Data Prefetching for Distributed Shared Memory Systems

Alexander I-Chi Lai and Chin-Laung Lei

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

E-mail: alex@fractal.ee.ntu.edu.tw, lei@thunder.ee.ntu.edu.tw.

Abstract

Data prefetching is a technique that a processing unit issues one or more non-blocking load operations before the very data items are actually required. The access latency of prefetching can be alleviated by overlapping it with other executions which are independent of the prefetched data. In distributed shared memory (DSM) systems, remote memory accesses take much longer than local ones and hence data prefetching should be effective for such systems. However, to our knowledge relatively few researches have been done for data prefetching on DSM systems.

This paper is concerned with issues of supporting data prefetching on DSM systems. Our approach is to develop a new memory consistency semantic (MCS) model under which the prefetchable shared data objects, as well as the best moment to launch a prefetching operation, can be easily identified. Our new MCS, called the aggressive consistency, utilizes the coherence-on-demand concept and supports a special synchronization operation called SYNC, which also acts as the prefetching indicator. Preliminary simulation results show that our prefetching approach combined with the aggressive consistency can substantially improve the performance of DSM systems.

1 Introduction

Recently, distributed shared memory (DSM) becomes a very active area in distributed system research [1, 2, 3]. In brief, a DSM is an abstraction of shared memory address space on a distributed-memory multicomputer, permitting shared memory parallel programs to be executed on interconnection network architecture. DSM possesses great potential because it combines simplicity of shared-memory programming paradigm with scalability of distributed-memory systems. Nevertheless, it is still a

challenge to build an efficient DSM system and maintain the system integrity at the minimum cost. Indeed, DSM researchers often focus on reducing the impact of remote memory accesses since the remote access latency is quite large.

Data prefetching [4, 5] is one of the several techniques for tolerating long memory access latency. In a word, prefetching means that a processor issues one or more non-blocking load operations before the very data items are actually required. The access latency of prefetching can be alleviated by overlapping it with other executions which are independent of the prefetched data. This technique has been studied for a long time [6], and it can be realized through various approaches from hardware-based [4, 5] to software-based [7, 8, 9]. It is also applicable to shared-memory multiprocessors [9, 10, 11]. However, relatively few studies have been done for relationship between DSM and data prefetching, to our best knowledge. Meanwhile, we also noticed that the memory consistency semantic (MCS) models obeyed by currently existing DSM systems lack intrinsic support of data prefetching. In this paper we study data prefetching issues on DSM systems, develop our experimental prefetching-aware MCS, and present our preliminary yet positive and inspiring results.

The remainder of this paper is organized as follows. Section 2 is an overview of the context of this work. In Section 3, we describe the principle and characteristics of our new MCS and its associated data prefetching mechanisms. Results of performance evaluation by simulation are summarized in Section 4. Finally, a brief conclusion and possible future works will be given.

2 Related works

2.1 Memory consistency semantic models

In order to enhance performance of a DSM system, a number of memory consistency semantic (MCS) models have been proposed in recent years. Generally speaking,

*This research was partially supported by the National Science Council of Republic of China, under Grant NSC84-2213-E-002-007.

an MCS is a set of criteria describing when, where, and how to perform a specific operation. An MCS defines legal operations, controls perspective of processors, forces executing order of operations, and most importantly, checks and maintains memory coherence in a DSM system. It is not overemphasized to say that the underlying MCS determines the personality of a DSM system.

Among the many MCSes, perhaps the most intuitive one to us is the sequential consistency (SC), adopted by IVY [12] and many other multiprocessor and DSM systems. SC is natural to (virtually all) programmers familiar to von Neumann computation model -- the result of any valid execution under SC is the same as if the operations of all processors were executed in some sequential order, and the operations of each processor appear in this sequence [13]. While SC is a conceptually friendly MCS, however, its performance is not so satisfactory. One way to improve the efficiency of SC is to relax its ordering constraint. For instance, the processor consistency (PC) [14] uses out-of-order execution technique, allowing load operations to bypass pending store operations. Another example is the causal memory [15], which replaces the serializability in SC by causality. The pipelined RAM, or PRAM [16], only guarantees that store operations of a single processor are done in the order in which they were issued. It is an interesting contrast to SC: PRAM imposes very few restrictions to the ordering, making it quite efficient but also hard to program.

