JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 14, 725-741 (1998)

Design and Implementation of Cohesion

CE-KUEN SHIEH, JYH-CHANG UENG, AN-CHOW LAI,
TYNG-YUE LIANG AND SU-CHEONG MAC
Department of Electrical Engineering
National Cheng Kung University
Tainan, Taiwan 701, R. O. C.
E-mail: shieh@ee.ncku.edu.tw

This paper describes a prototype DSM system called Cohesion which supports two memory consistency models, namely, Sequential consistency and Release consistency, within a single program to improve performance and support a wide variety of parallel programs for the system. Memory that is sequentially consistent is further divided into object-based and conventional (page-based) memory, where they are constructed at the user-level and kernel-level, respectively. In object-based memory, the shared data are kept consistent in terms of the granularity of an object; this is provided to improve the performance of fine-grained parallel applications that may incur a significant overhead in conventional or release memory as well as to eliminate unnecessary movement of pages which are protected in a critical section. On the other hand, the Release consistency model is supported in Cohesion to attack the problem of excessive network traffic and false sharing. Cohesion programs are written in C++, and the annotation of shared objects for release and object-based memory is accomplished by inheriting a system-provided base class. Cohesion is built up on a network of Intel 486-33 personal computers which are connected by a 10Mbps Ethernet. Three application programs, including Matrix Multiplication, SOR, and N-body, have been employed to evaluate the efficiency of Cohesion. In addition, a Producer-Consumer program has been tested to show that the object-based memory will benefit us in a critical section.

Keywords: distributed shared memory, multiple consistency protocols, upcall mechanism.

1. INTRODUCTION

A distributed Shared Memory (DSM) system is a memory management system which provides a shared memory abstraction for the users of a distributed system. A DSM system makes programming on a distributed memory multiprocessor system easier with the shared variable paradigm. In addition, the ease of building up the system as well as the high scalability of a loosely-coupled multiprocessor system is kept in a DSM system. In all fairness, it is not easy to construct an efficient DSM due to considerations like the choice of memory consistency, granularity, programming ease, etc.

In order to emulate the abstraction of shared memory, consideration of memory consistency models is important. We may state that the stricter the model

Received July 22, 1995; accepted June 18, 1998
Communicated by Chuan-lin Wu.

is, the more requirements the system has to meet; as efforts have been invested on different memory consistency models by the forerunners. These imposed requirements may induce unnecessary overheads. Nevertheless, there are advantages and disadvantages in each memory consistency model. A DSM system that employs a strict consistency model, such as Sequential consistency, allows programs written for shared memory multiprocessors to be ported easily. Users may know very little about the system. However, the performance of the system may not be good enough. On the other hand, a DSM system that supports a weaker consistency model may have better system performance and reduced network traffic; but there are some constraints the programmer must keep in mind, such as the use of "enough" synchronization operations. In conclusion, a system that supports a single memory consistency may not compromise the efficiency and flexibility in programming. Therefore, it is substantial for an efficient DSM system to support both strict and weak consistency models. Then, the users will be able to select the appropriate consistency model in programming for different shared data without loss of performance.

Another consideration in designing a DSM system is the choice of granularity. Without the assistance of a compiler or preprocessor, a page is typically suitable. However, when a page of 4Kbytes on a PC486 is adopted in a sequentially consistent system, contention will reduce the performance heavily. This is especially an issue if there are pages which are falsely shared among processors. This will cause a phenomenon called page thrashing that may not happen in Release consistency model [1, 7].

Release consistency may efficiently relieve us of the trouble of false sharing in coarse-grained data only. For fine-grained data, neither release nor sequential (page-based) consistency can efficiently reduce latency and frequency in transmission. For example, even though only one processor is assured to enter the protected region in a parallel program, (1) additional overheads are required to maintain the small shared data protected in a critical section when buffering and a delayed update strategy are employed to enforce release consistency because updates have to be flushed to other caching nodes in release consistency; and (2) page thrashing occurs in page-based sequential consistency due to redundant page faults and requests initiated in page-based sequential consistency. Accordingly, longer latency in a non-parallelizable critical section will definitely affect the system performance heavily.

In this paper, we present a distributed shared memory system called Cohesion. Cohesion supports both sequential and release consistency. It easily overcomes the aforementioned problems by taking advantage of a combination of the two consistency models. There are two features which distinguish Cohesion from other DSM systems. First, two kinds of granularity are applied for sequential memory in Cohesion, ie., *page* and *object*. Page-based memory is unstructured and shared at the physical layer while object-based memory structures the shared data as an object. Typically, object-based memory alleviates the overhead of accesses in a critical section and benefits the synchronization operations. Second, Cohesion is an object-oriented DSM. Annotation of shared objects in Cohesion can, therefore, be done by inheriting a system-provided base class, where the types of objects will be recorded in the system. This prevents us from modifying the compiler or creating

a special purpose preprocessor. Moreover, the infrastructure of Cohesion is composed of a cluster of identical Intel 486-33 processors connected by a 10Mbit Ethernet.

