[DSB88] Michel Dubois, Christoph Scheurich, and Faye Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–22, February 1988.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LR91] J. Lee and U. Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, 1991.

[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, pages 52–60, August 1991.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 18th International Conference on Parallel Processing*, pages 160–169, August 1989.

[RTY+87] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectu ral Support for Programming Languages and Operating Systems*, pages 31–39, 0ct 1987.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986.

References

[ACD+90] Mustaque Ahamad, Muthusamy Chelliah, Partha Dasgupta, Richard J. LeBlanc, and Mark Pearson. Shared memory programming in a distributed system. Technical Report GIT-CC-90/63, College of Computing, Georgia Institute of Technology, November 1990.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[AHJ91] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *11th International Conference on Dist. Comput.*, May 1991.

[Ana91] R. Ananthanarayanan. CC++ Reference Manual. Technical Report GIT-CC-91-07, Georgia Institute of Technology, College of Computing, Atlanta, GA, 1991.

[BAHKi87] J. M. Bernabéu Aubán, Phillip W. Hutto, and M. Yousef A. Khalid i. The Architecture of the Ra Kernel. Technical Report GIT-ICS-87/35, Georgia Institute of Technology, College of Computing, Atlanta, GA, 1987.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–177, March 1990.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Che86] D. R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems design. In *Proc. of the 6th Int'l Conf. on Distr. Computing Sys.*, pages 190–197, May 1986.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed v kernel and its performance on diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[DCM+90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46, Winter 1990.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

activities that take place in a message passing system; messages are sent only to processes that need the information. If the cost of synchronization is included, our approach appears to perform worse than the message passing approach, since synchronization is automatically achieved on receipt of messages. However, in a message passing system the sender has to explicitly name the set of receiving processes. Explicit synchronization in the ASM system provides a level of decoupling, since the 'sender', or the process performing the unlock does not have to know the identity of the 'receivers' statically. Thus, our approach allows for dynamic data sharing patterns. However, if the sharing pattern is well known, this information can be included in the implementation of synchronization, our approach performs as well as the message passing approach. Efficient realization of synchronization is one of the goals of future research.

In previously proposed research, synchronization information has been used to realize *release consistency* [GLL+90], for example, in Munin [CBZ91]. Release consistency requires updates to be sent to all processors on an acquire (lock) operation, that is, in the terms of the authors, *globally performed*. Our goal is to identify and implement synchronization operations such that only nodes that can obtain access to the data can see the changes to it. Thus, there may be inconsistent copies of the data in the system as long as no process accesses them. Only the *buffered consistency* model [LR91] allows such copies to exist. Intuitively, our system allows the data to *flow* along the *path* of synchronization, and does not care about inconsistent copies, as long as they are is not in this path.

7 Concluding remarks

Traditional Distributed Shared Memory systems perform poorly due to their mismatch with application requirements. Recent research on the Munin system has shown that such information can be obtained from programs without unduly complicating the programming model presented by the system. Our work focused on investigating the source of the performance problems and designing primitives to realize application specific coherence control. The use of synchronization information in the programs leads us to a way to automatically use these primitives through higher level program directives. This approach is based on notions of flow of data along with transfer of control on synchronization operations, rather than on global performance of updates found in release consistency. Since, the programmer provides only synchronization information and the association between synchronization and data usage, we believe that program complexity is kept to a minimum. Factoring out the primitives gives us the option of adding transformations of higher level program directives in the future, if better algorithms to maintain coherence based on synchronization or some other directives are found.

One of the directions of future research is to realize synchronization mechanisms efficiently. Such an effort is needed as the performance of the programs will depend not only on the efficiency of coherence maintenance operations, but also that of synchronization. In this work we have assumed that pages are not falsely shared. However, data placement to avoid false sharing cannot be performed by compilers in an efficient manner and the burden is usually passed on to the programmer. Avoiding the effects of false sharing and support for fine-grained data items are, thus, other areas of our future research.

The exact steps of consistency maintenance depend on the type of protection that the *synv* provides. If the *synv* protects the shared variable exactly, then the pages made accessible (which will be actually accessed) are pre-determined. Consistency maintenance, then, simply consists of updating the pages containing the shared variable using the `update` primitive to the set of nodes where the processes being granted the lock execute. If the *synv* protects the shared variable approximately, the cached copy of the particular page of the shared variable is written back to the owner by using an `unget`, and other copies of the page are invalidated by using an `invalidate`. As noted before in Section 3.3, invalidate is preferred over update since the node at which the new value of the data will be used is not known at the time of the generation of the new value.

When a directive to a establish sharing pattern is encountered along with an unlock operation, the `getcopyset` primitive is used to obtain the set of nodes that share the particular page. This information is used during the second phase of the unlock operation on the subsequent encounter of the unlock.

Figure 5 shows the bounded buffer example along with directives on the use of synchronization variables. The sequence of activities that take place during the *DeleteItem* operation are similar.

```
Shared Object BoundedBuffer;
        buf :  parray [0..n-1] of item; (* partitioned array *)
        InIndex, OutIndex :  integer;  (* allocated on separate pages *)
        InMutex :  semaphore exactly protects InIndex;
        OutMutex :  semaphore exactly protects OutIndex;
        empty, full :  semaphore approximately protects buf;

(* Operation Init as before *)

Operation AddItem(in data : item);
begin
        P(empty);              (* allows access to buf *)
        P(InMutex);            (* when allowed to proceed, InIndex has latest value *)
        buf[InIndex] := data;  (* might page fault to fetch current value,
                                  due to previous invalidation *)
        InIndex := InIndex + 1 mod n;
        V(InMutex);            (* updates InIndex at the node of first blocked process *)
        V(full);               (* Invalidates particular pages of buf that were accessed *)
end
```

Figure 5: Bounded Buffer : with synchronization directives

6.1 Discussion

Using synchronization information in the above manner has several interesting characteristics. When shared variables are exactly protected, updates are used to make the memory of the processes granted a lock consistent. The associated actions are exactly the same as the

potential access to a set of pages of memory, whereas only a subset of the pages will be actually accessed.