Some researchers take a rather different approach for better performance. In almost every parallel program, accesses to shared data appear in a kind of special program building blocks, called *critical sections*, whose boundaries are marked by interprocessor synchronization mechanisms. A critical section allows only one processor at a time to access shared data, and results of those shared accesses are revealed to other processors only at the moment that the current processor leaves this critical section. Bearing these facts in mind, researchers add synchronization primitives into their MCS designs and force coherence checking only at synchronization points; in other words, they follow the *coherence-at-synchronization* principle. The first member of the synchronization-equipped MCS family is the weak consistency (WC), described in [17, 18], which requires its synchronization operations to be sequentially consistent. A direct descendant of WC is the release consistency (RC) [14, 19], which provides two kinds of synchronization primitives: the acquire operations (used to enter critical sections) and release operations (to exit them). RC also relaxes ordering of synchronization primitives to be processor consistent. The success of RC causes debut of several variants, such as the lazy release consistency (LRC) which does not eagerly send the

released data to other processors [20]. The entry consistency (EC) takes a different approach, associating each synchronization variable with its guarded shared variables explicitly [21].

In summary, many MCSes have been proposed in the past to address the overhead problem of DSM systems. Either synchronization-free or synchronization-equipped, each MCS has its own design goals and advantages. Nevertheless, the trend tends to favor synchronization-equipped MCSes not only because they are possible to offer better performance [22], but synchronization primitives are vital to practically every parallel program, and modern DSM systems such as Munin [19] and Midway [21] usually integrate them as a part of system services. Readers are encouraged to refer to other sources [22, 23] for more details.

2.2 Data Prefetching Techniques

The cornerstone of prefetching is *predictability* of memory access: if the processor are currently using some memory location, we can probably figure out which is the next location to be accessed, and launch a (pre)fetch operation in advance. The most familiar kind of predictability is *spatial locality*, reflecting the static correlationship of one location to another. For example, the "naive" prefetching, also known as *one-block lookahead* (OBL) [6] is a simple and widely used spatial-locality scheme because a processor is likely to access the $(i+1)$ -th block upon accessing the i -th block. The OBL is so popular that some extensions to OBL have been proposed. One of them is FIFO *stream buffer* [24] that prefetches multiple consecutive blocks rather than one. Another is *stride-prefetching* that prefetches data at a predetermined or run-time calculated distance. Many scientific and engineering applications operate large amount of vectored (array) data, fitting the property of stride prefetching [25].

Data prefetching can be implemented by hardware, software, or some kind of combination of both. Hardware-based techniques determine prefetching location by information stored in some kind of special buffers or registers. For instance, Chen and Baer [4] proposed a dynamic approach utilizing a stride register called lookahead program counter (LA-PC). On the other hand, software-oriented approaches rely on optimizing compilers to insert prefetching instructions [7, 8]. In fact, pure hardware- or software-based prefetching schemes are rare because a special hardware cannot be fully utilized without software assistance, or vice versa. A good example can be found in [26], where compiler optimizing techniques are proposed for a novel processor with a small

content-addressable memory (CAM) as a scoreboard of stride distance.

Most discussions of data prefetching listed above are devoted to uniprocessor systems. On the parallel-computation side, data prefetching on shared-memory multiprocessors are of interest and extensively studied recently due to the growing demand of high-performance parallel computers. For example, Torrellas et al. [9] studied false sharing and spatial locality on multiprocessors; they also proposed several optimizing compiler techniques. Bianchini et al. [10] also studied cache line size of multiprocessors and concluded that large cache line size cannot be justified in most cases, confirming of result of [9]. For more thorough information of prefetching on multiprocessor (as well as uniprocessor) systems, the work of Gornish [5] is an excellent source.