This paper is organized as follows. An overview of the system is given in section 2. Section 3 discusses the design methodology of Cohesion. Subsequently, the implementation of the system is described briefly in section 4. In section 5, the performance of Cohesion is shown. Section 6 presents features that set Cohesion apart from others. Finally, section 7 gives a conclusion of our work.

2. SYSTEM OVERVIEW

Basically, Cohesion encompasses an object-oriented run-time thread system, similar to PRESTO [2], that provides a convenient parallel programming environment at the user-level and a kernel based on iRMK (intel Real Time Kernel) for each node where the coherence manager is incorporated. To implement the idea of DSM, another three important components are constructed, as illustrated in Fig. 1; there are reliable communication, memory coherence and kernel supporting subsystems, where a kernel thread is assigned to execute each of them. In addition, two other kernel threads are employed to execute the user-level threads and initialization of the user program.

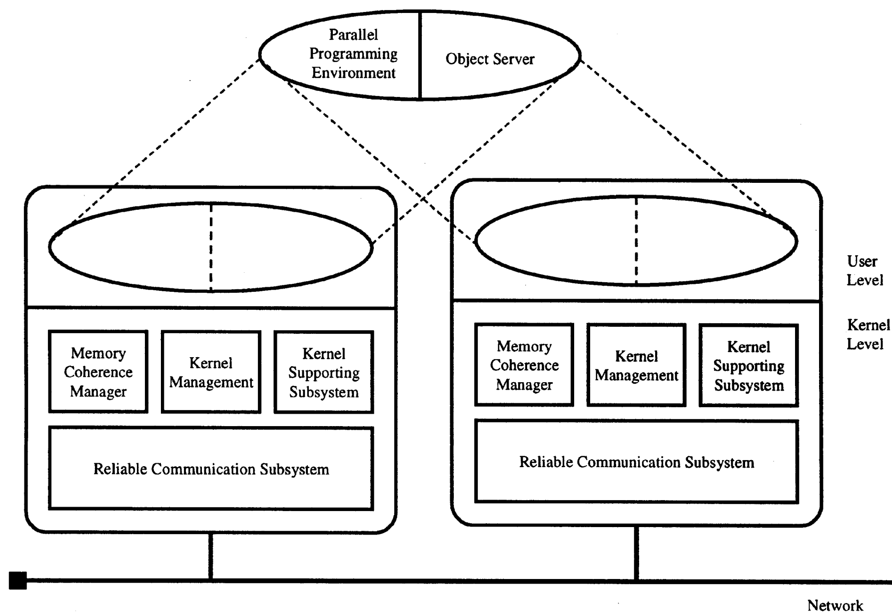


Fig. 1. System overview.

The reliable communication component is responsible for providing a rapid and reliable vessel to exchange messages in the emulation of shared address space since DSM is very sensitive to the communication delay. On the other hand, the

memory coherence component is particularly designed to enforce the consistency of virtually shared memory among the processors and is adaptive to different consistency protocols employed in Cohesion. The kernel supporting subsystem facilitates the Release consistency protocol as well as the object-based sequential memory. In other words, it provides a bridge, which is called the upcall mechanism, to integrate the user-level and kernel-level for the purpose of constructing a more efficient DSM system.

In a user-level thread system, affinity scheduling is implemented, and programmers are able to locate user-threads to any node in the system statically or dynamically. Furthermore, an object server is erected which works co-operatively with other scheduling objects in PRESTO. There are two goals for an object server : (1) to support synchronization operations, such as queue-based locks, which will be described in section 4; and (2) to support the release consistency protocol. This object server in Cohesion is not run as a user thread but is incorporated into the runtime thread system as an object. Therefore, it is worth noting that data maintained by the object server on different processors are not related.

2.1 Shared Address Space

Cohesion provides a global address space for users. Typically, the shared address space of a program running on Cohesion can be divided into three parts. They can be categorized as object-based, conventional, and release memories, as shown in Fig. 2. The coherence protocol of the object-based memory employs a migratory protocol; it is dedicated for shared memory protected in a critical section and synchronization of objects. While that in the conventional memory uses write-invalidation; this simplifies the task for developers of locating the shared system data, such as the global queue in the affinity scheduler, and of porting a program written for shared memory multiprocessors to Cohesion. For release memory, a synonym of write-update that remedies the problem caused by false sharing is adopted. In particularly, the size of each memory can be dynamically changed by the user. However, they are handled separately.

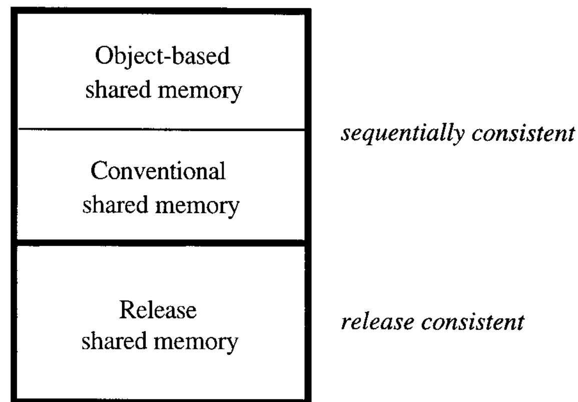


Fig. 2. Shared address space in Cohesion.