- Certain algorithms follow a predefined data sharing pattern. That is, an unlock makes the shared accesses visible to certain fixed number of processes, regardless of the contention for the lock variable in question. For example, in the linear solver application, once the location of the workers and the coordinator are fixed, the sharing pattern of the data is fixed, assuming absence of process migration. Many parallel programs partition the data into a number of non-overlapping parts and a 'parent' process creates 'worker' processes to operate on the data. The workers work in phases, each phase consisting of computing and producing a new value for the data partition allocated to the particular worker, possibly depending on the previously computed values of other data partitions. This class of programs is known as Single Program Multiple Data (SPMD) programs.

We next consider automatic insertion of the primitives discussed in Section 4.1, by making use of synchronization operation and the above observations. In place of directly programming with the primitives, the programmer is required to provide the following information:

- Identify variables used in synchronization, collectively termed *synv*. These constitute shared synchronization variables, and locks and semaphores of the ASM system.
- Protect access to each non-synchronizing shared variable by using exactly one *synv*, that is, bracket all accesses to the shared variable by operations equivalent to lock(*synv*) and unlock(*synv*), such that at most one process could be writing into the shared variable at any one time. While each shared variable is protected by exactly one *synv*, a *synv* may protect access to multiple shared variables.
- Provide association of shared variables and *synv*'s. This could be either on a per *synv* basis or on a particular operation on the *synv*, and classify each association as being exact or approximate.
- Optionally, identify points in the program, in the form of unlock operations, where sharing patterns get established.

The sequence of activities that take place on lock and unlock operations are as follows: An unlock operation proceeds in two phases. During the first phase, the list of processes waiting on the *synv* is determined. Access is to be given to a set of such processes. In case of a semaphore, there is exactly one process, while in case of a read-write lock, a number of readers could be granted the lock. The set of nodes at which these processes execute is then determined. At this point, consistency maintenance activities are performed, as discussed below. After consistency maintenance activities, the processes granted the lock are allowed to proceed.

Consequently, the lock operation can be thought of as proceeding in two steps. During the first step, the request is sent for obtaining the lock. This request may result in the process being blocked. In the second step, consistency maintenance activities take place, and then the process is allowed to proceed.

processes (or threads) on the same node. If the threads share the same page at the hardware level, the changes will be visible to all processes without the update request. Thus, the update primitive should be seen as a way of guaranteeing memory consistency; it cannot be assumed that changes to memory will *not* be observed by the processes if the the update is not explicitly requested.

6 Using Synchronization Information

The primitives discussed in Section 4.1, if used directly by the programmer, take away the simplicity of shared memory programming. Ideally, the programmer should provide directives at a 'higher' level which can be used by a transforming algorithm to insert the primitives that correspond to the directives. Such an approach is also followed in the design of the Munin system [CBZ91], but the directives tag data objects primarily based on the sharing pattern of the object, as opposed to using synchronization information as a basis in our design. The rest of this section outlines the feasibility of our approach.

Events in a pure shared memory system consist only of reads and writes to shared memory. Some of these reads and writes are used in synchronizing access to other parts of shared memory. For example, in the linear solver implementation (Figure 1), `wait`($\neg complete_i$) in the Worker is used as a *lock* primitive and the statement $complete_i := F$ in the Coordinator is used as an *unlock*. These statements synchronize access to the variable x_i.

In an *Augmented Shared Memory* (ASM) system, other events such as general synchronization operations may affect the observability of changes to shared memory. For example, creation and termination of a process can be detected by other processes. In such a case, a programmer can assume to observer changes made by the terminating process to shared memory after detection such an event. Other synchronization operations in an ASM consist of system-provided synchronization, in the form of semaphore, locks, monitors, conditional critical regions, etc. Hence forth, we assume an ASM system.

A *weakly consistent* shared memory system [DSB86] maintains coherence of memory on synchronizing accesses rather than on every access to shared memory. While our solution is based on a similar approach, we extend the weak consistency model with the following observations:

- Some synchronization variables *exactly* protect particular pages of memory. For synchronization operations on these variables, every time the lock is granted, only a fixed set of pages are accessed. An example of this type of synchronization variable is $InMutex$ (see Figure 2) semaphore; it protects access to the shared variable $InIndex$.

- The particular pages of memory made accessible by a lock may be decided at run-time. In this case the synchronization variable is said to *approximately* protect the pages of memory. Although the exact page accessed may be 'refined' by an inner lock, and only a portion of the memory may be accessed in an exclusive manner, the outer lock protects, in general, a bigger area of memory. For example, the operation `P`($full$) of the bounded buffer obtains access to a 'full' buffer element, which, at the point of the `P` operation could be any of the buffer elements. Hence, the `P`($full$) operation potentially protects all of the buffer elements. In general, counting semaphores provide

Activity being monitored (All times are in milliseconds)	Measured Total Time
Basic context switching	0.3
Page transfer at the transport layer	24.4
Initial Page fault	
Pages in memory of server	27.7
Pages to be retrieved from disk	34.5
Updating a single dirty page	
Back to server	26.0
Server and one other node	52.3
Server and two other nodes	78.3
Updating a single dirty page directly using copy set (without server as intermediary)	
Update one other node	33.8
Update two other nodes	66.3
Page fault after an invalidate	38.3
Time to invalidate	
Notifying the server (no other nodes cache page)	9.1
One node to be invalidated	18.5
Two nodes to be invalidated	28.1
Time to get copy set from server	9.2

Table 1: Timings of Primitives

the update primitive is used, then the action would take 33.8 ms[2]. On the other hand, if an invalidate and a subsequent page fault is used, then the total cost would be 56.8 (18.5 + 38.3). Thus, the update primitive, if used appropriately, would show a performance improvement of about 40% over using invalidations.

