Towards an Adaptive Distributed Shared Memory[1]

(Preliminary Version[2])

Jai-Hoon Kim Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
E-mail: {jhkim,vaidya}@cs.tamu.edu
Web: http://www.cs.tamu.edu/faculty/vaidya/

Technical Report 95-037

September 1995

Index Terms: **distributed shared memory, adaptive protocol, multiple protocol, hybrid protocol, competitive update protocol, invalidate protocol, update protocol.**

Abstract

The focus of this report is on software implementations of Distributed Shared Memory (DSM). In the recent years, many protocols for implementing DSM have been proposed. The protocols can be broadly divided into two classes: invalidation-based schemes and update-based schemes. Performance of these protocols depends on the memory access behavior of the applications. Some researchers have proposed DSMs that provide a family of consistency protocols or application specific protocols, and the programmer is allowed to choose any one of them for each shared memory object (or page) or each stage of an application. While such implementations have a potential for achieving optimal performance, they impose undue burden on the programmer. An *adaptive* implementation that automatically chooses the appropriate protocol for each shared memory page (at run-time) will ease the task of programming for DSM.

This report presents a simple approach for implementing *adaptive* DSMs. The approach is illustrated with the example of an adaptive DSM based on the *competitive* update protocol. The objective of the adaptive scheme is to minimize a pre-defined "cost" function. The cost functions considered here are *number of messages* and *size of messages.* (Other cost functions can also be used similarly.)

The proposed scheme allows each node to independently choose (at run-time) a different protocol for each page. The report presents preliminary evaluation of the *adaptive* DSM. Preliminary results shows that the performance is improved by *dynamically* selecting the appropriate protocol.

[1]This work is supported in part by the National Science Foundation under grant MIP-9502563.

[2]This report will be revised to include more discussion, generalizations of the proposed approach, and further experimental results. This preliminary version presents the basic adaptive scheme and evaluation of a synthetic application.

1 Introduction

Distributed shared memory (DSM) systems have many advantages over message passing systems [30, 23]. Since DSM provides a user a simple shared memory abstraction, the user does not have to be concerned with data movement between hosts. Many applications programmed for a multiprocessor system with shared memory can be executed in DSM without significant modifications.

Many approaches have been proposed to implement distributed shared memory [7, 12, 19, 30, 16, 6, 15, 26, 9, 27]. The DSM implementations are based on variations of *write-invalidate* and/or *write-update* protocols. Recent implementations of DSM use relaxed memory consistency models such as *release* consistency [7]. As no single protocol is optimal for all applications, researchers have proposed DSM implementations that provide a choice of multiple consistency protocols (e.g. [7]). The programmer may specify the appropriate protocol to be used for each shared memory object (or page). While this approach has the potential for achieving good performance, it imposes undue burden on the programmer. An *adaptive* implementation that automatically chooses the appropriate protocol (at run-time) for each shared memory page will ease the task of programming for DSM.

This report considers one approach for implementing adaptive DSM. This approach is similar to adaptive mechanisms used to solve many other problems [3], and can be summarized as follows (to be elaborated later):

1. Collect statistics over a *sampling period.* (Accesses to each memory page are divided into *sampling periods.*)

2. Using the statistics, determine the protocol that is "optimal" for the page.

3. Use the optimal protocol over the next sampling period.

4. Repeat above steps.

Essentially, the proposed implementation would use statistics collected during current execution to predict the optimal consistency protocol for the near-future. This prediction should

[3] For example, to predict the next CPU burst of a task, a Shortest-Job-First CPU scheduling algorithm may use an exponential average of the measured lengths of previous CPU bursts [25].

be accurate, provided that the memory access patterns change relatively infrequently. (The research presented in [26, 31] is closely related to that presented in this report. Section 6 discusses [26, 31] and other related research.)

