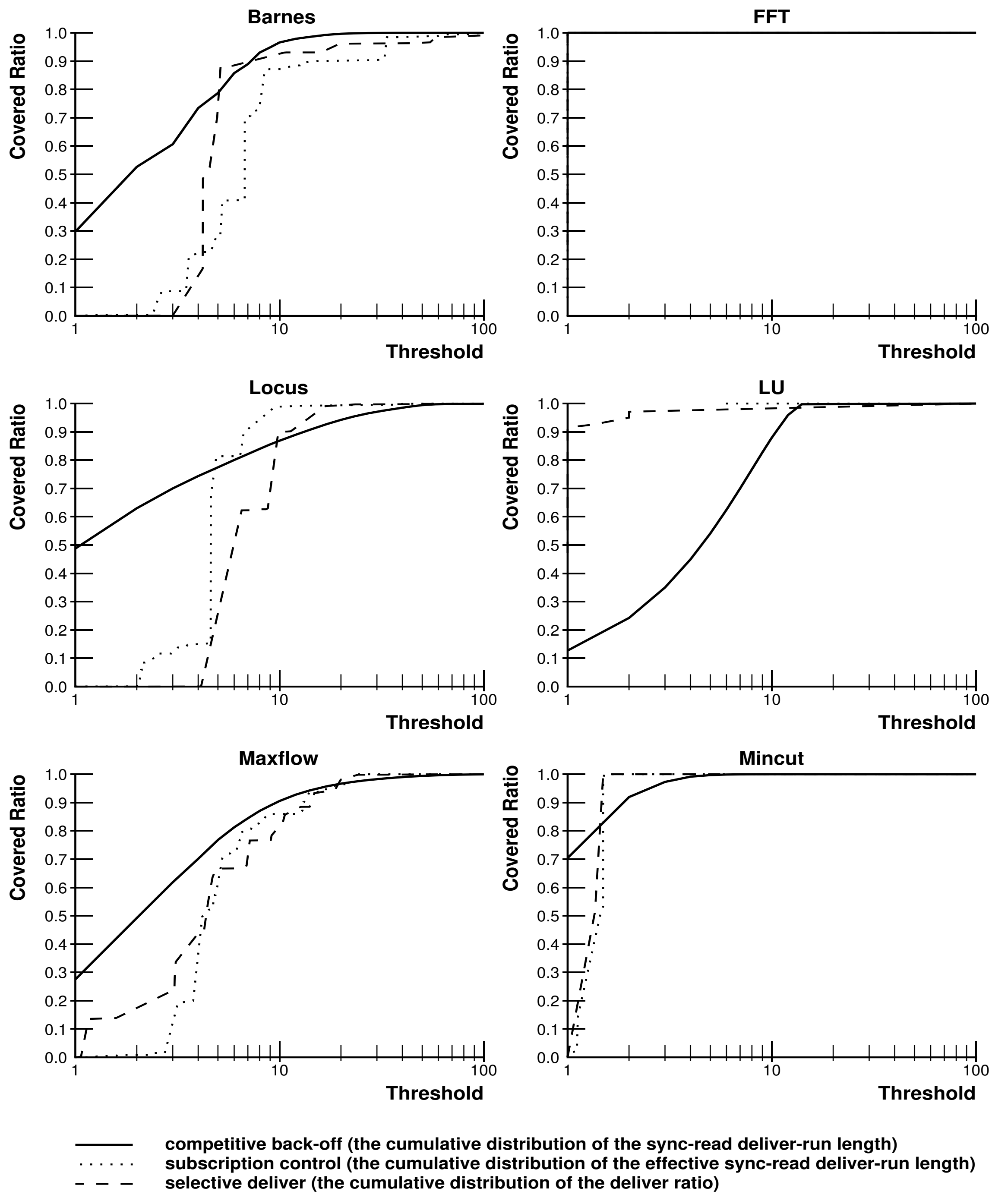

Figure 4-9: Covered Ratio versus Threshold.

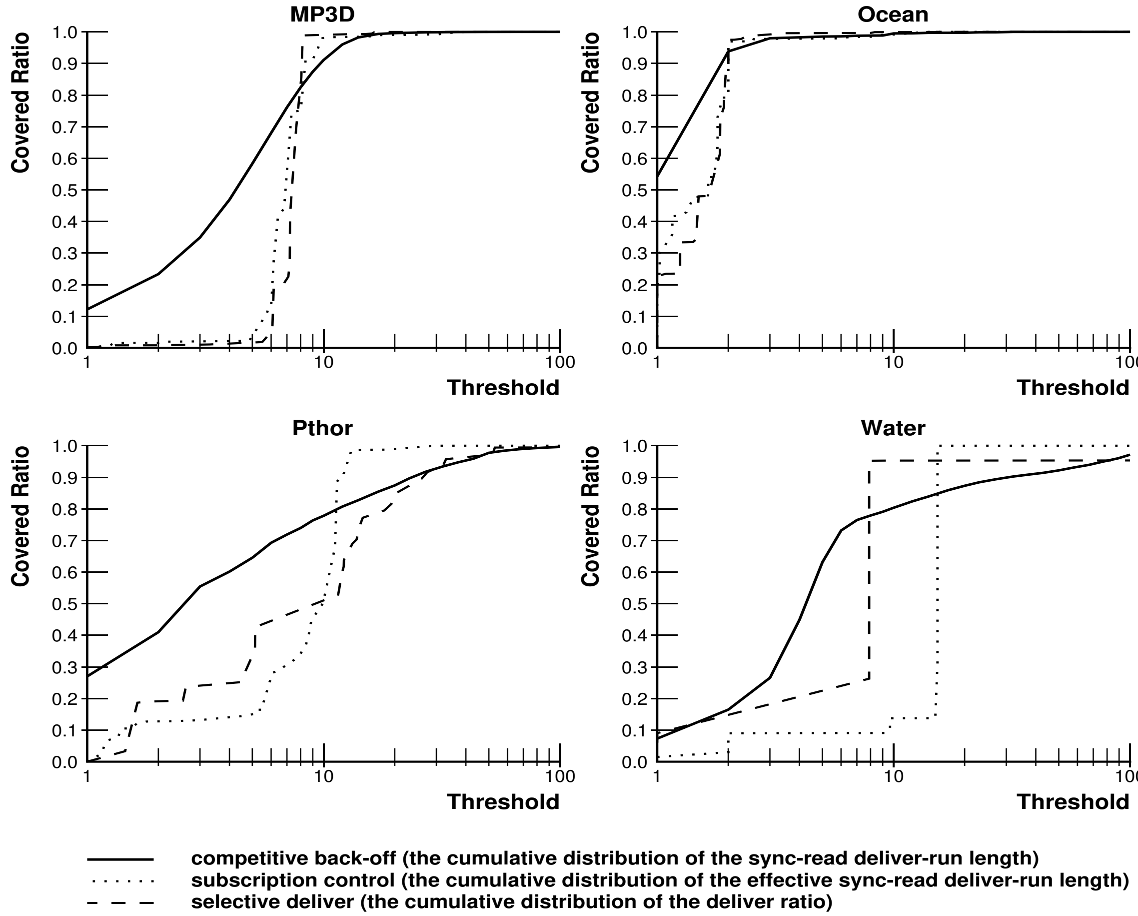

Figure 4-10: Covered Ratio versus Threshold. (Continued.)

For most of the applications, as shown in Figure 4-11 and Figure 4-12, selective deliver generates the lowest (or roughly equal) deliver message overhead among the three techniques for a wide range of the covered ratio. Only Locus is exceptional; competitive back-off performs significantly better than selective deliver for a wide range of the covered ratio. This is because the memory-access behavior changes significantly for the same memory operation at run-time in Locus, so that an adaptive technique — competitive back-off — performs better than other non-adaptive techniques. The performance difference among the techniques depends on how consumers and producers change their memory-access patterns in the program. For example, if producers display single-processor reuse only in a part of the program, selective deliver can reduce the deliver message overhead by avoiding deliver operations in that part of the program. However, if producers display single-processor reuse with the same frequency in all places where the producer

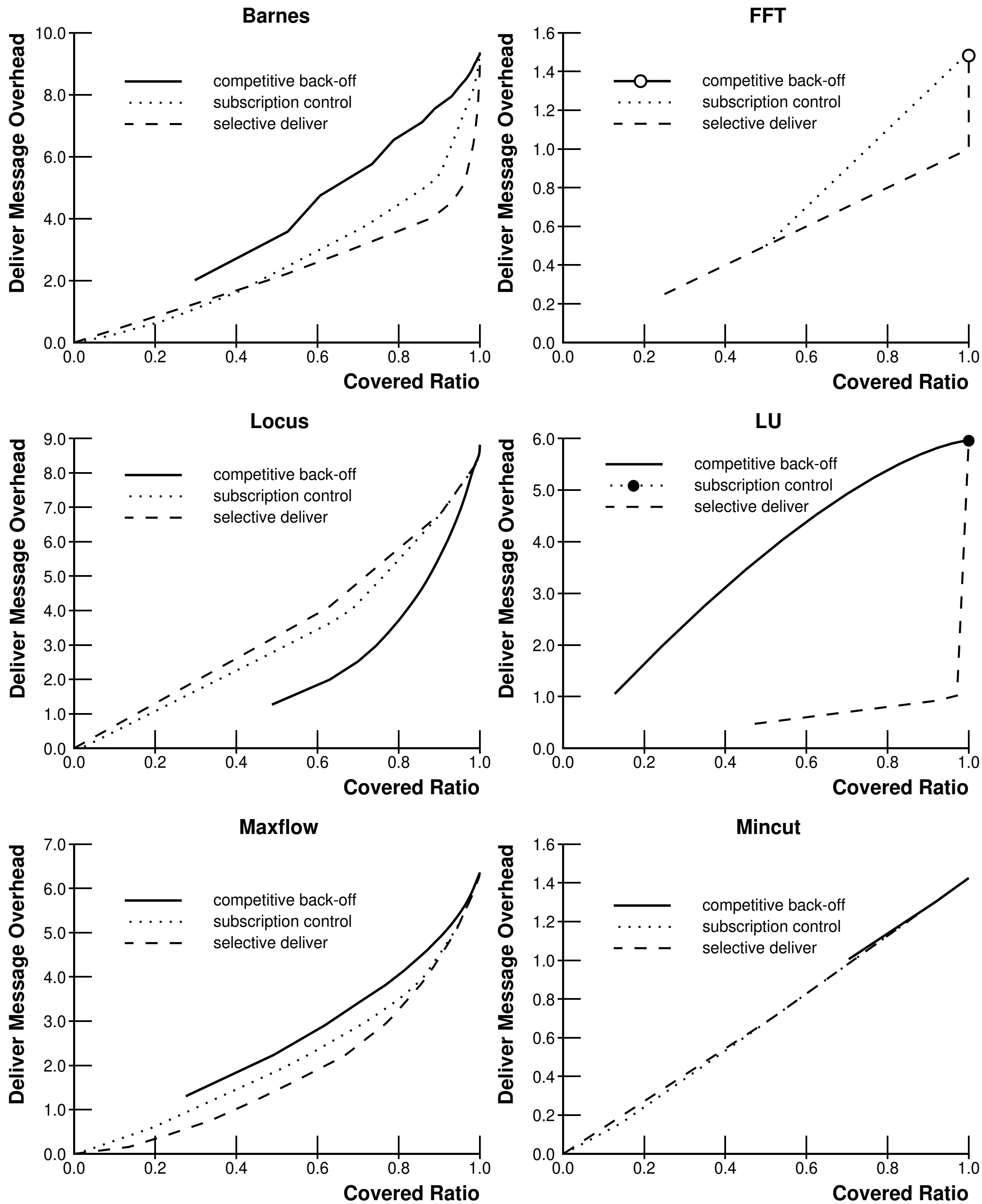

Figure 4-11: Deliver Message Overhead versus Covered Ratio.

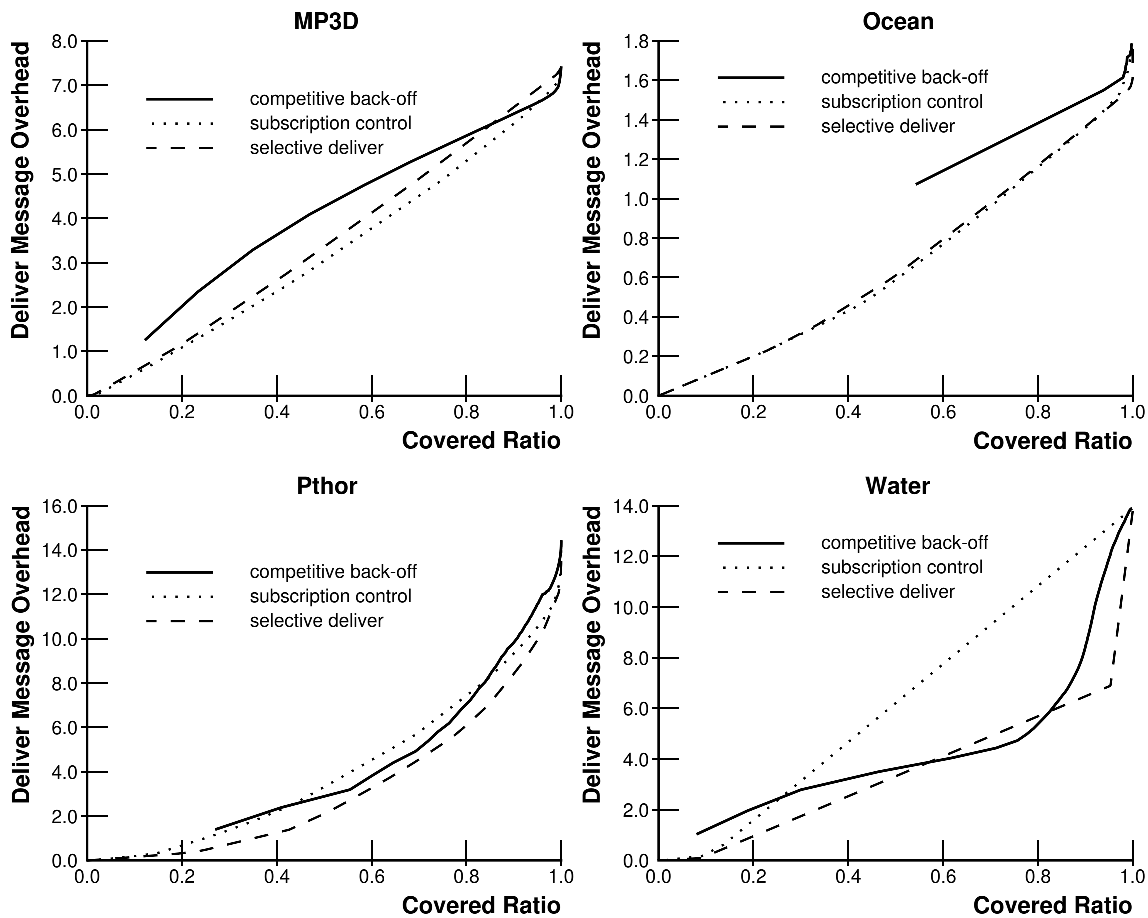

Figure 4-12: Deliver Message Overhead versus Covered Ratio. (Continued.)

updates a shared variable, perhaps another trade-off technique is more effective than selective deliver. Thus, to determine the best trade-off technique, we need to understand details of the memory-access behavior of each application.

4.1.4.1 Barnes

The main data structure is a Barnes-Hut tree that represents a three-dimensional space. The computation of each time step consists of four phases: tree construction, tree partition, tree traverse, and body update. For each time step, the program partitions the physical space into subspaces. The program assigns each processor to a similar subspace to which the processor was assigned in the previous time step, so that the processor exploits the memory locality for accessing body records. Cell records, however, are dynamically allocated when processors construct a tree from scratch for each time step, and the same cell record may not be used for the tree node that represents the same physical space. Therefore, even if a processor accesses the same physical space that the processor had accessed in the previous time step, the processor may access a different cell record. Thus, the deliver vector of the cell record does not represent the future access pattern.

Another characteristic of Barnes is that the access pattern of body and cell records varies significantly among those four phases. First, the tree construction updates cell records to generate single-processor reuse accesses. At the end of the tree construction phase, however, processors update the center of mass for each cell, and the update will be read by other processors in later phases. Second, the tree partition phase reads cell and body records but does not write them. Third, the tree traverse phase updates a part of the body records (e.g., acceleration and gravity potential) but does not update cell records. Fourth, the body update phase updates a part of the body records (e.g., position). Therefore, competitive back-off causes some overhead when the scheme dynamically adapts to the changing access pattern. Our software-controlled techniques (selective deliver and subscription control), on the other hand, do not cause such overhead because these techniques statically choose the places to use deliver operations by using the knowledge of the access pattern. As shown in Figure 4-11, both trade-off curves of the two software-controlled techniques have a knee when the covered ratio is about 90%. The right hand side of the knee exhibits the behavior of the deliver operation due to single-processor reuse for the tree construction. We can insert deliver or subscription control operations so that the deliver message overhead is in the left hand side of the knee. That is, for either software-controlled technique, we can avoid deliver operations during the tree construction phase. Competitive back-off, on the other hand, cannot statically avoid using deliver operations for the phase, so that competitive back-off generates more deliver message overhead than the other two techniques for the same covered ratio, as shown in Figure 4-11.

Figure 4-11 also shows that selective deliver and subscription control cause slightly different trade-off curves. Actually, selective deliver is better for cell records than subscription control, but subscription control is better than selective deliver for body records. For the cell record, single-processor reuse frequently occurs for some write instructions that update cell records (e.g., cell initialization and tree construction), and selective deliver can reduce unused deliver messages for these writes more efficiently than subscription control.

For the body record, on the other hand, subscription control performs better than selective deliver because of false sharing. Since each element in the body record has a different sharing pattern, the deliver operation and the deliver vector should ideally be defined for each element. In selective deliver, the producer sends a deliver message to all processors that have used any part of the cache line since the unit of the deliver vector is a cache line. Subscription control, however, allows the consumer to unsubscribe a cache line when the consumer accesses a particular record element. Thus, subscription control can effectively use deliver operations only for record elements for which deliver operations work efficiently. By applying selective deliver to the cell record and subscription control to the body record, we can generate less deliver message overhead than either scheme. (At a 70% covered rate, the deliver message overhead is 2.7 for the combined technique, 3.1 for selective deliver, and 3.6 for subscription control).

4.1.4.2 FFT

In FFT, communication occurs during a transpose of a $\sqrt{n} \times \sqrt{n}$ complex matrix to another matrix. This program uses a six-step FFT algorithm, and three steps out of six perform a transpose between two matrices. As shown in Figure 4-9, the length of all sync-read deliver-runs is one since the producer-consumer relationship is static and one-to-one except for the last transpose. Out of the three transpose operations, the first two transpose operations cause sync-read deliver-runs but the last transpose causes open deliver-runs because the final result of FFT obtained at the last transpose is not used. This is why the deliver message overhead becomes 1.5 when the covered rate is 1.0, as shown in Figure 4-11. Selective deliver can eliminate those open deliver-runs by avoiding deliver operations immediately before the last transpose. Thus, selective deliver can achieve the optimal result; the deliver message overhead is 1.0 when the covered rate is 1.0.

Subscription control, on the other hand, does not achieve the optimal result as shown in Figure 4-11. This is because all transpose steps use the same code so that this technique cannot unsubscribe deliver operations selectively during the last transpose. Simple source-code modifications (e.g., subroutine expansion), however, can eliminate this problem. Competitive back-off cannot eliminate open deliver-runs that occur in the last transpose since the length of those deliver-runs is one and the same as other deliver-runs.

4.1.4.3 Locus

For Locus, most of the deliver messages occur because of updates of CostArray elements. Each processor picks up a wire from one of the task queues and removes the evidence of the wire from CostArray if the wire was routed previously. The processor then scans CostArray for all candidate routes for the wire and updates CostArray for the chosen route. After that, the processor tries to pick up another wire from the same task queue, but if the queue is empty, the processor moves to another task queue. We use a geographical partitioning so that each task queue only contains wires that start from the same geographical partition. Thus, if a processor moves from one task queue to another, the processor will probably not access the CostArray elements for wires in the previous queue soon.

As shown in Figure 4-11, competitive back-off reduces the number of deliver messages more than the other two techniques. Figure 4-9 explains this behavior. The distribution of the sync-read deliver-run length indicates that the ratio of short deliver runs is relatively large (the length of about 50% of deliver runs is only one) and that the length of deliver runs dynamically changes for a wide range (from one to more than 50). Thus, competitive deliver can exploit only short deliver runs and cancel the subscription of deliver messages for long deliver runs. The distribution of the effective deliver-run length, on the other hand, indicates that the effective deliver-run length does not differ dramatically among read instructions. That is, short and long deliver runs occur at the same static instruction, and thus, the static technique cannot exploit only short deliver runs. The distribution of the deliver ratio in Figure 4-9 also indicates that the deliver ratio does not differ dramatically among write instructions. Thus, selective deliver is less effective than competitive back-off for Locus.

In selective deliver, unless the program is substantially modified, the producer does not know if other processors have moved out from the queue that the producer is working on. Selective deliver thus generates more deliver messages than the other two techniques. In subscription control, similarly, the consumer does not know which CostArray elements the consumer will be unlikely to access soon. Thus, the consumer cannot effectively unsubscribe those elements, and subscription control generates about the same number of deliver messages that selective deliver generates. There is, however, a small chance that a processor moves out from a task queue every time after the processor finishes routing a wire. Thus, if a processor unsubscribes those CostArray elements, the processor can avoid receiving deliver messages for those elements when the processor moves out from the current queue. This is why subscription control generates slightly less deliver message overhead than selective deliver. The number of reduced deliver messages due to subscription control, however, is limited because the processor does not unsubscribe all of the unnecessary CostArray elements.

4.1.4.4 LU

For LU, communication occurs in two cases for the factorization of a blocked dense matrix, $A_{ij} (0 \le i, j < n)$. First, after a diagonal block A_{kk} is factorized, perimeter blocks in the subsequent row and column, A_{ik} and $A_{kj} (k < i, j < n)$, are updated by using the diagonal block A_{kk}. Second, after those perimeter blocks are updated, interior blocks $A_{ij} (k < i, j < n)$ are updated by using those perimeter blocks. Thus, we can insert deliver operations for the following three computations that update a matrix block: the diagonal block factorization, the perimeter block update, and the interior block update. During the first two computations, the producer-consumer relation is static, and consumers use all deliver messages. That is, the deliver ratio is one. During the last computation, however, the producer becomes the next consumer (i.e., single-processor reuse), and none of delivered messages are used. That is why the deliver message overhead is large in LU (6.0 when the covered rate is 1.0) if deliver messages are used for all computations that update a matrix block.

Selective deliver can reduce the number of deliver messages substantially by avoiding deliver operations for the interior block update where single-processor reuse occurs. The interior block update, however, shares some subroutines with the other two computations (i.e., the diagonal block factorization and the perimeter block update). If selective deliver avoids deliver operations in such shared subroutines, both the deliver message overhead and the covered ratio decrease (0.47 deliver message overhead at 0.47 covered ratio). I modified the source code so that the interior block update does not share subroutines with the other two computations. As a result, selective deliver can eliminate most of the misses with very small overhead; the deliver message overhead is 1.0 when the covered ratio is 0.97.

Subscription control, on the other hand, reduces the number of deliver messages less successfully than selective deliver. Because all deliver-runs start at the same place in the initialization code, subscription control cannot unsubscribe only deliver messages with long deliver-runs. To allow subscription control to unsubscribe long deliver-runs, we need to add code that selects long deliver-runs during the initialization. We assume that subscription control performs the subscribe and unsubscribe operations immediately after an access to the cache line. If we use a more general subscription control technique that can perform such operations at arbitrary places, the technique could reduce the deliver message overhead more effectively. Competitive back-off does not reduce the number of deliver messages as effectively as selective deliver because deliver-runs are moderately long (6.0 in the average) as shown in Figure 4-9.

4.1.4.5 Maxflow

Maxflow traverses the input graph three times to find the maximum flow in the graph. The first step labels each node with the minimum distance from the source node. The second step pushes flow excesses towards the sink node. The third step pushes remaining flow excesses backwards towards the source node. There is one task queue for each processor and one global task queue. Each task queue contains graph nodes to be computed. Each processor picks up a node from its local queue, unless the queue is empty, and updates the node and its neighbor nodes. Then, the processor enqueues uncompleted neighbor nodes either to its local queue, if no other processor is idle, or to the global queue. Unlike Barnes, graph nodes do not have affinity to processors and are accessed by a different set of processors for each of the three graph traversals in the program. Thus, the past access pattern stored in the deliver vector does not represent the future access pattern, and the deliver message overhead is large (6.4 when the covered rate is 1.0).

In Maxflow, selective deliver generates less deliver message overhead than the other two techniques for the entire range of the covered rate as shown in Figure 4-11. This is because selective deliver can utilize an access pattern that causes efficient deliver operations. When a node pushes out all flow excesses to neighbor nodes, the node becomes an inactive node that is read but not updated until a neighbor node pushes back some flow. Thus, inactive nodes are only read, while active nodes that have flow excesses are read and written. Therefore, the last deliver messages for inactive nodes are more likely to be used than the deliver messages for active nodes. Because the producer determines the node state that affects the behavior of deliver messages, the producer can control the deliver traffic more efficiently than the consumer.

4.1.4.6 Mincut

In Mincut, communication occurs through two global variables and through graph node records. As discussed previously, each deliver message is sent to most of the rest of the processors (14.9 processors in average), and most of those processors use the message (10.4 processors in average). That is, processors use about 70% of received messages. This access pattern occurs because the communication pattern is broadcasting. With this access pattern, it is unlikely that Mincut can ever scale to significant number of processors. This access pattern does not occur in the other applications that we have examined.

Figure 4-11 shows that the trade-off curves for the three techniques have almost the same slope and are almost linear. This result indicates several characteristics in the access pattern: (1) All candidate places for deliver or unsubscribe operations have about the same deliver ratio or effective deliver-run length, respectively (Figure 4-9). Thus, the trade-off curve is linear for selective deliver and subscription control; (2) competitive back-off has, in general, an advantage in that it can cut long deliver-runs at the threshold; however, competitive back-off has a disadvantage in that it generates unused deliver messages up to the threshold for long deliver-runs. Nevertheless, the advantage and disadvantage are insignificant in Mincut because most of the deliver-runs are very short; (3) neither the producer nor the consumer has more opportunities for selecting places for deliver or unsubscribe operations than the other. Because processors pick a graph node randomly, neither the consumer nor the producer can predict future access patterns; the consumer does not know if the consumer will access the node soon or not, and the producer does not know if the access to the node will be likely to cause single-processor reuse or not. Therefore, both selective deliver and subscription control result in about the same trade-off curve.

4.1.4.7 MP3D

For MP3D, most of the coherency traffic occurs for updates of cell records. When a processor calculates a movement of a particle, the processor updates the cell record that contains the particle. The access pattern to the cell record is typically a migratory access. Thus, while each deliver operation sends a deliver message to all processors that have touched the record, only one of them usually uses the deliver message. This is why the deliver message overhead is large in MP3D (7.4 when the covered ratio is 1.0). The trade-off curve for selective deliver is almost linear. This is because most of the deliver operations generate about the same number of messages (about seven) to eliminate one read miss, as shown in Figure 4-10. Subscription control, on the other hand, generates slightly less deliver message overhead than selective deliver for a wide range of the covered ratio as shown in Figure 4-12. This is because each processor usually has a single particle in a cell. If the particle moves out from the cell, the processor is unlikely to access the cell record again soon. Thus, long deliver-runs occur for such cells. Because subscription control can unsubscribe those cells, subscription control performs slightly better than selective deliver.

4.1.4.8 Ocean

Ocean uses several grids that represent a physical value on a two-dimensional plane. Each grid is partitioned into square subgrids and allocated to a processor. Communication occurs when processors access another processor's subgrid to perform nearest-neighbor operations (i.e., Laplacian, Jacobian, and Multigrid Solver). Although the communication pattern is regular in Ocean, simple deliver annotations produce some unused deliver messages (1.8 delivered messages per used message). Most of the unused deliver messages occur because the access pattern of nearest-neighbor grid elements differs among computations in the multigrid solver. For example, processors usually access nearest neighbors to calculate each grid element ($A_{i+1,j}$, $A_{i-1,j}$, $A_{i,j+1}$, and $A_{i,j-1}$ for the computation of $A_{i,j}$). Processors, however, access some of surrounding elements of those nearest neighbors when processors calculate grid elements at a coarser level to sweep another hierarchy of grids. When this access occurs, the consumer automatically subscribes those surrounding elements for subsequent deliver operations and receives deliver messages when a producer updates those elements. The consumer, however, does not use those grid elements during other computations (e.g., relaxation operation). Therefore, unused deliver messages occur in the multigrid solver.

Selective deliver can reduce the number of unused deliver messages by avoiding deliver annotations at source-code lines where single-processor reuse may occur. Those source-code lines, however, also generate used deliver messages. Thus, removing deliver annotations from those lines also reduces the covered ratio. Because computations in Ocean are regular, however, it is relatively easy to modify the source code so that deliver operations are applied only to nearest-neighbor elements that cause communication. Such modifications improve the efficiency of deliver operations over the simple selective deliver technique.

Figure 4-12 shows that the trade-off curve of subscription control is about the same as that of selective deliver. This indicates that neither the producer nor the consumer has an advantage in controlling unused deliver messages. Competitive back-off, on the other hand, is less effective than the other two techniques. This is because the communication pattern for each computation does not change significantly at run-time. Thus, the static information that selective deliver and subscription control use is more useful for reducing the number of unused deliver messages than the dynamic information that competitive back-off uses.

4.1.4.9 Pthor

The main computation in Pthor involves picking a circuit-element record from one of distributed task queues and evaluating that element. If the new output of the element affects the input of other elements connected to the element, those elements are inserted in a task queue. When all task queues become empty, a deadlock occurs. The program resolves the deadlock and the simulation proceeds again. As discussed previously, processors move among task queues to find a circuit-element record to be evaluated. As processors move from one task queue to another, processors generate migratory accesses that cause a large number of unused deliver messages. Namely, about 50% of read misses occur for task queue records that are usually accessed in a migratory way. When a task queue is empty, however, the access pattern is not migratory. This is because processors read but do not write the task queue until another processor enqueues an element record to the queue.

Selective deliver performs slightly better than subscription control as shown in Figure 4-12. This is because selective deliver can avoid deliver operations on the task queue records when the access pattern is migratory (i.e., when the task queue is not empty). Competitive back-off, on the other hand, generates about the same mount of deliver message overhead as subscription control since competitive back-off has both an advantage and a disadvantage over subscription control. Similar to Locus, the access pattern for data records associated with each queue changes dynamically as processors move among task queues. Such a changing access pattern is an advantage for competitive back-off since competitive back-off is adaptable to such a pattern. Unlike Locus, however, the bulk of deliver-runs are very long in Pthor because migratory accesses occur more often in Pthor than Locus. Such distribution of deliver-runs is a disadvantage for competitive back-off because the back-off threshold has to be long enough for a significant reduction of read misses and competitive back-off generates unused deliver messages up to the threshold for deliver-runs longer than the threshold.[6] As a result, competitive back-off and subscription control have about the same level of the efficiency.