Several issues need to be considered in the implementation. Firstly, granularity of data items in a shared memory page could cause correctness problems. Consider for instance, two data items, d_1 and d_2 co-located in a page P. The page is fetched in read-write mode and cached at two nodes, A and B. d_1 is modified at A and d_2 at B, simultaneously. When the updates from each node are sent to other node, only one of the modifications will be survive, A's changes at B and vice-versa. This problem is due to false-write-sharing of the page. Thus, it is necessary to ensure that simultaneously writable data items are not co-located on the same page. The compiler, with the aid of the programmer must place the data appropriately. For example, the language used to program the applications in this paper, Distributed C++ [Ana91], supports partitioned arrays. In the linear solver example, the vector of x_i's is such an array: it is divided into contiguous parts, one per worker. Each part of the partitioned array is placed in a different page, but the access to the array elements is provided through overloading the index operator [Str86], which hides the partitioning.

Another issue is that of visibility of the changes to shared memory. Our model assumes that unless updates (or invalidates) are requested, the other nodes do not observe the changes made to shared memory at a node. It does not specify visibility of changes to

[2]We assume that copyset was obtained at a one time cost.

can change, if the changes occur at known points. The new value of the copyset can then be obtained from the owner at such points.

While the primitives discussed above provide the performance benefits of a message passing system, such an approach is not without drawbacks. The user has to be completely aware of the underlying DSM mechanisms and has to explicitly activate the coherence operations. Hence, with this approach, transparency is compromised for the potential gain in performance. In section 6, we consider how more transparency can be provided without loss of efficiency.

5 Implementation

The implementation environment consists of a set of Sun 3/50's and 3/60's connected through an Ethernet. These machines run the native CLOUDS operating system, directly on top of the hardware. The hardware and the virtual memory management provide the characteristics of the model described in section 4. The MMU supports pages of size 8K. The backing store abstraction is supported by CLOUDS through *Partitions* [DCM+90]. The name of a memory page (identified by `token` in the backing store interface of section 4) consists of a system-wide unique segment name and an offset. It should be noted that although the CLOUDS system, was used as a testbed, systems such as Mach, or the augmented V system [CZ83] with programmer specified paging capability could easily serve the same purpose.

The backing store is implemented as a server which uses a disk to store the pages of shared memory. A get request is initially processed by reading the appropriate page from the disk. Once this is done, the page is kept in the memory of the server, so that subsequent get requests can be processed without the overhead of disk access. The server keeps a copyset for each page, which is a list of nodes currently caching the page.

Each node, in addition, has a slave server that processes incoming update and invalidate requests. On the receipt of an update message, the following actions are taken: software tables associated with the existing page are changed so that the address translation points to the updated page and hardware contexts (essentially PTE's) are cleared of any pointers to the old page. Thus, access to the virtual address will generate a page fault and the software tables will be used to 'pick up' the new page. This technique of swapping of pointers to a page avoids copying of data, which requires 1-2 milliseconds, as compared to a page fault service of a in-memory page, which takes 0.25-0.5 milliseconds[1]. However, the technique requires the ability to extract the page of data from an incoming message, dissociate it from the message and map it into a different virtual address region. Such functionality is provided by the RA kernel [BAHKi87] on which Clouds is built. Invalidate messages are processed in a similar manner.

Performance figures for the DSM primitives are given in Table 1. The first two entries give the cost of the system underlying the DSM system. Subsequent entries tabulate the performance of the primitives discussed in section 4.1. In all of these measurements, the server owns the data pages and other nodes fetch the pages from it. The relative performance of using invalidate and update primitives is as follows. Assume that two nodes, N_1 and N_2 share a page and that node N_1 writes into the page. To make N_2's copy consistent, if

[1] The actual time depends on whether we use a 3/50 or a 3/60.

```
global
  (* a and b declared as before *)
  (* x, complete and changed declared as partitioned array's *)
```

COORDINATOR	WORKER$_i$
local	
$converged()$: returns boolean	t_i: real initially 0
while $(\neg done)$	**while** $(\neg done)$
wait $(\forall i\, complete_i)$	$t_i := \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j - \sum_{j=i+1}^{n} a_{i,j}\, x_j\right) \Big/ a_{i,i}$
if $(converged())$	$complete_i := T$
$done := T$	**update**(**pageof**($complete_i$), ALL)
update(**pageof**($done$), ALL)	**wait** $(\neg complete_i)$
$\forall i\, complete_i := F$	$x_i := t_i$
$\forall i$ **update**(**pageof**($complete_i$), ALL)	**update**(**pageof**(x_i), ALL)
wait $(\forall i\, changed_i)$	$changed_i := T$
$\forall i\, changed_i := F$	**update**(**pageof**($changed_i$), ALL)
$\forall i$ **update**(**pageof**($changed_i$), ALL)	**wait** $(\neg changed_i)$

Figure 4: Linear Solver : with DSM primitives inserted

It is easy to see that the number of messages for coherence maintenance is minimal. Similarly, the invalidate primitive can be used to maintain coherence of shared variables in the bounded buffer example while optimizing the performance.