To demonstrate our approach, we choose the *competitive* update protocol [1, 9, 14, 10]. This protocol is defined by a "threshold" parameter (we will rename the threshold as the "limit"). Different choices of the *limit* parameter yield different consistency protocols. Our goal is to determine the appropriate value of the limit so as to minimize a pre-defined "cost" metric. The preliminary experiments show that our approach can indeed reduce the cost, thus motivating further work.

This report is organized as follows. Section 2 summarizes the *competitive* update protocol [1, 9, 14, 10] and its generalizations. The proposed *adaptive* protocol is presented in Section 3. Section 4 shows the performance evaluation of the proposed scheme. Related work is discussed in Section 6. Section 7 concludes the report.

2 *Competitive* Update Protocol [1, 9, 14, 10] and Its Generalizations

A simple implementation of a *write-update* protocol is likely to be inefficient, because many copies of a page may be updated, even if some of them are not going to be accessed in the near future. Munin [7] incorporates a time-out mechanism to invalidate those copies of a page which have not been accessed by a node for a long time. [1, 9, 14, 10] present *competitive-update* mechanisms to invalidate a copy of a page at a node, if the copy is updated by other nodes "too many times" without an intervening local access. ([2] presents a similar scheme.) The advantage of this approach, as compared to [7], is as follows: the decision mechanism used in this approach (to determine when to invalidate a page) is dependent only on the application's access pattern, instead of *real time* as in Munin [7]. Quarks [16] also incorporates a mechanism similar to that presented in [1, 9, 14, 10].

We consider a software implementation of the *competitive* update protocol for a DSM that uses *release* consistency. This section presents details of the implementation, and also discusses some generalizations of the basic competitive update protocol. The adaptive scheme uses these generalizations of the original protocol.

The *competitive* update protocol defines a *limit* for each page at each node. If the number of update messages received for a page P at some node A – without an intervening access by node A – exceeds the *limit* for page P at node A, then the local copy of the page at node A is invalidated (other copies of the page are not affected).

Information Structure

We assume an implementation that is similar to Munin [7] and Quarks [16], with a few modifications to facilitate *competitive updates*. Each node maintains an information structure for each page resident in its memory. The information structure contains many pieces of information, as summarized below.

- *update_counter*: Counts how many times this page has been updated by other nodes, since the last *local* access to this page. When a page is brought into the local memory of a node, the counter is initialized to 0. Also, when a local process accesses (read or write) this page, the counter is cleared to 0.[4] The counter is incremented at every remote update of the page by any other node.

- *limit L*: Either set by user or transparently by the DSM protocol. The *limit* for each page determines the performance of the *competitive* update protocol. (As discussed later, we allow a different *limit* for each copy of each page.)

  Quarks[5] [16] also maintains information similar to our *update_counter* and *limit*. Section 6 discusses the differences.

- *last_updater*: Identity of the node that updated this page most recently. (*last_updater* is the originator of the most recent update message for the page).

- *copyset*: Set of nodes that are assumed to have a copy of this page. The *copyset* at different nodes, that have a copy of the same page, may be different. In general, a node

[4] A simple optimization can avoid clearing the counter on *every* local access.

[5] Quarks Beta release 0.8.

may not know exactly which other nodes have a copy of the page [7]. However, when a node updates remote copies of a page (when it does a *release*), at the end of the update procedure, that node knows precisely the set of nodes, that hold the copies of the page, that were updated. The "updater" node collects this information by means of acknowledgment messages sent as a part of the update procedure.

- *probOwner*: Points towards the "owner" of the page [7]. On a page fault, a node requests the page from the *probOwner*. If the *probOwner* does not have a copy of the page, it *forwards* the request to its *probOwner*. The request is thus *forwarded* until a node having a copy of the page is reached.

Note that each node maintains the above structure for *each* page in its local memory. In the following, when we use a phrase such as "the update_counter at node A", we are referring to "the update-counter for the page under consideration at node A". It is implicit that the "update_counter" (or some other information) pertains to a specific page.