2.2 Integrating Convenience in the User-level and Kernel-level

In DSM, since a page is usually taken as a promising shared granularity, the detection of writes in a kernel is simple and easy to implement. However, the enforcement of consistency is not necessarily efficient in the kernel because this will induce overhead especially when buffering techniques and more complicated algorithms are employed to meet the consistency requirements. Alternatively, this can be accomplished at the user-level, where much overlapping seems to be in the run to reduce the impact of communication in the consistency handlers by making use of convenient and low cost context switching of user threads. Also, if two or more consistency models are managed by a single kernel, the kernel will apparently become large and more difficult to debug. As a result, to facilitate management of the release memory, most of the required state information can be kept in user space while detection of writes is retained in the kernel.

Overall, the flexibility and convenience provided by the user-level and kernel-level approaches are incorporated into Cohesion. Conventional memory is maintained throughout the kernel while release memory is enforced at the user-level with kernel support. For object-based memory, everything is carried out at the user-level without kernel intervention. To make this idea work, an object-server in the run-time threads system serves as an intermediary.

2.3 Annotation

As mentioned earlier, Cohesion distinguishes itself from others in the annotation of shared data. Every shared object in Cohesion is considered to be an item in a class. When this object inherits the base class specially provided by the system, the memory consistency model as well as the coherence protocol applied to the object is set. Therefore, no effort in compiler or preprocessor is needed. However, the type of a shared object may not be changed once it is set.

Nevertheless, for conventional shared data, there is no need to specify. In Cohesion, they are kept sequentially consistent by default. This is reasonable and may simplify implementation. Thus, only the objects in release and object-based memory have to be annotated in Cohesion. Overall, doing an annotation by inheriting a base class is not cumbersome because it is a unique feature of object-oriented programming. To briefly illustrate how all this works, an example is shown in Fig. 3.

```
class Matrix : public Release { // declaring the Matrix as a Release object.
    private : shared data items;
    public : member functions;
};
class Sor { // declaring the Sor as a sequential object.
    private : shared data items; // this is default without inheriting.
    public : member functions;
};
```

Fig. 3. An example of annotation.

3. DESIGN METHODOLOGY

Since network latency is a major concern, we need to carefully choose algorithms and strategies for the coherence protocol and for buffering of updates. When dealing with presentation of writes in the system, either write update or write invalidate is chosen for the coherence protocol. In the write invalidation protocol, short messages are sent for invalidation; however, requests for data and invalidations will occur too often when the data are heavily referenced by different processors. On the other hand, in the write update strategy, long update messages are propagated to other nodes that share the data; this may cause additional network traffic when other processors do not reference the data often.

3.1 Sequential Consistency Model

The Sequential consistency model applied in Cohesion follows the definition given by Leslie Lamport [8]. In Cohesion, sequential consistency of conventional memory is maintained at a page granularity, and the algorithm used to implement this model is similar to that of IVY [10]. For that of object-based shared memory is maintained at an object granularity.

Definition 1: Sequential Consistency (SC)

A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

3.1.1 Conventional Memory

In our system, the write invalidation approach is employed to present a write since this protocol is more efficient in a sequential consistency algorithm. The management of the pages' ownership applies a distributed algorithm [9]. Moreover, to deal with page thrashing, a time window paradigm [6] is employed. The idea is to hold a page in a processor for at least a quantum of time even if there is a write request for the page.

3.1.2 Object-based Shared Memory

Object-based shared memory is designed to handle fine-grained shared data protected in a critical section and synchronization of objects. Typically, all shared objects allocated in this memory are assumed to be migratory, where an object is migrated to the first requester and only one owner exists among the processors.

Because the object-shared memory is handled at the user-level, the detection of writes can not make use of any hardware supports as in conventional memory. To make this process work, some requirements have to be known by the programmers. Since Cohesion provides an object-oriented programming environment, users may easily declare an object-based shared data as the data item in a user-defined

class which inherits the base class dedicated for object-based memory. Then, users may create this shared object dynamically by overloading the operator *new()*, such that the object server may seize this opportunity to insert an entry for it in an object table. Furthermore, access to this object should be done through its member functions. Within each member function, before the shared accesses are allowed to be initiated, a function *check()* provided in the object server has to be called to check if the data item resides. If not, the data item will be requested. When the member function has finished accessing the data, another function, *release()*, has to be called to notify others that the data are available. Within the period of time between *check()* and *release()*, the object being accessed cannot be migrated. This will assure the integrity of the accessed data.