The update primitive has several interesting characteristics: First, it is a modified form of the *send* primitive in a message passing model. However, the *receive* primitive of the message passing model has no analogous primitive in our model: the 'message' is received by checking for the changes in the value of the memory being updated. For example, each worker i in the linear solver waits for the value of $complete_i$ to change to false before it writes x_i. Second, some of the updates can be combined. For instance, the update of x_i and $changed_i$ can be combined, since the 'reception' of this 'message' is performed through the checking of all $changed_i$'s in the coordinator. In other words, the value of x_i is not checked by the coordinator before the value of $changed_i$ is checked and hence the update of x_i can be combined into the DSM message that updates $changed_i$.

The `getcopyset` primitive can be used to reduce the number of messages in certain cases. If set of the nodes which cache a particular page do not change during the execution of a program, then the copyset for that page needs to be obtained only once; the writer can then send updated value directly to these nodes, avoiding the request to be routed through the owner. The linear solver is an example of such an application. After the end of the execution of the first iteration, the set of nodes which cache copies of any page will remain fixed. The copyset can be obtained at this point and can be used until the termination of the program. In fact, the technique even works when the set of nodes which cache the page

primitives that can be used to maintain coherence of pages held in the caches of different nodes of a system.

4.1 Primitives

We propose six primitives that can be used to program the DSM system. These are as follows :

get(page, mode) : gets the page (the page is a pointer in the DSM shared space) in the specified mode from the owner. Mode can be read-only or read-write. The owner adds the requesting node to the list of nodes (copy-set) having a cached copy. Notice that the page, once received, will be stored in a page of the physical memory at the requesting node and the appropriate PTE will reflect this.

unget(page) : a message containing a notification is sent to the owner and the cached copy is invalidated. If page was brought in read-write mode and if the page was modified (dirtied), the notification also includes the modified page. The owner node removes the requesting node from the copy-set.

invalidate(attributes, page, node-set) : sends invalidate messages to specified nodes; attributes and node-set are discussed below.

update(attributes, page, node-set) : updates the cached copies of the page in the set of nodes specified using the contents of the locally cached copy.

getcopyset(page) returns node-set : returns the set of nodes that have a cached copy of the specified page; involves a request/reply message exchange with the owner.

flush(page) : flushes the outq of messages belonging to the page. Such messages could be updates, invalidates, etc. and are processed in the order in which they were queued. As a special case, page could be ALL, in which case all the messages in the outq are processed.

node-set is a list of nodes to which the invalidation or the update message is to be sent. It can be a subset of the nodes at which the cached copy of the page is kept. The actual nodes in the set are determined by the set of nodes which can potentially read an invalid data copy. As a special case, node-set ALL sends invalidate or update message to all nodes having a cached copy of the page (the nodes in the copy-set, maintained by the owner).

Attributes is used to classify the invalidate or update request. One of the attributes is the time at which the request is to be carried out. *Synchronous* requests are processed immediately. *Asynchronous* requests are queued in the outq and are processed later on a flush request.

These primitives are intended to provide the basic functionality for cache management. Unless explicitly directed through these primitives, the DSM system does not perform any coherence maintenance activity. Given these primitives, it is relatively easy to obtain the performance benefits of write-update or the write-invalidate protocol. The programmer analyzes the pattern of interaction in the program, chooses the appropriate primitives and inserts them in the code. Figure 4 shows the linear solver with the primitives inserted; all calls are assumed to have the synchronous update attribute.

the model.

The distributed system is composed of a number of workstations connected by a network (see figure 3). Each of the machines in the system consist of primary memory and one or more CPUs. The workstations do not share physical memory and an implementation of a DSM treats the primary memories of the workstations as caches of the global DSM address space. The primary memory is divided into a number of physical pages. The size of a physical page is determined by the Memory Management Unit (MMU). A page is also the smallest unit of memory for which protection can be specified through the use of page table entries (PTE). A PTE typically maps a virtual page into a physical page with a specific protection attribute. Common protection attributes are read-only, read-write and invalid.

The CPUs generate read and write requests to memory and for each specify the virtual address of the location that is to be read or written. In either case, the virtual address is translated into a physical page number and a byte (or word) offset within it by looking up the address in the page table. If an address cannot be translated, a *page fault* is generated.

A page fault is serviced by filling a free physical page with appropriate data and setting the PTE mapping of the faulting virtual page to map to the physical page with suitable protection attributes. The 'appropriate data' is determined by the software tables maintained by the operating system. Such a table essentially specifies the characteristics of the virtual address space. Let us call this the Virtual Address Characteristic Table (VACT). Some of typical characteristics are zero-filled data, data backed by the swapping system and data to be demand paged from the disk. The VACT is a structure that could be used to characterize a virtual page as belonging to the distributed shared memory system. Note that due to characteristics of the MMU dictating that the smallest unit of protection be a physical page, the VACT cannot effectively characterize virtual address regions of less than a page size.

In general, we can classify the pointers to 'appropriate data' from the VACT as pointers to *backing store*s. A backing store provides storage for pages of memory and is said to *own* the page of memory. The backing store could use a part of the main memory itself as a store (sometimes called a *ram disk*) or could use a secondary storage device. We call the node that controls the backing store for a particular page its *owner*. A backing store should offer at least the following interface to the Virtual Memory System, particularly the page fault handler:

```
get(token, physical page)
token = unget(physical page)
```

`token` is an identifier used to name the data that is to be requested. The `get` routine fetches the data identified by the token from the owner and copies it into the physical page provided. The `unget` routine writes out the data in the physical page to the owner (when the page is dirty), and returns a token that could later be used to get the data again. Once a `unget` has been completed, the physical page could be used for other purposes. `get` and `unget` are generic routines. The DSM system, in particular, should offer such an interface.