4.1.4.10 Water

For Water, molecules are statically partitioned and assigned to each processor, as discussed previously. We call a molecule that is assigned to the local processor a *local molecule* and a molecule that is assigned to another processor a *remote molecule*. Most of the deliver operations occur during a inter-molecule force calculation, which computes the inter-molecule force for all pairs of N molecules. For each local molecule, a processor computes the inter-molecule force with about $N/2$ molecules. This computation ends up with a deliver vector of each molecule that holds the identity of 7 to 9 processors. Thus, a deliver operation sends a molecule record to those 7 to 9 processors. However, usually only one of them uses the delivered record. This access pattern causes a sharp knee in the distribution curve of the deliver ratio when the threshold is about 8, as shown in Figure 4-10.

6. Refer EQ 4-4 on page 48 and EQ 4-5 on page 49.

The inter-molecule force computation is partitioned so that each molecule is accessed by the local processor much more often than by any other processors. Therefore, the access pattern for local molecules is likely single-processor reuse, and deliver operations for local molecules generate deliver messages that no destination processor uses. In fact, our simulation results show that the deliver ratio — the number of delivered messages per used message — is 147 for local molecules and 8 for remote molecules! The access pattern for remote molecules, on the other hand, is very likely a migratory access since the next user of a remote molecule usually differs from the current user. Consequently, selective deliver can significantly reduce deliver message overhead without significantly reducing the covered ratio by using deliver operations only for remote molecules. The effectiveness of selective deliver appears as a sharp knee of the slope of the trade-off curve when the covered ratio is about 96% in Figure 4-12. The right hand side of the knee represents the behavior of deliver operations due to single-processor reuse for local molecules.

Subscription control, on the other hand, reduces the deliver message overhead less successfully than selective deliver. Since processors do not access remote molecules often, accesses to remote molecules often cause long deliver-runs. The distribution of the effective deliver-run length in Figure 4-10 exhibits a sharp knee when the length is about 15. This indicates that processors receive about 15 deliver messages in average between two consecutive accesses for a remote molecule. Remote molecules cause most of sharing misses (86%). Thus, if consumers unsubscribe deliver messages for remote molecules, subscription control significantly reduces the deliver message overhead as well as the covered ratio. Therefore, subscription control generates more deliver messages than selective deliver for the same covered ratio. The distribution of the sync-read deliver-run length for remote molecules, however, has a large deviation, as shown in Figure 4-10. While subscription control cannot dynamically unsubscribe only deliver messages with long deliver-runs, competitive back-off can. Therefore, as shown in Figure 4-12, competitive back-off can reduce the deliver message overhead as effectively as the selective deliver scheme for a wide range of covered ratio.

4.1.5 Summary

In Section 4.1, we have discussed the behavior of deliver operations from the consumer and the producer perspectives by analyzing the simulation results of ten parallel applications. While the statistics of deliver-runs indicate the sharing pattern from the consumer perspective, the statistics of delivered and used messages indicate the sharing pattern from the producer perspective. We identified two common access patterns — single-processor reuse and reader migration — that generate unused deliver messages.

The detailed statistics about sharing patterns allow us to evaluate techniques that trade off the deliver message overhead for the covered ratio. We discussed three techniques: competitive back-off (consumer-initiated hardware-controlled), subscription control (consumer-initiated software-controlled), and selective deliver (producer-initiated software-controlled). For nine out of the ten applications that we examined, selective deliver performs best or nearly best. Only for Locus does competitive back-off significantly outperform the other two techniques. We discussed how the access pattern and the program structure affects the performance of these techniques in Section 4.1.4. Table 4-3 summarizes the effect of the trade-off techniques. The table shows the number of delivered messages per used message (deliver ratio) for two covered ratios (50% and 100%) for the best technique among the three techniques. For the 100% covered ratio, all the three techniques generate the same deliver ratio for most of our applications because the threshold for the trade-off is basically infinite for either technique. For the 50% covered ratio, the best technique is shown in table. When the covered ratio is 100%, the deliver operation generates more than six deliver messages to eliminate one cache miss for seven applications. When the covered ratio is 50%, on the other hand, the trade-off technique significantly reduces the ratio between the number of delivered messages and the number of used messages — more than a factor of two for those seven applications with the exception of MP3D. The ratio, however, is smaller than two only for four applications: LU, FFT, Ocean, and Mincut. These applications have a static communication pattern except for Mincut, which has a dynamic broadcasting pattern. For the rest of the applications, although our trade-off techniques can significantly reduce the number of unused deliver messages, deliver operations still generate substantial traffic overhead.

application	$\frac{\text{delivered messages}}{\text{used messages}}$		
	at 100% covered ratio	at 50% covered ratio[a]	
Barnes	8.7	4.2	selective deliver
FFT	1.0	1.0	selective deliver and subscription control
Locus	8.8	2.6	competitive backoff
LU	6.0	1.0	selective deliver
Maxflow	6.3	3.0	selective deliver
Mincut	1.4	1.4	selective deliver and subscription control
MP3D	7.4	6.0	subscription control
Ocean	1.8	1.2	selective deliver and subscription control
Pthor	14.4	4.3	selective deliver
Water	13.9	6.5	selective deliver

Table 4-3: The Number of Delivered Messages per Used Message.

a. The trade-off technique that produces the lowest deliver message overhead is shown in the table.

4.2 Consumer-oriented Prefetch

This section discusses the behavior of prefetch operations and compares the prefetch and the deliver schemes. Our simulation results indicate that the prefetch scheme can significantly reduce the number of read misses while it can generate a large number of unnecessary prefetch operations. We will show, however, that it is simple for the prefetch scheme to prevent most of the unnecessary operations from generating traffic while it is difficult for the deliver scheme to do so.

In Section 4.1, we assumed that processors perform deliver operations at synchronization or pseudo-synchronization operations, so the simulator automatically inserts deliver operations in application programs. This is a reasonable assumption for the analysis of the deliver scheme, because communication typically occurs across synchronization points and because the producer generally can identify the addresses to deliver at synchronization points. For prefetch operations, however, synchronization or pseudo-synchronization operations are not a practical place to insert prefetch operations. This is because at synchronization (e.g., acquire) points, the consumer generally cannot identify the addresses to prefetch without causing significant instruction overhead. The places where we should insert prefetch operations depend significantly on the data and the program structures of the application. Therefore, for evaluating prefetch operations, we use a version of application programs in which prefetch operations are inserted manually.

I used several techniques that Mowry [45] has proposed and evaluated for automated insertion of prefetch operations. The techniques include software pipelining and loop splitting. Software pipelining allows us to schedule prefetch operations for a future iteration of a loop in the current iteration. Loop splitting is used to eliminate predicate statements that select loop iterations that should perform prefetch operations. When the Mowry's algorithm is not applicable, I simply extend the algorithm to insert prefetch operations by hand. For example, I used subroutine inlining to eliminate some predicate statements that select computations that should perform prefetch operations. I used a profiling system that identifies the number of cache misses for each source-code line. We will discuss details about our prefetching strategy for each application in Subsection 4.2.3.

For comparing the simulation results of the deliver and the prefetch schemes, we use the same memory model as the one used for the deliver scheme (an infinite cache with 16-byte lines and a unit-delay memory). Because of this model, we ignore the effect of the timing of prefetch operations which will be discussed in Subsection 5.1.2.

In the following subsections, we first discuss the covered ratio and the efficiency of prefetch operations and identify important differences between the prefetch and the deliver schemes. Second, we discuss a technique similar to selective deliver for improving the efficiency of prefetch operations, and we compare the efficiency of deliver and prefetch operations. Third, we discuss our strategy for inserting prefetch operations in the source code and detailed characteristics of our applications that affect the behavior of prefetch operations. Last, we summarize our discussions about the prefetch scheme.

4.2.1 Coverage and Efficiency of Prefetch Operations

While I manually inserted prefetch operations to eliminate the bulk of read misses, some read misses still remain. Figure 4-13 shows the ratio of eliminated and remaining read misses normalized by the number of read misses without prefetch operations. The *covered* fraction indicates the ratio of read misses that are eliminated by hand-inserted prefetch operations. The remaining read misses are divided into two types: invalidated and not-prefetched. The *invalidated* fraction indicates the ratio of remaining read misses for cache lines that are prefetched but invalidated before they are used.[7] These types of misses occur because processors conflict each other for accessing the same shared line. The *not-prefetched* fraction indicates the ratio of remaining read misses for cache lines for which prefetch operations are not inserted.

I inserted prefetch operations by using a profiling system so that the ratio of covered misses (we simply call it covered ratio) is approximately 80% or larger for most of the applications. For LU and FFT, Figure 4-13 shows that the covered ratio is 100%. This is because the memory access pattern is regular in the two applications so that it is easy to insert prefetch operations to eliminate virtually all misses. In real machines, however, the covered ratio may be smaller than the ratio shown in Figure 4-13 for some applications. This is because, for some applications, prefetch operations do not bring cache lines early

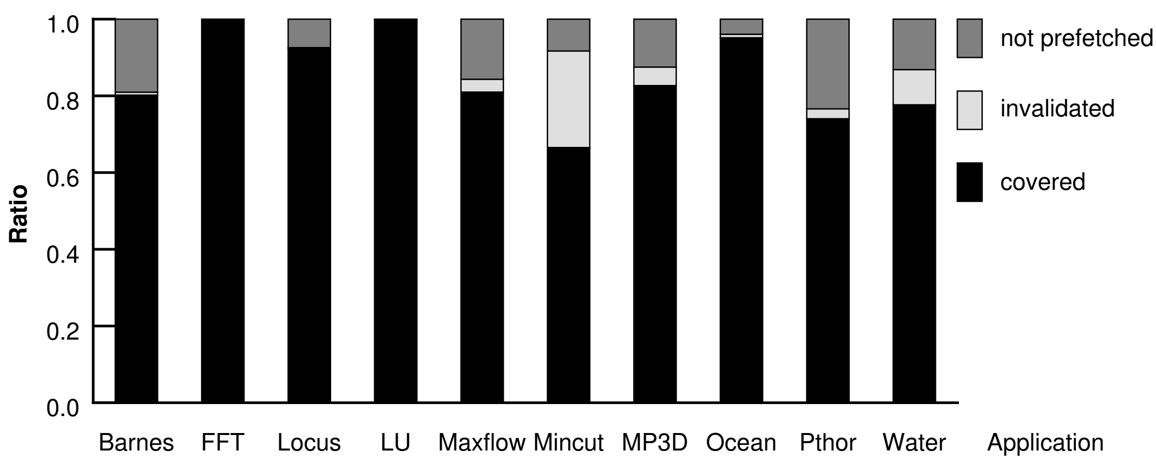

Figure 4-13: Coverage of Prefetch Operations.

7. The *invalidated* type includes cache misses for cache lines that were valid when the processor performed a prefetch operation but have been invalidated before they are used.

enough to eliminate a read miss when memory latencies are very large. We will examine the effect of large memory latencies in Section 5.4. The ratio of the invalidated misses is generally small, except for Mincut. In Mincut, the shared data structures are not well partitioned so that conflicting accesses interfere with prefetch operations.

Now, we examine the efficiency of prefetch operations. As shown in Figure 4-14, we divide prefetch operations into three types: used, not-used, and not-issued. The *used* category indicates the ratio of prefetch operations that transfer a cache line that the processor actually uses. The *not-used* category indicates the ratio of prefetch operations that transfer a cache line that the processor does not use. This type of prefetch operation occurs, for example, if another processor invalidates the prefetched line because of a conflicting access, or if a processor speculatively prefetches an unnecessary line. The *not-issued* category indicates the ratio of prefetch operations that are not issued to the network because the cache line state is valid (or owned for prefetch-exclusive operations) in the processor cache when the processor performs the prefetch operation.

Not-used prefetches cause network traffic overhead as well as instruction overhead, but the ratio of not-used prefetches is relatively small (< 22%) for all applications that we examined. Not-issued prefetches, on the other hand, cause only instruction overhead but no network traffic overhead. Figure 4-14 shows that most of the unused prefetch operations are not-issued prefetches. This indicates that the processor cache can filter out most of the unused prefetch operations by checking the processor cache. This is the *most* signif-

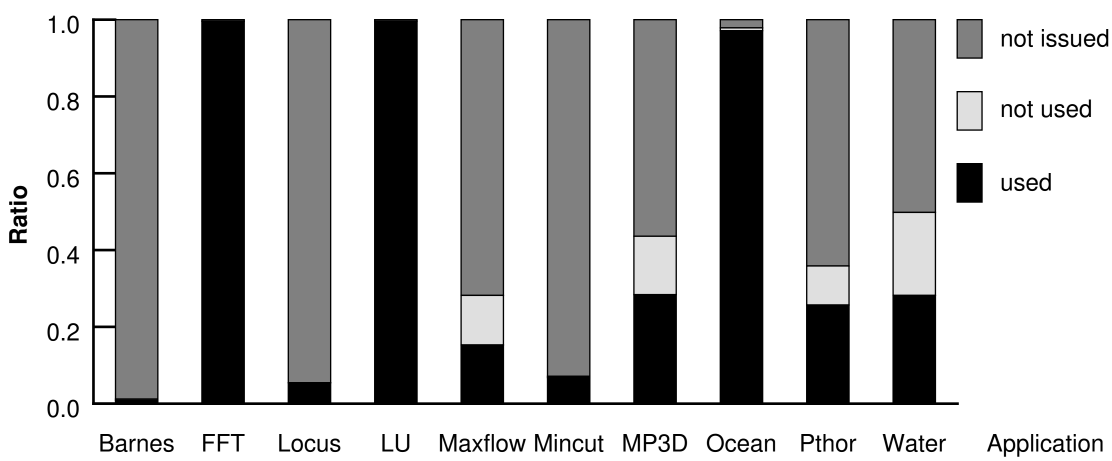

Figure 4-14: Efficiency of Prefetch Operations.

icant advantage of the prefetch scheme over the deliver scheme. For example, in Section 4.1.3, we have discussed that single-processor reuse generates useless deliver messages to the network. In our prefetch scheme, single-processor reuse does not generate unnecessary prefetch requests to the network. Since the same processor is using the same data, the cache line for the data is valid in the processor cache so that the processor cache prevents prefetch operations from sending prefetch requests to the network. It is therefore very important for prefetch schemes to implement a filtering function that eliminates unnecessary prefetch requests for valid cache lines.

The ratio of not-issued prefetch operations is generally large, more than 50%, except for LU, FFT, and Ocean, which perform regular computations.[8] This indicates that prefetch operations are often unnecessary because the prefetch target line is valid in the processor cache. Based on my experience of inserting prefetch operations, I have found that it is challenging to identify such unnecessary prefetch operations statically unless the memory access pattern is regular. Most of our applications, in fact, use irregular data structures (such as linked lists) or access regular data structures in a irregular way, so that a large number of unnecessary prefetch operations occur. Not-issued prefetch operations, however, can be filtered out by using the processor cache.

4.2.2 Efficiency Comparison between Prefetch and Deliver Operations

In Subsection 4.1.3, we discussed a selective deliver technique which uses profiling information for a trade-off between the number of misses and the number of deliver messages. In this subsection, we consider a similar technique for prefetch operations and compare the efficiency of prefetch and deliver operations.

We use a simulator that counts the number of executions and the number of eliminated misses for each prefetch operation in the source code. This information leads us to a trade-

8. The ratio of unused prefetch operations is generally higher than the one that Mowry has shown in [45] because of two reasons. First, we assume an infinite cache while Mowry assumed a finite cache. When the cache is finite, a memory-access instruction that causes sharing misses for an infinite cache may also cause capacity misses. Thus, the ratio of unused prefetch operations for an infinite cache is higher than one for finite caches. Second, we inserted more prefetch operations than Mowry [45] did, and the ratio of unused prefetch operations is relatively high for some prefetch operations in our annotation. We discuss a trade-off between the covered ratio and the efficiency of prefetch operations in the next subsection.

off technique, *selective prefetch*, which is similar to selective deliver. For each prefetch operation in the source code, we define a *prefetch ratio*, which is the ratio between the number of executions of the prefetch operation and the number of eliminated misses due to the prefetch operation. The prefetch ratio corresponds to the deliver ratio, which is defined at EQ 4-6 on page 55. Selective prefetch uses a prefetch operation only if the prefetch ratio is equal to or smaller than a certain threshold. As the threshold increases, the number of misses decreases while the number of prefetch operations per eliminated miss increases. Thus, this technique allows us to trade off the number of unnecessary prefetch operations for the number of eliminated misses.

Figure 4-15 and Figure 4-16 show the trade-off curve of selective prefetch and selective deliver. The solid lines indicate the trade-off between the number of prefetch operations and the number of eliminated misses for selective prefetch, and the dotted lines indicate the trade-off between the number of deliver messages and the number of eliminated misses for selective deliver.[9] Both X and Y axes are normalized by the number of read misses for the invalidate-only protocol.

For applications with a regular communication pattern (i.e., FFT, LU, and Ocean), both prefetch and deliver schemes can eliminate most of the read misses with either no or small overhead, as shown in Figure 4-15 and Figure 4-16. For the prefetch scheme, one prefetch operation generally eliminates one read miss in these applications, so that the normalized number of prefetch operations is about one when the normalized number of eliminated misses is about one. For the deliver scheme, as we discussed in Subsection 4.1.4, selective deliver can eliminate most of the deliver operations that generate unnecessary deliver messages so that selective deliver can reduce the miss rate without significant traffic overhead. The efficiency of prefetch and deliver operations is high for these applications because the communication pattern is generally predictable for consumers and producers. Thus, we can insert prefetch or deliver operations only where communication frequently occurs.

For applications with an irregular communication pattern, on the other hand, the efficiency is generally very different between the prefetch and the deliver schemes, as shown in Fig-

9. These graphs are similar to ones in Figure 4-11 and Figure 4-12 but use a different scale.

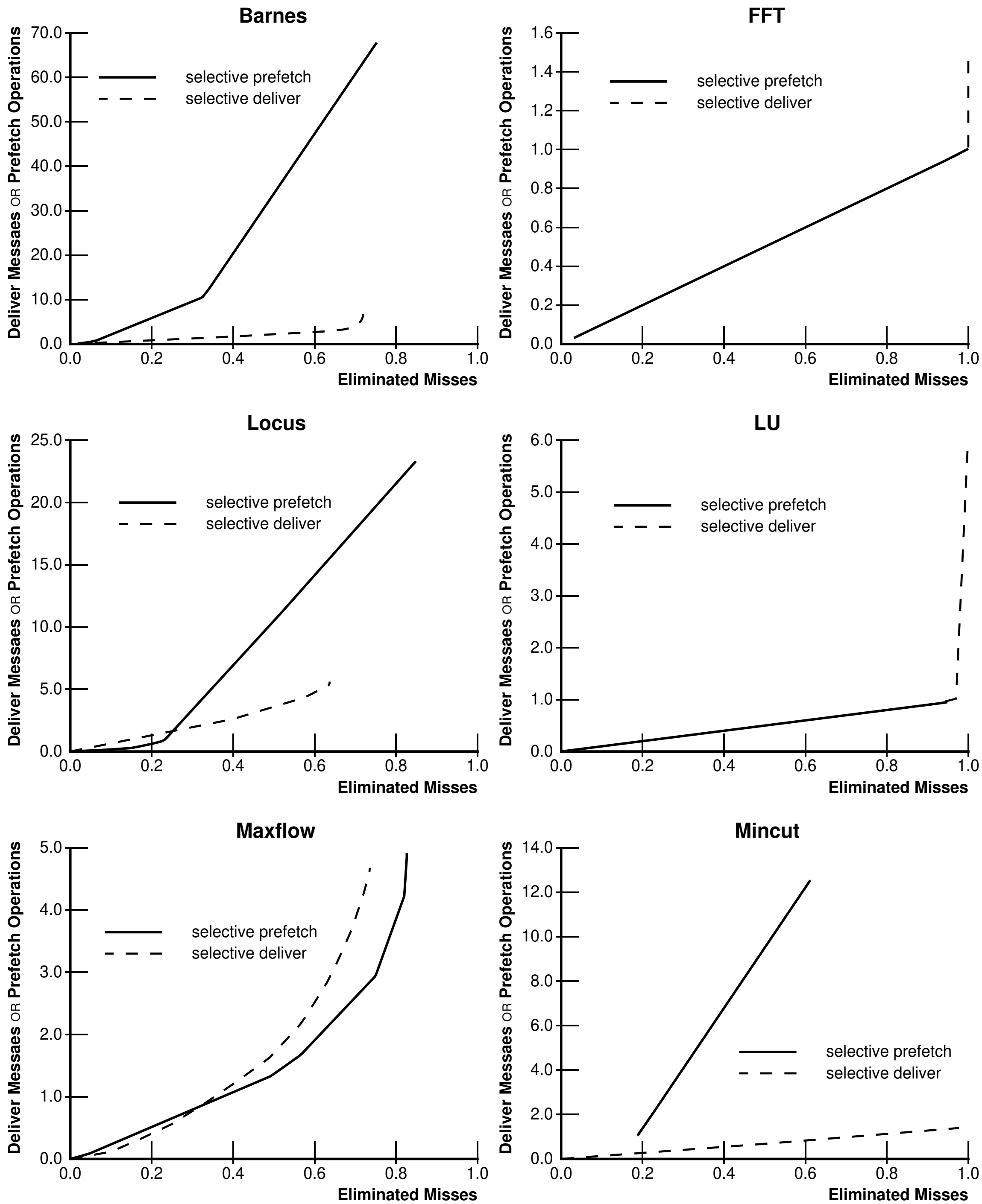

Figure 4-15: Efficiency of Selective Prefetch and Selective Deliver.

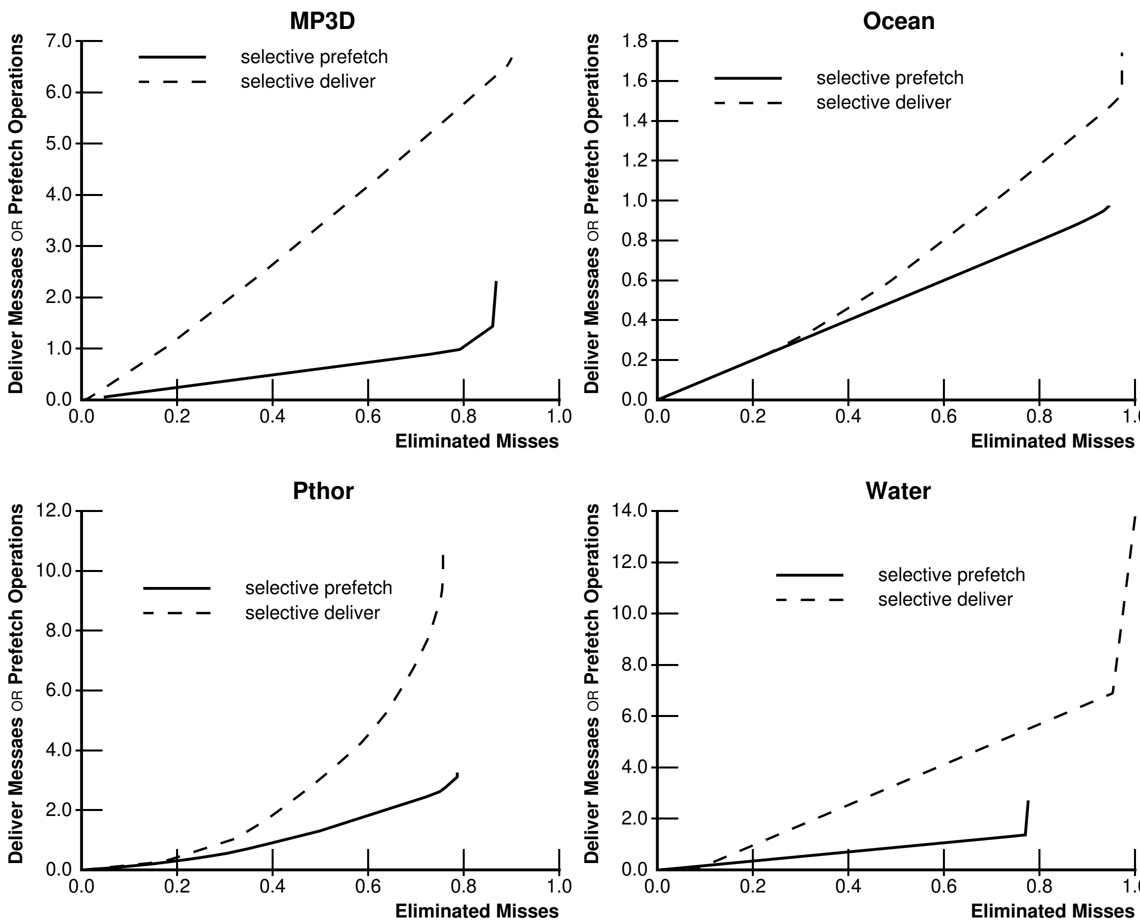

Figure 4-16: Efficiency of Selective Prefetch and Selective Deliver. (Continued.)

ure 4-15 and Figure 4-16. For Barnes, Locus, and Mincut, the number of prefetch operations is usually much larger than the number of deliver messages when the two schemes eliminate the same number of misses. This is because it is difficult to identify memory accesses for cache lines that are already fetched, so that processors frequently perform prefetch operations for cache lines that exist in the cache. For these applications, although it is relatively easy to statically identify memory access instructions that are responsible for most the cache misses in the code, those instructions also cause a large number of cache hits at run-time. Because of the irregular access pattern, it is difficult to separate the instances of those memory accesses that cause a cache miss from those that cause a cache hit. Thus, selective prefetch does not significantly improve the efficiency of prefetch operations for these applications. When the cache is smaller than the data-set size, the efficiency of prefetch operations increases because prefetch operations eliminate capacity misses that do not occur for an infinite cache.

For Maxflow, MP3D, Pthor, and Water, the deliver scheme usually generates more overhead than the prefetch scheme. For these applications, it is relatively easy to statically identify memory access instructions that are responsible for most of the cache misses in the code, and those instructions do not frequently cause a cache hit. Since selective prefetch can insert prefetch operations only for those memory accesses, the efficiency of prefetch operations is relatively high. For MP3D and Water, the curve for selective prefetch has a sharp knee as shown in Figure 4-15. The knee indicates that the efficiency differs significantly among different prefetch operations, and thus, selective prefetch can avoid inserting prefetch operations that do not significantly increase the hit rate. For the deliver scheme, selective deliver does not significantly improve the efficiency of deliver operations because of reader migrations, as we discussed in Subsection 4.1.4.

4.2.3 Application Algorithms and Prefetch Operation Behavior

The behavior of prefetch operations depends significantly on two factors: the prefetch insertion strategy and the program structure. In this subsection, we examine details about the two factors for each application to understand the efficiency of prefetch operations.

4.2.3.1 Barnes

I inserted prefetch operations for four computations: gravity-potential calculation, body partition, center-of-mass calculation, and body insertion.[10] In the gravity-potential calculation and the body partition, each processor traverses the Barnes-Hut tree in a depth-first manner. When a processor reaches a tree node, the processor prefetches all immediate subnodes. In the center-of-mass calculation, on the other hand, each processor sequentially accesses an array of pointers to cell records that are allocated to the processor. Each processor sequentially computes the center-of-mass of cells in the array. Each processor prefetches a cell record one iteration ahead in the loop for the center-of-mass calculation. In the body insertion, processors go down the tree from the root node to find an appropri-

10. The prefetching strategy for the gravity-potential calculation is similar to the hand-inserted prefetching strategy described in [44]. In this study, more prefetch operations are added for the other three computations.

ate place to insert a body. When a processor reaches a tree node, the processor prefetches a cell record that the processor visits in the next iteration.

In Barnes, 99% of the prefetch operations are performed to a valid cache line and not issued to the network. This ratio is largest among our ten applications. Since each processor traverses the tree multiple times in the gravity-potential calculation and always prefetches the subnodes of visited nodes, processors usually prefetch the same nodes multiple times. It turns out that those nodes are often valid in the processor cache because of its infinite capacity and that the prefetches for those nodes cause most of the not-issued prefetches in Barnes. As the cache size decreases, the ratio of not-issued prefetch operations should decrease because capacity misses occur for node accesses in the gravity-potential calculation.

4.2.3.2 FFT

I inserted prefetch operations for a transpose operation, which causes most of the sharing misses in FFT. Since the computation is regular, we can eliminate unnecessary prefetches by carefully inserting prefetch operations. The simulation results, in fact, show that the ratio of covered misses and used prefetch operations are 100%. Loop splitting and software pipelining techniques maximize the efficiency of prefetch operations. Because of the infinite cache assumption, prefetch operations are inserted only for sharing misses. For finite cache cases, more prefetch operations should be inserted where capacity misses occur.

4.2.3.3 Locus

I inserted prefetch operations for CostArray accesses that cause the bulk of sharing misses in Locus. The access pattern to the elements of CostArray depends on the input and is not statically predictable. Therefore, it is difficult to identify CostArray elements that need prefetching (ones that have not been accessed or have been invalidated). Thus, I inserted prefetch operations at all loops that use CostArray, and I unrolled those loops to avoid multiple prefetches of the same cache line in the same loop. It turns out that most of the prefetch operations (95%) are unnecessary because the prefetch target line is usually valid in the cache. I also inserted prefetch operations for DensityArray accesses, which have a pattern similar to CostArray accesses.

4.2.3.4 LU

I inserted prefetch operations in three loops that access matrix blocks that another processor updates. Since the computation is regular in LU, we can insert prefetch operations only for memory accesses that cause misses. The simulation results, in fact, show that prefetch operations cover 100% of read misses and that 100% of prefetch operations prefetch a cache line that is actually used. I used loop splitting and software pipelining techniques to avoid unnecessary prefetches. I assumed an infinite cache to insert prefetch operations for this simulation. For finite cache cases, more prefetch operations should be inserted for capacity misses.

4.2.3.5 Maxflow

Most of the sharing misses occur for edge and node records of the input graph. I inserted prefetch operations for two while-loops that cause a large number of sharing misses. Those loops perform several computations for one graph node. For each iteration, a processor accesses one edge and two nodes that are directly connected by the edge. We use a software pipelining technique to prefetch these edge and node records. In one loop, for example, a processor prefetches an edge record two iterations ahead. The edge record contains the node address that the processor needs to access. During the next iteration after the edge record is prefetched, the node address becomes available, and the processor prefetches the node. Although this application extensively uses linked lists, the access pattern is structured. Inserting prefetch operations is thus a relatively simple task, and the covered ratio is relatively large (81%).

Maxflow, however, dynamically allocates graph nodes to processors, and processors sometimes conflict with each other for accessing the same data record. It is therefore hard to avoid unnecessary (not-issued and not-used) prefetches. Because of the dynamic allocation and the conflicting access, when a processor accesses a data record that the processor prefetched for another computation, the prefetched data record may or may not have been invalidated by another processor. Hence, processors conservatively prefetch edge and node records for each access to them. This conservative prefetching increases the number of unnecessary prefetches. As a result, the ratio of used prefetch operations is 15% for Maxflow.