3 The Aggressive Consistency (AC)

As we stated in the previous section, many MCSes have been proposed and each has its own features. However, a closer inspection reveals that existing MCSes have several common restrictions:

- ♦ *They induce unnecessary coherence checking and maintenance actions.* In synchronization-free MCSes such as SC and PC, every operation is forced visible to all processors -- even those irrelevant ones -- in this system, causing extra coherence actions. The synchronization-equipped MCSes are likely to handle this situation better; however, they are not free of this problem because they either group all coherence actions (including unnecessary ones) at the synchronization point, or rely on programmers to identify every sharing "pattern" to take proper actions.
- ♦ *Accesses of shared data are not well defined.* They must be confined in critical sections to ensure well-behaved. Outside the acquire-release fences of critical sections, shared accesses are considered "dangling" and the results will be totally undefined [21]. Such a limitation leads to programming inconvenience and debugging difficulty. For example, one may acquire a shared variable x and then forgets to release it: its effect will not be discovered until all other processes are blocked at acquiring x .
- ♦ *They do not support temporary inconsistency.* In fact, this restriction is a side effect of critical sections. Since only one processor is allowed to enter the critical section at a time, there is no chance to produce even temporarily two or more versions of the same shared data object. However, in many cases allowing temporary inconsistency

can reduce coherence maintenance overhead and improve system performance.

To address all problems at a sweep, a new MCS called the *aggressive consistency* (AC) will be designed. In the following subsection, we will describe the motivation and underlying principle, specify the definition, and discuss the data prefetching features, of the aggressive consistency model.

3.1 The Principle of Coherence-on-Demand

One of the features of shared memory access operations, especially STORE operations, is seldom referred: the propagation property. When one STORE access is issued, the result of this STORE will eventually propagate to every corner of this system; otherwise at least one notification of this STORE will be spread. However, there is no means to control or even know *when* this objective is done, or in DSM terminology, when the STORE is *performed* [14]. (An operation O is considered *performed with respect to a processor P* if and only if the effect of O is visible to P , and O is considered *globally performed*, or simply *performed*, if and only if O is performed with respect to all processors in the system.) For example, an issued STORE cannot be postponed or withdrawn even in SC, and the complete time cannot be predicted in PC or PRAM. Synchronization-equipped MCSes try to remedy this problem by introducing the RELEASE operation: any (shared) STORE will be performed definitely after the associated RELEASE. However, another problem arises, namely the dangling shared access stated above. Moreover, critical sections are a kind of exclusive resources contended by all processors, one processor cannot hold the result of a shared memory STORE too long without RELEASE it. This reduces the effectiveness of RELEASE.

Our answer to such situations is to redefine the propagation property. In our new scheme, result of any STORE operation will never be exposed to other processors without any explicit request. In other words, temporary inconsistency is allowed under this new scheme. If one processor decides to get the current version of some shared data object, it is responsible for locating and fetching that object *by itself* through an explicit command. Such command is a unary synchronization primitive that searches for an agreement of shared data objects among processors, and is named SYNC after its functionality. The syntax of SYNC is as follows (the real format depends on the language implementing SYNC):

.....SYNC x

where x is a shared data object of some type.

The SYNC command is designed to make a consistent view of its associated data object. To achieve this, the

programmer should SYNC a shared data object before use it. A piece of code fragment containing SYNC may look like this:

```
....{program statements}....
SYNC x;      // x is a shared variable
....        // code irrelevant to x
proc(x);     // code that uses x
x=x+1;       // can be of any style
....{other program statements}....
```