The DSM system should also handle problems that arise due to sharing. In doing so, messages are exchanged between nodes. Outgoing messages from a node are queued in a *outq* and incoming messages are queued in an *inq*. In the next section, we present a set of

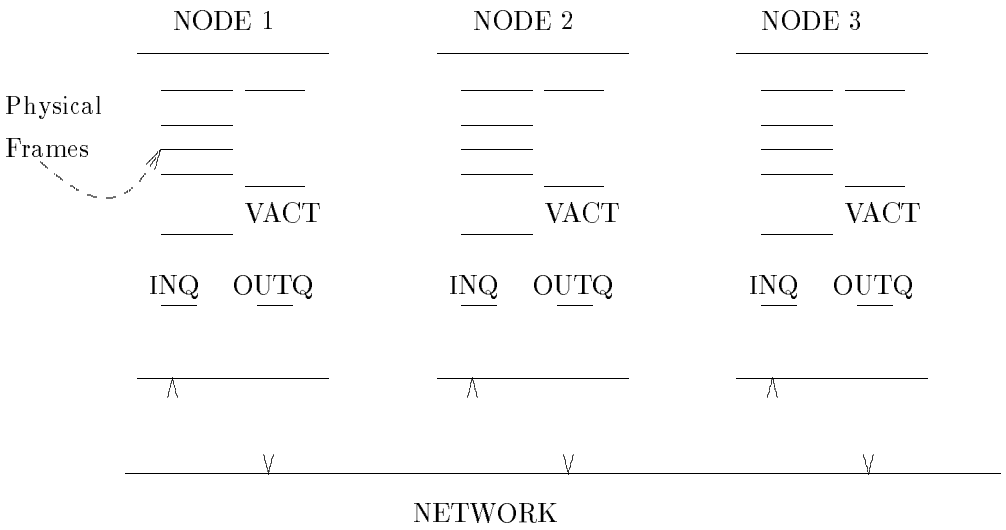

Figure 3: Model of Distributed System

given page and the get message for the same page on subsequent access by the node. This is a fundamental problem with invalidate protocols and cannot be solved with any amount of assistance from the applications. However, we believe that the update protocol can be modified, again with assistance from program directives, such that the disadvantages cited above can be mitigated, resulting in an efficient coherence scheme that reflects the nature of the application.

The invalidation technique may still be useful under certain scenarios. In some cases, nodes to which the update is sent may not need the information at the point of the update: either the node may not need it at all or may not use it for a 'long' time. In such a situation, it is better to invalidate the copy of the page at the node so that the node is forced to explicitly request the page when it is needed. Invalidating a copy makes the reader request the data on demand, while, update pre-prepares the data for the user. Thus, invalidation is preferable when the access pattern of shared memory is not known, and dynamic coherence maintenance activity is required.

Although we used the two common protocols to argue that no single protocol can perform well for all types of applications, the same can be shown to be true when other protocols are considered. We believe that since the programmer of an application best knows how data is shared, it is necessary to provide primitives that allow programmers to customize the coherence of the shared data. In the following sections, we consider a 'first cut' attempt at specifying features which can be used to control memory coherence operations. In later sections, we develop a programming model which allows automatic initiation of coherence operations from programs which follow certain constraints.

4 Basic Distributed Computing Model

In this section, we present a simple model of a distributed system. The model is intended to be general and we characterize the memory management with a functional description and procedural interfaces. Later, in the section on implementation, we give an instance of

(read/write-runs) by the writer. Strictly speaking, this is not a disadvantage when the data-item is a buffer element, since producers are pure writers of the buffer. However, if the data is being read in a run, the update will be useful and necessary. For example, consider the shared variables *InIndex* and *OutIndex*. As before, we assume these variables are each in a separate page. *InIndex* is potentially cached at all nodes where the producers execute. Let us assume that there are N such nodes and that each producer is continuously active at each of these nodes; that is, once started the producer executes forever.

In the case of invalidation protocol, since all producers are writers of *InIndex*, at most one node has a cache of *InIndex*. Thus, each access to *InIndex* incurs 3 messages : the request to DS, forward from DS to current keeper (where the last access to *InIndex* was made by a producer) and data transfer from keeper to the requesting node. The keeper node invalidates its copy on forwarding the data. In case of update protocol, the cached copies are not invalidated. Hence, at a steady state, all N nodes have cached copies. Thus, every write to *InIndex* will incur N messages: update message to owner, and $N-1$ propagates to keepers. Thus, for this example, the update protocol performs poorly.

3.3 Discussion

We can characterize the access patterns in the above two examples as follows: the linear solver is an example where for a particular data item, there is a single writer and multiple readers. The bounded buffer is an example of multiple reader-writers with respect to *InIndex* and *OutIndex* variables. With respect to the buffer elements, the access pattern is that of a single pure writer (producer) and a single reader (consumer). We conclude that in the case of a single writer and multiple readers, the update protocol will outperform the invalidation protocol as long as there is at least one read by each of the readers for every three writes by the writer.

Invalidating or updating of potentially incoherent data are two fundamental techniques that can be employed in maintaining coherence of shared distributed data. However, the environment in which it is implemented may determine the effectiveness of either of the technique. Invalidate protocols are suited to implementation in a distributed setting, where no special hardware support is available. Once the cached copy of the incoherent data is invalidated, the correct copy can be supplied when a subsequent request is generated for that data, through a page fault, for example. However, update protocols are difficult to implement without hardware support. First, it is cumbersome to implement tracking of every write, since each update to cached data should be sent to nodes in the copyset. Second, updates are sent to nodes which no longer use the data, but the data exists in their caches due to relaxed garbage collection of pages. For these reasons, update protocols have to be assisted by program directives, so that the system will know when and where to send the update messages.