3.2 Release Consistency Model

Basically, the Release consistency model employed in Cohesion is based on the definition given by Gharachorloo [7]. Experience with release consistent memories indicates that it is necessary to send messages before announcing that the release operation is performed. Since there are two or more threads on a processor, threads may be preempted instead of waiting for acknowledgement. Based on this fact, the effect of overlapping communication latency may also be achieved by running another ready thread. Consequently, the requirement that the synchronization operations be processor consistent is not so essential. In Cohesion, the synchronization operations in the Release model are kept sequentially consistent.

To deal with updates by a processor, the delay update scheme is employed, and updates are presented to other caching nodes during release operations. A consistency protocol for this memory can be stated as a write-update.

Definition 2: Release Consistency (RC)

(a) Before an ordinary LOAD or STORE access is allowed to be performed with respect to any other processor, all previous acquire accesses must be performed.
(b) Before a release access is allowed to be performed with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed.
(c) Special accesses are sequentially consistent with respect to one another.

During the execution of a parallel program on Cohesion, a release object may be allocated dynamically through an overloaded *new()* operator in C++. The object server is then invoked by assigning several pages that are large enough from the pre-allocated release memory pool, as discussed in 4.1, for the object.

4. IMPLEMENTATION

In the following sections, only the most subtle portions in implementation will be introduced. They include the routines used to allocate shared address space, to

handle page faults together with the time the upcall is invoked, and to also handle object management at the user-level as well as synchronization in Cohesion.

4.1 Allocation of Shared Address Space

In Cohesion, efforts are devoted to reduce the possibility of thrashing of system data. Allocation of memory is specially designed in two levels, the kernel and user levels. Initially, shared memory space is reserved and is kept sequentially consistent by the coherence manager. Subsequently, part of this space is allocated as object-based shared memory during the initialization stage of the thread system. In order to provide release memory, pages of shared memory that are large enough to meet user requirements are again pre-allocated. Eventually, this requires a manager, the object server mentioned previously, who is aware and takes care of these pages at the user-level. The access right, ownership and copyset of each of these pages are kept individually in the private space of each processor by the object server; this will be discussed in 4.3. For release memory, the boundaries of shared pages are registered by instructing the coherence manager to handle them differently from conventional pages at each node. In this situation, all future accesses to the release memory will be intercepted and redirected to the object server. The advantage of doing so has already been described in section 2.2 and details are given in 4.2. In Fig. 4, routines described for initialization are shown.

```
void thread_system_initial() {
  ......
  start_co-schedulers_of_thread_system_on_each_node()
  clear_object_based_memory_description_table()
  allocate_object_based_shared_memory()
  clear_release_page_description_table();
  allocate_release_shared_memory()
  register_boundary_of_release_memory( start, end );
  ......
}
```

Fig. 4. Initialization for release memory.

4.2 Page Fault Handling and Upcalling

The first unique feature that Cohesion possesses is its support of two memory consistency models in the kernel of the system. In a sense, two memory handlers in the coherence manager are used to cope with memory violations. In the following, the focus is the release memory since the conventional part [10] is very familiar. Events in the manager are shown in Fig. 5.

In particular, when there is a read fault of release memory, the fault handling response is similar to that in a conventional memory handler. A page will be fetched, and the access right will be set as read-only. Since a read miss does not

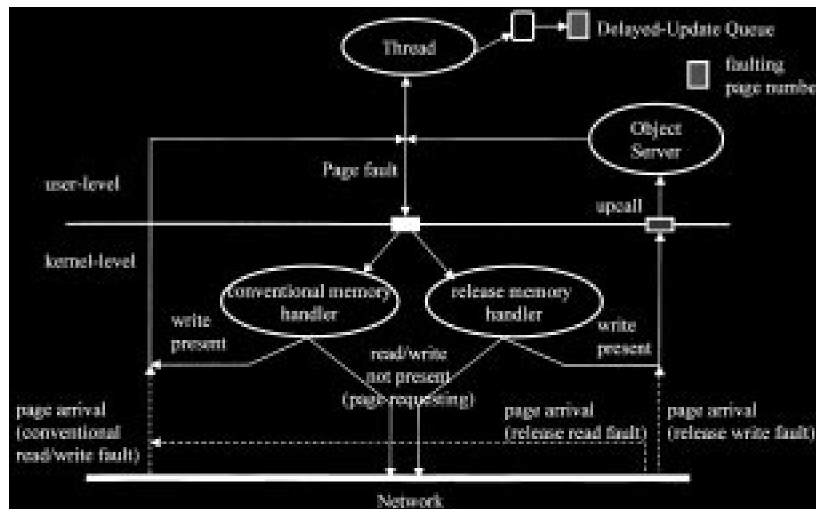


Fig. 5. Page fault handling and upcalling.

affect buffering or write sharing in the release protocol, no upcall will be outgoing, and control will return to normal in this case. On write fault, special care is given. The release memory handler will acquire the page if the page is not present. After setting its access right, the coherence manager will pass the control to the object server together with the faulted page number via upcalling. Then, the upcall handler in the object server may realize that this particular page is dirty, and the number will be appended to the faulting thread's delay queue [5]. Subsequently, control will be returned to the faulting thread.