Use of the Update_counter

During the execution, *update_counter* of each copy of a page changes dynamically, as explained below. A copy of a page is *invalidated* whenever its *update-counter* becomes equal to the *limit* [6]. (Initially, we assume that the *limit* for a page is fixed. In the *adaptive* protocol, however, the *limit* changes with time, depending on the memory access pattern.)

When a node, say A, receives from another node a copy of a page, say P, its update-counter is initialized to 0. The update-counter is incremented whenever node A receives an update message, for page P, from any other node. If a process on node A accesses page P, then the update-counter for page P at node A is cleared to 0. If, at any time, the update counter for page P at node A becomes equal to the *limit* L, then node A invalidates its copy of page P. Thus, page P on node A is invalidated *only when* L updates to the page, by other nodes, occur without an intervening access to the page by a process on node A. Thus, the competitive update protocol invalidates those pages that are accessed "less frequently" – the protocol can be tuned to a given application by a proper choice of limit L [1, 9, 14, 10].

[6] In case of *write-sharing*, a local copy of a page should not be invalidated if local copy has been modified (the modification needs to be sent to the other nodes).

As discussed later, the limit can also be changed dynamically (at run-time) to adapt to the time-varying memory access patterns.

Example

Figure 1 illustrates how the competitive update protocol works, by focusing on a single page in the DSM. For this example, let us assume that the *limit* L associated with this page is fixed at 3. In the figure, iL and iU denote *acquire* and *release* operations by node i.[7] Also, iR and iW denote *read* and *write* operations performed on this page by node i. The second row of the table presents a total ordering on the memory accesses performed by nodes 0, 1 and 2 on the page under consideration. The last three rows list the values of the update-counters at the three nodes at various times, e.g., the *update-counter:0* row corresponds to node 0. A "blank" in the table implies that the corresponding node does not have a copy of the page at that time. The reader familiar with *competitive* updates can omit rest of this example without a loss of continuity.

Initially, only node 0 has a copy of the page whose update-counter is 0 (column 0 in the table). Next, node 1 performs an *acquire* and tries to *read* the page (columns 1-2 in the figure). As it does not have a copy of the page yet, it receives a copy from node 0, and the update-counter for this copy is set to 0 (column 2 in the figure). Node 1 then performs a *release* (column 3). As node 1 did not perform any *writes*, no updates are needed at the *release*. Next, node 2 also performs *acquire-read-release* (columns 4-6). Again, a copy of the page is brought to the local memory of node 2, and its update-counter is set to 0 (column 5).

Next, node 0 performs an *acquire* and performs two *write* accesses to the page, and then performs a *release*. As node 0 has performed a local access to the page, its update-counter is cleared[8] to 0 (in this case, of course, it was already 0). In the software implementation

[7]Although we obtained the notation *iL* and *iU* by abbreviating *i-Lock* and *i-Unlock*, it should be noted that *acquire* and *release* operations in release consistency are not equivalent to *lock* and *unlock*.

[8]Actually, in practice, only if the update-counter is non-zero, any action need be taken. This can be implemented as follows. When the update-counter for a page becomes non-zero, the page is read/write-protected, so that any local access to the page will result in a page fault. On such a page fault, the fault handler will set the update-counter to 0, and remove the read/write-protection. Thus, any further local access to the page can proceed without any performance penalty. A similar optimization is used in Munin

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Memory Access		1L	1R	1U	2L	2R	2U	0L	0W	0W	0U	1L	1R	1W	1U	0L	0W	0U	0L	0W	0U	0L	0W	0U
Update-counter: 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
Update-counter: 1			0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	2	2	2	$\cancel{3}$
Update-counter: 2						0	0	0	0	0	1	1	1	1	2	2	2	$\cancel{3}$						