Now we explain why the SYNC operation can address the problems mentioned in the beginning of this section. First, it is obvious that the dangling shared memory access problem no longer exists. For the same reason, temporarily inconsistency is supported now: updating a shared variable without SYNCing it immediately is legal, although it should be used with care. The third and most important is that SYNC reduces unnecessary coherence overhead. To validate this assertion, observe that only the demanding processor(s), who issues the SYNC operation, requires to get a consistent view of the SYNCed objects; any other processor is irrelevant to the very synchronization operation. Furthermore, only those data objects specified explicitly in the SYNC operations will participate in coherence maintenance actions, further reducing the probability of unnecessary sharing. To contrast, critical-section style programming uses an extra "synchronization" variable to protect a group of shared data objects (even those who need no coherence checking), inducing more unnecessary sharing. This concept--updates never propagate until SYNCed and only processors launching SYNC participate in coherence maintenance--is called the principle of *coherence-on-demand*, which is the essence of AC, our new MCS. Note that after the SYNC point and before the SYNCed data object is really used, that data object is subject to change. A program utilizing SYNC operations should be aware of this possibility. A typical method to deal with such nondeterminism is to avoid it by allowing only one new update of a shared object to be generated at a time (most conventional parallel programs belong to this category). Another approach is to take advantage of this feature (for example, a heuristic searching program).

3.2 Characteristics of AC

After introducing the SYNC operation, we proceed to investigate other properties of AC. Under AC, interprocessor coherence is controlled by synchronization operations. More concretely, AC specifies the ordering relationship among ordinary access operations, i.e. LOADs and STOREs, and synchronization operations,

including SYNCs, ACQUIREs and RELEASEs. Traditional ACQUIREs and RELEASEs are not discarded because it is still required to serialize the exclusive uses of shared data, and keeping these synchronization operations increases the compatibility between AC and other MCSes. Nevertheless, the syntax and semantic of ACQUIREs and RELEASEs are different from other MCSes. In AC, special synchronization variables do not exist any longer; if a processor wants to exclusively access a shared data object, it must directly ACQUIRE, use, and then RELEASE that data object. In addition, ordinary accesses within an ACQUIRE-RELEASE pair may or may not be forced completion at the RELEASE point: only those who access the ACQUIRED variable (called *bound* accesses) will be. All these changes are intended to make the ACQUIRE-RELEASE mechanism conform the principle of coherence-on-demand.

Now we are ready to present the definition of AC; these definitions are originated from our earlier work [27]. In general, a memory subsystem is considered aggressively consistent if it obeys the following conditions:

- ◆ Before any bound memory access is allowed to be performed with respect to any other processor, the associated ACQUIRE must be performed.
- ◆ Before an ACQUIRE is allowed to be performed with respect to any other processor, the most recent RELEASE of the same shared object must be performed with respect to the processor issuing the acquire operation.
- ◆ Before a RELEASE is allowed to be performed with respect to any other processor, all previous bound memory accesses after the associated ACQUIRE in program context must be performed.
- ◆ Before an ordinary operation is performed, all previous SYNC operations to the same object accessed must be performed.
- ◆ All synchronization primitives follow the *partial causal order*; that is, they either follow dependency order, or they are in the same program except that ACQUIREs can bypass outstanding RELEASEs if they are mutually independent.

For the implementation aspects, it is surprising that implementing a DSM system under the AC criteria shown above is not too difficult. Since programs under AC directly ACQUIRE and RELEASE demanded shared variables, any distributed locking mechanism is sufficient to provide such services. The RELEASE will merely act as an end mark of a critical section and do nothing at all, just as in LRC or EC. As to SYNC, a timestamp is generated when a *free* (unbound) STORE access occurs, and those

timestamps are compared at the SYNC point to determine which copy is the freshest. If the program guarantees no more than one free copy will be generated for each variable between two synchronization points, the complexity of SYNC implementation can be further reduced: a SYNC broadcasts the new value (or invalidation) if the associated variable is updated, or searches for a new copy otherwise. Although such a "fast" implementation is not strictly adhere to the requirements of SYNC, the run-time overhead is substantially lowered and better performance can be achieved.

3.3 Data Prefetching and AC

Besides potential performance edge over previous MCSes, AC possess another advantage: data prefetching support. Here we mean *interprocessor* prefetching, namely prefetching shared data among processors.