In spite of the drawbacks, update protocols are promising in one respect: the number of messages involved. In a way, updates directly reflect the message passing nature of the underlying system. Updates can be thought of as sending a message containing the state that the application wishes to share among the different parts of the program. Hence, when used carefully, we can expect the update protocol to perform as well as any message passing implementation of an application. In contrast, the invalidate protocol involves two 'extra' messages, to achieve the same effect: the invalidate message to a node caching a

```
Shared Object BoundedBuffer;
        buf :  array [0..n-1] of item;
        InMutex, OutMutex, empty, full :  semaphore;
        InIndex, OutIndex :  integer;
Operation Init;
begin
        InIndex := 0; OutIndex := 0;
        empty := n; full := 0;
        InMutex := 1; OutMutex := 1;
end

Operation AddItem(in data : item);
begin
        P(empty); P(InMutex);
        buf[InIndex] := data; InIndex := InIndex + 1 mod n;
        V(InMutex); V(full);
end

Operation DeleteItem(out data : item);
begin
        P(full); P(OutMutex);
        data := buf[OutIndex]; OutIndex := OutIndex + 1 mod n;
        V(OutMutex); V(empty);
end
end BoundedBuffer;
```

Figure 2: Bounded Buffer

Using the update scheme, all cached copies have to be updated, even though only one read is ever executed on the copies before they are updated again. For example, consider the reuse of buffer elements: a producer at N_1 writes into a buffer element, say $buf[i]$, and later a consumer at N_2 reads from $buf[i]$. At this point, with the update scheme, cached copies exist at N_1 and N_2. Now, a different producer, say at N_3, tries to write into $buf[i]$ due to the reuse of buffer elements. With a simple update scheme, updates are sent to all nodes containing cached copies. The following variations are possible when N_3 writes $buf[i]$.

- Copy at N_1 is invalidated; Multiple writers cannot simultaneously hold cached copies.
- Cached copy is retained at N_1; Multiple writers simultaneously hold copies and updates are serialized using synchronization and/or through a single data server.

With the first variation, updates, in step 5, are sent to N_2 only. Updates are sent to N_1 and N_2 in the second variation. Updates to N_1 are redundant, since producers are *pure* writers. The updates sent to N_2 could be useful, if the consumer running at N_2 happens to read from $buf[i]$ (again); however, such an assumption cannot be guaranteed. Variation 2 of the update scheme has the disadvantage of not taking care of reuse of the data-item

of a page and the list is maintained by the owner node [LH89]. We assume standard implementation of the write-update and write-invalidate schemes.

The message cost of the two schemes in each iteration of the algorithm is as follows. Since the handshake bits $complete_i$ and $changed_i$ are used for synchronization, we exclude the messages generated when these variables are accessed (when each worker computes a large number of elements of x, the synchronization cost will not be significant). To simplify the analysis, we assume that each x_i is in a separate page owned by the corresponding worker. We analyze the message cost for each iteration of a worker; total message cost for an iteration of the algorithm can be obtained by multiplying the cost by the number of workers. In particular, we will consider worker i.

First, we consider the coherence cost of the invalidation scheme. In the steady state, at the beginning of an iteration, all x_j's ($j \neq i$) have to be fetched to the node where worker i runs because cached values were invalidated when these variables were written at the end of the previous iteration. This fetching involves $n-1$ request messages and $n-1$ replies. Once the coordinator indicates that it has read all the newly computed values to test for convergence, the worker i writes the newly computed value into x_i. This generates $n-1$ invalidation messages to other worker nodes. An additional invalidation message is sent to coordinator node when t_i is written. Thus, the total message cost is $3n-2$ for each iteration for worker i.

In case of the write-update protocol, at steady state, the data at a node is always consistent with the latest value. Although $2(n-1)$ messages are needed to perform the initial fetching of data to the worker nodes in the first iteration, in the steady state the only messages generated are due to the updates sent to other workers when new values of x_i are produced. Since there are $n-1$ other workers and one coordinator, each iteration of a worker costs n messages.

Clearly, write-updates are better for executing the linear solver application. However, such arguments do not universally hold true. In fact, it may not be possible to clearly state which of the protocols will perform better. To illustrate this, we present another example.

3.2 Bounded Buffer

In the bounded buffer example (figure 2), a producer executes *AddItem* to insert items into the bounded buffer. Consumers execute *DeleteItem* to take out an item. To simplify the discussion, assume that each element of the buffer (*buf*) is located in a separate page and that all the data is owned by a data server (DS), which is different from the nodes where the producers and consumers execute.