4.3 Management in the Object Server

For shared objects declared as release or object-based, special care is required. This management is carried out by the object server at the user-level of each machine. Although these servers work co-operatively, they have private data and tables to avoid message exchanges and to maintain the states of shared objects.

In practice, each object has a header that is created during the construction of the system-provided base class. The information included in the header is listed below:

(a) **type:** a flag to specify the memory model applied, i.e., object-based, page-based or release;
(b) **size:** the size, in bytes, of the object;
(c) **access right:** a flag to indicate if the page or the object is clean (read-only), dirty (read/write) or absent (invalidated);
(d) **copyset:** a best guess of the set of processors with the page or the object;
(e) **probable owner:** a best guess of the owner of the page or the object;
(f) **twin pointer:** a pointer of the page's twin copy (for Release consistency only).

For object-based objects, as mentioned before, the *check()* function call will look at the bit set in the *access right* field to find out the state of the data item. The *probable owner* will help the object server to retrieve valid data.

For page-based objects, headers are dummies. They are inserted for the purpose of uniformity.

For release objects, the *access right* is substantial. The reason is that the delayed modifications received will be decoded and merged into the page accordingly by the relevant processors. During merging, updates will also be written to the twin copy of a page if the page is already dirty in order to avoid retransmission of updates just received upon the next synchronization by the relevant thread on the receiving processor. The access right of the page is used to check whether it is dirty. The *copyset* tells the releasing thread to which locations the delayed updates are to be forwarded and is updated when a new caching node is added. The *twincopy* may help the object server to find dirty parts of a page for the releasing thread.

4.4 Synchronization

Parallel programs usually employ synchronization objects, such as locks and barriers, to remove data races and ambiguities. In Cohesion, locks and barriers are allocated in the object-based shared memory such that users may consider synchronization objects as shared data, and no additional manager, such as the lock manager in other systems, is required. Moreover, they are managed in a distributed manner, so that the synchronization load can be spread among all the nodes. Typically, the locks apply a queue-based algorithm [11].

5. PERFORMANCE ANALYSIS

We evaluated the power of two memory consistency models by considering the speedup of three applications in two versions, i.e., release and conventional versions, and the results are presented in Figs. 6 through 8. The X-axis represents the number of processors, and the Y-axis represents the amount of speedup achieved. To evaluate the efficiency of object-shared memory in a critical section, a Producer-Consumer was tested.

All of the experiments were carried out on a network of eight 486-33 personal computers. These computers contained 33MHz INTEL 80486 microprocessors and the 3Com Ethernet interface. They were connected by a 10 Mbps Ethernet which was disconnected from other campus networks during the experiments. In all the tests, the starting time was read when the main thread started off, and the ending time was recorded when all the worker threads had joined. The elapsed computation time was the time interval between the starting time and the ending time.

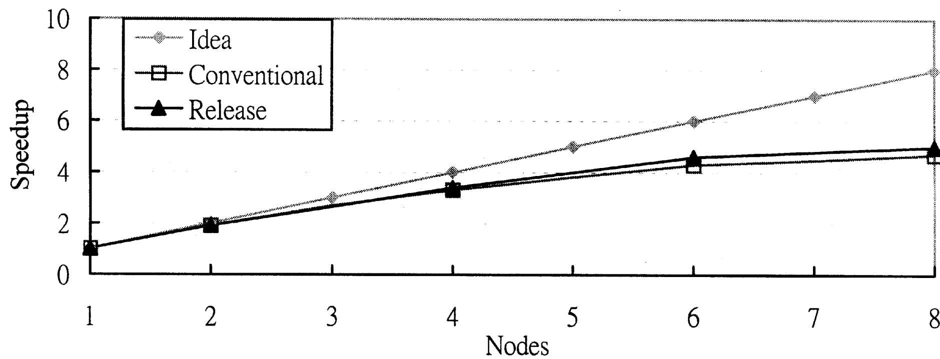


Fig. 6. Speedup of matrix multiplication in Cohesion.

5.1 Matrix Multiplication

Matrix Multiplication computes $C = A B$, where A, B, and C are $N \times N$ square matrices. In our experiment, a 400×400 matrix was used for A, B, and C; only a single thread was forked on each node during execution. Initially, data were stored in the master node. During processing, matrix A and B were read-only data. Fig. 6 shows that a compute-bound program performed pretty well in Cohesion for both versions. In the release version, the performance loss was due to the communications required for each page that was primarily accessed, and updates of messages at synchronization points after every thread had completed computation. In the conventional version, pages were transmitted for reads at the initial stage and writes were initiated by processors to the same page in matrix C during execution. Despite the false sharing, we found that the conventional version was just slightly poorer in performance than the release version. This is because the data size was large in this problem, and the possibility of false sharing in a page was relatively small.