Figure 1: Update_counter

of *release* consistency protocol, the updates are not propagated to other nodes until node 0 performs a *release*. Therefore, the update-counters at nodes 1 and 2 are unchanged in columns 7, 8 and 9. When node 0 performs a *release*, the updates by node 0 are propagated to nodes 1 and 2. When the update message is received, nodes 1 and 2 incorporate the updates, and increment the respective update-counters by 1 (column 10). Next, node 1 performs an *acquire-read-write-release* sequence (columns 11-14). As node 1 has performed a local access, its update-counter is cleared to 0 (column 12). When node 1 performs the *release*, the updates are propagated to nodes 0 and 2, and their update-counters are incremented to 1 and 2, respectively (column 14). Next, node 0 performs a *acquire-write-release* sequence (columns 15-17). The update-counter of node 0 is cleared because of the local access (column 16). The update-counters at nodes 1 and 2 are incremented to 1 and 3, respectively, when the update is received from node 0 (column 17). Now, because the update-counter at node 2 has become equal to 3, the *limit* for the page, the local copy of the page at node 2 is *invalidated*. (The X in the figure denotes an invalidation.) When the copy is invalidated, node 2 sends a negative acknowledgment to the node from which the update message is received. Subsequently, node 0 performs two *acquire-write-release* sequences, at the end of which, the update-counter at node 1 becomes 3 (column 23). Therefore, the page copy at node 1 is also invalidated. At the end, only node 0 has a copy of the page, with update-counter 0 (column 23). □

[7] to implement its time-out mechanism.

2.1 Generalization of the Competitive Update Protocol

Unlike other similar schemes [2, 1, 9, 14, 10], implemented in hardware caches, the software implementation can be more flexible and complex (without affecting the performance adversely). The basic competitive protocol can be generalized in four ways, as summarized below. The adaptive protocol presented later uses these generalizations.

- The generalized competitive update approach can provide a different protocol for each page by adjusting the *limit* parameter independently for each page. By setting the *limit* to 1, the competitive update protocol becomes similar to the invalidate protocol, while the protocol is equivalent to the traditional update protocol when *limit* is infinity (or large). Thus, the generalized competitive update protocol can effectively be used as a "multiple" consistency protocol [7], simply by using a different *limit* for each page.

- If the access pattern to the same page is different for different nodes, a "hybrid" protocol [24] is more appropriate than a "pure" protocol. The *competitive* update protocol can act as a "hybrid" protocol by allowing each node to use a *different* limit for the local copy of the *same* page.

- It is possible to change the *limit* associated with each page (in fact, each copy of a page) dynamically. This feature is useful when the pattern of accesses to a page changes with time. This feature is also useful when, initially, the "optimal" value of the *limit* is unknown. Some heuristic can be used to adapt the protocol to dynamically determine the appropriate limit, so as to minimize the overhead. Such an *adaptive* protocol is the subject of this report.

- Instead of incrementing the limit by 1 each time an update occurs, we can allow the limit to be incremented by a different amount. When an update message is received by a node, the amount of increment in the update-counter can be made a function of the node from which the update message is received. When the "cost" of an update is dependent on the identity of the sender node, this feature is useful to tune the competitive update protocol to the underlying hardware architecture.

3 *Adaptive* Protocol

Our ultimate objective is to implement an *adaptive* DSM that can adapt to the time-varying memory access patterns of an application. The *competitive* update approach defines a family of protocols each characterized by a different value of limit L. An optimal implementation must, of course, choose the appropriate value of the limit.

An *adaptive* DSM based on the competitive update protocol would choose, at run-time, the appropriate value of L (between 1 and ∞) for each copy of each page. Also, if necessary, the limit will be changed, at run-time, if the access patterns change. The range of possible values of limit is large (1 to ∞), making it hard to design a heuristic for choosing the appropriate limit.

As a first step, this report focuses on a much simpler problem, where the choice of the limit is restricted to either 1 or ∞. With limit 1, the competitive update protocol is similar (but not exactly identical) to the traditional invalidate protocol, and with limit ∞, similar to traditional update protocol. Thus, the simplified problem is to choose between invalidate (limit $=1$) and update (limit $=\infty$) for each *copy* of each *page*.