First, we consider the overhead involved in maintaining consistency of buffer elements. Initially a producer at a node, say, N_0 adds an item to the bounded buffer. This action involves a page fetch request message from N_0 to DS and the data transfer from DS to N_0. Next, a consumer runs at node N_1, which generates a page fetch message to DS. DS forwards the message to N_0, which has the latest copy. N_0 sends the page to N_1. Thus, for each execution of *DeleteItem*, a cached copy of the page containing the consumed buffer element is created. With the invalidation scheme the cached copy is invalidated as soon as the buffer element is written into by a producer on the reuse of the bounded buffer element. Counting the invalidation as part of the read, each execution of the consumer involves 4 messages (get, forward, data-transfer, invalidation).

computed in the t^{th} iteration. Also, let $a_{i,j}$ be the element in row i and column j of matrix A, b_i be the i^{th} element of vector b. The values of x in the $t+1^{th}$ iteration are computed using the following relationship:

$$x_i^{t+1} = \frac{b_i - \sum_{k=1}^{i-1} a_{i,k} x_k^t - \sum_{k=i+1}^{n} a_{i,k} x_k^t}{a_{i,i}} \quad (1)$$

The computation terminates when the following convergence condition is satisfied.

$$\sum_{i=1}^{n} abs(x_i^{t+1} - x_i^t) < error_tolerance$$

The pseudo-code that implements this algorithm is shown in figure 1. The workers are the processes that compute the new values for each element of x and they synchronize via the coordinator process. Since we use the synchronous iterative algorithm, the coordinator ensures that a new iteration is started by a worker only after all the values in the previous iteration have been computed. Although each worker should compute a number of elements, to simplify the program, we assume that each worker computes a single element (worker i computes x_i). `wait` B is the notation for `while not` B where B is a boolean condition.

```
global
  a[n, n]        : matrix of real
  b[n]           : vector of real
  x[n]           : vector of real each initially 0
  complete[n]    : vector of boolean each initially F
  changed[n]     : vector of boolean each initially F
  done           : boolean initially F

  COORDINATOR                          WORKER_i

local
  converged(): returns boolean         t_i: real initially 0

while (¬done)                          while (¬done)
  wait (∀i complete_i)                   t_i := (b_i − Σ_{j=1}^{i−1} a_{i,j} x_j − Σ_{j=i+1}^{n} a_{i,j} x_j) / a_{i,i}
  if (converged())                       complete_i := T
    done := T                            wait (¬complete_i)
  ∀i complete_i := F                     x_i := t_i
  wait (∀i changed_i)                    changed_i := T
  ∀i changed_i := F                      wait (¬changed_i)
```

Figure 1: Linear Solver : synchronous iterative solution

Before we analyze the algorithm, we define some necessary terminology. A page is the smallest unit of memory on which coherence is enforced. Each page in the shared memory system is *owned* by a particular node. *Copyset* is a list of nodes having a cached copy

exploit synchronization information to reduce the cost of memory coherence maintenance [DSB88, GLL+90, AH90, LR91]. For example, as mentioned earlier, the DASH project only guarantees strict coherence for synchronization variables and writes to other variables can be propagated without blocking the processor that writes these variables (the propagation of the new values must be done by the time a certain type of operation is executed by the processor on a synchronization variable). These systems can execute programs written for coherent memory correctly as long as the programs follow the appropriate synchronization model. Thus, programmers do not see the weakened memory model. Other weak memory models have been described in [AHJ91, LS88].

Since we are dealing with a loosely coupled environment where nodes do not share physical memory, we are interested in protocols that can be used in an operating system or by the runtime support system of a programming language. These protocols are software-based and certain functionality easily realized in hardware-based solutions cannot be considered. For instance, writes to shared memory cannot be tracked as in a write-through cache coherence protocol. The emphasis of our work is not on developing a new DSM protocol or to define a new memory model but to identify, implement and evaluate operating system primitives that can be used to implement a variety of data coherence protocols. These primitives will allow users to tailor the coherence requirements of shared data. Thus, our work is unique in that it segregates coherence mechanisms and allows policies to be specified at the user level. Furthermore, when we restrict the program model to automatically transform shared memory programs to ones that use the primitives proposed by us, our goal is to develop efficient software-based implementations of synchronization mechanisms so that they can be used to maintain the coherence of other data efficiently. This differs from the approach taken by the hardware designers, where the synchronization operations are also of the type read and write but their execution ensures that the variables on which they operate are kept consistent.

3 Coherence Mismatch

In this section, we argue that for any given DSM protocol, there exist application programs with coherence requirements that do not match the way the DSM protocol implements the data coherence. This mismatch results in the loss of performance for the application which can be measured in a distributed system by the number of messages exchanged by the protocol to maintain coherence. We consider two common protocols: write-update and write-invalidate. The memory coherence provided by the write-update and write-invalidate protocols, however, is the same — sequential consistency [Lam79]. To illustrate the mismatch problem, we consider several examples. First, we present an implementation of a linear solver, a problem which often arises in a variety of numerical applications.

3.1 Linear Solver

A linear solver computes a solution to a set of linear equations of the form $Ax = b$ where A is an $n \times n$ matrix of real numbers, b is a vector of n real numbers, and x is a vector of size n of the unknowns to be computed. We consider a parallel version of the Gauss-Siedel algorithm for solving the linear equations. In this algorithm, new values of the elements of x are computed iteratively until the solution converges. Let us denote by x_i^t the value of x_i

clients can request data from a server and read it. The server generally does not notify the clients when the data value changes because they must explicitly request the new data from the server when necessary. On the other hand, a DSM protocol will try to reflect the change in the data value at all nodes where copies of the data exist. This results in invalidation or update messages which will be wasteful if the clients do not need to read the data again.

Our goal is to provide a DSM system that allows users to develop applications that exhibit performance comparable to an implementation of the application on a message passing system. Instead of identifying a single program model or a fixed set of data types and coherence protocols to reduce the cost of maintaining data coherence, our work focuses on operating system level primitives that can be used by programmers to manage the coherence of shared data. The following are the main issues that addressed in this paper:

- We analyze the performance problems of DSM systems by analyzing sample applications which help us to identify the different scenarios under which the existing protocols do not perform well (section 3).
- We identify operating system primitives which can be used to implement shared memory coherence without loss of performance, by customizing the coherence of memory according to the needs of the application. The primitives are to be directly used by programmers or automatically inserted into programs that assume a particular program and memory model (section 4).
- An implementation of the above primitives is described and performance figures are presented (section 5).
- We discuss how the task of programming can be simplified by inserting our primitives in in the code when the programming model is restricted (section 6).