5.2 Successive Over-Relaxation

SOR is an algorithm that simply processes iteratively to solve a problem. Basically, we may regard a matrix as representing a grid of points in a pending area of the problem. During each iteration, every matrix element for next iteration is updated to some function of the elements near it. In the experiment, this function was the average of its nearest neighbors (above, below, left, and right). There are two matrices in the program, both of which are in turn considered as current and scratch arrays before an iteration is started. Every element calculated for the next iteration is written to the scratch array. In our experiment, there were 2000×1000 points in the pending area. A single thread which was forked on each node would synchronize via a barrier at the end of each iteration, and twenty iterations were presumed for convergence in the program. Comparing the speedup of the two developed consistency models, the release model did not perform very much better than the conventional model. The main reason is that two matrices were alternatively

used for each iteration. This required only that an entry be read from a matrix but written to another matrix. As a consequence, there was little contention in memory, and false sharing only caused a problem when several processors were trying to write the result to a same page in the conventional version. Moreover, there was unlikely to be much false sharing in this algorithm when the size of the matrix was large.

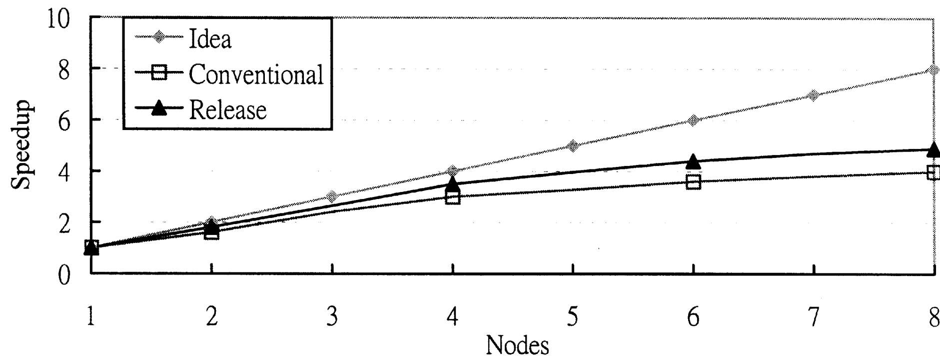


Fig. 7. Speedup of SOR in Cohesion.

5.3 N-Body

N-Body is a force calculation problem found in the area of astrophysics. It calculates the total force perceived by each particle in a self-gravitating space system according to Newton acceleration theorem. Fundamentally, there are $N(N - 1)/2$ forces between all pairs of bodies. We may arrange an array of structures that represent the mass and the coordinates of all particles. In this problem, for parallelism without contention, we apply four arrays to temporarily store the result, and barriers are required. Lastly, the final result may be calculated by adding up all these four arrays for each entry. In the experiment, 8192 particles were considered. As shown in Fig. 8, we found that the release consistency model indeed

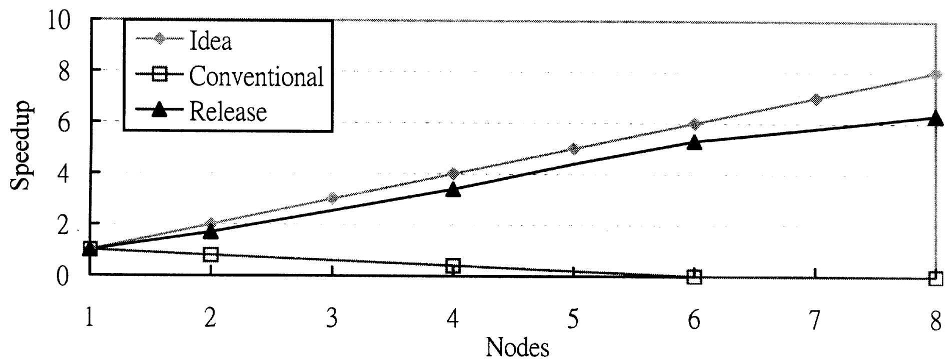


Fig. 8. Speedup of N-body in Cohesion.

relieved the program of page thrashing. Although the algorithm in our experiment required several barriers for synchronization, fully partitioning of the problem allowed high utilization of the processors. On the other hand, the number of pages needed for the arrays was small and there apparently was false sharing in every page. Consequently, pages were thrashed heavily among the processors in the conventional version. This became a serious problem when there were more than two processors in the system.

5.4 Efficiency of Object-Shared Memory

In a critical section, data that are truly shared among processors are protected. It is trivial to state that there is only a single owner of these data at any instant throughout the execution. Accordingly, a migratory protocol is more suitable. However, if these data are declared to be objects in a page-based sequential memory, write invalidation protocol is imposed in Cohesion; if release consistency is used, write update protocol is concerned. In both methods, additional messages and delays are involved. Moreover, shared data in a critical section are usually fine-grained. Consequently, object-shared memory is important. As shown in Fig. 9, three versions of the P-C program, sequential (page-based, and object-based) and release, with 16 Producers and 16 Consumers accessing a protected 8192 byte buffer were tested to evaluate the overhead in the critical sections. Each time the thread entered the critical section, and 128 bytes of "good" were produced or consumed.