4.2.3.6 Mincut

I inserted prefetch operations for graph nodes and some other global variables. Each processor randomly picks a graph node and examines other nodes that are directly connected to the node. Each processor prefetches those connected nodes as soon as the processor picks the first node. Since graph nodes are not well divided and distributed to processors in the program, processors cause conflicting accesses that invalidate prefetched graph nodes in other processor caches. Because of the conflicting access, I chose a conservative strategy that always prefetches graph nodes even though they were prefetched previously. The conservative prefetching causes a large number of not-issued prefetches. Although the time distance between the prefetch operation and the subsequent use is usually small, conflicting accesses still sometimes invalidate a cache line between the prefetch operation and the subsequent use. In fact, for 25% of read misses that occur without prefetch operations, the cache line is valid when the prefetch operation is performed but invalid when the line is accessed. As a result, prefetch operations cover only 67% of read misses.

4.2.3.7 MP3D

I inserted prefetch operations for a particle movement computation and some other computations.[11] During the particle movement computation, processors prefetch particle and cell records by using a software pipelining technique. Since particle records hold the pointer to a cell record, a processor prefetches a particle record two iterations ahead. In the next iteration, the address of the cell record becomes available, and the processor prefetches the cell record. I used prefetch-exclusive operations, instead of normal prefetch operations, since processors modify prefetched particle and cell records during the particle movement computation.

Most of the prefetch operations for particle records are not actually issued to the network since the corresponding cache lines are often valid when a processor performs prefetch

11. The prefetching strategy is similar to the hand-inserted prefetching strategy described in [44].

operations. This is because particles do not frequently collide with each other and particle records are not frequently updated by other processors. Thus, by simply avoiding prefetch operations for particle records, we can significantly decrease the number of unnecessary prefetch operations with a small increase of the number of cache misses. As the cache size decreases, however, the number of capacity misses increases for particle records, so that the ratio of unnecessary prefetches for particle records decreases.

Not-used prefetches occur because of conflicting accesses for cell records. Although a cell record is prefetched only one iteration ahead, another processor invalidates the prefetched line before it is used. For 15% of prefetch operations, prefetched cache lines are wasted because of such conflicting accesses.

4.2.3.8 Ocean

Most of the sharing misses occur when processors access grid elements that are allocated to another processor for nearest-neighbor operations. I inserted prefetch operations in all loops that access such grid elements. Each processor prefetches a grid element two iterations ahead, so that a pipelining effect reduces the average of memory latencies even when the cache-miss time is large. Moreover, I unrolled some loops to avoid applying multiple prefetch operations to the same cache line within the same loop. Although false sharing causes conflicting accesses that invalidate prefetched cache lines, the number of false sharing misses is very small. Because of regular access patterns, the covered ratio and the efficiency of prefetch operations are very large in Ocean.

4.2.3.9 Pthor

Unlike other applications in our study, memory accesses that cause read misses are widespread in the source program. I inserted prefetch operations at several places in each of three subroutines.[12] First, processors prefetch task-queue entries and a circuit-element record in a subroutine that dequeues a circuit element to be evaluated. Second, processors prefetch input and output records of the circuit element and some other data structures in a subroutine that evaluates the circuit element. Last, processors prefetch a task-queue entry and an enqueued circuit-element record in a subroutine that enqueues a circuit element.

12. The prefetching approach is similar to the hand-inserted prefetching approach described in [44].

The simulation results show that these prefetch operations cover 74% of read misses that occur without prefetch operations.

Conflicting and irregular access patterns make it difficult to use prefetch operations efficiently in Pthor. Although there are multiple task queues, processors sometimes compete by accessing the same task queue and invalidate prefetched cache lines in other processor caches before those lines are used. Prefetch operations for circuit-element records, moreover, are not always useful. Each circuit-element is always enqueued to the same task queue and each processor is assigned to the same task queue unless the queue is empty. Thus, a processor often evaluates a circuit-element that the processor has evaluated. As a result, a processor performs prefetch operations for circuit-element records that are valid in the cache. The simulation results show that only 26% of prefetch operations transfer cache lines that are actually used.

4.2.3.10 Water

I inserted prefetch operations for molecule records during the inter-molecule-force computation. Each iteration of the inner-most loop computes one pair of molecules and updates the record of both molecules. Processors prefetch both molecules one iteration ahead. Since multiple processors update the same molecule, processors may conflict with each other for accessing the same molecule record and invalidate a prefetched molecule record in other processor caches. Thus, I have chosen a conservative strategy; a processor prefetches each pair of molecules even though the processor prefetched those molecules beforehand. When the cache size is infinite, however, the processor cache typically has a valid copy of cache lines for molecules that are allocated to the local processor. Thus, most of the prefetch operations for those molecules are unnecessary. If we avoid prefetch operations for those molecules, we can eliminate about 50% of prefetch operations with a very small increase of the number of cache misses. About 44% of the remaining prefetch operations are still useless because conflicting accesses invalidate prefetched lines before they are used.

4.2.4 Summary

We have discussed application characteristics and their effects on the behavior of prefetch operations. Our simulation results show that hand-inserted prefetch operations cover a sufficiently large ratio of the read misses that occur without prefetch operations. Our simulation results also show that the ratio of used prefetch operations is very large (> 95%) for applications with regular computations such as LU, FFT, and Ocean, while the ratio is very small (1 to 28%) for applications with irregular computations. Most of unused prefetches occur because the prefetch target line is valid in the cache. The prefetch scheme can prevent such unused prefetches from generating network traffic because the processor cache can filter out unused prefetch traffic. The deliver scheme, however, cannot prevent unused deliver traffic in a manner as simple as that the prefetch can do. The filtering function of the processor cache has a significant advantage for the prefetch scheme.

4.3 Chapter Summary

In this chapter, we have examined the sharing characteristics of parallel applications that affect the behavior of deliver and prefetch operations. Our simulation results show that applications with a static producer-consumer relation generate no or little traffic overhead for the deliver scheme while applications with a dynamic producer-consumer relation usually generate substantial traffic overhead. We identified two common access patterns that cause unnecessary deliver messages: single-processor reuse and reader migration.

We examined three techniques for a trade-off between the covered ratio and the deliver message overhead: competitive back-off, subscription control, and selective deliver. In competitive back-off and subscription control, the consumer cancels the subscription of deliver messages if they are likely to be unnecessary. In selective deliver, the producer omits deliver operations if they are likely to generate substantial traffic overhead. Competitive back-off takes advantage of dynamically changing behavior of the application, while subscription control and selective deliver take advantage of static software knowledge of the application. Our simulation results show that selective deliver performs best or nearly best for most of our applications.

The prefetch scheme also generates a large number of unnecessary prefetch operations for applications with a dynamic producer-consumer relation while the prefetch scheme generates a small number of unnecessary prefetch operations for applications with a static producer-consumer relation. Since most of the unnecessary prefetch operations occur for cache lines that are already in the cache, the prefetch scheme can eliminate most of the network traffic due to unnecessary prefetch operations by simply checking the processor cache before issuing a prefetch request to the network. This is the most important advantage of the prefetch scheme over the deliver scheme; the cache system works as a filter to eliminate unnecessary traffic for prefetches but not for delivers.

Chapter 5

Real Architectural Effects

The discussion in the previous chapter assumes an ideal machine model that has an infinite cache, a short cache-line, and a unit memory latency. Although this model is useful for discussions about intrinsic sharing characteristics of parallel applications, the results do not reflect the impact of architectural parameters, such as cache configurations and memory bandwidth, which can significantly affect the performance of parallel programs. In this chapter, we discuss the effect of important architectural parameters on the performance of deliver and prefetch operations. We also analyze application characteristics that interact with those parameters, such as temporal and spatial locality of memory access behavior.

We begin with examining the effect of three important cache parameters — cache size, associativity, and line size — in Section 5.1 and Section 5.2. In Section 5.1, we also examine temporal characteristics of deliver and prefetch operations and working-set characteristics of application programs to understand the effect of cache size and associativity on the performance of deliver and prefetch operations. In Section 5.2, we examine spatial and false-sharing characteristics of application programs to understand the effect of line size. In Section 5.3 and Section 5.4, we discuss the effect of memory latency and bandwidth. Finally, we summarize our discussions about the effect of architectural parameters and application characteristics on the performance of deliver and prefetch operations.

5.1 Cache Size

Unlike an infinite cache in our ideal machine model, real caches can store only a certain number of cache lines. A cache conflict occurs when a cache needs to evict a valid cache line to accommodate a new line in the cache. This conflict causes a cache miss if the local processor accesses the evicted cache line later. Any data transfer that brings in a cache line (e.g., cache-misses and deliver/prefetch operations) may cause a cache conflict. Generally, the smaller the cache, the more cache conflicts occur.

In this section, we first classify cache conflicts due to deliver/prefetch operations and qualitatively discuss important factors that affect the impact of cache conflicts; namely temporal characteristics and replacement policies of deliver and prefetch operations. Second, we quantitatively examine temporal characteristics of deliver and prefetch operations. Third, we discuss replacement polices that we use for our simulation study. Fourth, we classify cache misses and analyze the lower-bound miss rate that deliver operations could achieve. Finally, we quantitatively discuss the impact of cache size and associativity for deliver and prefetch operations by analyzing working-sets of application programs.

5.1.1 Cache Conflicts

Figure 5-1 illustrates two types of cache conflicts that deliver and prefetch may suffer from. The first type (TYPE I) occurs when a transfer operation — deliver or prefetch — evicts any valid cache line. If the processor uses the evicted cache line and does not use the transferred line, the number of cache misses increases. The second type (TYPE II) occurs when an on-demand memory operation — read or write — evicts a line transferred by a deliver or prefetch before the local processor uses the line. If this type of conflict occurs, the transfer operation does not decrease the number of cache misses. The number of cache misses due to the first conflict depends on the replacement policy for the transfer operation, while the number of cache misses due to the second conflict depends on temporal characteristics of the transfer operation.

A replacement policy determines a cache line to be evicted — a victim line — when a cache conflict occurs. If a replacement policy could choose a line that the local processor would not use in future, a cache conflict would not cause a cache miss. This is not always possible since information about future access patterns is generally unavailable. Thus, replacement policies generally use information about past access patterns to predict a cache line that will be used with lower probability.

Unlike on-demand memory operations, deliver and prefetch operations may transfer a cache line that the consumer does not use before the line is replaced or invalidated. Thus,

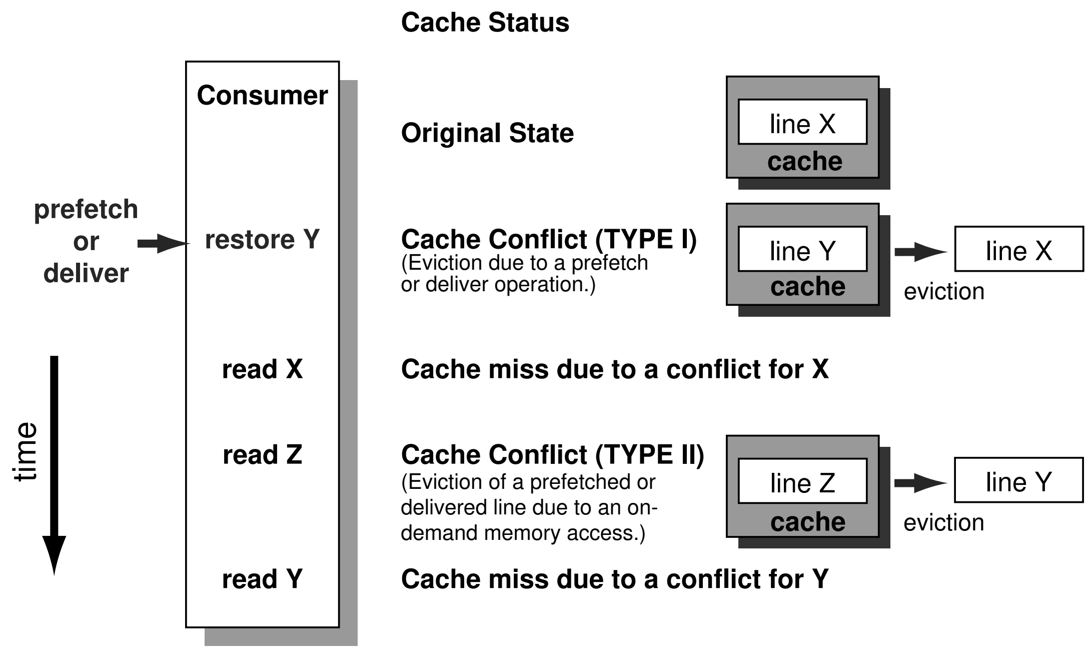

Figure 5-1: Two Types of Cache Conflicts due to Prefetch and Deliver.

the delivered or prefetched line itself can become a victim. As we discussed in Section 4.1, in fact, deliver operations transfer many cache lines that the receiver does not need for some applications. Thus, a replacement policy for deliver operations may need to consider a delivered line as a victim to minimize the number of cache misses due to cache conflicts. Prefetch operations, on the other hand, mostly transfer cache lines that the receiver needs, so that the replacement policy probably does not need to consider a prefetched line as a victim.

Temporal characteristics of the cache-line transfer also affect the number of cache conflicts. If the cache-line is transferred too early, the cache line is probably evicted before the local processor uses it, so that the cache-line transfer does not eliminate a cache miss that the transfer is supposed to eliminate. The earlier the cache-line is transferred, generally, the more likely the cache-line is evicted before the local processor uses it. We should note that if the cache-line is transferred too late, however, it does not arrive at the local processor before the cache-line is needed, and thus the transfer does not completely hide the memory latency. As we will discuss in the next subsection, deliver operations usually transfer cache lines much earlier than prefetch operations, so that deliver operations suffer more from cache conflicts than prefetch operations, while potentially hiding larger latencies.

5.1.2 Temporal Characteristics

As we discussed in the previous subsection, temporal characteristics of deliver and prefetch operations are important for latency hiding. In this subsection, we examine the statistics of the time between a cache-line transfer due to a prefetch or deliver operation and the subsequent use of the transferred line. Our simulation results show that prefetch operations generally transfer cache lines much later than deliver operations but still early enough to hide typical memory latencies. If the memory latency becomes much larger than what is currently typical, however, it becomes more difficult to insert prefetch operations to hide the memory latency completely.

As illustrated in Figure 5-2, a *prefetch distance* is defined as the number of clock cycles between a prefetch operation and the subsequent use of the prefetched line. Similarly, a *deliver distance* is defined as the number of clock cycles between a deliver operation and the subsequent use of the delivered line. Figure 5-3 and Figure 5-4 show a histogram of deliver and prefetch distances for each application. The X axis shows the number of processor-clock cycles on a log scale. The Y axis shows the ratio of the number of the cache-

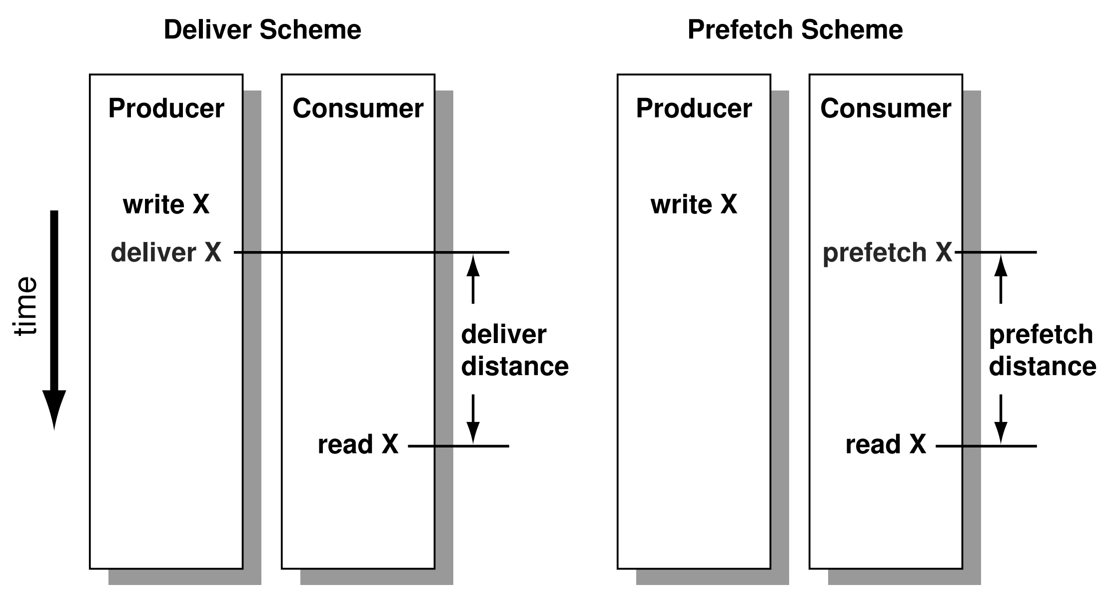

Figure 5-2: Prefetch and Deliver Distances.

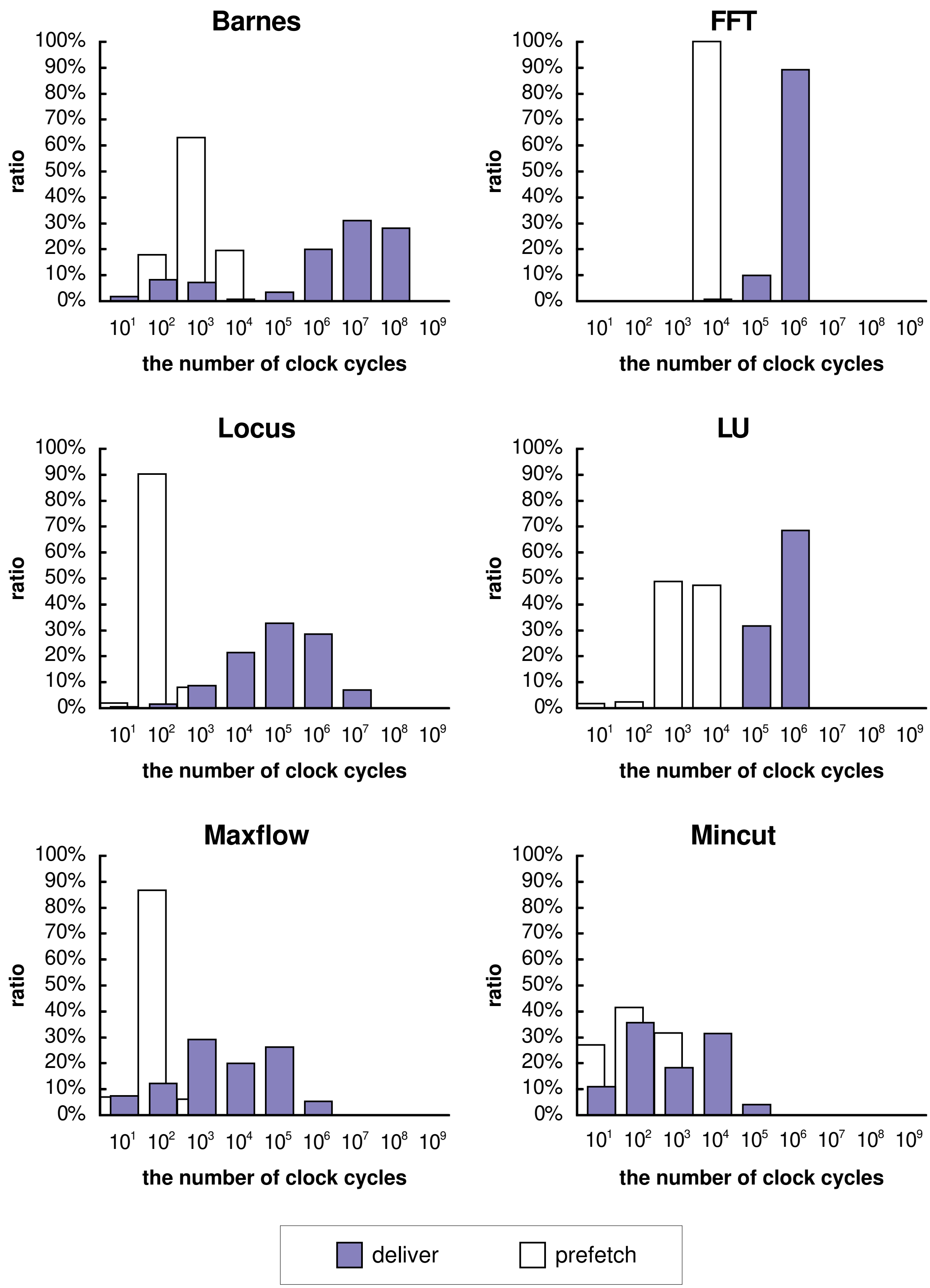

Figure 5-3: Histograms of Deliver and Prefetch Distances.

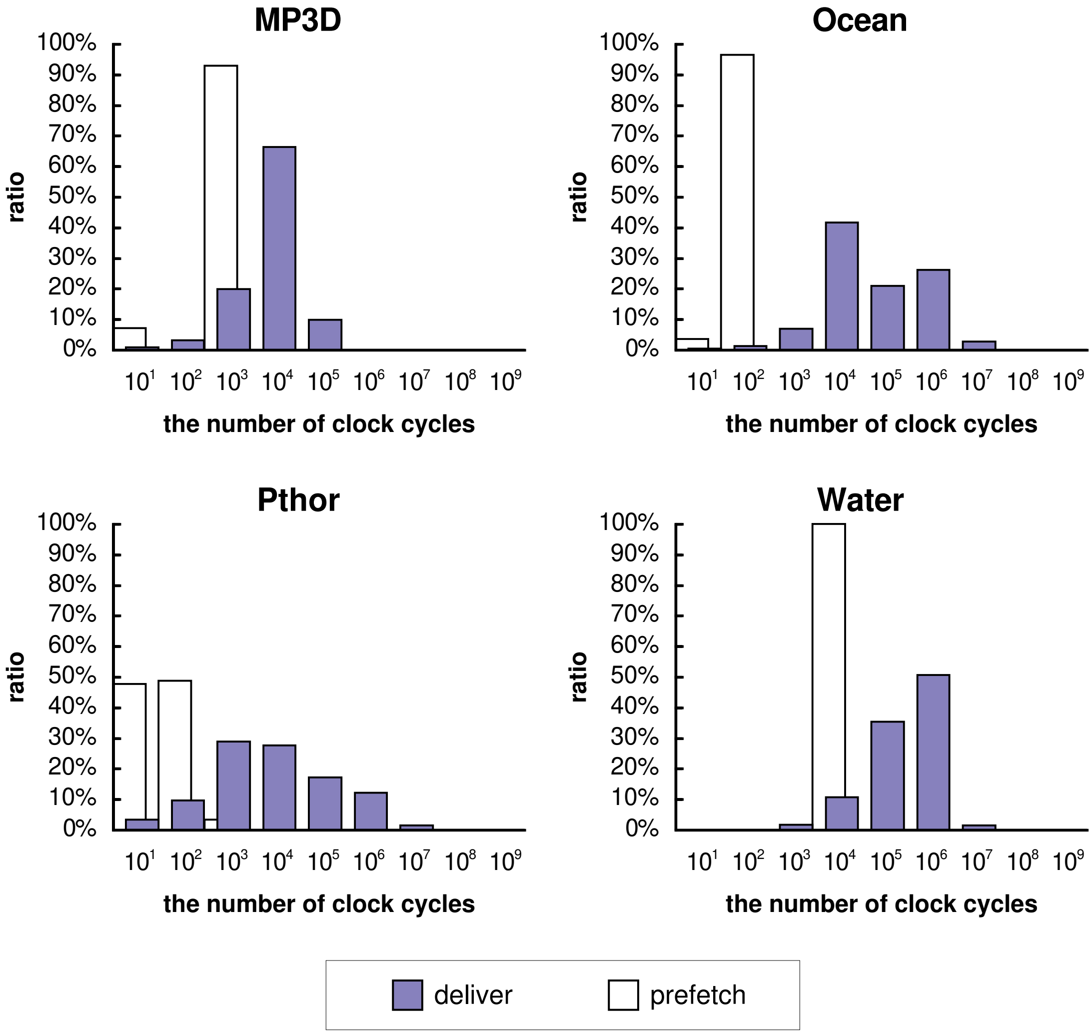

Figure 5-4: Histograms of Deliver and Prefetch Distances. (Continued.)

line transfer events for the corresponding clock range. We assume 16-byte cache lines and a unit-latency memory model for these simulations. The histograms clearly show that deliver operations occur much earlier than prefetch operations except for Mincut. These results indicate that, generally, delivered lines have a lower probability of staying in the cache until they are used compared to prefetched lines. The results also indicate that, for most of our applications, the prefetch distance is usually at least as long as 10^2 clocks, as the prefetch annotation intended. Thus, prefetch operations can usually hide typical remote memory latencies (roughly 10^2 clocks) that we find in current shared-memory multiprocessors.

The temporal characteristics we see depend on the places where we insert deliver and prefetch operations in the application program.[1] Prefetch operations are typically inserted in a loop so as to prefetch a variable that becomes necessary in a future iteration of the loop. Mowry [45] showed that prefetch operations should be scheduled $\lceil l/s \rceil$ iteration ahead in the loop, where l is an expected memory latency and s is the length of the shortest path of the loop body. Mowry also showed that it is generally straightforward to schedule prefetch operations early enough to hide memory latencies if the target variable is indexed by a loop induction variable (or more generally an affine function of the loop induction variable). The temporal characteristics of prefetch operations, in fact, confirm that this prefetch annotation strategy can sufficiently hide memory latencies we assume.

If memory latency significantly increases, however, it is difficult to hide enough memory latency for applications that have conflicting or irregular memory accesses (e.g., intensive uses of pointers). When an application has a conflicting access pattern, another processor may overwrite prefetched variables before they are used. If this happens, the prefetch operation does not hide memory latencies at all; the prefetching processor causes a cache miss to fetch the data from the main memory. Therefore, even if we schedule a prefetch operation early to try to hide large memory latencies, conflicting accesses may reduce the benefit of the prefetch operation. Mincut, MP3D, and Water have this type of access pattern. Moreover, if we need to insert prefetch operations for irregular memory accesses, it may be difficult to calculate the address to be prefetched early enough to hide large memory latencies. Mowry [45] showed that irregular access patterns with an intensive use of pointers in Pthor prevent prefetch operations from significantly hiding memory latencies. For such applications, as the memory latency increases, prefetch operations become even less effective for performance improvement unless the program is significantly modified. Furthermore, as the memory latency increases, we may need to apply software pipelining to an outer loop to hide memory latencies sufficiently. Thus, even if an inner loop includes only regular accesses and is easy for prefetch insertion, if the outer loop includes irregular accesses, prefetch insertion may become a complicated task. Therefore, if the memory latency is much larger than that we assume, deliver operations are potentially more advan-

1. For details of deliver and prefetch annotations, refer Subsection 4.1.4 and 4.2.3.

tageous than prefetch operations for applications that generate conflicting or irregular access patterns.

Deliver operations, on the other hand, have very different temporal characteristics from prefetch operations. The deliver distance is much longer, typically 10^4 to 10^7 clocks, except for Maxflow, Mincut, and Pthor. This is basically because most of our applications partition the computations and the shared data so that the production of the shared data is far away from the use of the data. Figure 5-5 illustrates two common structures of parallel programs. The diagram is slightly different from actual programs but close enough to explain the temporal characteristics of deliver and prefetch operations.[2] The first diagram in Figure 5-5 has two loops, producer and consumer loops, inside the outermost loop. The producer loop writes some shared data, and the consumer loop reads the shared data that other processors write in the producer loop. Both producer and consumer loops are generally nested. While we insert deliver operations in the producer loop for the deliver scheme, we insert prefetch operations in the consumer loop for the prefetch scheme. This program structure represents Barnes, FFT, LU, and Ocean. For these programs, the producer and consumer loops are relatively large: each loop usually accesses a significant portion of the shared data, so that cache systems can exploit the locality of memory accesses. Therefore, the deliver distance is very long for these programs. The deliver distance of Ocean is relatively short (peak at 10^4 clocks) among the applications with the same type of the structure. This is because of the multigrid computation, which consists of nearest-neighbor operations on hierarchical grids. The higher the grid level becomes, the fewer points the grid has, so the smaller the consumer and producer loops become. Thus, the deliver distance is relatively short for higher level grids. The prefetch distance, on the other hand, is relatively short for all applications because prefetch operations are inserted in the consumer loop, often in the innermost loop of the consumer loop.

The second diagram in Figure 5-5 shows another common structure, a merged producer/consumer loop inside the outermost loop. The merged loop reads shared data produced by other processors and writes shared data read by other processors. This program

2. For some programs, the number of loop nests is different from the diagram, some loops are a while-loop instead of a for-loop, and some data structures are not an array (e.g., a tree or a graph).

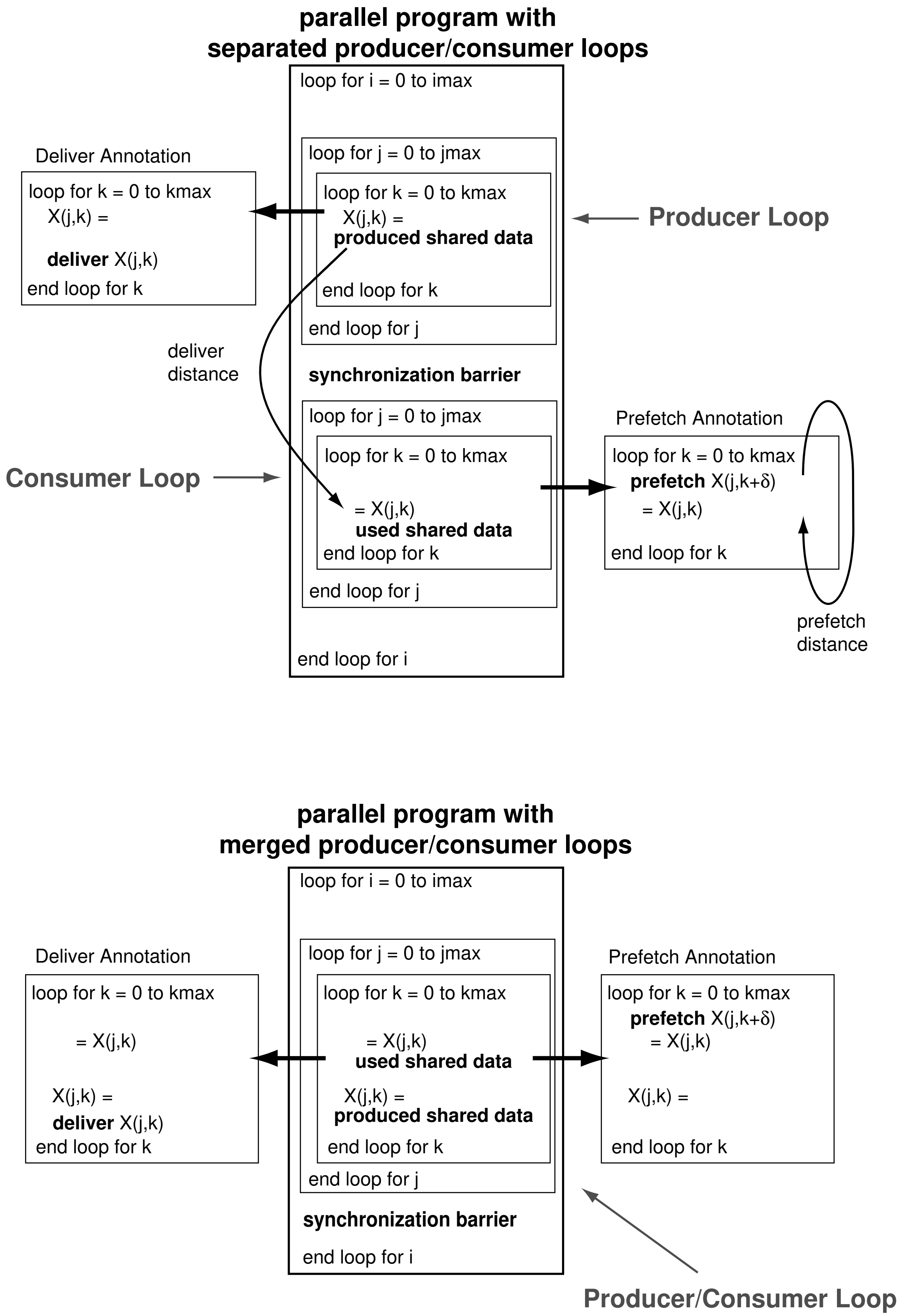

Figure 5-5: Two Common Structures of Parallel Programs.

structure represents Locus, Maxflow, Mincut, MP3D, Pthor, and Water.[3] For these programs, the deliver distance depends on the data partitioning technique, the input data, and sometimes small details in the program structure. If the application is well-written, the deliver distance is generally long. Locus, for example, uses geographical partitioning; each processor mostly accesses shared data of one partition at a time, and after finishing tasks for the partition, the processor moves to another partition. Thus, other processors do not use most of the produced data in a partition until they move to the partition. Therefore, the deliver distance is long for Locus. For MP3D, the deliver distance is moderately long (roughly 10^4 clocks), as shown in Figure 5-4. If we increase the number of particles, however, particles collide frequently, so that the deliver distance decreases. The deliver distance of Pthor is relatively short because false sharing causes frequent communication. The deliver distance of Maxflow and Mincut is also relatively short because a lack of effective data partitioning technique causes frequent communication.