Traditionally, there are two major obstacles of incorporating data prefetching into DSM designs. The first is the risk of increasing communication traffic and execution overhead. Since communication bandwidth is, more or less, an exclusive resource contended by all processors, unsuccessful prefetching actions will bring stress to the communication subsystem. The situation becomes more serious for DSMs than uniprocessor or tightly-coupled multiprocessor systems because the communication overhead is substantially higher than pure memory access overhead, and so does the mis-prefetch penalty. Even the stride or adaptive prefetching schemes cannot totally eliminate the possibility of mis-prefetching. Furthermore, there is no standard method to predict whether prefetching is beneficial for general systems, not to mention already-complicated DSMs.

The other difficulty comes from the critical-section synchronization mechanism: a processor cannot access a shared data object without ACQUIRE-ing it. That is, a prefetching operation must be attached to the entry point (an ACQUIRE) of some critical section, otherwise the prefetching will induce extra synchronization and hence block other processors. However, for the sake of progress a processor should free the current critical section as soon as possible, which means a shorter critical section is preferable. Consequently, in practice there is not enough time for the prefetched data to arrive before use, making the prefetching null and void.

Under AC, however, the prior difficulties can be easily overcome. Since a shared data object should be SYNCed in advance and SYNC operations will not obstruct other processors, the SYNC point is the best indicator to launch a prefetching action. This policy is called *prefetch-on-demand* scheme, which is named after the coherence-on-demand property of the SYNC operation. To

accomplish this, the implementation of SYNC should be *non-blocking* to allow the succeeding operations be executed simultaneously. The following code fragment shows this concept.

```
....{program statements}....
a=b+c;          // code irrelevant
SYNC x;         // non-blocking SYNC
y=func(z);      // code irrelevant to x
                // overlapped with SYNC
proc(x);        // code that uses x
....{other program statements}....
```

The only technique required for prefetching under AC is the non-blocking capability of SYNC. One possible approach of implementation is to invalidate the virtual memory location where the SYNCed variable resides, and to trap all access to that location before the prefetched data arrives. Such a capability can be completely integrated into the AC mechanisms without any other special requirements, like hardware lookahead registers or compiler-inserted prefetching instructions. Also because no extra prefetching instruction is ever generated, communication traffic will remain the same; that is, prefetching at SYNC is penalty-free. The remaining question is whether prefetching is really beneficial under AC. We answer this question in the next section.

4 Performance Evaluation and Analysis

4.1 Methodology

To verify the effectiveness of AC with prefetching, we conducted a simulation of a DSM environment of several workstations connected by Ethernet. In order to simulate the traffic and collisions of Ethernet, we use the simplified analysis result of binary exponential backoff algorithm of the IEEE 802.3 standard [28] to obtain the latency of each network transaction at run-time. Network parameters used in our simulation are summarized in Table 1.

Table 1: Network parameters of Ethernet

Parameter	Value
Communication Bandwidth (Mbits/second)	10
Packet Head Size (Bytes)	32
Minimal Packet Data Length (Bytes)	32
Maximal Packet Data Length (Bytes)	1,500
Packet Round-Trip Latency (microsecond)	51.2
Packet Pack/Unpack overhead (microsecond)	51.2

Four MCSes are investigated in this simulation: WC, LRC, AC, and AC with prefetching. All four MCSes in this simulation are synchronization-equipped; the reason that synchronization-equipped ones are favorable has been described in Section 2. In order to execute the same benchmark suite without modifying the source code, SYNCs are legal under all four MCSes except that they are treated as NOPs (no operations) under WC and LRC. Besides, we incorporate into these MCSes as many performance optimization options as possible, which are summarized in Table 2.