At run-time, the proposed adaptive scheme collects some information (number and size of messages sent) that is used to periodically determine the new value of the limit for each copy of a page. The protocol is completely distributed. Now, we present a *cost analysis* to motivate our heuristics for choosing the limit.

3.1 Cost Analysis

The objective of our *adaptive* protocol is to minimize the "cost" metric of interest. Three possibilities for the cost metric are:

- The number of messages.
- The amount of data transferred.
- Execution time.

The objectives of minimizing these cost are inter-related. This report deals with the first two metrics.

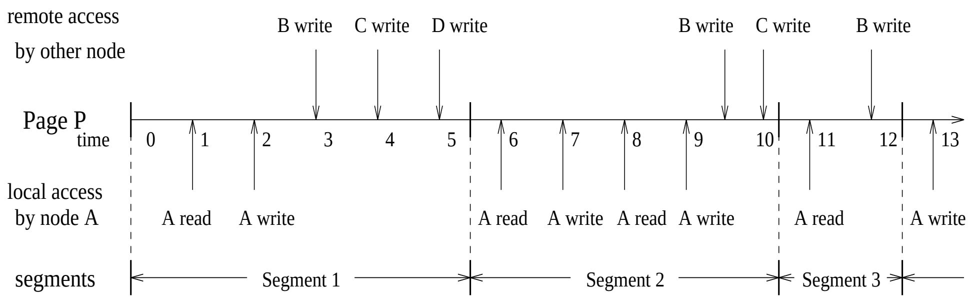

Figure 2: Segments

[30, 13, 31] present similar cost analysis for coherency overhead. [30] is based on read to write ratio and fault ratio, and [13] is based on the *write-run* model. [31] associates different costs with different events (such as cache hit, invalidate, update, and cache load). Our approach is based on the number of updates by other nodes between consecutive local accesses.

Number of Messages

Let us focus on the accesses to a particular page P as observed at a node A. These accesses can be partitioned into "segments". A new segment begins with the first access by node A following an update to the page by another node [9]. Figure 2 illustrates this with an example: (a) node A starts segment 1 for page P at time 1, (b) copy of page P on node A is updated by nodes B, C, and D. After that, (c) node A starts segment 2 by local access at time 6. Similarly, (d) node A starts segment 3 by local access at time 11 following remote updates by nodes B and C at time 9 and 10, respectively.

Now we evaluate the number of messages sent during each segment when $L = 1$ and $L = \infty$. For simplicity, in the present discussion, we do not consider the messages required to perform an *acquire*.

[9]Segment is a sequence of remote updates between two consecutive local accesses. *Write-run* [13] and *no-synch run* [4] models are introduced by others. A *write-run* is a sequence of local writes between two consecutive remote accesses. *no-synch run* is a sequence of accesses to a single object by any thread between two synchronization points in a particular thread. The *write-run* model was presented by Eggers [13] to predict the cache coherency overhead for a bus based multiprocessor system. This model measures the overhead by the number of sequences of single processor writes to an individual shared address, and the length of these sequences.

- update protocol (limit $L = \infty$): When $L = \infty$, a copy of the page is never invalidated. To evaluate the number of messages sent in each segment, we need to measure the number of updates made by other nodes during the segment. Let U be the *average* number of updates to the local copy during a segment. An acknowledgement is sent for each update message received. Therefore, the *average* number of messages needed in one segment, denoted by M_{update}, is $2U$. As shown in Figure 3, for example, 6 messages are needed in segment 1 because page P is updated 3 times by other nodes. (The numbers in parentheses in the figure denote number of messages associated with an event.) Similarly, 4 and 2 messages are needed in segment 2 and segment 3, respectively. For the three segments shown in the figure, the average number of updates per segment is $U = \frac{3+2+1}{3} = 2$. Therefore, $M_{update} = 2U = 4$.