Because the deliver distance is generally much longer than the prefetch distance, deliver operations potentially tolerate larger memory latencies than prefetch operations, but delivered data are more probably evicted from the cache before they are used than prefetched data. This raises an interesting question; can we delay the timing of deliver operations so that the deliver distance is short enough to minimize the impact of cache conflicts but still long enough to hide memory latencies? We qualitatively discuss this question.

We could probably use a software pipelining technique similar to one that we used for scheduling prefetch operations. In general, however, it is more difficult to delay deliver operations for reducing cache conflict effects than to schedule prefetch operations for hiding sufficient memory latencies for several reasons. First, to identify how long we need to delay deliver operations, we need to identify when consumers use the shared data to be delivered. It is not a simple task for producers to predict the consumer's access pattern unless the access pattern is regular. For most of our applications, the access pattern dynamically changes. For example, Locus, Maxflow, and Pthor use distributed task queues. Since the access pattern dynamically changes for load balancing in these pro-

3. Although MP3D and Water have a combined structure of these two common structures, the merged consumer/producer loop represents the majority of the communication in the program.

grams, it is difficult, or at least expensive, for producers to determine when they should perform deliver operations. For Locus, for example, Figure 5-3 shows that the distribution of deliver distances is widespread. Since this distribution mainly exhibits the characteristics of only one data structure (CostArray), this distribution implies that, for Locus, the consumer's access pattern of the data structure changes dramatically and is hard to predict. Moreover, each consumer generally uses the delivered data in a different timing. To minimize the cache conflict effect, therefore, producers need to keep track of each consumer's access pattern and to send the shared data to each consumer with a different timing. Even if this is possible, the extra code for delaying deliver operations is probably complex and may cause significant overhead unless the computation is regular.

5.1.3 Replacement Policies

When a cache conflict occurs, a replacement policy determines a victim line to be discarded. The choice of a replacement policy is especially important for deliver operations because deliver operations suffer significantly from cache conflicts as we discussed in Subsection 5.1.2. In this subsection, we describe replacement policies that we use for our simulation study and discuss the motivation for the choice of the replacement policies.

An optimal, if impractical, replacement policy is known for uniprocessors [6, 42]: when a cache needs to replace a cache line, the chosen line is the one whose next reference is farthest in future. We can extend this optimal policy for multiprocessors with deliver and prefetch operations by adding two rules. First, if another processor invalidates a cache line before the local processor references it next time, we consider that the cache line is no longer used and that the cache line is a candidate for replacement. Second, when a deliver or prefetch operation transfers a cache line to a cache, the cache line itself can become a victim for replacement if the local processor will not access the cache line for a longer period than any other cache lines.

While the optimal policy for multiprocessors is of interest for theoretical studies, the policy is not implementable since it requires knowledge about future memory accesses. Among implementable policies, we use two variations of a LRU policy for our simulation study. The first variation handles delivered (or prefetched) cache lines as if a cache miss brought them into the cache. That is, if a cache conflict occurs, the replacement policy evicts the least-recently-used line and considers the new line as the most-recently-used line. We call this policy an *optimistic policy* since this policy assumes that the local processor will use the transferred line very soon. This replacement policy should be reasonable for prefetched cache lines because the local processor usually uses prefetched cache lines soon after they are prefetched as we discussed in Subsection 5.1.2. For delivered cache lines, however, this replacement policy may increase the impact of cache conflicts, especially for small caches. When a cache conflict occurs, we want to evict a cache line that will not be used for the longest time. Since the deliver distance is generally very large, delivered cache lines may not be used sooner than any other cache lines if the cache is small. Thus, we consider another variation of a LRU policy for deliver operations, a pessimistic policy.

The *pessimistic policy* assumes that the local processor will use the delivered cache line only if the cache has an invalidated copy of the cache line. When a cache recycles an invalidated line to store another line, the cache uses the least-recently invalidated line. The pessimistic policy, therefore, stores a delivered line only if the local processor used the line so recently that the cache has a valid copy of the line when another processor writes the line and if the invalidation was so recent that the cache still has the invalidated copy.[4] The pessimistic policy also assumes that the local processor will not use the delivered line sooner than any valid lines in the cache. Thus, the cache stores the delivered line as if the local processor accessed the line least recently. In other words, when the cache needs to replace a valid line, the cache replaces unused delivered lines before any lines that the local processor has used. Therefore, deliver operations will never interfere with cache lines that the local processor used, so that the miss rate will never be larger than that of the invalidate-only protocol even if the cache size is very small.

For simplicity, we call the deliver operation with the optimistic replacement policy *optimistic deliver*, and we call the deliver operation with the pessimistic replacement policy *pessimistic deliver*. If the cache is infinite, the optimistic and pessimistic deliver operations should generate the same miss rate since the cache always has an invalidated copy

4. Deliver with the pessimistic policy is similar to an update protocol with a write cache in a sense that both protocols transfer cache lines to consumers only if the consumer's cache has a copy (valid or invalidated) of the cache line.

for delivered lines. If the cache is very small, the optimistic deliver probably generates a larger miss rate than the invalidate-only protocol since delivered lines evict necessary cache lines, while the pessimistic deliver probably generates the same miss rate as the invalidate-only protocol. In Section 5.1.5, we will discuss how the cache size and the replacement policy affect the miss rate by examining our simulation results.

For some cache and application configurations, there may be better implementable replacement policies than the two replacement policies that we have discussed. Our goal, however, is not to find the best replacement policy for some configurations but to obtain an intuition about the interplay between replacement policies and deliver/prefetch operations. Our analyses of these two replacement policies provides this insight.

5.1.4 Classification of Cache Misses

In this subsection, we classify cache misses to understand the lower bound of the miss rate due to deliver operations. Cache misses are generally divided into three types: cold, capacity, and sharing misses.[5] A cold miss occurs because of the first access for the memory block. A capacity miss occurs because of a replacement of the line due to finite cache capacity. A sharing miss occurs because of an invalidation of the line due to an invalidation-based coherency protocol. This simple classification, however, is not helpful to understand the number of cache misses that deliver operations can eliminate. This is because a deliver operation may not eliminate some sharing misses and may eliminate some capacity misses. A pessimistic deliver operation, for example, cannot eliminate a sharing miss if the local processor replaces the cache line after some processor wrote it. An optimistic deliver operation, on the other hand, may eliminate a capacity miss if some processor writes the cache line after the local processor replaced it.

To understand the number of cache misses that deliver operations can reduce, we further divide the above three miss-types as illustrated in Figure 5-6. We divide capacity misses into two types: if someone writes the cache line after the local processor replaced it, we call the miss a *sharing capacity miss*. If no one writes the cache line, we call the miss a *pure capacity miss*. We also divide sharing misses into two types: if the local processor

5. The sharing miss is also called the coherency miss.

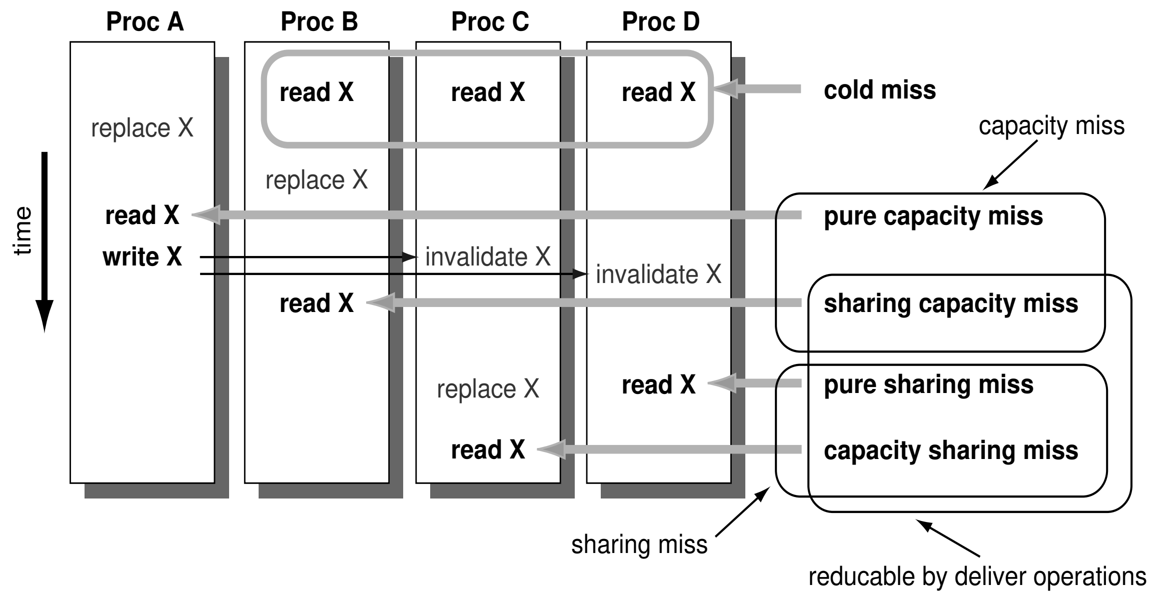

Figure 5-6: A Classification of Cache Misses.

retains the invalidated line until the local processor accesses it, we call the miss a *pure sharing miss*. If the local processor does not retain the invalidated line, we call the miss a *capacity sharing miss*. Deliver operations do not eliminate cold misses since deliver operations do not send a cache line to caches that have never held it, likewise deliver operations do not eliminate pure capacity misses since deliver operations follow a write operation and no write operation occurs between the replacement and the pure capacity miss. Therefore, the number of cold and pure capacity misses gives us the lower bound of the miss rate for the deliver scheme. Deliver operations may eliminate the rest of the cache misses (sharing capacity, pure sharing, and capacity sharing misses) if we insert a deliver operation after every write operation that is followed by a consumer's read operation. Deliver operations, however, can generally eliminate only a part of these misses because of cache conflicts.

Figure 5-7 shows the cache miss breakdown with our classification for Locus and FFT as an example. We assume fully-associative caches with 16-byte lines. We can obtain a rough idea about the miss rate due to deliver operations from this cache miss breakdown without simulating deliver-annotated applications. For both applications, when the cache is 16KB,

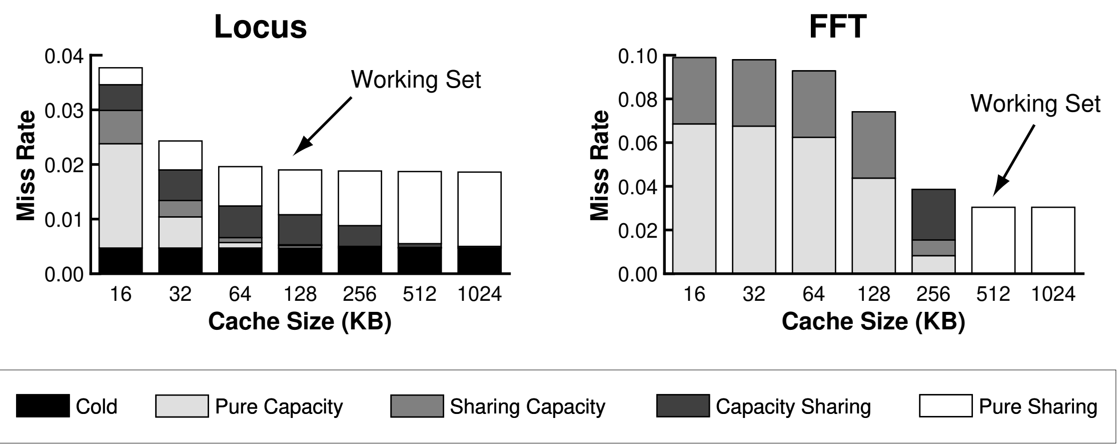

Figure 5-7: Breakdown of Miss Rate versus Cache Size. No deliver or prefetch operations are used.

the number of pure sharing misses is very small and capacity-related misses (pure-capacity, sharing-capacity, and capacity-sharing) dominate the overall cache misses. As the cache size increases, the number of these misses decreases. When the cache is 1MB, virtually no capacity-related misses occur. The arrow in each graph indicates the largest working set for the application, which represents the whole data partition for a single processor. As we discussed previously, the number of cold and pure capacity misses indicates the lower-bound miss rate due to deliver operations. Figure 5-7 shows that the lower-bound reaches about a half of the total miss rate when the cache is a quarter of the data partition (32KB for Locus and 128KB for FFT). Deliver operations, therefore, cannot significantly reduce the number of cache misses if the cache size is smaller than a quarter of the data partition for both applications. Figure 5-7 also shows that no pure capacity misses occur when the cache is larger than the data partition. Thus, the lower-bound is as low as that of an infinite cache when the cache is larger than the data partition. For Locus, capacity sharing misses still occur when the cache is as large as the data partition. This indicates that deliver operations may suffer from cache conflicts even if the whole data partition fits in the cache.

5.1.5 Cache Miss Rate versus Cache Size

In this subsection, we quantitatively evaluate the impact of the cache size and the replacement policy by using deliver-annotated and prefetch-annotated applications. Our simulator simulates the optimistic and pessimistic replacement policies that we discussed in Subsection 5.1.3. Figure 5-8 and Figure 5-9 show the results of our simulation. The X axis shows the cache size and the Y axis shows the miss rate. We assume a unit-delay model and fully associative caches with 16-byte lines. Four solid miss-rate curves in each graph correspond to the invalidate-only protocol (invalidate only), the deliver protocol with the pessimistic replacement policy (pessimistic deliver), the deliver protocol with the optimistic replacement policy (optimistic deliver), and the prefetch protocol with the optimistic replacement policy (optimistic prefetch). The arrow in each graph shows a working set knee in the miss-rate curve. Two dotted curves in each graph show a hypothetical miss-rate curve: lower-bound deliver shows the lower-bound miss rate due to deliver operations that we discussed in Subsection 5.1.4, that is the sum of the pure capacity miss rate and the cold miss rate of the invalidate-only protocol. No-conflict deliver shows the miss rate that we would observe if delivered messages did not conflict with any other cache lines.[6] The no-conflict deliver represents the lower-bound miss rate for a given deliver annotation. Lower-bound deliver, on the other hand, represents the lower-bound miss rate if a deliver operation follows all write operations immediately before a cache miss. The difference between these two is noticeable only for Locus, Maxflow, and MP3D.

The miss-rate curves in Figure 5-8 and Figure 5-9 show that the cache size affects the miss rate of the optimistic prefetch scheme less than the optimistic and pessimistic deliver schemes. This happens because of two reasons. First, prefetch can eliminate pure capacity and cold misses, while deliver cannot eliminate these misses. Since the number of pure capacity misses increases as the cache size decreases, prefetch has more opportunities to reduce the miss rate than deliver for small caches. Second, as we discussed in Subsection 5.1.2, the deliver operation generally occurs much earlier than the prefetch operation so that delivered cache lines need to stay in the cache longer to eliminate a cache miss than prefetched cache lines. Deliver operation, therefore, suffers more from cache conflicts than prefetch for small caches.

---

6. Our simulator simulates the no-conflict deliver by using an infinite buffer to store received deliver messages for each cache. The cache does not store received messages in the cache until the local processor accesses them. Thus, until the local processor uses deliver messages, the deliver messages conflicts with neither other unused delivered lines nor other cache lines that the local processor has used.

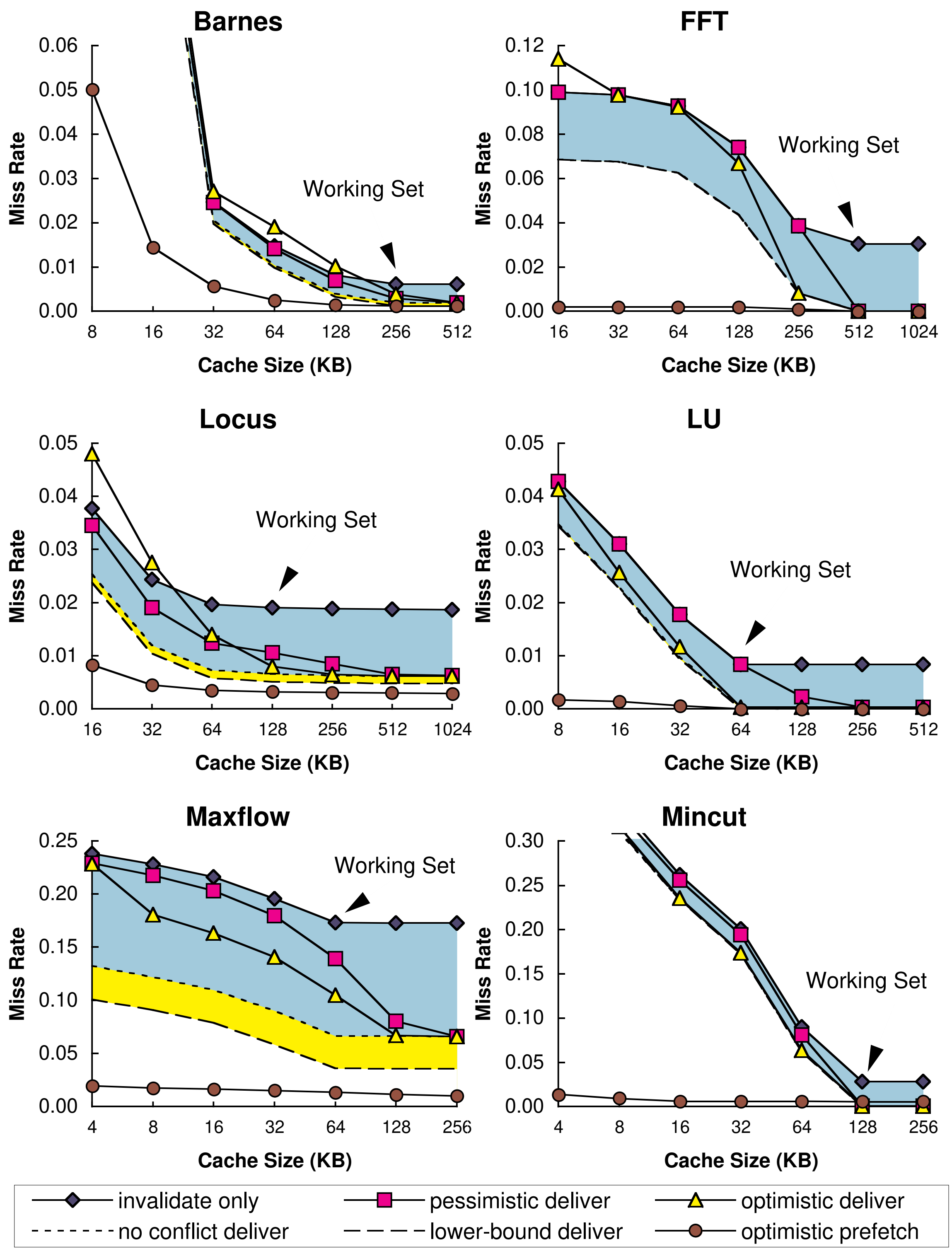

Figure 5-8: Miss Rate versus Cache Size.

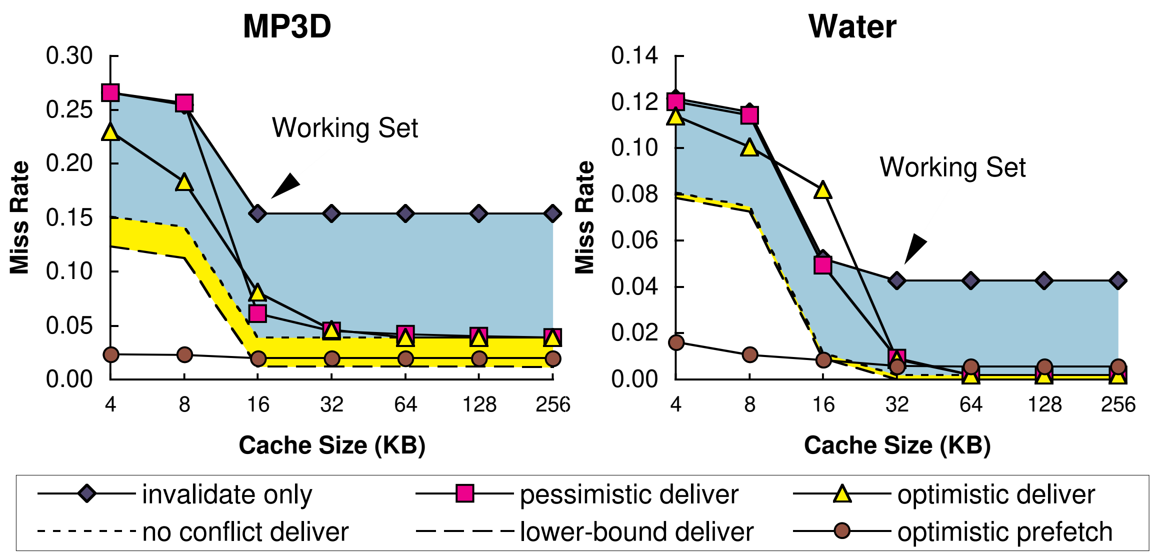

Figure 5-9: Miss Rate versus Cache Size. (Continued.)

Let's compare the miss rate of the optimistic and pessimistic policies for deliver. If the cache is large, it is essentially infinite, so that no cache conflicts occur. Thus, the optimistic and pessimistic policies generate the same miss rate as a deliver with no conflict. The largest cache size in each graph represents a cache size that is effectively equivalent to an infinite cache. As the cache size decreases, the miss rate of these two behaves differently. Generally, if the cache size is relatively large, optimistic deliver generates a smaller miss rate than pessimistic deliver. If the cache size is relatively small, however, pessimistic deliver generates a smaller miss rate than optimistic deliver. In FFT, for example, optimistic deliver is better for cache sizes between 32KB and 512KB, and pessimistic deliver is better for cache sizes smaller than 32KB. We can intuitively explain this behavior in the following way. Optimistic replacement policy assumes that the local processor will use delivered lines before other lines in the cache, while pessimistic replacement policy assumes that the local processor will not use delivered lines before other lines in the cache. Which replacement policy performs better depends on which assumption is more accurate. As we discussed previously, if the cache is sufficiently large, the two replacement policies generate the same miss rate. As the cache size decreases, optimistic deliver performs better than pessimistic deliver because the cache is still large enough to hold the cache lines that the local processor will not use sooner than delivered lines. That is, the assumption of the optimistic policy is more accurate than that of the pessimistic policy. When the cache size reaches a certain point, however, the cache can hold only the cache lines that the local processor will use relatively soon, so that the assumption of the pessimistic policy becomes more accurate than that of the optimistic policy. Thus, pessimistic deliver performs better than optimistic deliver for cache sizes smaller than that point. When the cache size decreases even further, optimistic deliver generates a larger miss rate than the invalidate-only protocol because of severe cache conflicts. The miss-rate behavior of deliver operations is similar for most of our applications, although these applications are very different in the sharing pattern and the program structure.

5.1.6 Working-Set Characteristics

An important question is how large the cache needs to be for the deliver operation to eliminate a significant number of cache misses. Figure 5-8 and Figure 5-9 indicate that, for most of our applications, the cache needs to be as large as the largest working-set so that the deliver operation can reduce the miss rate to about one half of the miss rate of the invalidate-only protocol. In Subsection 5.1.2, we discussed that deliver operations may transfer cache lines too early to reduce the miss rate significantly unless the cache is very large. In this subsection, we examine working-set characteristics to understand how large the cache needs to be to capture delivered cache lines.

First, we discuss equations to derive the working-set size and the cache miss rate from the inter-reference distance distribution for cache-coherent multiprocessors. Then, we compare the distribution of inter-reference distances with that of deliver distances to understand why the cache needs to be as large as the largest working-set to exploit deliver operations.

5.1.6.1 Inter-reference Distance Distribution and Working Set

Denning [17, 18] has shown a set of simple equations to estimate a program's average working set and the miss rate by using an inter-reference distance distribution. We extend the equations for multiprocessor caches with a coherency protocol.

First, we consider the invalidate-only protocol. Later, we add the effect of deliver operations. Let $S(R)$ be the average of the number of distinct cache lines that a processor accessed during the last R references and have not been invalidated by another processor. That is, $S(R)$ indicates the average size of a working set in terms of the number of cache lines when the working-set window is R references.[7] The following equations hold.

$$S(R+1) - S(R) = m(R) - i(R) \qquad \text{EQ 5-1}$$

$$m(R) = 1 - F(R) \qquad \text{EQ 5-2}$$

Where $m(R)$ is the probability of references for cache lines that are not included in $S(R)$, and $i(R)$ is the probability of invalidations for cache lines whose last reference occurred within the last R references. These two probabilities are per memory reference. $F(R)$ is the cumulative distribution of inter-reference distances, which indicates the number of references for cache lines that are referenced within the last R references and have not been invalidated by another processor. $F(R)$ is normalized by the number of all memory references. Thus, for example, $i(\infty)$ is the ratio between the number of received invalidation requests and the number of memory references, and $1 - F(\infty)$ is the ratio of references for cache lines that have never been accessed (i.e., cold misses) or have been invalidated since the last reference (i.e., sharing misses).

In EQ 5-1, $i(R)$ represents the effect of cache coherency that we add to an equivalent equation for uniprocessors that Denning has shown [18]. We assume that a fully-associative LRU cache recycles invalidated lines to store new cache lines before it recycles any valid lines. We use $i(R)$ to take into account this effect. With this assumption of the replacement policy, $S(R)$ corresponds to the number of cache lines in a fully-associative LRU cache that captures past R references, and $m(R)$ corresponds to the miss rate of the cache. We obtain $F(R)$ and $i(R)$ by simulations and calculate $S(R)$ and $m(R)$ by using EQ 5-1 and EQ 5-2. The calculated $S(R)$ and $m(R)$ are very close to the miss rate curve shown in Figure 5-8 and Figure 5-9.

Now, we consider the effect of deliver operations for the working set. We assume the optimistic deliver scheme, in which the cache stores a delivered line as if the local processor

---

7. While Denning used the number of clocks to measure the interval between references, we use the number of references to directly obtain the miss ratio — the number of cache misses per memory reference — instead of the instantaneous miss rate — the number of cache misses per clock cycle.

references the line most recently. We add another term to EQ 5-1 and EQ 5-2 to take into account the effect of deliver operations, as shown below.

$$S(R+1) - S(R) = m(R) - i(R) + d \qquad \text{EQ 5-3}$$

$$m(R) = 1 - F(R) - D(R) \qquad \text{EQ 5-4}$$

We slightly change the definition of the working set so that the working set includes a set of cache lines that are delivered to the processor. In EQ 5-3, *d* is the number of delivered cache lines per reference, and *i(R)* is the probability of invalidations for cache lines that are referenced or delivered within the last *R* references. *S(R)* is now the average of the number of cache lines that are referenced or delivered in the cache within the last *R* references and have not invalidated by another processor. *D(R)* is the ratio of references for cache lines that are delivered to the processor within the last *R* references and have not been invalidated by another processor. *D(R)* represents the decrease of the miss rate because of deliver operations. Thus, *m(R)* corresponds to the miss rate for the optimistic deliver scheme.

For a cache with *S(R)* lines, EQ 5-3 and EQ 5-4 indicate that the cache does not hold cache lines that were delivered outside the last *R* reference window. In other words, for a cache with *S(R)* lines, a deliver operation does not eliminate a cache miss if the distance between the deliver and the following reference is larger than *R* references.

5.1.6.2 Hierarchical Working Sets

Parallel applications typically have hierarchical working sets. Each working set corresponds to a knee in the miss rate curve for the application [52, 59], as illustrated in Figure 5-10. From EQ 5-1 and EQ 5-2, we derive the following equations;

$$\frac{\Delta S}{\Delta R} = S(R+1) - S(R) = 1 - F(R) - i(R) \qquad \text{EQ 5-5}$$

$$\begin{aligned}\frac{\Delta m}{\Delta R} &= m(R+1) - m(R) \\ &= (1 - F(R+1)) - (1 - F(R)) \\ &= -\frac{\Delta F}{\Delta R}\end{aligned} \qquad \text{EQ 5-6}$$

$$\frac{\Delta m}{\Delta S} = \frac{\Delta R}{\Delta S} \cdot \frac{\Delta m}{\Delta R} = -\frac{1}{1 - F(R) - i(R)} \frac{\Delta F}{\Delta R} \qquad \text{EQ 5-7}$$

EQ 5-7 indicates that the working-set knee in the miss rate curve corresponds to a knee in the distribution curve of the inter-reference distance.

Figure 5-11 shows the cumulative distribution of inter-reference distances and deliver distances. The X axis shows the distance in the number of memory references, and the Y axis shows the normalized ratio (*F(R)/F(∞)* for the inter-reference distance, and *D(R)/D(∞)* for the deliver distance). We assume that the number of processors is 16 and that the line size is 16 bytes. Arrows in Figure 5-11 indicate the reference interval that corresponds to the largest working set (WS2) and the second largest working set (WS1).