Table 2: Optimization options for simulated MCSes

Optimization/MCS	WC	LRC	AC	AC w/ prefetch
Read Caching	Yes	Yes	Yes	Yes
Write Invalidation	Yes	Yes	Yes	Yes
Coherence-at-Synchronization	Yes	Yes		
ACQUIRE bypass RELEASE		Yes	Yes	Yes
Lazy Updating		Yes	Yes	Yes
Coherence-on-Demand			Yes	Yes
Data Prefetching at SYNC				Yes

4.2 The Benchmark Suite

Our benchmark suite consists of three programs. In order to be executed under WC and LRC, all benchmark programs utilize ACQUIRE and RELEASE operations to synchronize shared data uses, and SYNCs are manually inserted in proper positions for prefetching. The first program, *MatMul*, is to multiply 32 matrices with 256*256 doubles each. The computation of *Matmul* is arranged as follows. First, divide matrices into equal-sized groups, and assign one processor to each group to get the product matrix of that group. Then we pairwise multiply those intermediate product matrices in parallel, and repeat this step until the final result is obtained. This strategy is the *parallel-prefix* method which can be found in classical parallel algorithm texts [29] [30]; it is also suitable for other divide-and-conquer style algorithms such as Quicksort and FFT.

The second benchmark program is *Neuron*, running a pre-trained artificial neural network to recognizes 200 patterns. The configuration of the simulated neural network is a multilayer feedforward network of 32 layers with 256 elements ("neurons") in each layer. Each neuron is supplied by a set of inputs which are outputs of the neurons in the previous layer (except the first-layer ones,

which obtain the inputs directly from the patterns). Each input is multiplied by a corresponding weight value which is precomputed ("trained"), and the weighted sum of these inputs determines whether this neuron is activated. Since input patterns flow through the neural network in a fixed direction, it is natural to partition the whole network by layers and establish a pipeline computation model. The sharing occurs between each stage of the pipeline, in a fashion of producer-consumer relationship because data values are generated in one stage and used in the next stage.

The last one is *Relax*, which is an iterative process on a square matrix of 1536 x 1536 elements, repeated 100 times in each execution of *Relax*. During each iteration, the computation task of *Relax* is to replace each matrix element with the average of eight nearest neighbors and the element itself. This process is useful in many directions. For example, low-pass filtering of an image is a representative application of *Relax*, and a 3D-*Relax* can be used to determine the temperature gradients in a closed space. To parallelize *Relax*, we partition the whole matrix into several horizontal strips and assign one processor for a strip; thus sharing occurs across the boundaries of those strips.

4.3 Results and Analysis

Figure 1 illustrates the simulated result of *Matmul*, in elapsed executing time (seconds) versus number of processors. The performance result of *Matmul* matches our expectation: data prefetching on SYNC consistently improves the efficiency of AC, which already performs better than LRC and WC. Quantitative observation shows that for *Matmul*, prefetching brings moderate performance gain to AC, from less than 2% (2 processors) to about 15% (16 processors). The performance gain cannot be very high because the prefetched data cannot all arrive in time. In the merging stage of parallel-prefix computation, intermediate product matrices are pairwise multiplied, which implies one matrix must be remotely accessed. In *Matmul*, the amount of data of a matrix is so large (512K bytes each) that although the remote matrix can be prefetched one row at a time, the first arrived rows are sooner or later "used up" and thus the processor must wait for the next row. Also notice that the overhead of WC is relatively higher than other three MCSes; the other two benchmark programs also deliver similar results. The difference becomes obvious when the number of processors increase -- we even got negative speedup from 8 to 16 processors. This phenomenon comes from the limitation of WC definition [18], which states that synchronization operations must be globally performed in

sequential consistency.

Figure 1: Performance of *Matmul*

The simulation result of *Neuron* is charted in Figure 2. The performance curves are similar to those of *Matmul*; however, while data prefetching still improves the performance of AC, the gain is not so evident. A closer inspection reveals that the communication traffic of data packets only (acknowledges excluded) is quite low. Recall that we set up a virtual pipeline to execute *Neuron*. In each iteration, one stage of pipeline fetches the raw outputs (not links; link values are private to each stage) from its predecessor, or equivalently 256 double floating-point numbers, which can be put in two long packets. Since the traffic is already low, enabling data prefetching does not help too much to performance.