Before presenting the main issues of the paper, we revisit some of the work in the area of distributed shared memories and memory consistency in the next section.

2 Related Work

Several researchers have proposed and implemented protocols for DSM systems. In his pioneering work in this area, Kai Li implemented and evaluated several variants of an invalidation-style protocol for multiprocessor cache consistency in a distributed environment. In the Clouds system [DCM+90], since concurrently executing threads in the same object may be at different nodes of a loosely coupled system, an underlying DSM facility maintains the coherence of data shared by such threads. The Clouds protocol exploits synchronization information in maintaining memory coherence by combining data transfer with synchronization requests [RAK89]. External pagers in Mach can also be used to share data across nodes [RTY+87]. The Munin system implements DSM on top of the V distributed system [CBZ91]. As described earlier, Munin exploits information about how a particular data item is shared between nodes to reduce the cost of maintaining its coherence. Other DSM systems are reported in [NL91].

Large scale multiprocessor systems also need to deal with problems similar to the ones that arise in DSM systems. In recent research, systems have been proposed that

processors do not share physical memory, it is possible to implement a shared memory abstraction. In such a shared memory system, the address space of each process consists of both a private space and space that is shared with the address spaces of one or more other processes. In this case, the processes of a distributed application have a uniform way of accessing state information, whether local or belonging to another process (which may be executing on a different processor): by examining memory.

Remote Procedure Calls(RPC) is another way of structuring distributed applications. RPC's hide message-passing by providing a procedural abstraction in place of sending messages. That is, instead of a process sending a message to another, which subsequently processes the message by doing some computation, the computation is encapsulated in a procedure inside the second process, and the first process directly calls (RPC) this procedure. RPC is more structured than sending messages but different mechanisms are used for operating on shared state: directly examining memory for local state and using procedural calls for remote state.

A shared memory abstraction in a distributed system, called *distributed shared memory* (DSM) [LH89], is desirable not only because it provides a uniform mechanism for accessing both local and remote state but also because it offers several other benefits. For example, if a server that implements RPCs is executed by several threads, DSM allows these threads to be scheduled at different processors to speed up execution of pending calls. In a message-passing or RPC system without DSM, the instances of the server at different processors will have to be programmed explicitly. The claim that DSM simplifies distributed programming is supported by the experiences of a number of researchers [ACD+90, BCZ90, Che86].

The DSM abstraction is implemented by treating the private memory of the processors as caches of the globally shared memory and enforcing coherence of these caches through the use of appropriate protocols whenever processes read from or write to the shared memory. Several common DSM protocols are adapted from multiprocessors and do not perform well due to the lack of capabilities in distributed systems that are easily and efficiently implemented in hardware in multiprocessor systems. Thus, the cost of a simpler programming model in DSM systems is the overhead of maintaining the coherence of the shared memory and resulting loss in performance. Recently, two general approaches have been proposed for reducing the performance degradation in DSM systems:

- Restrictions on the program behavior for its correct execution on a particular memory model. This approach is best exemplified by the *Release Consistency* model in which memory accesses are classified in two categories: *synchronization* and *data* accesses [GLL+90]. This approach permits a more efficient implementation of data accesses but programs only execute correctly if they follow the required synchronization model.

- Additional information provided about the use of shared memory. For example, in the Munin system [CBZ91], data items are classified by the programmer according to how they are shared, and rather than using a general protocol, and the coherence of each data type is implemented by a specialized protocol.

Although the above approaches lead to much more efficient DSM coherence protocols, performance of applications that use DSM is still poor when compared to similar applications written for message based systems. This can be easily explained by observing that a programmer explicitly deals with data transfer in message systems. For example, a set of

Application Specific Coherence Control for High Performance Distributed Shared Memory *

R. Ananthanarayanan *Mustaque Ahamad* *Richard J. LeBlanc*

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332
Ph: (404) 853-9390
e-mail: ananth@cc.gatech.edu

Abstract

Shared Memory is an attractive alternative to message passing for structuring distributed applications. However, the performance of applications written for shared memory is poor when compared to the performance of the same applications implemented using message passing. One of the reasons for this inefficiency is the lack of use of application specific knowledge in maintaining coherence of shared memory. When a fixed number of protocols are implemented by the system, programmers are unable to match the coherence specifications to the requirements of the application.

In this paper, we consider examples of programs that use shared memory and analyze the coherence requirements that they actually impose on shared memory. We present a cost analysis of these applications using existing distributed shared memory protocols. Our initial thrust in solving the efficiency problem is towards developing primitives that can be supported at the operating system level to maintain coherence of shared memory. These primitives can be used to exploit application semantics and can be either explicitly inserted by the programmer or by transformation programs. Thus, the primitives are the mechanisms that allow users to implement the policies that correspond to the coherence protocols. Initial results from in implementing these primitives are encouraging.

An alternate view of memory coherence is to consider how the events in the memory system relate to synchronization. We outline how these events can be used to obtain the performance benefits without complicating the programming of distributed applications.

1 Introduction

A loosely coupled distributed system consists of a set of processors with private memory that communicate through a network. Two well-known techniques for inter-process communication in such a system are message passing and distributed shared memory. The model provided in a message-passing system consists of processes with private address spaces communicating through messages. A distributed application can be programmed using a set of processes which synchronize and exchange state information using messages. Although

*This work was supported in part by NSF contracts CCR-91-06627 and CCR-86-19886.