Figure 5-11 indicates that, for most of the applications, the cumulative distribution of deliver distances reaches 100% when the deliver distance is about the same as the reference interval for WS2. Thus, the cache needs to be as large as WS2 to capture most of the delivered lines until they are used. This directly explains the miss-rate behavior of the

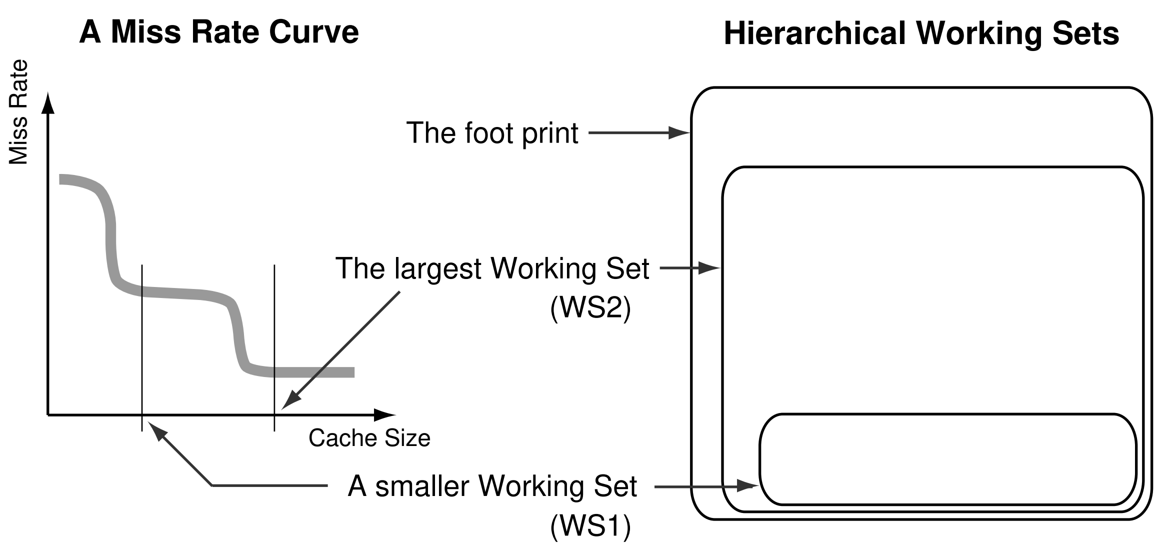

Figure 5-10: Miss Rate Curve and Hierarchical Working Sets.

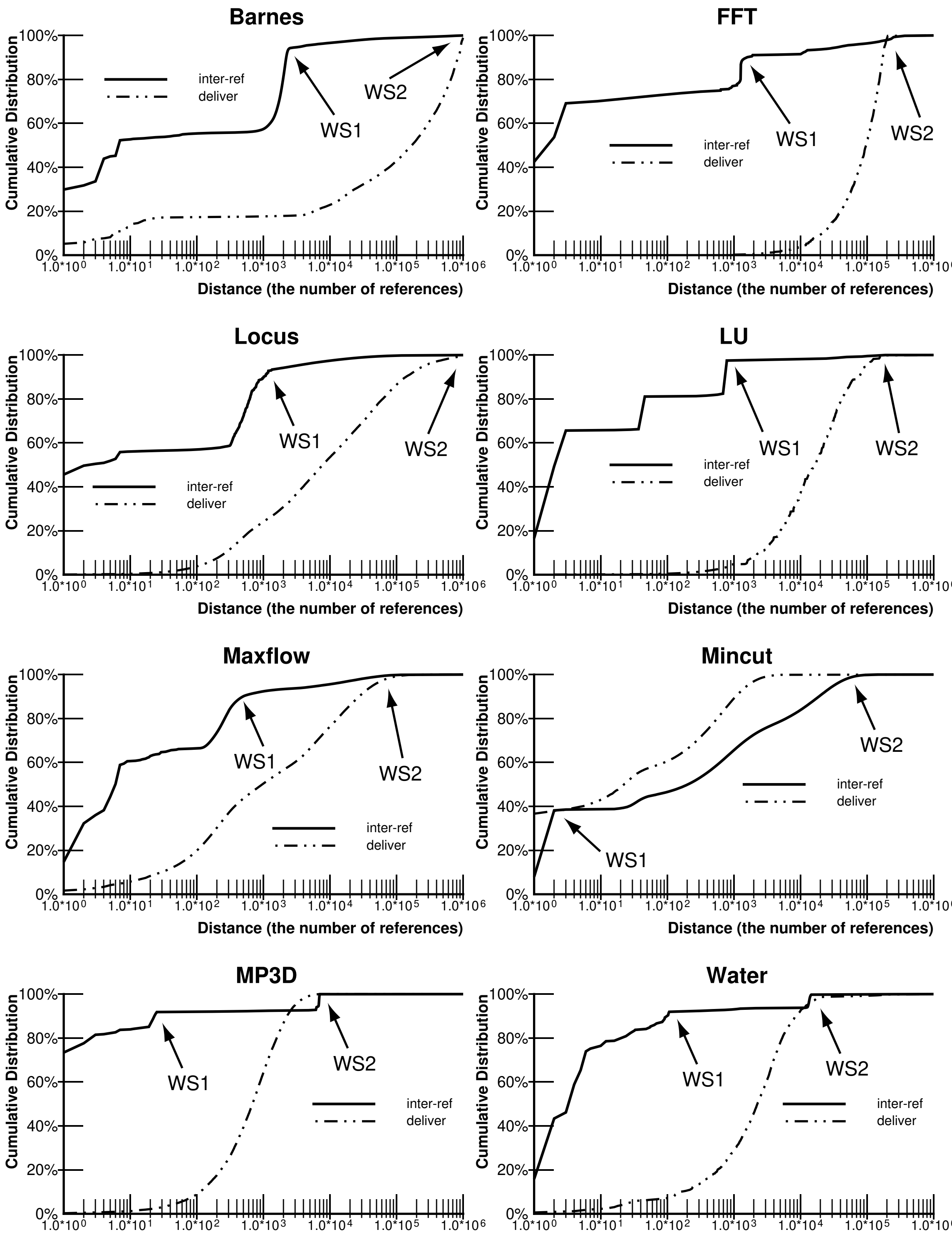

Figure 5-11: Cumulative Distribution of Inter-Reference Distances and Deliver Distances.

deliver scheme that we discussed in Subsection 5.1.5; the cache needs to be as large as the largest working-set to obtain most of the benefit due to deliver operations for most of our applications. Mincut exhibits a different distribution pattern from the rest of the applications. For Mincut, the cumulative distribution of deliver distances reaches 100% for a distance that is much shorter than the reference interval for WS2. This explains the miss rate curve of Mincut in Figure 5-8; for Mincut, deliver operations can eliminate bulk of sharing misses even when the cache size is much smaller than the largest working-set.

Now, we intuitively discuss why the deliver distance is typically much longer than the reference interval for WS1. The largest working-set (WS2) generally corresponds to the data set that the processor accesses in the outermost loop, and the second largest working-set (WS1) generally corresponds to the data set that the processor accesses in a second-outermost loop. Thus, the reference interval for each working-set roughly represents the number of memory references in the corresponding loop. In Subsection 5.1.2, we have discussed typical loop structures that explain the temporal characteristics of deliver operations (see Figure 5-5). We have discussed that the consumer processor generally does not use a produced value during the same instance of the second outermost loop in which another processor produced the value. Therefore, the deliver distance generally becomes much longer than the reference interval for WS1.

5.1.7 Cache Associativity

So far we have assumed fully associative caches to exclude the artifacts due to small associativity from our discussions. In this subsection, we examine the effect of small associativity that we observe in real machines. Small associativity causes mapping conflicts, so that the miss rate of set-associative caches is generally higher than that of fully associative caches. The deliver scheme especially suffers from mapping conflicts because deliver operations transfer cache lines very early before the receiver uses them so that delivered lines are likely replaced due to mapping conflicts. In this subsection, first, we pick FFT as an example and examine the impact of small associativity for several protocols. Second, we analyze the optimistic deliver protocol and discuss whether small associativity affects the cache size that the deliver operation needs to reduce a significant number of cache misses.

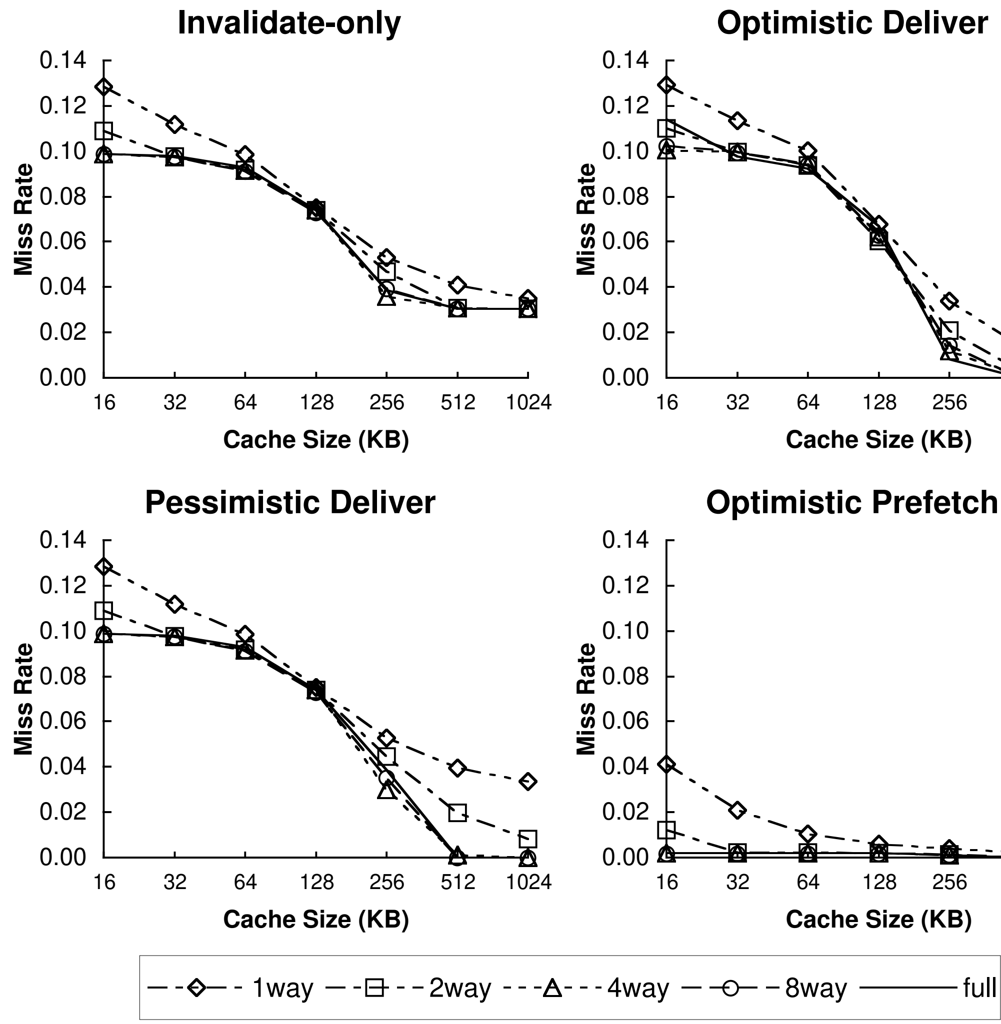

Figure 5-12: Miss Rate of Four Protocols versus Cache Size for Various Associativities. The application is FFT.

Figure 5-12 shows the miss rate of FFT for four protocols — invalidate-only, optimistic deliver, pessimistic deliver, and optimistic prefetch — when the line size is 16 bytes. The knee of the miss-rate curve for the invalidate-only protocol moves toward right as the associativity decreases. This indicates that the cache needs to be larger to hold the working set as the associativity decreases. This is because mapping conflicts due to small associativity prevent caches from utilizing the cache memory efficiently. Figure 5-12 also shows that the optimistic prefetch scheme suffers much less from small associativity than the deliver scheme with either replacement policies because prefetch operations transfer cache lines much later than deliver operations. If the associativity is four or larger, however, the impact of mapping conflicts on the deliver scheme with either replacement policies is so small that the miss rate is almost the same as that of fully associative caches.

In Subsection 5.1.5, we showed why the deliver operation needs a large cache — roughly as large as the largest working set — to reduce the miss rate significantly. An interesting question is whether the deliver operation needs even a larger cache to reduce the miss rate significantly as the associativity decreases. Figure 5-13 answers this question. Figure 5-13 shows the normalized miss rate of the optimistic deliver scheme for 16-byte lines; that is the miss rate of the optimistic deliver scheme divided by that of the invalidate-only protocol. The normalized miss rate usually increases as the associativity decreases, so that we need a larger cache to reduce the same ratio of cache misses of the invalidate-only protocol as the associativity decreases. This is again because deliver operations transfer cache lines very early before the local processor uses, so that the delivered lines suffer more from mapping conflicts than the cache lines that the local processor fetches. When the cache size is about the same as the largest working-set size, the impact of small associativity is especially large. This is because caches with small associativity do not use cache memory efficiently, hence associativity is critical for caches to hold the working set when the cache size is about the same as the size of the working set. If the cache size is much smaller or larger than the size of the working set, the impact of small associativity is generally small.

5.2 Line Size

In Section 5.1, we assumed a very small line size, 16 bytes, to simplify our discussions. In this section, we analyze the effect of large line sizes for deliver and prefetch operations. We can consider large cache lines as a hardware mechanism that provides simple prefetching. As the line size increases, the miss rate usually decreases because a single miss can "prefetch" additional data that the processor may use later. Ideally, when the line size is doubled, the miss rate is halved. The miss rate, however, may not be halved because the processor does not necessarily use the prefetched data before they are replaced. When the line size increases, moreover, false sharing may invalidate cache lines before they are used. Similarly, when the line size increases, false sharing may invalidate delivered or prefetched lines. For some applications, because of false sharing, the miss rate of the

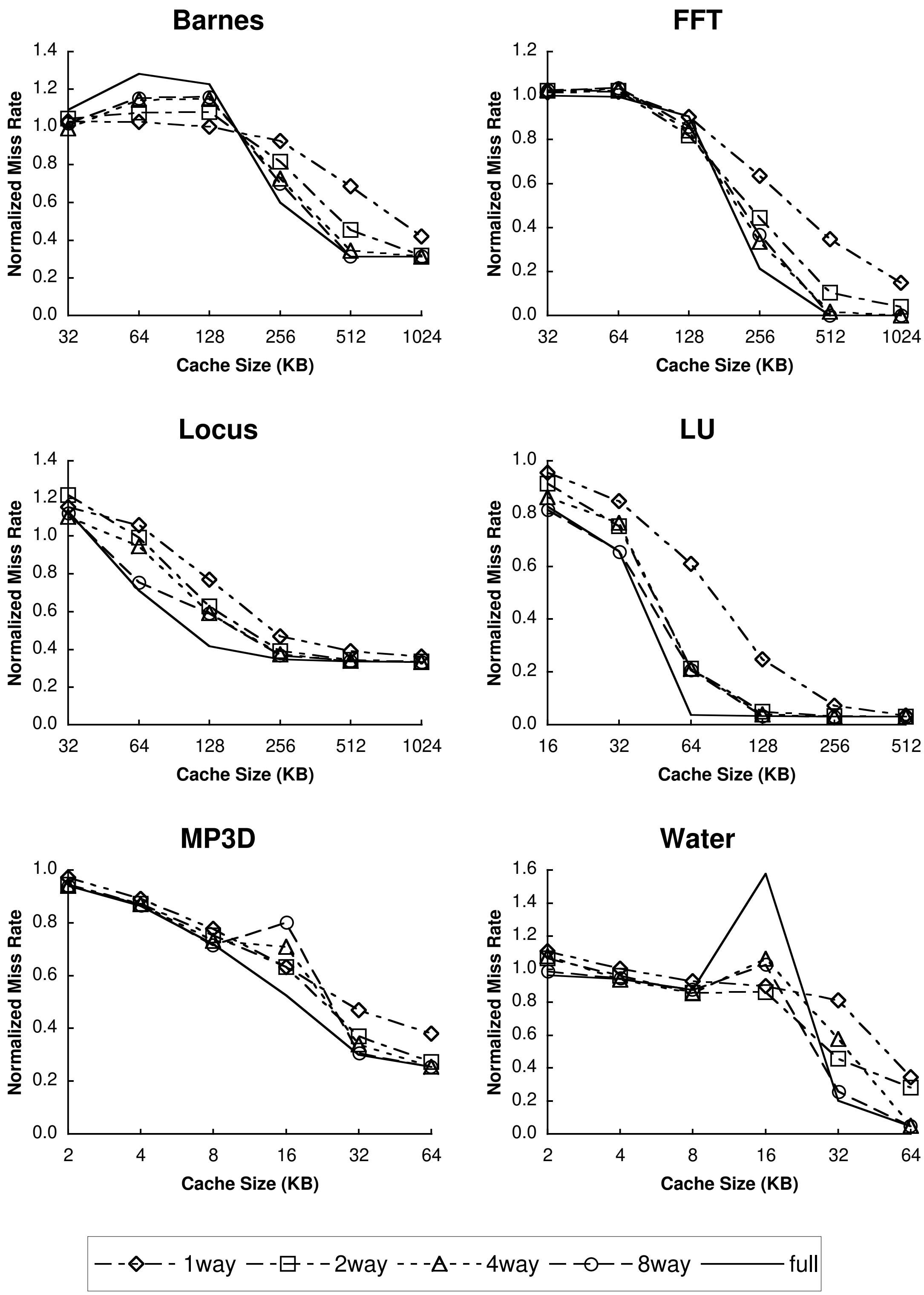

Figure 5-13: Miss Rate of Optimistic Deliver versus Cache Size for Various Associativities. The miss rate is normalized by that of the invalidate-only protocol.

prefetch or the deliver scheme does not decrease proportionally as the miss rate of the invalidate-only mechanism decreases. In this section, we discuss sharing patterns that interfere with deliver or prefetch operations.

Since cache misses are a result of interactions among the spatial locality, the temporal locality, and the sharing activity, the effect of large line sizes for the miss rate depends on the cache size. In this section, therefore, we first discuss the effect of large line sizes for a large cache size to isolate the effect of small cache sizes. Then, we discuss additional effects of large line sizes when the cache size varies.

5.2.1 Effect of Large Lines for Large Caches

Figure 5-14 shows the miss rate for our benchmark applications when we vary the line size from 16 to 128 bytes. We assume fully-associative caches. The bar graph shows the miss rate of the invalidate-only protocol. The line graph shows the normalized miss-rate — the miss rate divided by that of the invalidate-only protocol — for optimistic deliver and optimistic prefetch. The cache size is so large that we can ignore cache conflict effects. Thus, the miss rate of optimistic deliver should be about the same as the that of pessimistic deliver.

The miss-rate graph of the invalidate-only protocol (the bar graph in Figure 5-14) indicates the degree of the spatial locality of the application; the application has good spatial locality if doubling the line size nearly halves the miss rate of the invalidate-only protocol. In our applications, FFT, LU, to some extent, Locus, and Water have good spatial locality. The normalized miss-rate graph of the deliver and prefetch protocols (the line graph in Figure 5-14) on the other hand, indicates the degree of the interference for delivered and prefetched lines due to false sharing; the normalized miss-rate curve is flat if false sharing does not invalidate delivered or prefetched lines before they are used. In our applications, FFT, LU, to some extent, Maxflow, Mincut, and Water do not suffer from the interference for both deliver and prefetch schemes.

Note that the behavior of the invalidate-only protocol does not give us good insight into the behavior of the deliver and prefetch schemes. For example, the miss-rate graph of the invalidate-only protocol indicates that Locus has better spatial locality than Maxflow and

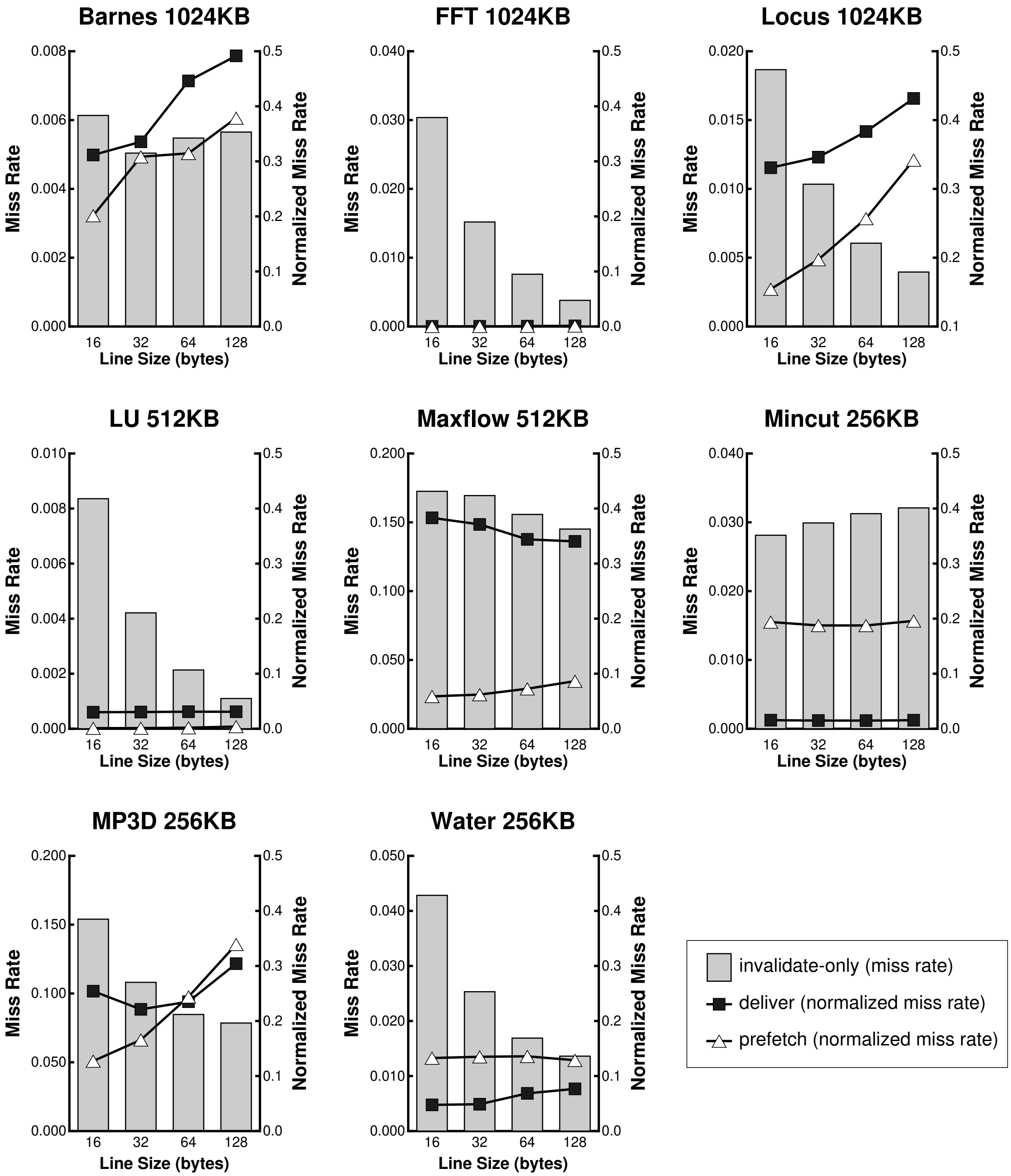

Figure 5-14: Miss Rate versus Line Size for a Large Cache. The bar graph shows the miss rate of the invalidate-only protocol. The line graph shows the normalized miss-rate (the miss rate divided by that of the invalidate-only protocol) for the deliver and prefetch protocols. The optimistic replacement policy is used for both protocols. The cache size is shown at the top of each graph and is large enough to eliminate capacity misses.

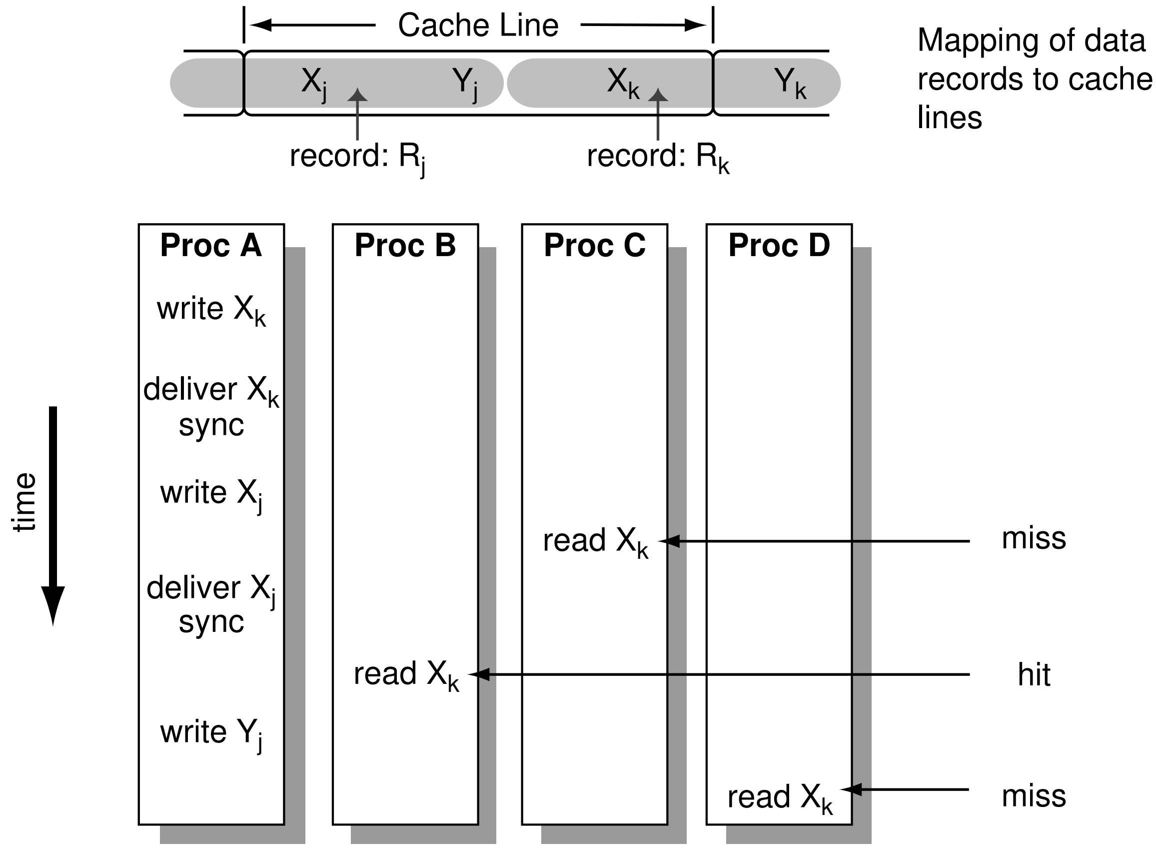

Figure 5-15: False Sharing with Deliver Operations. Proc B does not cause a miss because the deliver operation for X_j also sends X_k to Proc B. Proc C causes a miss because Proc A invalidates the delivered X_k. Proc A does not deliver Y_j because of a single-processor reuse of Y_j, so that Proc D causes a miss when Proc D reads X_k.

Mincut. The normalized miss-rate graph of the deliver and prefetch protocols, however, indicates that Locus suffers more from the interference due to false sharing than Maxflow and Mincut. This is because deliver and prefetch operations cannot always take advantage of the spatial locality of the application. For the rest of this subsection, we discuss sharing patterns that interfere with delivered or prefetched lines.

First, we focus on the deliver scheme. Figure 5-15 illustrates three sharing patterns that we consider. Proc A is a producer of the two records (R_j and R_k) and the other three processors are a consumer. Because producer-consumer communication occurs through X_j and X_k, we insert a deliver operation before consumers access them. We do not, however, insert a deliver operation for Y_j because only the producer uses it (i.e., single-processor reuse).

If the line size is small enough, the three consumers do not cause a cache miss for reading X_k since Proc A delivers X_k to them. If the line size is large enough to hold X_j, Y_j, and X_k as shown in Figure 5-15, however, Proc C and Proc D cause a cache miss for reading X_k since Proc A invalidates the delivered line because of false sharing. Proc B still does not cause a cache miss for reading X_k since the deliver operation for X_j sends X_k to Proc B. This is because our deliver mechanism relies on a cache directory so that a cache line is sent to processors that have accessed the cache line because of false sharing. This access pattern occurs in Mincut and Maxflow. Thus, although false sharing occurs, the normalized miss-rate curve of the deliver scheme is almost flat as shown in Figure 5-14 for the two applications.

The access pattern similar to the Proc C's read occurs in Locus. In Locus, when a processor updates CostArray elements, we delay sending a deliver message to aggregate all updates for the same cache line into a single deliver message. As the line size increases, the delay increases. This increases the chance that another processor accesses the cache line before the producer sends the deliver message. The access pattern similar to the Proc D's read occurs in Barnes. Proc D causes a cache miss for reading X_k, because Proc A invalidates the delivered copy of X_k when Proc A writes Y_j. While the access pattern of Y_j is single-processor reuse, the access pattern of the cache line is not single-processor reuse because of false sharing, so that Proc A needs to deliver the cache line to Proc D after writing Y_j to prevent the cache miss at Proc D. For Barnes, because of this type of sharing pattern, the normalized miss-rate of the deliver scheme increases as the line size increases. For both sharing patterns of Proc C and Proc D, if cache-line-level sharing information is available, we can add deliver operations at appropriate places — immediately after "write X_j" and "write Y_j" in Figure 5-15 — to eliminate the cache misses due to false sharing.