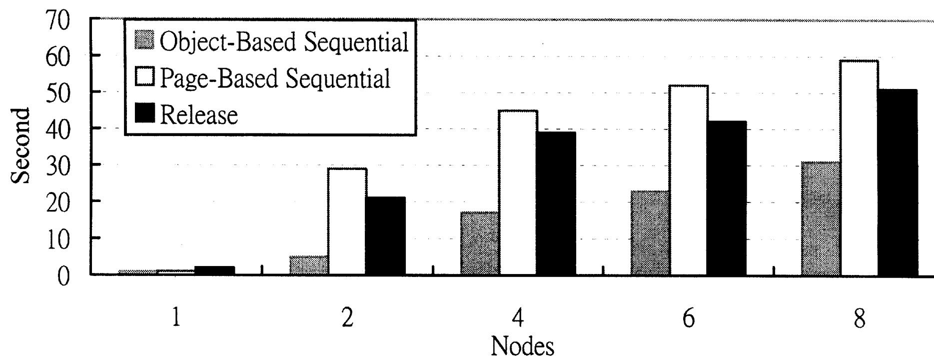


Fig. 9. Overhead of different consistency models in P-C program.

6. RELATED WORKS AND DISCUSSIONS

Munin [1, 5] is a shared memory system developed to overcome the architecture limitations of shared memory machines while keeping their advantages in terms of programming ease. A multiple-protocol release consistency in memory coherence is employed. Most parts of this system are implemented at the user-level. Although virtual memory manipulation is employed, consistency is enforced on a per-variable granularity. Munin employs several different strategies instead of a single memory coherence strategy for all shared data objects to improve performance.

Each data object is associated with a type accepted as a hint to the system via the user or the preprocessor. A write-shared consistency protocol is employed to achieve release consistency protocol to attack the problem of false sharing.

Midway [3] is a distributed shared memory system that supports multiple consistency models. Data in the program may be declared as processor consistent, release consistent, or entry consistent. Within a single run of a program, several multiple consistency models may be active at the same time. Midway tries to minimize communications costs by aggressively exploiting the relationship between shared variables and the synchronization objects that protect them. Midway does not rely on virtual memory system, and consistency models are supported in the granularity of individual data items. On the other hand, it detects access violations without taking page faults, and only a small amount of compiler-time support is required to generate codes that store a new value in a shared data item and mark the item as 'dirty' in an auxiliary data structure. Synchronization objects are permitted to be cached and further specified as exclusive and non-exclusive. In this way, simultaneous reads in a critical section are allowed when non-exclusive synchronization objects are used.

Compared to the above systems, Cohesion differs in several ways. Cohesion is a software distributed shared memory system supporting multiple consistency models, i.e., sequential and release consistency. The sequential consistency model includes the designation of conventional and object-based shared memory. It is worth noting that the release consistency provided in Cohesion does not apply a multi-protocol. Unlike Munin and Midway, there is no compiler or preprocessor intervention in the annotation of shared objects. On the other hand, a system-provided base class is provided. Object-based and release shared objects are specified by inheriting the system-provided base class. Cohesion mitigates the overhead in a critical section via object-based shared memory while Midway employs the entry consistency model. Typically, shared memory is supported in the granularity of an object and a page. In addition, threads running under Cohesion are created during run-time, and their locations need not be specified. This offers us higher programming ease than Munin as well as flexibility for load balancing in the future. A brief comparison is given in Fig. 10.

	Cohesion	Munin	Midway
Memory Consistency Models	2	1	3
Special Purpose Preprocessor	No	Yes	Yes
Threads Location	Dynamic & Static	Static	Static
Granularity	Per Page and Object	Per variable	Per Data Item
Type Specifying	Base Class Inheritance	Well-defined Annotation	Well-defined Annotation
Reduce overhead in Critical Section	Object-based Memory	Multi-protocol	Entry Consistency
Object-Oriented	Yes	No	No

Fig. 10. Comparison of cohesion, munin and midway.

7. CONCLUSIONS

In this paper, we have shown that it is essential for an efficient DSM system to support at least two consistency models, a strict and a relaxed model. Cohesion provides programming transparency and eases porting by supporting the Sequential consistency model. We can reduce the impact of excessive networking and page thrashing due to false sharing in a coarse-grained unit using the Release consistency model. Fine-grained objects may be allocated in object-based memory. Furthermore, delay in a critical section can be shortened by applying object-based shared memory. At this point, users are able to select the consistency type of a shared object according to the object's behaviour. Therefore, Cohesion provides benefits in a wide variety of applications although there is a trade-off to explicitly annotate the consistency type in the program.