- invalidate protocol (limit $L = 1$): From the definition of a segment, it is clear that, when $L = 1$, each segment begins with a page fault. On a page fault, $F + 1$ messages are required to obtain the page, where F is the average number of times the request for the page is forwarded before reaching a node that has the page (one additional message is required to send the page). With $L = 1$, when the first update message for the page (during the segment) is received from another node, the local copy of the page is invalidated. This invalidation requires two messages – one for the update message and one for a *negative* acknowledgement to the sender of the update. Ideally, once a page is invalidated, no more update messages will be sent to the node during the segment. (The reality, however, can be a little different, as discussed later.) Therefore, when $L = 1$, (ideally) the average number of messages needed in one segment (denoted by $M_{invalidate}$), is $F + 3$ (regardless of the value of U). As shown in Figure 4, $F + 3$ messages are needed in a segment on the average.

The analysis for arbitrary limit (other than 1 and ∞) is excluded in this report (to be considered in future work). Critical value of the number of updates, $U_{critical}$, where $L = 1$ and $L = \infty$ require the same number of messages, is computed as follows:

$$\begin{aligned} M_{update} &= M_{invalidate} \\ \Rightarrow 2\,U_{critical} &= F + 3 \end{aligned}$$

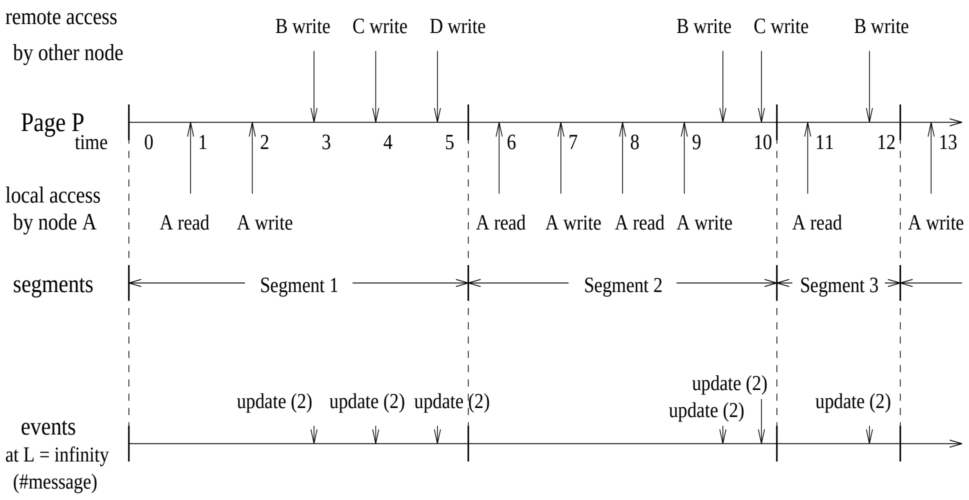

Figure 3: Illustrations for memory access and cost (update protocol)