Now, we examine the effect of large lines for the prefetch scheme. False sharing also reduces the benefit of prefetch operations as illustrated in Figure 5-16. A false sharing miss occurs when Proc B reads a prefetched word X_k because Proc A writes another word in the same cache line. This type of misses occurs for Barnes, Locus, and MP3D. We schedule prefetch operations early enough to hide memory latencies. As we discussed in Subsection 5.1.2, the prefetch distance is typically 10^2 to 10^3 clocks. If another processor

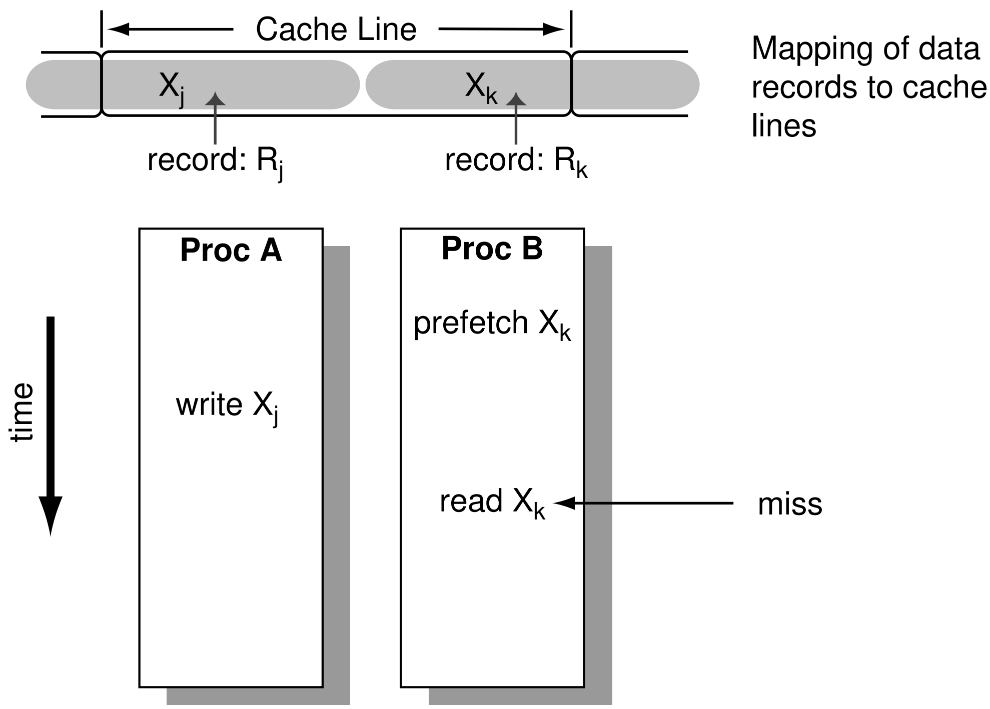

Figure 5-16: False Sharing with Prefetch Operations. Proc B causes a cache miss because Proc A invalidates the prefetched X_k.

writes a value in the same cache line during this period, the prefetch operation becomes useless. If processors use exclusive-mode prefetching, the impact of this access pattern increases since the probability of the invalidation due to false sharing increases. As the memory latency or the line size increases, moreover, we encounter a dilemma: whether to schedule prefetch operations early to hide the latency or late to reduce the false sharing effect. If we schedule prefetch operations early, we potentially hide the memory latency completely but may increase the false sharing effect. Note that adding deliver operations can reduce the number of false sharing misses at the expense of additional deliver traffic. Although the miss rate of the prefetch scheme is smaller than that of the deliver scheme except for a few applications as shown in Figure 5-14, it will become more difficult for the prefetch operation to reduce the false sharing effect than the deliver operation when the line size and the memory latency become very large.

5.2.2 Interactions Between Line-Size and Cache-Size Effects

We have discussed cache parameter effects in one-dimensional spaces: cache size effects for a small line size (Subsection 5.1.5) and line size effects for a large cache size (Subsection 5.2.1). In this subsection, we extend our scope to a two-dimensional space of cache and line sizes. Figure 5-17 and Figure 5-18 show the normalized miss-rate of optimistic deliver and optimistic prefetch for four line sizes from 16 bytes to 128 bytes. The X axis shows the cache size, and the Y axis shows the normalized miss-rate (normalized by the miss rate of the invalidate-only protocol with the same cache size and the same line size). We assume fully-associative caches.

In the previous subsection, we discussed that increasing the line size causes false sharing that interferes with delivered or prefetched lines for some applications. Varying the cache size adds two effects for the exploitation of the spatial locality in the deliver scheme. First, as the cache size decreases, the number of capacity misses increases for the deliver scheme. For some applications, capacity and sharing misses occur for memory accesses with different spatial locality (e.g., different data sets or the same data sets with different access patterns). For those applications, the effect of large cache lines varies as the cache size varies. Second, as the line size increases, the utilization of cache lines generally decreases. As we discussed in Subsection 5.1.5, the cache needs to be roughly large enough to capture the largest working set so that deliver scheme can eliminate most of the cache misses. If the utilization of cache lines decreases, the cache size needs to increase to capture the working set.

For FFT and LU, the normalized miss-rate of the deliver scheme does not change noticeably when the line size increases at all cache sizes that we examine, as shown in Figure 5-17. This is because the two above-mentioned effects due to cache size variations do not occur; memory access patterns that cause cache misses have good spatial locality for all cache sizes that we examine, and the utilization of cache lines does not decrease as the line size increases. For the two applications, the normalized miss-rate of the prefetch scheme does not change as the line size increases since the prefetch scheme eliminates virtually all cache misses.

For Locus and MP3D, as we discussed in Subsection 5.2.1, the normalized miss-rate of prefetch increases more significantly than that of deliver when the line size increases because exclusive-mode prefetching enlarges the impact of false sharing. This is true for all cache sizes that are shown in Figure 5-17 and Figure 5-18. As long as the cache size is

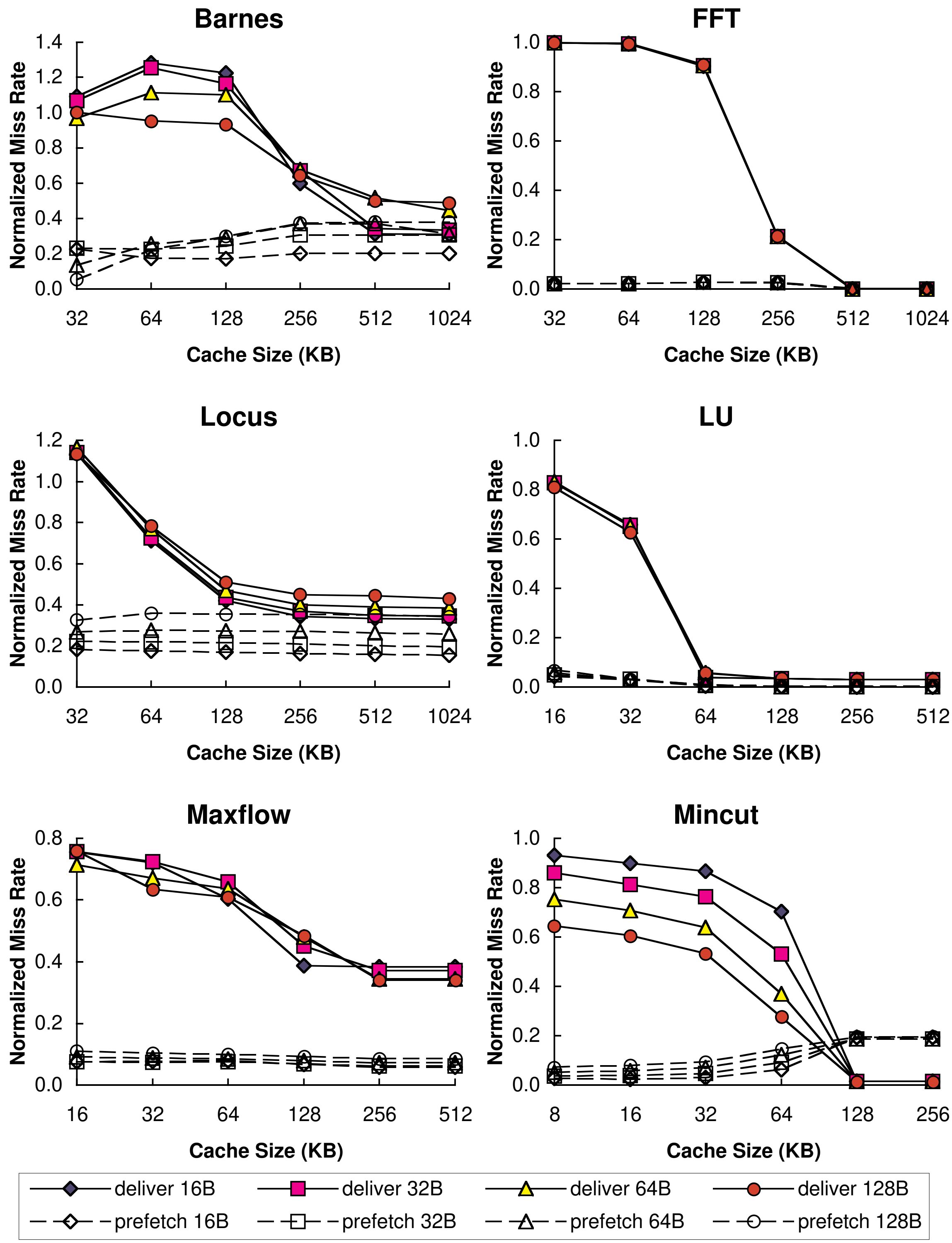

Figure 5-17: Miss Rate of Optimistic Deliver and Optimistic Prefetch for Cache and Line Size Variations. The miss rate is normalized by that of the invalidate-only protocol with the same cache size and the same line size.

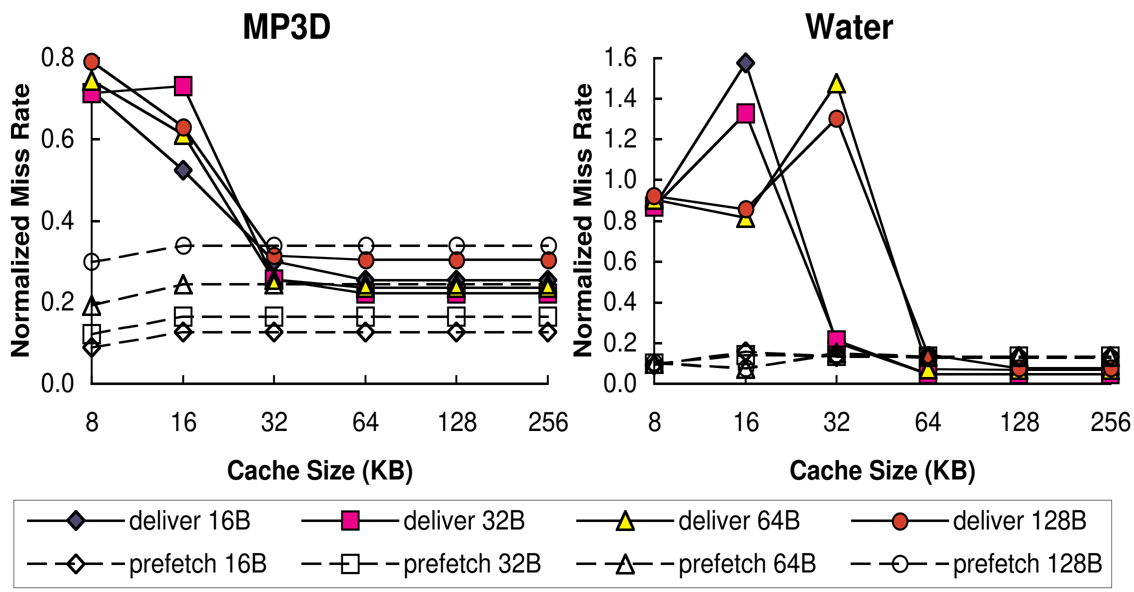

Figure 5-18: Miss Rate of Optimistic Deliver and Optimistic Prefetch for Cache and Line Size Variations. (Continued.) The miss rate is normalized by that of the invalidate-only protocol with the same cache size and the same line size.

smaller than the largest working set, however, the prefetch scheme still generates a smaller miss rate than the deliver scheme since the deliver scheme cannot eliminate capacity misses.

For Mincut, the normalized miss-rate curves show interesting characteristics. When the cache size is larger than the largest working set (128 Kbytes), the normalized miss-rate of the prefetch and deliver protocols does not change as the line size increases. When the cache size is smaller than the working set, on the other hand, as the line size increases, the normalized miss-rate of the deliver protocol decreases while that of the prefetch protocol increases. This is because the spatial locality of memory accesses that cause remaining misses differs between the deliver and prefetch schemes. In the deliver scheme, most of the remaining misses are capacity misses for data records that exhibit good spatial locality. In the prefetch scheme, by contrast, most of the remaining misses are sharing misses for data records that exhibit poor spatial locality. Therefore, when the cache size is smaller than the largest working set, the exploitation of the spatial locality differs between the deliver and prefetch schemes for Mincut.

For Barnes, the normalized miss-rate curve of the deliver scheme in Figure 5-17 indicates that the effect of large cache lines depends on the cache size. When the cache size is larger than the largest working set (256 Kbytes), the normalized miss-rate increases when the line size increases. This is because, as discussed in Subsection 5.2.1, increasing the line size causes false sharing that interferes with delivered lines. When the cache size is smaller than the working set, on the other hand, the normalized miss-rate decreases when the line size increases for the deliver scheme. This is because, similar to the deliver scheme in Mincut, most of the cache misses are capacity misses for data records that have relatively good spatial locality.

For Water, the normalized miss-rate curve of the deliver scheme exhibits a very different pattern from the rest of the applications. As shown in Figure 5-18, the curve has a sharp peak when the cache size is slightly smaller than the largest working set. At this cache size, the cache almost captures the working set when we use the invalidate-only protocol, so that the miss rate is significantly smaller than that for a slightly smaller cache. When we use deliver operations, however, the cache cannot capture the working set. This is because processors do not use about 87% of delivered lines so that the useful space in the cache is much smaller than the actual size. Thus, the normalized miss-rate of the deliver scheme becomes significantly large when the cache size is slightly smaller than the working set. Furthermore, the cache size that produces a sharp peak increases when the line size increases as shown in Figure 5-18. This is because the utilization of cache lines decreases so that the cache size needs to increase to capture the same working set. The decrease of the cache-line utilization also occurs for Maxflow. As shown in Figure 5-17, a knee of the normalized miss-rate curve for the deliver scheme moves from 128 Kbytes to 256 Kbytes when the line size increases from 16 bytes to 32 bytes because of the decrease of the cache-line utilization.

5.3 Memory/Network Bandwidth

In this section, we analyze the characteristics of the memory and network traffic for our applications by examining the simulation results of a unit-delay memory model, and we compare the demand traffic with an upper-bound bandwidth of a feasible memory system.

We vary three architectural parameters — cache size, line size, and the number of processors — to analyze their effects for the traffic behavior.

As discussed in Chapter 4, the deliver scheme usually generates much more traffic than the invalidate-only and prefetch protocols for applications with dynamic producer-consumer patterns. The simulation results, however, show that the demand traffic with the deliver protocol is within the upper-bound bandwidth for most of our applications when the number of processors is 16. As the number of processors increases, the ratio between the traffic and the bandwidth increases dramatically for applications with dynamic producer-consumer patterns. As the line size increase up to 128 bytes, the ratio increases only slightly or decreases for most of our applications.

The simulation results also show that the pessimistic deliver protocol produces significantly less traffic than the optimistic deliver protocol unless the cache size is much larger than the largest working set of the application. If the cache size is much smaller than the largest working set, however, the pessimistic deliver protocol does not significantly reduce the miss rate, as we discussed in Subsection 5.1.5. Thus, the pessimistic deliver protocol may be an attractive solution only when the cache is about the same as the largest working set.

In the following subsections, we first discuss our assumptions for the processor-node and the cache protocol. Second, we examine the traffic characteristics for major system components in the processor node when the cache size varies, and we compare the traffic with a feasible bandwidth for each component. We discuss the producer-consumer relation to understand the traffic behavior. Third, we examine the traffic characteristics when the line size varies. Fourth, we discuss the traffic characteristics when the number of processors varies. Finally, we summarize our discussions about the traffic characteristics of our applications.

5.3.1 Architectural Model

The multiprocessor system consists of a set of network-connected processor nodes. Figure 5-19 shows the processor node structure that we assume. Each processor node consists of six major components: a processor, a cache, a directory controller, a directory memory, a

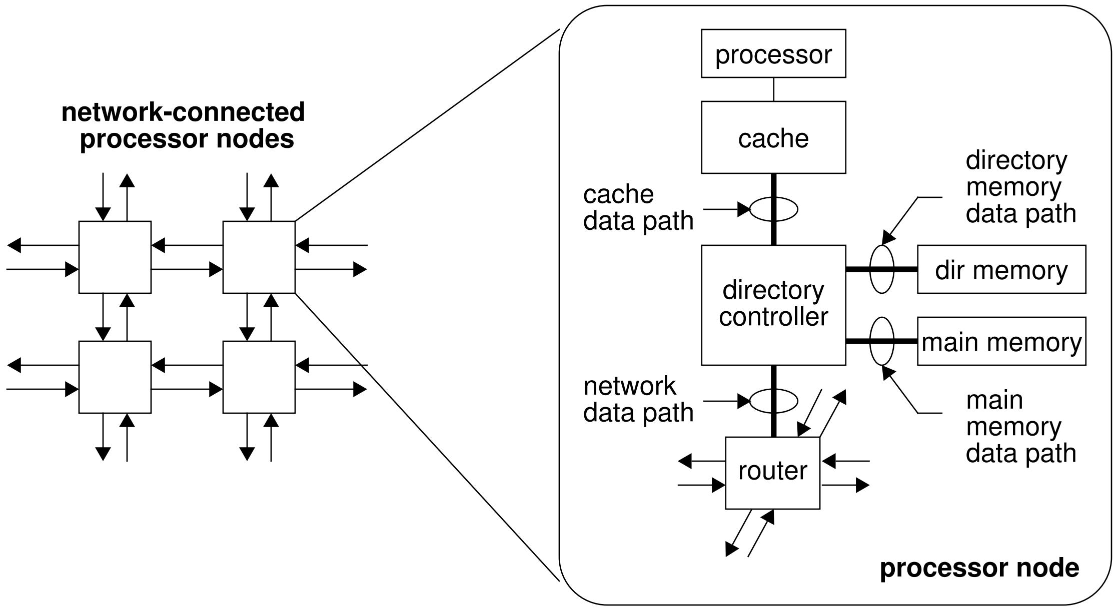

Figure 5-19: Processor Node Model.

main memory, and a network router. Each data path between processor-node components has a finite bandwidth and becomes a bottleneck if the demand traffic for the data path exceeds its bandwidth. We focus on the four data paths between the directory controller and another component, as illustrated in Figure 5-19, since the traffic of these data paths is directly relating to the sharing pattern of applications as well as the behavior of deliver and prefetch operations.

We assume that each processor cache is kept coherent using a directory-based Illinois protocol [48]; each cache line is in one of invalid, shared, dirty, or valid-exclusive state. A cache-directory maintains the state of each memory block and keeps track of processors that have a copy of the block. The memory block is in one of clean, dirty, or valid-exclusive state. If a cache line in a processor-cache is in dirty or valid-exclusive state, the corresponding memory block in the directory is in dirty or valid-exclusive state, respectively. Otherwise, the memory block is in clean state.

Our directory-based Illinois protocol has three assumptions that are not included in bus-based Illinois protocols. First, we assume that processor caches send a replacement hint to the directory when they replace a clean cache line so that the directory does not need to send invalidation requests to processor caches that no longer have the copy. Second, the directory collects acknowledge messages of write and deliver operations to determine the completion of the operation. Third, processors can write a valid-exclusive cache line without accessing the directory only if the home node of the cache line is the local node. In other words, when processor caches have a valid-exclusive line, we assume that the cache has an ownership of the line only if the home of the cache line is the same as the node of the processor.

Table 5-1 shows the number of transactions that our cache protocol generates at each processor-node component for different type of memory accesses. For main and cache memory transactions, the table shows the number of transferred cache lines per memory operation. For main memory transactions, Table 5-1 shows two different numbers for some conditions. The numbers that are not enclosed in a parenthesis indicate the number of memory transactions if the directory controller sequentially accesses the directory and the main memory. In this case, the directory controller accesses the main memory only if the main memory has a latest value of the cache line. The numbers that are enclosed in parentheses, on the other hand, indicate the number of extra memory transactions if the directory controller speculatively accesses the main memory before the controller finds that the main memory has a latest value. The speculative access makes the memory access time shorter but increases the main memory traffic. For directory transactions, the table shows the number of directory transactions per memory operation. We assume that each acknowledge message for write and deliver operations causes one directory transaction. For network transactions, the table shows the number of transferred cache lines and headers per memory operation. Headers include those for data packets and those for control (header-only) packets. The numbers of network transactions correspond to the traffic for a single direction (incoming or outgoing) of the data path between the directory controller and the network router. We calculate the network traffic in bytes per instruction for different line sizes, assuming that the header size is 8 bytes. Prefetch and prefetch-exclusive operations generate the same number of transactions as read and write operations, respectively.

Figure 5-20 shows our data transfer model for the cache and main memory. Each access to the cache and main memory includes a fixed occupancy time and data transfer cycles. The

memory state	home/owner location	access type	main memory[a]	directory memory	cache memory	network	
						cache line	header
clean	home = requester	read	1	1	1	0	0
		write	1	1 + S	1	0	2 S’
		upgrade	0	1 + S	0	0	2 S’
	home requester	read	1	1	1	1	2
		write	1	1 + S	1	1	2 S’ + 3
		upgrade	0	1 + S	0	0	2 S’ + 3
dirty	home = requester owner home	read	1 (1)	2	2	1	2
		write	0 (1)	2	2	1	2
	home requester owner = home	read	1 (1)	2	2	1	2
		write	0 (1)	2	2	1	2
	home requester owner home	read	1 (1)	2	2	2	4
		write	0 (1)	2	2	2	4
valid exclu-sive	home = requester owner home	read	1	1	1	0	0
		write	1	1 + S	1	0	2 S’
	home requester owner = home	read	0 (1)	2	2	1	2
		write	0 (1)	2	2	1	2
	home requester owner home	read	1	1	1	1	2
		write	1	1 + S	1	1	2 S’ + 3

cache state	home/owner location	access type	main memory	directory memory	cache memory	network	
						cache line	header
shared	home = requester	replace	0	1	0	0	0
	home requester	replace	0	1	0	0	1
dirty	home = requester	replace	1	1	1	0	0
	home requester	replace	1	1	1	1	1
—	home = requester	deliver	1	1 + S	1 + S	S’	2 S’
	home requester	deliver	1	1 + S	1 + S	1 + S’	2 S’ + 1

Table 5-1: Traffic Table for read, write, upgrade, replace, and deliver operations.[b]

a. The numbers in the parenthesis indicate the number of extra memory operations when the directory controller speculatively accesses the main memory.

b. S denotes the number of sharing processors except for the requester, and S’ denotes the number of sharing processors except for the requester and the home node. For optimistic deliver operations, sharing processors mean processors that have shared the cache line. For pessimistic deliver operations, sharing processors mean processors that have an invalidated copy of the cache line.

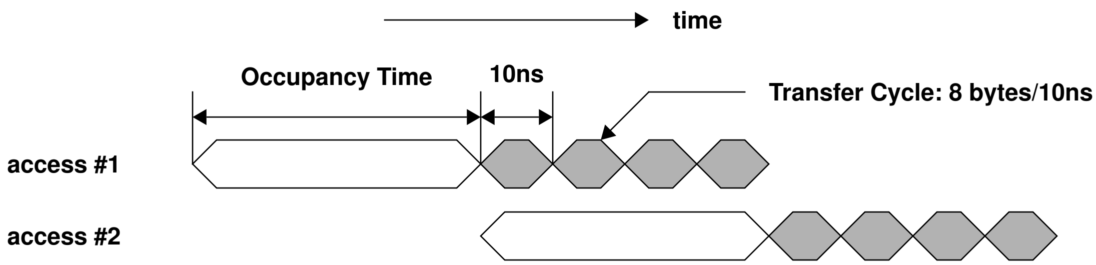

Figure 5-20: Data Transfer Model for Cache and Main Memory.

occupancy time represents the bandwidth of the memory, and the data transfer cycles represent the bandwidth of the data path for the memory.[8] We assume that the data path is 8 bytes and each transfer takes 10ns. (i.e., the maximum bandwidth is 800 Mbytes/sec.) Memory accesses are pipelined so that the memory can process the next access during the data transfer. Thus, the bandwidth is the line size divided by the occupancy time, if the data transfer time is shorter than the occupancy time. The bandwidth increases up to the maximum bandwidth as the line size increases until a certain size for which the transfer time becomes equal to the occupancy time. The directory bandwidth, on the other hand, is determined only by the occupancy time of the directory memory since a directory access transfers a fixed amount of information. Figure 5-21 illustrates the bandwidth of the cache and main memory and the directory when the access latency varies. The X axis shows the occupancy time of one transaction, and the Y axis shows the bandwidth. For the cache and main memory traffic, we count the traffic in the number of bytes per instruction. For the directory traffic, we count the traffic in the number of transactions per instruction. We assume that processors execute 200 million instructions per second.

The occupancy time strongly depends on the technology of the implementation. For the main memory, it is probably feasible to have 100ns occupancy time with a DRAM technology. For the cache memory, it is probably feasible to have 40ns occupancy time with a SRAM technology. Thus, the main memory bandwidth reaches the maximum bandwidth (800 Mbytes/sec = 4 bytes/instruction) when the line size is 128 bytes, and the cache memory bandwidth reaches the maximum bandwidth when the line size is 32 bytes. For

8. In this section, we discuss only the bandwidth, not the latency. We will discuss the latency in Section 5.4.

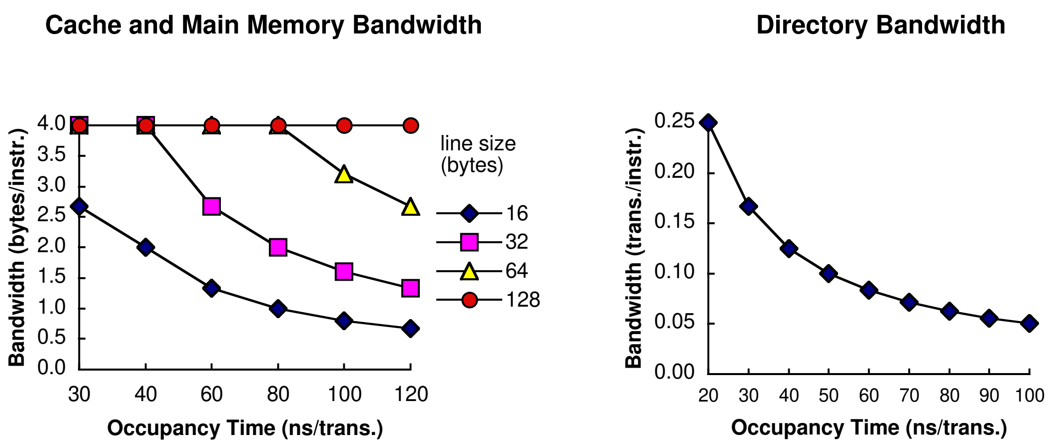

Figure 5-21: Bandwidth versus Occupancy Time. Processor = 200 M instructions/sec.

simplicity, we assume that the cache memory is dual-ported so that cache accesses from the local processor do not affect the bandwidth for cache accesses from the directory controller.[9] For the directory, we assume an aggressive design (e.g., FLASH Architecture [34, 38]) in which the directory information is cached in a SRAM. We assume that the directory occupancy time is 60ns. This occupancy time provides 0.08 transactions/instruction bandwidth. The directory bandwidth would be smaller in a traditional implementation that uses only a DRAM technology to store directory information. (If the occupancy time of the DRAM is 100ns, the bandwidth is 0.05 transactions/instruction.) For the network, we assume that the data path between the directory controller and the network route is bidirectional and the network data path transfers 4 bytes per 10ns (400 Mbytes/sec = 2 bytes/instruction) for each direction.

The bandwidth of these four system components provides a reasonable upper-bound bandwidth of a feasible memory system, which is fairly independent from the implementation of the directory controller. For real implementations, the actual memory bandwidth may be smaller than the one that we assume since some data or control paths inside the directory controller may become a bottleneck. Detailed design issues of the directory controller are, however, beyond the scope of this thesis. We compare the demand traffic of our appli-

9. Other components (e.g., main memory and directory) can be also multi-ported (or multi-banked) to provide large bandwidth. For simplicity, however, we assume that other components serve one transaction at a time.

cations with the bandwidth for each system component to examine the feasibility of the deliver and prefetch schemes.

5.3.2 Traffic versus Cache Size

In this subsection, first, we focus on the traffic characteristics for FFT and Locus, which have distinct producer-consumer patterns. Later, we extend our discussions to other applications. Figure 5-22 shows the traffic at the four system components — the directory, the main memory, the cache memory, and the network — for FFT and Locus. The cache is fully associative, and the line size is 16 bytes. In Figure 5-22, the main memory traffic does not include extra transactions due to speculative memory accesses. The X axis shows the cache size, and the Y axis shows the traffic in the number of transactions per instruction for the directory traffic and in the number of bytes per instruction for the rest. Four lines of each graph in Figure 5-22 correspond to four protocols (i.e., invalidate-only, optimistic prefetch, optimistic deliver, and pessimistic deliver). FFT represents applications that have a static consumer-producer pattern. Locus, on the other hand, represents applications that have a dynamically changing consumer-producer pattern.

For FFT and Locus, the optimistic prefetch and invalidate-only protocols produce about the same amount of traffic for the four system components, as shown in Figure 5-22. This is because processors use virtually all of the prefetched lines so that prefetch operations do not generate unnecessary traffic. For Locus, the optimistic prefetch protocol actually produces slightly less traffic than the invalidate-only protocol except for the cache memory traffic. This is because Locus uses exclusive-mode prefetching. As discussed in [45], exclusive-mode prefetching reduces the number of directory lookups and the number of network packets by combining read-miss and write-upgrade operations. Exclusive-mode prefetching also reduces the number of write-backs at the main memory because exclusive-mode prefetch operations do not write dirty lines back to the main memory while read-miss operations do.

5.3.2.1 Traffic Characteristics of FFT — a static producer-consumer pattern

Let's examine the traffic characteristics of the optimistic and pessimistic deliver protocols. First, we focus on the traffic behavior of FFT. Since FFT has a static one-to-one producer-

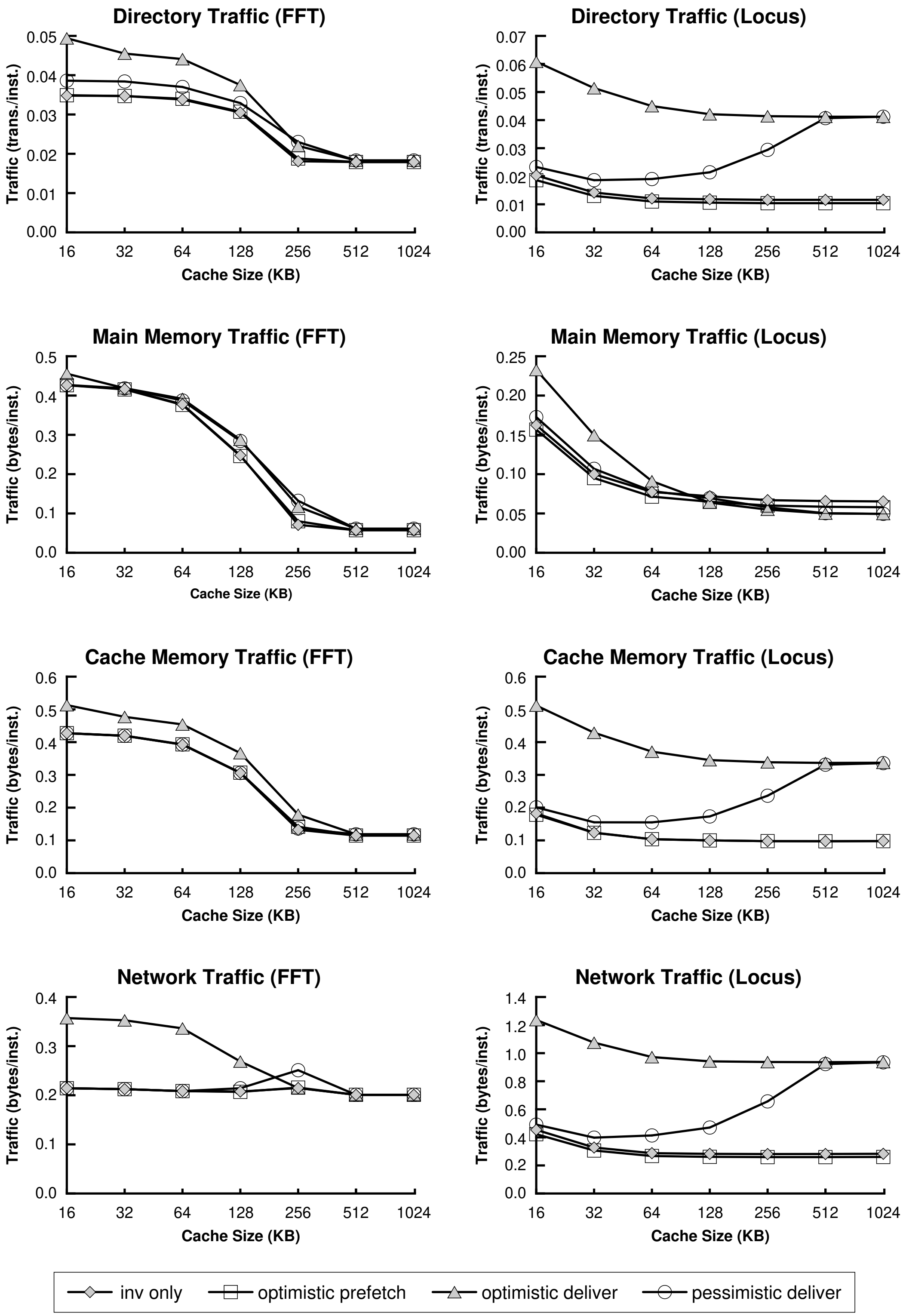

Figure 5-22: Directory, Main Memory, Cache Memory, and Network Traffic of versus Cache Size for FFT and Locus.

consumer pattern, a producer sends cache lines only to a consumer that uses the cache lines. As long as the consumer's cache is large enough to keep the received cache lines until they are used, the optimistic and pessimistic deliver protocols do not produce extra traffic over the invalidate-only protocol.