Figure 2: Performance of *Neuron*

The scenario is different for *Relax*, whose simulation result is shown in Figure 3. We observed that data prefetching brings a large performance improvement to AC, especially when the system is scaled up: the efficiency almost doubles for 16 processors. Recall that in *Relax*, the whole matrix is partitioned into horizontal strips and each processor is responsible to update one designated strip. During each iteration, each processor must remotely read the cells lying on the two nearest rows outside the upper- and lower-boundaries of its own strip to update its own boundary cells. Such remote access latencies are quite large (due to the size of the matrix) and

inevitable even in AC because these two rows are just updated (and thus owned and cached) by other processors during the previous iteration. Fortunately, the latencies can be completely recovered by launching two prefetching operations and overlapping them with the processing time of the inner non-boundary cells. Since the cross-boundary access latencies are the source of most communication overhead in *Relax*, recovering them by prefetching delivers significant performance improvement.

Figure 3: Performance of *Relax*

5 Conclusion and Future Works

In this paper we present the aggressive consistency (AC), a new memory consistency semantic model that utilizes coherence-on-demand principle and can easily supports data prefetching, for distributed shared memory systems. Simulation results show that AC consistently offers better performance than other MCSes, and data prefetching can further improve the efficiency of AC to a various extent, depending on the application.

In the near future, our objective is to obtain real world performance characteristics of AC and AC with prefetching. An experimental AC-based DSM system, running on SPARC workstations connected by Ethernet, is now under construction. Once our prototype system is stabilized, we will report the results as soon as possible.

Acknowledgement

The authors wish to appreciate the valuable assistance and encouragement of the anonymous referees.

References

- [1] Bill Nitzberg and Virginia Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", *Computer*, August 1991, pp. 52~60.

- [2] Michael Stumm and Songnian Ziou, "Algorithms Implementing Distributed Shared Memory", *Computer*, May 1990, pp. 54~64.
- [3] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber, "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems", *ACM Operating System Review*, Vol. 14 No. 3, May 1990, pp. 40~67.
- [4] Tien-Fu Chen, *Data Prefetching for High-Performance Processors*, PhD dissertation, Department of Computer Science and Engineering, University of Washington, July 1993.
- [5] Edward H. Gornish, *Adaptive and Integrated Data Cache Prefetching for Shared-Memory Multiprocessors*, PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.
- [6] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14 No. 3, 1982, pp. 473~530.
- [7] David Callahan, Ken Kennedy and Allan Porterfield, "Software Prefetching", *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 40~54.
- [8] T. C. Mowry, Monica S. Lam and Anoop Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 62~73.
- [9] J. Torrellas, Monica S. Lam and John Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches", *IEEE Transaction on Parallel and Distributed Systems*, Vol. 33 No. 6, June 1994.
- [10] Ricardo Bianchini and Thomas J. LeBlanc, "Can High Bandwidth and Latency Justify Large Cache Blocks in Scalable Multiprocessors?", *Proceedings of 1994 International Conference on Parallel Processing*, Vol. I, August 1994, pp. I-258~I-262.
- [11] Rafael H. Saavedra, Weihua Mao, and Kai Hwang, "Performance and Optimization of Data Prefetching Strategies in Scalable Multiprocessors", *Journal of Parallel and Distributed Computing*, Vol. 22 No. 3, September 1994, pp. 427~448.
- [12] Kai Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", *ACM Transaction on Computer Systems*, Vol. 7 No. 4, November 1989, pp. 321~359.
- [13] Leslie Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem", *Communications of the ACM*, Vol. 17 No.8, August 1974, pp. 453~455.
- [14] Kourosh Gharachorloo, Daniel Lenoskj, Phillip Gibbons, Anoop Gupta and John Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proceedings of the 17th International Symposium on Computer Architectures*, June 1990, pp. 15~26.
- [15] Mustaque Ahamad, James E. Burns, Phillip W. Hutto and Gil Neiger, "Causal Memory", from *Distributed Algorithms*, Lecture Notes on Computer Science #597, Springer Verlag Co., 1990, pp. 9~30.
- [16] Richard J. Lipton and Jonathan S. Sandberg, "PRAM: A Scalable Shared Memory", Technical Report 180-88, Dept. of Computer Science, Princeton University, September 1988.
- [17] Michel Dubois, Christoph Scheurich, and Faye A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors", *Computer*, February 1988, pp. 9~21.
- [18] Richard N. Zucker, *Relaxed Consistency and Synchronization in Parallel Processing*, PhD dissertation, Department of Computer Science and Engineering, University of Washington, December 1992.
- [19] John B. Carter, John K. Bennett, and Willy Zwaenepoel, "Implementation and Performance of Munin", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991, pp. 152~164.
- [20] Pete Keleher, Alan L. Cox and Willy Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory", *Proceedings of 19th Annual International Symposium on Computer Architecture*, 1992, PP. 13~21.
- [21] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon, "The Midway Distributed Shared Memory System", *Proceedings of IEEE COMPCON 93*, 1993, pp. 528~537.
- [22] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall Inc., 1995.
- [23] David Mosberger, "Memory Consistency Models", Technical report TR 93/11 of Dept. of Computer Science of the Univ. of Arizona. (An earlier version of this work appears on *ACM Operating System Review*, Vol 17, No. 1, January 1993, pp. 18~26.)
- [24] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a small Fully-Associative Cache and Prefetch Buffers", *Proceedings of 17th International Symposium on Computer Architecture*, 1990, pp. 364~373.
- [25] J. W. C. Fu and J. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories", *Proceedings*