Since object-oriented programming is emphasized, shared data can be annotated by inheriting a system-provided base class. This specification is easy and reasonable. Moreover, no compiler or preprocessor is required to handle this annotation. This makes Cohesion more familiar to a parallel programmer.

In Cohesion, the processors applied in our study were 33MHz Intel microprocessors. Therefore, the 10Mbits Ethernet, which was so much slower than the processors, became a bottleneck in the system. This constraint limited the speedup potential of the system. This conflict between processor and network in speed should be eliminated by applying a faster network, such as ATM. This may eliminate the large latency in communications in the network.

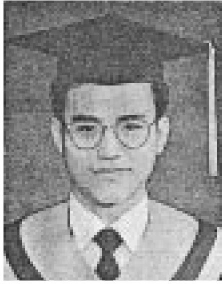
REFERENCES

1. John K. Bennett, John B. Carter and Willy Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Second ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming*, 1990, pp. 168-175.
2. Brian N. Bershad, Edward D. Lazowska and Henry M. Levy, "PRESTO: A system for object-oriented parallel programming," *Software - Practice and Experience*, Vol. 18, No. 8, 1988, pp. 713-732.
3. Brian N. Bershad, Matthew J. Zekauskas and Wayne A. Sawdon, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," Technical Report, CMU-CS-91-170, Department of Computer Science, Carnegie-Mellon University, 1991.
4. John B. Carter, "Efficient distributed shared memory based on multi-protocol release consistency," Ph. D. dissertation, Department of Computer Science, Rice University, 1993.
5. John B. Carter, John K. Bennett and Willy Zwaenepoel, "Implementation and performance of Munin," in *13th ACM Symposium of Operating Systems Principles*, 1991, pp. 152-164.
6. Brett D. Fleisch and Gerald J. Popek, "Mirage: A coherence distributed shared memory design," *Operating Systems Review*, Vol. 23, No. 5, 1989, pp. 211-223.

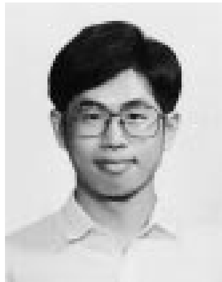
7. Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta and John Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *17th Annual International Symposium of Computer Architecture*, 1990, pp. 15-26.
8. Leslie Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computer*, Vol. 28, No. 9, 1979, pp. 690-691.
9. Kai Li, "Shared virtual memory on loosely coupled multiprocessors," Ph. D. dissertation, Department of Computer Science, Yale University, 1986.
10. Kai Li, "IVY: A shared virtual memory system for parallel computing," in *Proceedings of 1988 IEEE International Conference on Parallel Processing*, 1988, pp. 94-101.
11. John M. Mellor-Crummey and Micheal L. Scott, "Synchronization without contention," in *Fourth International Conference of Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 269-278.
12. Ajay Mohindra, "Issues in the design of distributed shared memory systems," Ph. D. dissertation, Department of Computer Science, Georgia Institue of Technology, 1993.
13. Bill Nitzberg, Virginia Lo, "Distributed shared memory : A survey of issues and algorithms," *Computer*, Vol. 30, No. 8, 1991, pp. 52-60.
14. Ce-Kuen Shieh, An-Chow Lai, Jyh-Chang Ueng, Tyng-Yeu Liang, Tzu-Chiang Chang, Su-Cheong Mac, "Cohesion: An efficient distributed shared memory system supporting multiple memory consistency models," in *AIZU International Symposium of Parallel Algorithms/Architecture Synthesis*, 1995, pp. 146-152.

Ce-Kuen Shieh (謝錫堃) is currently an associate professor in the Department of Electrical Engineering, National Cheng Kung University. He received his Ph. D., M. S., and B. S. degrees from National Cheng Kung University, all in electrical engineering. His current research interests include distributed/parallel processing, operating systems, computer networking, and compilers.

Jyh-Chang Ueng (翁志昌) is currently a Ph. D. candidate in the Department of Electrical Engineering at National Cheng Kung University. Ueng received his B. S. degree from National Sun Yat-Sen University in 1991, and M. S. degree from National Cheng Kung University in 1993. His main research interest is Distributed Shared Memory.



An-Chow Lai（賴安洲）is currently a Ph. D. candidate in the Department of Electrical and Computer Engineering at Purdue University, U. S. A. His main research interest is Distributed Shared Memory. Lai received his B. S. and M. S. degrees from National Cheng Kung University in 1992 and 1994, respectively.



Tyng-Yue Liang（梁廷宇）is currently a Ph. D. candidate in the Department of Electrical Engineering at National Cheng Kung University. His main research interest is Performance Optimization on Distributed Shared Memory. Liang received his B. S. and M. S. degrees from National Cheng Kung University in 1992 and 1994, respectively.

Su-Cheong Mac（麥樹翔）received his B. S., M. S., and Ph. D. degrees from National Cheng Kung University in 1990, 1992, and 1998, respectively. His research interests focus on integrating Distributed Shared Memory and parallel I/O.