In the optimistic deliver protocol, a deliver operation always sends a cache line to the consumer that has used the line. If the cache size is smaller than the largest working set, however, the consumer does not keep all of the delivered cache lines in the cache until they are used and fetches some of the delivered lines when the consumer needs them. Thus, although the communication pattern is regular, the optimistic deliver protocol produces extra traffic for the directory, the cache memory, and the network when the cache size is smaller than the largest working set, as shown in Figure 5-22. For the main memory traffic, on the other hand, the optimistic deliver protocol produces only a small amount of extra traffic. This is because both the optimistic deliver and the invalidate-only protocols usually generate two memory operations for each cache transfer between processors when the cache size is smaller than the largest working set. For the deliver protocol, a deliver operation writes the produced value back to the main memory and a consumer reads the value from the main memory. For the invalidate-only protocol, on the other hand, the produced value is replaced from the producer's cache and written back to the main memory, and the consumer reads the value from the main memory. Thus, as shown in Figure 5-22, the optimistic deliver and invalidate-only protocols produce about the same amount of the main memory traffic.

In the pessimistic deliver protocol, a deliver operation sends a cache line to a consumer only when the consumer has an invalidated copy of the line. For FFT, when a producer performs a deliver operation, the consumer does not have an invalidated copy in the cache if the cache is smaller than the largest working set. Thus, the deliver operation writes the line back to the main memory but does not send the line to the consumer. This is why the pessimistic deliver and invalidate-only protocols generate the same amount of the traffic for the main memory, the cache memory, and the network, as shown in Figure 5-22. For the directory, the pessimistic deliver protocol generates slightly more traffic than the invalidate-only protocol since a write-back and a replacement are done by a single transaction in the invalidate-only case but by two transactions (a deliver and a replacement-hint) in the pessimistic deliver protocol.

Applications with static producer-consumer patterns show similar deliver traffic characteristics versus the cache size; the deliver scheme does not produce a significant traffic overhead as long as the cache is large enough to capture most of the delivered messages.

5.3.2.2 Traffic Characteristics of Locus — a dynamic producer-consumer pattern

Unlike FFT, Locus has a dynamically changing producer-consumer pattern, which causes a very different traffic behavior. As discussed in Section 4.1, the producer-consumer pattern of Locus produces reader migration and single-processor reuse. In the optimistic deliver protocol, producers send unnecessary cache lines to consumers. Thus, as shown in Figure 5-22, the optimistic deliver protocol causes much larger traffic than the invalidate-only protocol for the directory, the cache memory, and the network since each unused deliver message generates unnecessary traffic at these components. The traffic overhead for the main memory, on the other hand, is not as large as that for other system components in the optimistic deliver protocol. This is because each deliver operation performs only one memory operation (a write back) so that deliver operations do not generate extra traffic if at least one of the deliver destinations uses the delivered line. If several destinations use the delivered line, the deliver operation reduces the amount of the main memory traffic by eliminating memory operations that each destination processor would cause for a cache miss in the invalidate-only protocol. In fact, as shown in Figure 5-22, the optimistic deliver protocol generates slightly less main-memory traffic than the invalidate-only protocol when the cache is larger than the largest working set. When the cache is smaller than the largest working set, however, deliver operations evict necessary cache lines in receiver's caches, so that cache conflicts due to deliver operations cause extra traffic at the main memory.

The pessimistic deliver protocol behaves very differently from the optimistic deliver protocol for Locus. As shown in Figure 5-22, when the cache size is much smaller than the largest working-set, the pessimistic deliver generates about the same amount of the traffic as the invalidate-only protocol at all of the four system components. This is because the cache is not large enough to keep invalidated cache lines until the producers sends them so that the pessimistic deliver protocol does not send cache lines to consumers. Thus, deliver operations do not significantly increase the amount of the traffic but do not significantly decrease the miss rate. When the cache size is much larger than the largest working set, on the other hand, the pessimistic deliver protocol generates the same amount of traffic as the optimistic deliver at the four system components because consumers do not replace cache lines.[10] The larger the cache size becomes, the more traffic overhead the pessimistic deliver protocol generates, but the more cache misses the pessimistic deliver protocol eliminates. For the pessimistic deliver protocol, therefore, if the four system components have a certain bandwidth, we may have a sweet spot in the cache size; for the cache size, the traffic overhead is not large enough to produce a significant contention delay but the cache miss rate is small enough to reduce memory latencies. The pessimistic deliver protocol exhibits similar traffic characteristics as other applications with a dynamic producer-consumer relation.

5.3.2.3 Traffic Characteristics of Other Applications and Feasible Bandwidth

Now, that we have discussed detailed traffic characteristics versus cache size for two very different applications, we pick two cache sizes to examine the traffic characteristics for all the applications. Then, we compare the amount of the traffic with the feasible bandwidth that we discussed in Section 5.3.1.

Figure 5-23 and Figure 5-24 show the traffic of the directory, the main memory, the cache memory, and the network for two different cache sizes. Figure 5-23 shows the traffic for a cache size larger than the largest working set of the application. This cache size is essentially infinite and does not cause cache conflicts. For this cache size, write-back traffic is negligible, and the pessimistic and optimistic deliver protocols exhibit almost the same behavior. Figure 5-24, on the other hand, shows the traffic for a cache size of about a quarter of the largest working set. This cache size is roughly the smallest size for which deliver operations can reduce the miss rate, and thus, is the smallest interesting cache size for the deliver scheme. For both cache sizes, we assume a fully-associative cache with 16-byte

10. This behavior is known as a dead-line effect for update protocols; as the cache size increases, the traffic of update protocols increases because the number of dead lines (cache lines that processors no longer use) increases.

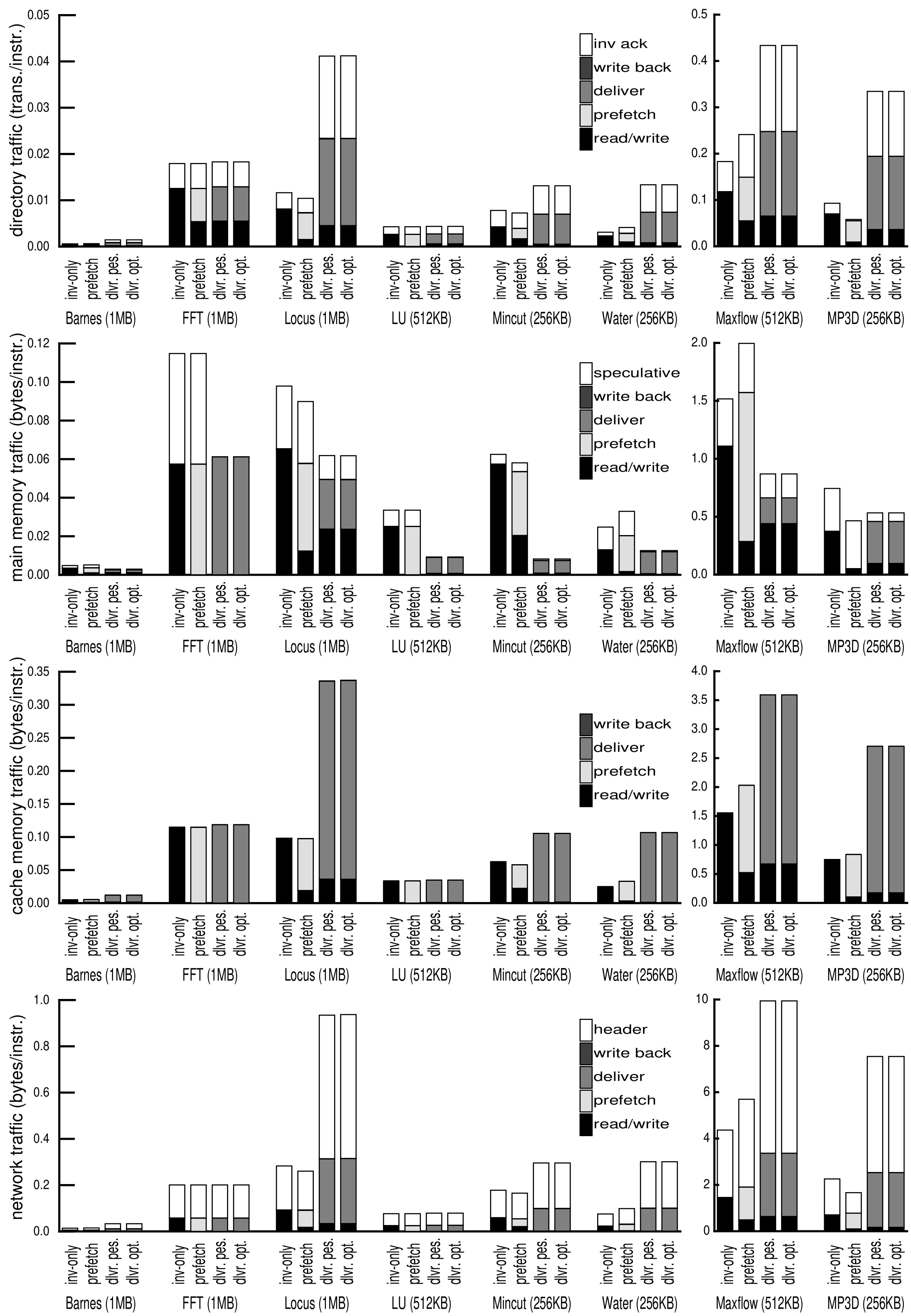

Figure 5-23: Breakdown of Directory, Main Memory, Cache Memory, and Network Traffic for a Large Cache.

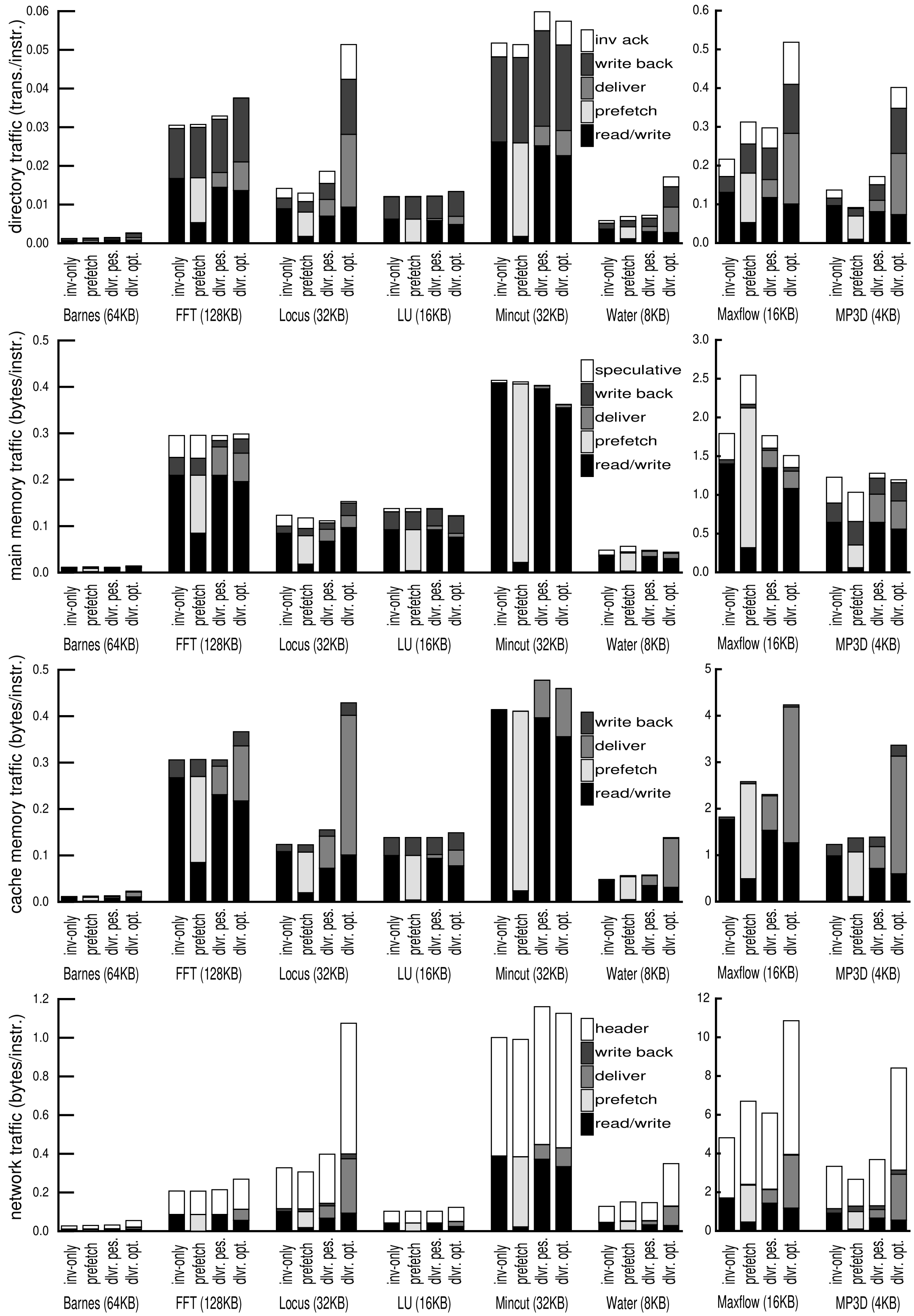

Figure 5-24: Breakdown of Directory, Main Memory, Cache Memory, and Network Traffic for a Small Cache.

lines. We use a different scale for Maxflow and MP3D since these two applications generate much more traffic than the rest of the applications.

Each bar graph in Figure 5-23 and Figure 5-24 divides the traffic into four to five components. The first four from the bottom indicate the amount of the traffic generated by a different type of memory accesses: read/write (read, write, and write-upgrade) misses, prefetch operations, deliver operations, and replace operations (writebacks and replacement hints). The fifth component shows a part of the traffic that we want to examine separately for each system component. For the directory traffic, the fifth component shows the invalidate acknowledgment due to write and write-upgrade misses and due to exclusive-mode prefetch operations. For the main memory traffic, the fifth component shows the traffic due to speculative accesses. For the network traffic, the fifth component shows the header traffic due to the header part in data and control packets.[11]

Some traffic characteristics of FFT and Locus that we discussed previously can be seen in the other applications. As shown in Figure 5-23 and Figure 5-24, the optimistic and pessimistic deliver protocols generate small or no traffic overhead for applications with static producer-consumer patterns (FFT and LU) but relatively large traffic overhead for applications with dynamic producer-consumer patterns (the rest of our applications). As discussed for Locus, when the cache size is sufficiently large, deliver operations reduce the amount of the main memory traffic for applications in which the producer-consumer relation is one-to-many (several consumers use a produced value). In fact, Figure 5-23 shows that the optimistic and pessimistic deliver protocols produce less traffic than the invalidate-only protocol at the main memory for Locus, LU, Mincut, and Maxflow when the cache is larger than the largest working set.

Moreover, as discussed for FFT and Locus, the prefetch protocol generates about the same amount of the traffic as the invalidate-only protocol for other applications except Maxflow and MP3D. For Maxflow, we prefetch the entire record of node and edge structures that the processor will use in future because it is difficult to determine which elements of the record will be used when the processor prefetches the record. This strategy generates more traffic than the invalidate-only case since the processor uses only a part of the prefetched

11. Control packets (i.e., request and acknowledge) consist only of a header.

record. For MP3D, the prefetch protocol produces much less traffic at the main memory than the invalidate-only protocol for the large cache case. This is because exclusive-mode prefetch operations transfer produced values (dirty cache lines) from cache to cache so that processors do not access the main memory to fetch produced values.

Let's compare the traffic of the deliver scheme with the feasible bandwidth that we discussed in Subsection 5.3.1. We first consider the large cache data of the six applications excepting Maxflow and MP3D. As shown in Figure 5-23, the optimistic and pessimistic deliver protocols generally produce more traffic than the invalidate-only protocol. The traffic of the deliver protocols is, however, still much smaller than the feasible bandwidth at each of the four system components for the six applications. For the main memory traffic, the deliver protocols generate less traffic than the prefetch and invalidate-only protocols. This is because the invalidate-only and prefetch protocols generate extra traffic for speculative memory accesses, which occur when processors access dirty cache lines. Speculative accesses do not occur often for the deliver scheme because deliver operations write dirty lines back to the main memory. The deliver operations, moreover, reduce the main memory traffic for one-to-many producer-consumer patterns, as discussed previously.

For the other three system components, the deliver scheme generates more traffic than the invalidate-only and prefetch protocols unless the application has a static producer-consumer pattern. As shown in Figure 5-23, Locus generates the largest amount of traffic among the six applications because of reader migration and single-processor reuse. For the directory and network traffic, the deliver scheme consumes bandwidth not only because of unused deliver messages but also because of acknowledge and invalidate messages for them, since producers need to invalidate unused cache lines that a previous deliver operation sends. This means that the header traffic creates significant overhead for the network.[12] For the large cache case, however, the traffic of the four system components is still less than a half of the bandwidth that we assume. Thus, the contention delay will not significantly affect the performance of the optimistic and pessimistic deliver protocols for

12. The header traffic component of the network traffic decreases, as the line size increases. We will discuss the traffic characteristics for various line sizes in the next subsection.

these six applications unless burst transfers or hot spots in the memory system cause significant contention delays.

Now, let's examine the traffic characteristics of the six applications for the small cache shown in Figure 5-24. Capacity misses and cache conflicts increase the traffic for the optimistic deliver protocol when the cache size is smaller than the largest working set. Especially for Mincut, the traffic at all of the four system components increases for all protocols, as shown in Figure 5-24. This is because a larger number of capacity misses occur for read-only data records. For Mincut, with any protocol, the directory traffic becomes more than 62% of its capacity. For Locus, with the optimistic deliver protocol, the directory traffic also becomes 62% of its capacity. Thus, the contention delay at the directory probably affects the performance of Mincut for any protocols and the performance of Locus for the optimistic deliver protocol. For Locus, however, the pessimistic deliver protocol generates much less traffic than the optimistic deliver protocol — hence, much less than the bandwidth — for the network and the directory because the pessimistic deliver protocol uses replacement hints. For the other four applications, the traffic of the four system components is less than 50% of the bandwidth.

Next, we discuss the traffic characteristics of Maxflow and MP3D. These two applications have a large communication-to-computation ratio and are not tuned for large-scale multiprocessors. Since the demand traffic for the invalidate-only protocol exceeds or nearly exceeds the feasible bandwidth at most of the four system components, the contention delay has a significant impact on performance. It is thus important to reduce the amount of the traffic to improve the performance of these two applications. For Maxflow, the optimistic prefetch protocol increases the amount of the traffic at all of the four system components, and the optimistic and pessimistic deliver protocols increase the amount of the traffic at the four system components except for the main memory. Thus, neither prefetch nor deliver operations would significantly improve the performance for Maxflow. For MP3D, since the optimistic and pessimistic deliver protocols increase the amount of the traffic, neither deliver technique would improve the performance. The optimistic prefetch protocol, on the other hand, reduces the amount of the traffic at all system components except the cache memory, so that it would improve the performance.

5.3.3 Traffic versus Line Size

In this subsection, we examine the traffic characteristics of the optimistic deliver protocol when the line size varies. Figure 5-25 shows the traffic of the optimistic deliver protocol at the four system components for line sizes from 16 bytes to 128 bytes. The eight graphs in Figure 5-25 consists of two sets: those on the left hand side and those on the right hand side. The two sets show the traffic for different cache sizes. The left hand side is for a cache size which is larger than the largest working set of the application (the same as the one used in Figure 5-23), and the right hand side is for a cache size which is about a quarter of the largest working set of the application (the same as the one used in Figure 5-24). For each graph, the right Y axis is for Maxflow and MP3D (dotted lines), and the left Y axis is for the rest of the applications (solid lines).

The cache and main memory traffic generally increases as the line size increases because the utilization of transferred cache lines decreases. In fact, Figure 5-25 shows that this is true for all applications except FFT and LU, in which processors fully utilize transferred cache lines at least up to 128 bytes. In our bandwidth model, the bandwidth is doubled as the line size is doubled up to 64 bytes for the main memory and up to 32 bytes for the cache memory. Thus, the ratio between the traffic and the bandwidth does not increase significantly except for Maxflow and MP3D. Except for these two applications, the traffic of the cache and main memory is much less than the capacity of the component for the small cache with all line sizes from 16 bytes to 128 bytes. (The cache memory traffic is less than 36% of its capacity, and the main memory traffic is less than 45% of its capacity.) The traffic for the large cache is even smaller than that for the small cache. Therefore, for most of our applications, the traffic at the cache and main memory will not cause a significant contention delay for line sizes up to 128 bytes.

The directory traffic generally decreases as the line size increases for most of the applications because the miss rate decreases. Since the directory bandwidth is independent of the line size in our model, the ratio between the traffic and the bandwidth decreases for these applications. Even for applications with poor spatial locality (i.e., Barnes, Mincut, and Maxflow), the amount of the directory traffic does not increase significantly. For line sizes

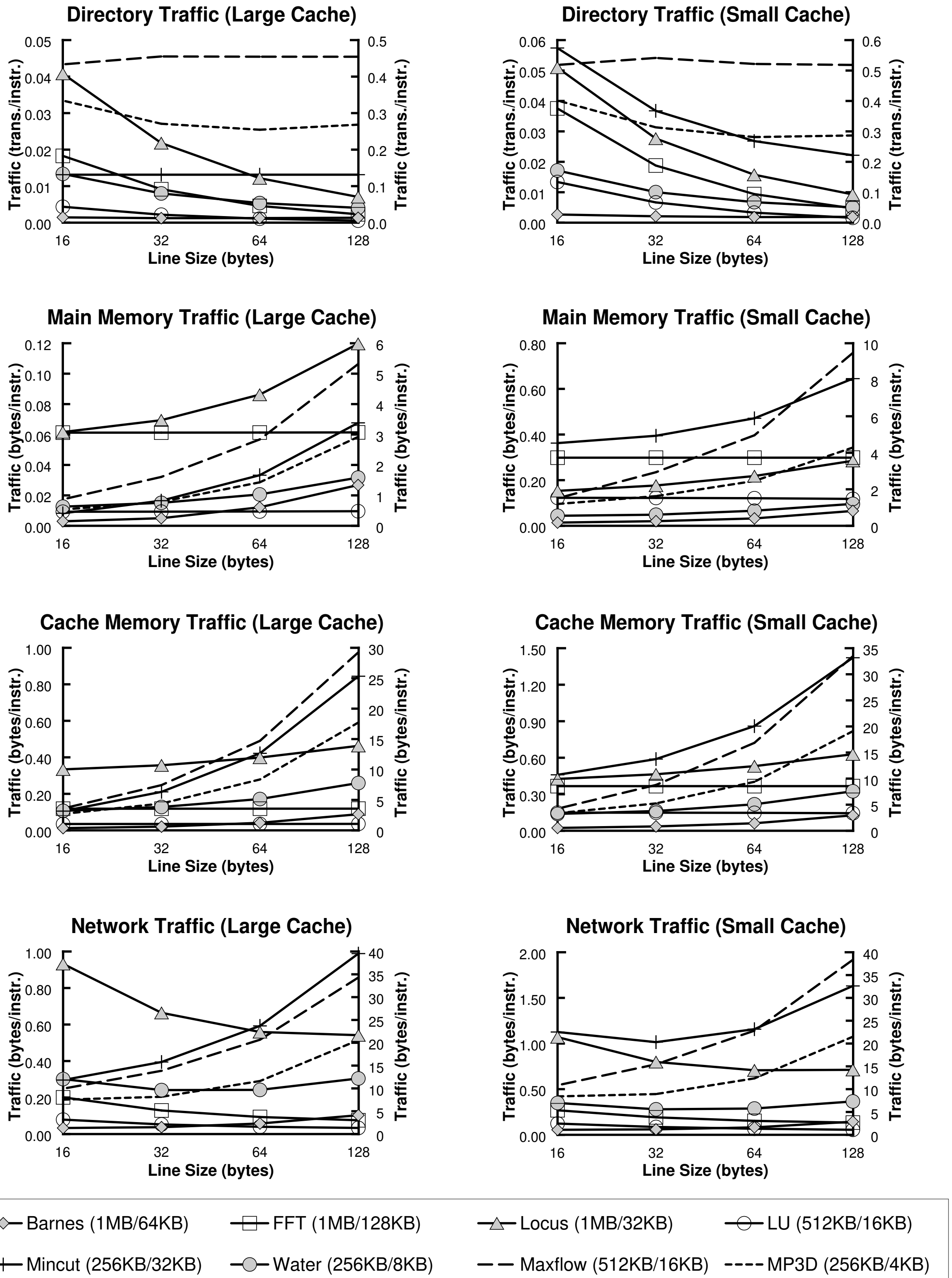

Figure 5-25: Directory, Main Memory, Cache Memory, and Network Traffic versus Line Size for Optimistic Deliver. The right Y axis is for Maxflow and MP3D (dotted lines), and the left Y axis is for the rest of the applications (solid lines). The graph legend shows two cache sizes for each application: a large size for the left graphs and a small size for the right graphs.

from 32 bytes to 128 bytes, the directory traffic is less than a half of the bandwidth for both cache sizes except for the two most demanding applications (Maxflow and MP3D).

For the network traffic, there are two conflicting components: data and header traffic. The header traffic causes more than a half of the network traffic when the line size is 16 bytes as shown in Figure 5-23 and Figure 5-24, but the amount of the header traffic generally decreases as the line size increases since the number of packets decreases. The amount of the data traffic, on the other hand, generally increases as the line size increases since the utilization of cache lines decreases. Therefore, as the line size increases, the network traffic generally decreases up to a certain line size and increases after the line size. For Maxflow and MP3D, however, because of the poor spatial locality, the network traffic of the optimistic deliver increases consistently as the line size increases. Mincut also has poor spatial locality so that the network traffic reaches 82% of the bandwidth for the small cache size when the line size is 128 bytes. For applications except Maxflow, Mincut, and MP3D, the network traffic is less than roughly a half of the bandwidth for line sizes from 16 bytes to 128 bytes.

5.3.4 Traffic versus the Number of Processors

In this subsection, we discuss the traffic behavior when the number of processors varies and the problem size is constant. As the number of processors increases, the communication-to-computation ratio generally increases so that the communication traffic increases. The traffic overhead due to deliver operations, on the other hand, may or may not increase as the number of processors increases. The characteristics of the deliver traffic significantly depends on the producer-consumer pattern of the application. As the number of processors increases, the traffic overhead due to deliver operations does not increase for applications with a static producer-consumer relation, while the traffic overhead increases for applications with a dynamic producer-consumer relation. Figure 5-26 shows the traffic at the four system components when the number of processors varies from 8 to 64 for three protocols — invalidate-only, optimistic prefetch, and optimistic deliver. The cache is a fully-associative cache with 16-byte lines, and the cache size is large enough to capture the largest working-set, the same as the one used in Figure 5-23. This large cache assumption allows us to focus on communication traffic. Figure 5-26 divides the traffic into four to five components in the same manner as Figure 5-23 and Figure 5-24.

For FFT and LU, as we discussed previously, the producer-consumer relation is static so that consumers use most of the delivered cache lines as long as the cache is large enough to capture the delivered lines. This characteristic does not change when the number of processors increases. Thus, deliver operations produce small or no traffic overhead for the directory, the cache memory, and the network, even if the number of processors increases. For the main memory, as we discussed, deliver operations reduce the amount of the traffic for applications with a one-to-many producer-consumer pattern (e.g., LU). For LU, Figure 5-26 shows that the amount of the main memory traffic increases for the invalidate-only and prefetch protocols as the number of processors increases, while the amount of the main memory traffic does not increase for the deliver protocol. This is because the number of consumers that use a produced value (submatrix) increases as the number of processors increases. For the invalidate-only protocol, the main memory traffic occurs when a consumer reads a produced value, so that the main memory traffic increases as the number of consumers per produced value increases. For the optimistic deliver protocol, on the other hand, the main memory traffic occurs when a producer writes a produced value, so that the main memory traffic is constant as long as the problem size is constant.