- of 18th International Symposium on Computer Architecture, 1991, pp. 54~63.
- [26] Chi-Hung Chi, "Compiler Optimization Technique for Data Cache Prefetching Using a Small CAM Array", *Proceedings of 1994 International Conference on Parallel Processing*, Vol. I, August 1994, pp. I-263~I-270.
 - [27] Alexander I-Chi Lai and Chin-Laung Lei, "The Formalization and Hierarchy of Memory Consistency Models", *Proceedings of International Computer Symposium*, December 1994, pp. 1060~1066.
 - [28] Andrew S. Tanenbaum, *Computer Networks* (2nd edition), Prentice-Hall Inc., 1989.
 - [29] Selim G. Akl, *The Design and Analysis of Parallel Algorithms*, Reading, Prentice-Hall Inc., 1989.
 - [30] Joseph JáJá, *An Introduction to Parallel Algorithms*, Reading, Addison-Wesley Publishing Company, 1993.
 - [31] Anant Agarwal and Anoop Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH", *Proceedings of Conference on Measurement and Modeling of Computer Systems*, May 1988, pp. 215~225.
 - [32] Brett D. Fleisch and Gerrald J. Popek, "Mirage: A Coherent Distributed Shared Memory Design", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989, pp. 211~223.
 - [33] Anne Dinning, "A Survey of Synchronization Methods for Parallel Computers", *Computer*, July 1989, pp. 66~77.
 - [34] John K. Bennett, John B. Carter, and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", *2nd Symposium on Principle and Practice Of Parallel Programming*, 1990, pp. 168~176.
 - [35] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models", *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. I, August 1991, pp. I-355~I-364.
 - [36] Sarita V. Adve and Mark D. Hill, "A Unified Formalization of Four Shared-Memory Models", *IEEE Transaction on Parallel and Distributed Systems*, Vol. 4 No. 6, June 1993, pp. 613~624.
 - [37] David K. Puolsen and Pen-Chung Yew, "Data Prefetching and Data Forwarding in Shared Memory Multiprocessors", *Proceedings of 1994 International Conference on Parallel Processing*, Vol. II, August 1994, pp. II-276~II-280.
 - [38] Tien-Fu Chen and Jean-Loup Baer, "A performance Study of Software and Hardware Data Prefetching Schemes", *Proceedings of the 21th International Symposium on Computer Architecture*, November 1994, pp. 223~232.