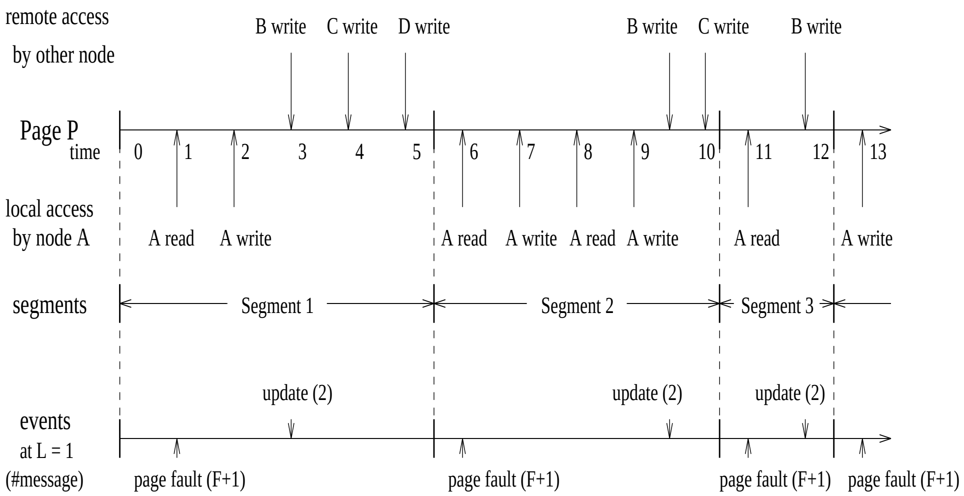

Figure 4: Illustrations for memory access and cost (invalidate protocol)

$$\Rightarrow U_{critical} = \frac{F+3}{2}$$

Therefore, if $U < \frac{F+3}{2}$, update protocol performs better. If $U > \frac{F+3}{2}$, invalidate protocol performs better. Our *adaptive* DSM implementation estimates U at run-time and sets $L = 1$ (for page P at node A) if $U > U_{critical}$, and $L = \infty$ otherwise. Note that F varies according to memory access patterns and the number of nodes. Although F can be estimated at run-time, previous work [19] suggests that F is less than 3 up to 32 nodes. For this discussion, we assume $F = 2$ (future work will consider estimating F at run-time). Then, $U_{critical} = \frac{F+3}{2} = \frac{2+3}{2} = 2.5$. In figure 2, update protocol would be chosen since $(U = 2) < (U_{critical} = 2.5)$.

Amount of Data Transferred

In the above analysis, we consider the number of messages as the cost. However, the amount of data transferred is also a factor in evaluating the cost of a DSM. The average amount of data transferred per segment can be evaluated similar to the number of messages.

- Let $D_{invalidate}$ be the average amount of data transferred in one segment for the invalidate protocol ($L = 1$). Then, $D_{invalidate} = p_{update} + (1 + F)\,p_{control} + p_{page}$, where p_{update} is the average size of an update message, $p_{control}$ is the size of a control message (page request, acknowledgment of update, etc.), and p_{page} is the size of a message that is required to send a page from one node to another.

- Let D_{update} be the average amount of data transferred in one segment for the update protocol ($L = \infty$). Then, it follows that, $D_{update} = (p_{update} + p_{control})\,U$.

Critical value of p_{update} where the two protocols require the same amount of data, is computed as follows:

$$\begin{aligned} D_{update} &= D_{invalidate} \\ \Rightarrow (p_{update(critical)} + p_{control})\,U &= p_{update(critical)} + (1 + F)\,p_{control} + p_{page} \\ p_{update(critical)} &= \frac{(1 + F - U)p_{control} + p_{page}}{U - 1} \end{aligned}$$

Since $p_{control}$, F, and p_{page} are constant [10], the critical value of p_{update} (i.e., $p_{update(critical)}$) can be determined, using the estimate for U. Having determined $p_{update(critical)}$, L is chosen to be 1 if the estimated $p_{update} > p_{update(critical)}$, and ∞ otherwise.

General Cost Functions

In general, the "cost" may be an arbitrary function. For instance, the *cost* may be some function of the message size. A procedure similar to that described above can be used to choose the appropriate value of L for such a cost function.

Let the "cost" of sending or receiving a message of size m be a function of m, say $c(m)$. (For example, $c(m)$ may be $K_1 + K_2\,m$, where K_1, K_2 are constants.) Total cost, C, is computed as follows:

- $C_{update} = (\overline{c(p_{update\_msg})} + c(p_{control}))\,U$
- $C_{invalidate} = \overline{c(p_{update\_msg})} + (1 + F)\,c(p_{control}) + c(p_{page})$

where $\overline{c(p_{update\_msg})}$ denotes the average cost of an update message. Appropriate limit can be chosen, by comparing the above costs estimated at run-time.

We believe that a general "cost" function can be used to reduce the total execution time. Such a cost function may have parameters other than message size. The difficulty, however, is in determining the appropriate cost function. We have not yet experimented with the general cost function.