For Locus and Water, the producer-consumer relation changes dynamically so that producers send cache lines to processors that no longer use them. Figure 5-26 indicates that, for both applications, the traffic of the optimistic deliver protocol increases more rapidly than the traffic of the invalidate-only protocol when the number of processors increases. For Water, while a producer delivers a molecule record to about a half of the processors, only one of them usually uses the record. Thus, a producer generates more deliver messages to eliminate a one cache miss as the number of processors increases, so that the traffic of the optimistic deliver increases more rapidly than the communication-to-computation ratio. Moreover, Figure 5-26 indicates that the traffic of the optimistic deliver protocol at the four system components is much less than the feasible bandwidth (cf. Subsection 5.3.1) up to 64 processors for the four applications except Locus. For Locus, the directory and network traffic exceed the capacity of the system component when the num-

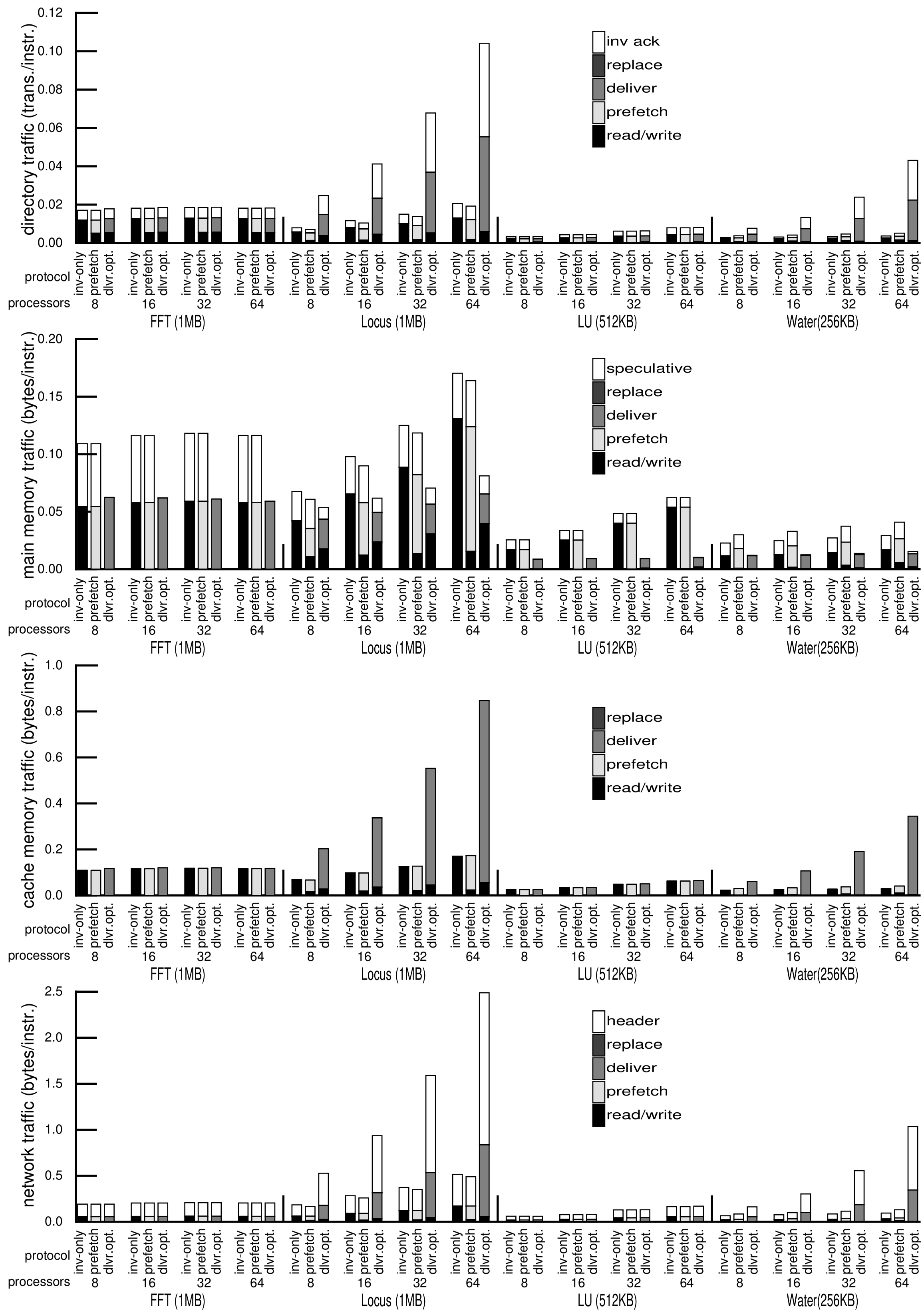

Figure 5-26: Breakdown of Directory, Main Memory, Cache Memory, and Network Traffic versus the Number of Processors for a Large Cache.

ber of processors becomes 64. If we use a larger line size, however, the directory and network traffic of Locus should become less than the capacity.

5.3.5 Summary

We have examined the traffic at the four system components — the directory, the cache memory, the main memory, and the network — and compared the traffic with a feasible bandwidth for each component. We vary three architectural parameters — cache size, line size, and the number of processors — to examine interactions between those parameters and the traffic characteristics of the deliver and prefetch schemes.

The traffic characteristics of the prefetch scheme is almost the same as those of the invalidate-only protocol for most of our applications. The deliver scheme, on the other hand, usually generates more traffic than the invalidate-only protocol for applications with a dynamically changing producer-consumer pattern. For the base configuration (16 processors with 16-byte cache lines), however, the averaged traffic at each system component is less than 50% of its capacity for the optimistic and pessimistic deliver protocols for most of our applications when the cache is large enough to capture the largest working set. For those applications, therefore, the deliver protocols with a sufficiently large cache will not cause a significant contention delay unless bursty transfers or hot spots impact significantly. As the cache size decreases, the traffic at each system component increases for the optimistic deliver protocol because of capacity misses and cache conflicts. Pessimistic deliver operations generate much less traffic than optimistic deliver operations when the cache size is not much larger than the largest working set because pessimistic deliver operations send a cache line only to processors that have an invalidated copy of the cache line. Pessimistic deliver operations, however, can be an attractive deliver technique only if the cache size is about the same as the largest working set of the application.

As the line size increases, the traffic of the optimistic deliver protocol behaves differently for each system component. As the line size increases, the bandwidth for the cache and main memory increases to some extent in our architectural model. Thus, the ratio between the traffic and the bandwidth does not increase significantly for these two system components. The directory traffic usually decreases, as the line size increases, since the number of cache misses decreases. The network traffic consists of two components (header and data) that exhibit different behavior. As the line size increases, the network traffic generally decreases up to a certain line size and increases after the line size. On the whole, as the line size increases up to 128 bytes, the ratio between the traffic and the bandwidth decreases or increases only slightly for most of our applications.

Finally, we have discussed the traffic characteristics when the number of processors varies. We examined four applications (FFT, Locus, LU, and Water) when the cache is large enough to capture the largest working set of the application. The characteristics of the deliver traffic strongly depends on the producer-consumer pattern of the application. For applications with a static producer-consumer relation (e.g., FFT and LU), the traffic overhead due to deliver operations does not increase as the number of processors increases. For applications with a dynamic producer-consumer pattern (e.g., Locus and Water), on the other hand, the traffic overhead due to deliver operations increases more rapidly than the communication-to-computation ratio as the number of processors increases.

5.4 Memory/Network Latency

In this section, we discuss the impact of more realistic traffic models on different protocols. Our simulator models a machine including contention in an interconnection network and the four components of the processor node (the directory, the cache memory, the main memory, and the network).

Our simulation results show that, for most of our applications with dynamic producer-consumer patterns, the deliver scheme suffers more from traffic overhead than the prefetch scheme, while the prefetch scheme suffers more from instruction overhead than the deliver scheme. Even if the average of the traffic of the deliver scheme is much less than the capacity for each system component, non uniform communication causes contention delays for some applications. When the memory latency is very large, as discussed in Subsection 5.1.2, the deliver scheme is qualitatively more advantageous than the prefetch scheme since deliver operations transfer cache lines much earlier than prefetch operations. Our simulation results, however, show that the prefetch scheme performs about the same as the deliver scheme for most of our applications because multiple prefetch requests can be overlapped.

In the following sections, we first discuss our bandwidth and latency assumptions for our simulations. Second, we discuss the execution time of applications for a base latency. Third, we discuss the simulation results for longer latency to examine the impact of memory latency variations for the prefetch and deliver schemes.

5.4.1 Latency/Bandwidth Assumption

We assume the maximum bandwidth that we discussed in Section 5.3 for each of the four system components in a processor node. Table 5-2 shows the bandwidth for caches with 64-byte lines. We assume a two-dimensional wormhole mesh-network that connects processor nodes as illustrated in Figure 5-19. The network bandwidth between nodes is 400 Mbytes/sec for each direction, the same as the bandwidth between the network router and the directory controller.

Table 5-3 shows unloaded read latencies for 64-byte lines when the number of network hops is two. (The average of the number of hops for a 4×4 mesh network is 2.6.) The latency numbers in Table 5-3 are the number of clocks until the processor receives the necessary word. We assume that the processor receives the necessary word first and proceeds without waiting for the whole cache line.

We assume a weak consistency model and a lockup-free single-level cache with a 8-deep write buffer for each processor node. All write and prefetch requests are stored in the buffer until the request completes. The write buffer merges a write (or prefetch) request with another pending request in the write buffer when the two requests are for the same cache line. Read requests are allowed to bypass pending requests in the write buffer if the requested cache line is different from that of any pending requests. Our simulator models

Line Size	64 bytes
Cache Directory (transactions/sec)	16.7M
Main Memory (bytes/sec)	640M
Cache Memory (bytes/sec)	800M
Network (bytes/sec)	400M

Table 5-2: Bandwidth of Four System Components in Processor Nodes.

Latency	Base	Long
Local (clean, home=local)	50	50
Remote (clean, home local)	210	1234
Remote (dirty, home=owner, home local)	220	1244
Remote (dirty, home owner, home=local)	220	1244
Remote (dirty, home owner, home local)	284	1820

Table 5-3: Typical Read Latency without Contention. (The number of 200MHz processor clocks, Two network hops per remote-node access.)

processor stalls due to write-buffer overflow and due to fences at synchronization operations.

5.4.2 Execution Time for the Base Latency Model

Figure 5-27 shows the execution time of four protocols — invalidate-only, optimistic prefetch, pessimistic deliver, and optimistic deliver — when the line size is 64 bytes. Each execution time is normalized by the busy time of the invalidate-only protocol. The busy time is the time during which the processor executes instructions. In addition to the busy time, the execution time has five other components as shown in Figure 5-27: the read time, the read contention time, the read pending time, the write-buffer stall time, and the synchronization time. The first three components — the read time, the read contention time, and the read pending time — comprise the processor stall time due to read misses. The read time is a portion of the stall time due to read misses excluding contention delays. The read contention time is a portion of the stall time due to contention at various system components. The read pending time is a portion of the stall time due to pending operations to the same cache line. This read pending time typically represents processor stalls because of prefetch or deliver operations that are executed too late for the consumer to receive the cache line before its use. The write-buffer stall time is the stall time due to write buffer overflow and due to fences for synchronization. The synchronization time is the stall time due to synchronization operations. The cache is fully associative and the cache size is large enough to capture the largest working set of the application and the same as the cache size used for Figure 5-23.

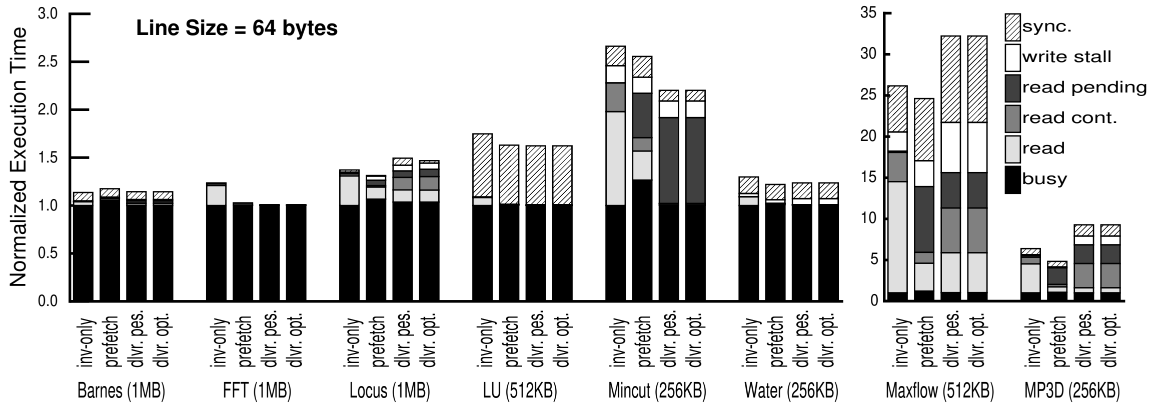

Figure 5-27: Execution Time of Four Protocols for Base Latency. The execution time is normalized by the busy time of the invalidate-only protocol.

Since the cache is large enough to capture the largest working set, the pessimistic and optimistic deliver protocols cause about the same execution time. The prefetch scheme performs better than the deliver scheme for Locus, Water, Maxflow, and MP3D because the contention delay of the deliver scheme is larger than that of the prefetch scheme. Especially for Locus, Maxflow, and MP3D, the execution time of the deliver scheme is even larger than that of the invalidate-only protocol. For Maxflow and MP3D, as illustrated in Figure 5-25, the directory and network traffic exceeds the capacity of the component for the deliver scheme, so that the contention delay degrades the performance for the deliver scheme. For Locus, on the other hand, although Figure 5-25 shows that the averaged traffic is much less than the bandwidth for the four system components, the contention delay degrades the performance for the deliver scheme. This is because the traffic is not uniformly distributed among processor nodes so that the traffic of the directory and the network at some processor nodes nearly exceeds the bandwidth. Thus, contention at those processor nodes causes a significant delay for Locus.

For Barnes and Mincut, the execution time of the deliver scheme is smaller than that of the prefetch scheme. This is because of instruction overhead due to prefetch operations. The instruction overhead of prefetch operations is generally larger than that of deliver operations. If we could insert prefetch or deliver operations only where communication occurs, the instruction overhead would be very small for both schemes. As discussed in Section 4.2, it is difficult to pinpoint read or write operations that cause communication, so that we need to insert deliver or prefetch operations conservatively for read or write operations that might cause communication. This approach causes unnecessary deliver or prefetch operations; processors perform deliver operations for cache lines that none of the destinations use (e.g., single-processor reuse), or processors perform prefetch operations for cache lines that are already in the cache. Prefetch operations are typically inserted before read operations while deliver operations are typically inserted after write operations. Since the number of read operations is much larger than that of write operations, the number of prefetch operations is potentially much larger than that of deliver operations. The simulation results show that the prefetch scheme actually produces more instruction overhead than the deliver scheme.

As the cache size decreases, the miss rate for the deliver scheme increases more rapidly than that for the prefetch scheme because capacity misses and cache conflicts occur for the deliver scheme, as discussed in Section 5.1. Thus, the prefetch scheme usually performs better than the deliver scheme for cache sizes smaller than the largest working set.

5.4.3 Execution Time for the Long Latency Model

In this section, we discuss the impact of long latency for the performance of deliver and prefetch operations. We assume that the bandwidth is the same as that used in the previous subsection (shown in Table 5-2) and that the latency for each communication between processor nodes is 512 clocks longer than the base latency (shown in Table 5-3). We use the same annotation of prefetch and deliver operations as the one we used for the base latency. The longer the memory latency becomes, the earlier we should schedule prefetch operations to hide the memory latency effectively. In this section, however, we leave open the question of how much improvement is possible in prefetch if prefetch operations are scheduled earlier.

Figure 5-28 shows the normalized execution time for the long memory latency. The cache size is larger than the largest working set (the same as the one used in Figure 5-27). The cache is fully associative, and the line size is 64 bytes. Hiding memory latencies becomes more important as the memory latency increases; the benefit of prefetch and deliver operations increases. As discussed in Subsection 5.1.2, deliver operations generally transfer cache lines much earlier than prefetch operations. Moreover, the data transfer latency of

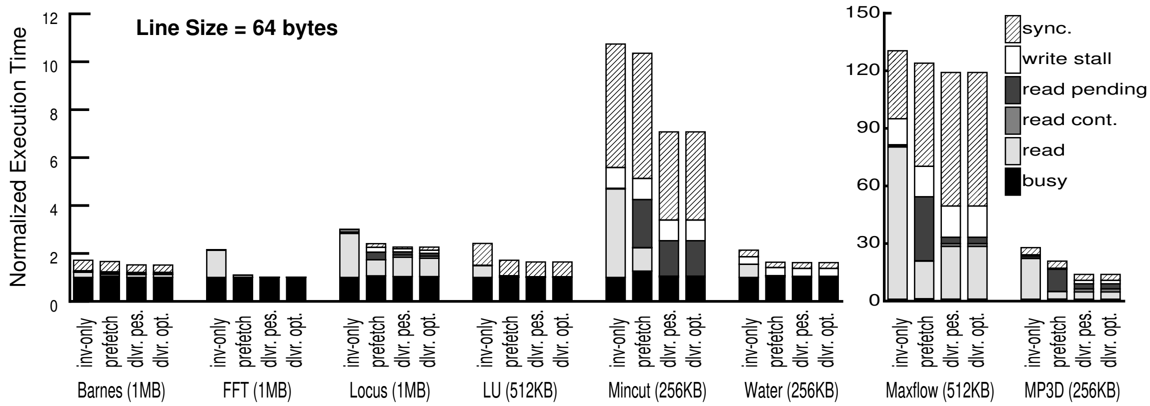

Figure 5-28: Execution Time of Four Protocols for Large Latency. The execution time is normalized by the busy time of the invalidate-only protocol.

the deliver operation is shorter than that of the prefetch operation; the latency of the deliver operation is a one-way latency from the producer to the consumer, while the latency of the prefetch operation is a round-trip latency between the consumer and the producer. Thus, the deliver scheme is qualitatively more advantageous than the prefetch scheme when the memory latency is large.

In fact, as shown in Figure 5-28, the deliver scheme performs better than the prefetch scheme for most of the applications when the memory latency is large. The deliver scheme, however, performs significantly better than the prefetch scheme only for Mincut. For the rest of the applications, the execution time for the prefetch scheme is about the same as or only slightly larger than that for the deliver scheme. This is because the cache is lock-up free; multiple memory accesses can be overlapped [45], even if the prefetch distance is not large enough to overlap memory accesses with computation perfectly. Thus, the latency of a memory access can be hidden by that of another memory access. Furthermore, Figure 5-28 implies that the prefetch scheme probably performs better than the deliver scheme if prefetch operations are scheduled earlier. The quantitative evaluation of the sensitivity for large memory latency, however, is beyond the scope of this thesis; we need to vary the scheduling of prefetch operations to compare the execution time of the prefetch and deliver schemes.

5.5 Chapter Summary

In this chapter, we examined the effect of various architectural parameters for the performance of deliver and prefetch operations and analyzed application characteristics that interact with those architectural parameters.

First, we examined the effect of two cache parameters: cache size and associativity. Deliver operations suffer much more from cache conflicts due to small cache size and small associativity than prefetch operations because deliver operations transfer cache lines much earlier than prefetch operations. Our analysis of the inter-reference distance distribution demonstrates that, for most of our applications, the cache needs to be as large as the largest working set to capture most of the delivered lines until they are used. Deliver operations, moreover, cannot eliminate cold and pure capacity misses, while prefetch operations can. Therefore, the prefetch scheme generally eliminates much more cache misses than the deliver scheme if the cache size is smaller than the largest working-set.

Second, we discussed the effect of line size for the prefetch and deliver schemes. The miss rate of the deliver and prefetch schemes does not decrease proportionally as the miss rate of the invalidate-only protocol decreases, because deliver and prefetch operations cause false sharing in some access patterns. We also discussed how the effect of large line sizes changes as the cache size changes.

Third, we discussed traffic characteristics when various architectural parameters change. Although the optimistic deliver protocol generally generates more traffic than the optimistic prefetch protocol for most of the system components, the average of the traffic for the deliver scheme is less than 50% of the capacity of the component for most of our applications when the cache is large enough to capture the largest working-set. We can reduce the traffic overhead due to deliver operations by using the pessimistic deliver protocol. The use of the pessimistic deliver protocol, however, is not a general solution; it can be advantageous only if the cache size is about the same as the largest working set.

The deliver scheme generates more traffic overhead for applications with dynamic producer-consumer patterns than applications with static producer-consumer patterns. For applications with dynamic producer-consumer patterns, the traffic overhead increases more rapidly than the communication-to-computation ratio, as the number of processors increases. For those applications, our simulation results show that the traffic overhead at the directory and the network probably produces a significant contention delay as the number of processors increases when the line size is 16 bytes. As the line size increases, however, the traffic at these system components generally decreases so that the contention delay at these system components probably does not significantly impact the performance for the deliver scheme.

Finally, we examined the effect of memory contention and latency for the performance of the prefetch and deliver schemes. The prefetch scheme performs better than or about the same as the deliver scheme for most of the applications because the prefetch scheme generates less traffic contention than the deliver scheme. For some applications, although the average of the traffic of the deliver scheme is less than the bandwidth that we assume, the traffic is not uniformly distributed so that the contention delay degrades the performance of the deliver scheme. Our simulation results also show that the prefetch scheme generally generates more instruction overhead than the deliver scheme. For two applications out of eight, the prefetch scheme performs worse than the deliver scheme because of the instruction overhead due to prefetch operations. For applications with static producer-consumer patterns, the performance of the two schemes is about the same.

When the memory latency is very large, qualitatively, the deliver scheme is more advantageous than the prefetch scheme. Our simulation results, however, indicate that the prefetch scheme performs about the same as the deliver scheme for most of our applications even if prefetch operations are not scheduled earlier. This is because multiple memory accesses can be overlapped so that the latency of a memory access is hidden by that of another access.

Chapter 6

Conclusions

Coping with the memory latency is becoming increasingly important for large-scale shared-memory multiprocessors because communication occurs by accessing remote memory. Prefetching is a technique to optimize the communication on the shared memory so that the communication and the computation overlap. This thesis has quantitatively evaluated two alternatives for software-controlled prefetching: consumer-oriented and producer-oriented schemes.

The consumer-oriented scheme — prefetch — has been shown to be effective for reducing the impact of large memory latencies. Qualitatively, the producer-oriented scheme — deliver — has an advantage over the prefetch; the deliver scheme can transfer produced values at the earliest possible timing, when the value is produced. The prefetch scheme, on the other hand, may not fetch cache lines early enough to hide the memory latency completely. For applications with irregular access patterns, for example, the address of the value to be prefetched may not be available early enough. By examining simulation results of scientific parallel applications, however, this thesis has shown that the prefetch scheme is generally more effective than the deliver scheme for NUMA machines with realistic architectural parameters.

The deliver scheme can be viewed as an optimized update protocol. Regular update protocols are known to generate substantial traffic overhead since each write operation for a shared variable generates traffic. The deliver scheme is an optimized update protocol in a sense that it aggregates multiple updates for the same cache line into a single message by using synchronization operations as a hint to find the place where communication occurs. Our analyzes of sharing patterns, however, have shown that the deliver scheme still generates high traffic overhead for applications with a dynamic producer-consumer relation although the deliver scheme generates little or no traffic overhead for applications with a static producer-consumer relation. This is because the past sharing behavior that the deliver scheme relies on is not a good predictor of future sharing behavior for applications with a dynamic producer-consumer relation. We identified two sharing patterns — reader migration and single-processor reuse — that cause traffic overhead for the deliver scheme.

We have evaluated three techniques — selective deliver, subscription control, and competitive back-off — to trade off the number of delivered messages for the number of eliminated cache misses. While the effectiveness of these techniques depends on the sharing pattern of the application, our simulation results show that selective deliver (software-based producer's control) is better than or roughly equal to the other two techniques for most of our applications. The prefetch scheme also generates a large number of unnecessary prefetch operations for applications with dynamic access patterns. Most of unnecessary prefetch operations occur when consumers try to prefetch cache lines that are already in the cache. This happens because consumers cannot identify the data set that had been fetched by the consumer and has not been updated by another processor. The prefetch scheme can use the processor cache as a filter to prevent such unnecessary prefetch operations from generating network traffic. This is a significant advantage of the prefetch scheme since the processor cache cannot work as a filter for the deliver scheme. Although the processor cache with replacement hints could be used to filter out some unnecessary deliver messages, it is not a general solution because the effect of replacement hints is extremely sensitive to the cache size (or the data-set size).

We have analyzed the miss-rate and traffic characteristics for the deliver and prefetch schemes when various architectural parameter changes, and we have examined the application characteristics that interact with those parameters. Our simulation results show that the deliver scheme suffers much more from cache conflicts due to small capacity and small associativity than the prefetch scheme because the deliver scheme transfers cache lines much earlier than the prefetch scheme. One of the most important results is that the cache needs to be as large as the largest working-set of the application to retain most of the received deliver messages until they are used. We analyzed the temporal characteristics of the communication and the working-set characteristics to understand this behavior of deliver operations. For analyzing the traffic characteristics, we examined demand traffic for major system components (the cache directory, the cache memory, the main memory, and the network). Our simulation results show that, although the deliver scheme generates high traffic overhead, the average of the traffic at each system component is less than a half of the bandwidth for most of our applications when the cache size is large enough to capture the largest working-set of the application. For some applications, however, non-uniform communication patterns (hot-spots or burstiness) cause traffic congestions for the deliver scheme. For applications with dynamic producer-consumer patterns, moreover, as the number of processors increases, the traffic for the deliver scheme increases more rapidly than the communication-to-computation ratio.

Finally, we have evaluated the performance impact of deliver and prefetch operations for realistic latency and bandwidth parameters. Even if the cache size is large enough for the deliver scheme to eliminate most of sharing misses, the prefetch scheme performs better than or roughly equal to the deliver scheme for most of our applications. This is because prefetch operations are inserted early enough to hide the bulk of the memory latency for most of the applications and because deliver operations cause contention delay due to traffic congestions for some applications. Even if memory latency is very large, our simulation results show that the prefetch scheme performs about the same as the deliver scheme for most of our applications. This is because prefetch operations can pipeline memory accesses. The deliver scheme performs better than the prefetch scheme for some applications because of the instruction overhead for the prefetch operation. As discussed previously, while the prefetch scheme can eliminate the traffic overhead due to unnecessary prefetches, the instruction overhead still remains. For applications with irregular access patterns, prefetch operations are conservatively inserted where a cache miss might occur. As a result, the prefetch scheme generates more instruction overhead than the deliver scheme.

To summarize, we have quantitatively evaluated the prefetch and deliver schemes by analyzing application-intrinsic characteristics (e.g., sharing patterns and working sets) and interactions between application characteristics and architectural parameters (e.g., miss rate, traffic, and execution time). This thesis has shown that the prefetch scheme is more effective than the deliver scheme for scientific applications on realistic NUMA machines. The advantages for the prefetch scheme come from several non-obvious sources including filtering by the cache, the inaccuracy of past behavior as a predictor for future behavior in the case of coherency accesses, and the program structure that causes too early transfers of deliver messages.

6.1 Future Work

The comparison of the producer-oriented and consumer-oriented schemes should be interesting in other contexts since prefetching techniques are applicable for various applications and architectures. For example, further research is needed to evaluate prefetching techniques for commercial applications and operating systems.

We qualitatively discussed that the deliver scheme is advantageous over the prefetch scheme when the communication latency and the cache size is very large. This condition may hold in other architectures (e.g., COMA with extremely high-speed processors, network-connected workstations). The quantitative comparison of the latency sensitivity for the two schemes requires further experiments.

Bibliography

[1] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104-114, May 1990.

[2] C. Anderson and J.-L. Baer. Two Techniques for Improving Performance on Bus-based Multiprocessors. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 264-275, January 1995.

[3] J. K. Archibald. A Cache Coherence Approach For Large Multiprocessor Systems. In *International Conference on Supercomputing*, pages 337-345, July 1988.

[4] R. R. Atkinson and E. M. McCreight. The Dragon Processor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 65-69, October 1987.

[5] D. H. Bailey. FFT's in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23-35, 1990.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78-101, 1966.

[7] M. Berry, et al. The perfect club benchmark: Effective performance evaluation of supercomputers. Technical Report CSRD 827, Center for Supercomputing Research and Development, May 1989.

[8] G. Byrd, B. Delagi, and M. Flynn. Communication mechanisms in shared memory multiprocessors. Technical Report CSL-TR-94-623, Stanford University Computer Systems Laboratory, May 1994.

[9] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40-52, April 1991.

[10] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 224-234, April 1991.

[11] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. *Communication of the ACM*, 24(11):198-206, 1981.

[12] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 223-232, April 1994.

[13] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98-108, May 1993.

[14] Cray Research Inc. Cray T3D System Architecture and Overview. Revision 1.c., Cray Research Inc., September 1993.

[15] F. Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 60-69, June 1995.

[16] F. Dahlgren and P. Stenström. Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 26(2):193-210, 1995.

[17] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323-333, 1968.

[18] P. J. Denning and S. C. Schwartz. Properties of the Working-Set Model. *Communications of the ACM*, 15(3):191-198, 1972.

[19] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88-97, May 1993.

[20] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In *Proceedings of Supercomputing '91*, pages 197-206, November 1991.

[21] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373-383, May 1988.

[22] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2-15, May 1989.

[23] S. Frank, H. B. III, and J. Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Compcon Spring '93*, pages 285-294, February 1993.

[24] D. B. Glasco, B. A. Delagi, and M. J. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 534-545, January 1994.

[25] A. Goldberg and R. Tarjan. A New Approach to the Maximum Flow Problem. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 136-146, May 1986.

[26] S. Goldschmidt. Simulation of multiprocessors: accuracy and performance. Ph.D. Thesis, Stanford University, 1993.

[27] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *International Conference on Supercomputing*, pages 354-368, June 1990.

[28] H. Grahn, P. Stenström, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247-271, 1995.

[29] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254-263, May 1991.

[30] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794-810, 1992.

[31] E. Hagersten. Towards scalable cache only memory architectures. Dissertation (Doctor of Technology), The Royal Institute of Technology, 1992.

[32] E. Hagersten, P. Anderson, A. Landin, and S. Haridi. A Performance Study of the DDM — a Cache-Only Memory Architecture. Research Report R91:17, Swedish Institute of Computer Science.

[33] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 38-50, October 1994.

[34] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 274-285, October 1994.

[35] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 262-273, September 1992.

[36] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 244-254, October 1986.

[37] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985.

[38] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[39] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63-79, 1992.

[40] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148-159, May 1990.

[41] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: analyzing memory system bottlenecks in programs. In *Proceedings of ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 1-12, June 1992.

[42] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78-117, 1970.

[43] J. D. McDonald and D. Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[44] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87-106, 1991.

[45] T. C. Mowry. Tolerating latency through software-controlled data prefetching. Ph.D. Thesis, Stanford University, 1994.

[46] T. C. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62-73, October 1992.

[47] H. Nilsson and P. Stenström. An Adaptive Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic. In *Proceedings of Parallel Architectures and Languages Europe (PARLE '94)*, pages 363-374, June 1994.

[48] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 348-354, 1984.

[49] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA)*, February 1996.

[50] J. S. Rose. LocusRoute: a parallel global router for standard cells. In *Proceedings of the 25th Design Automation Conference*, pages 189-195, June 1988.

[51] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L. W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proceedings of ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 74-85, May 1993.

[52] E. Rothberg, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 14-25, May 1993.

[53] J. P. Singh, C. Host, T. Totsuka, A. Gupta, and J. L. Hennessy. Load Balancing and Data Locality in Hierarchical N-body Methods. Technical Report CSL-TR-92-505, Stanford University Computer Systems Laboratory, January 1992.

[54] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University Computer Systems Laboratory, April 1991.

[55] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 109-118, May 1993.

[56] C. P. Thacker and L. C. Stewart. Firefly: A Multiprocessor Workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 164-172, October 1987.

[57] J. E. Veenstra and R. J. Fowler. A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols. In *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 149-160, September 1992.

[58] D. Windheiser, E. L. Boyd, E. Hao, S. G. Abraham, and E. S. Davidson. The KSR1: Analysis of Latency Hiding Techniques in a Sparse Solver. In *Proceedings of Seventh International Parallel Processing Symposium*, pages 454-461, April 1993.

[59] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24-36, June 1995.

[60] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. Technical Report CSL-TR-93-593, Stanford University, December 1993.

[61] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 219-229, October 1994.