The present implementation chooses the appropriate limit to minimize the number of messages <u>or</u> the amount of data transferred. (Any one of the two can be minimized at any time, not both.)

3.2 Implementation

As shown in the above analysis, the average number of updates since the last local access (U) and the average size of update message (p_{update}) are important factors to decide which protocol is better. Our *adaptive* protocol estimates these values over consecutive N_s segments

[10] F is actually not a constant – but, as discussed earlier, we will assume it to be a constant for this discussion.

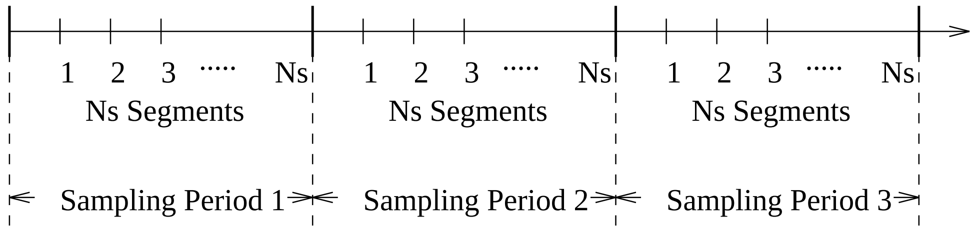

Figure 5: Segments and Sampling Periods

(let us call it a "sampling period") and selects appropriate protocol for the next sampling period. Figure 5 illustrates segments and sampling periods. The U (or p_{update}) value estimated during sampling period i is used to determine the value of limit L to be used during sampling period $i+1$.

Note that each node estimates U and p_{update} independently for each page. To facilitate estimation of U and p_{update} at run-time, each node maintains the following information for each page (in addition to that summarized in section 2).

- *version*: Counts how many times this page has been updated since the beginning of execution of the application. *version* is initialized to zero at the beginning of execution.

- *dynamic_version*: The *version* (defined above) of the page at the last local access. *dynamic_version* is initialized to zero at the beginning of execution, and set to *version* after a page fault or on performing an update. *dynamic_version* does not have to be updated on *every* local access – more details are presented below.

- *xdata*: Total amount of data transferred for updating copies of this page since the beginning of execution of the application. *xdata* is initialized to zero at the beginning of execution. (*xdata* is mnemonic for "exchanged data".)

- *dynamic_xdata*: The *xdata* (defined above) of the page at the last local access. *dynamic_xdata* is initialized to zero at the beginning of execution and set to *xdata* after a page fault or on performing an update (as described below).

- *update*: The number of updates by other nodes during the current sampling period. *update* is initialized to zero at the beginning of execution and is cleared to zero at the end of every sampling period.

- *d_update*: The amount of data received to update local copy of the page in the current sampling period. *d_update* is initialized to zero at the beginning of execution and is cleared to zero at the end of every sampling period.

- *counter*: Total number of segments during the current sampling period. *counter* is initialized to zero at the beginning of execution and is cleared to zero at the end of every sampling period.

The procedure for estimating U and p_{update} is as follows. In the following, we focus on a single page P at a node A – the same procedure is used for each page at each node.

1. On receiving an update message for page P, node A increments the *version* of page P by 1, and increments *xdata* by the size of the update message. Similarly, when node A modifies page P and sends update messages to other nodes that have a copy of page P, *version* is incremented by 1, and *xdata* is incremented by the size of the update message. This can be summarized as:

$$\begin{aligned} version &\longleftarrow version + 1 \\ xdata &\longleftarrow xdata + \text{size of the update message} \end{aligned}$$

   In addition, when node A sends update messages, *dynamic_version* is set equal to *version* and *dynamic_xdata* is set equal to *xdata*.

$$\begin{aligned} dynamic\_version &\longleftarrow version \\ dynamic\_xdata &\longleftarrow xdata \end{aligned}$$

2. On a page fault, when a copy of page P is received by node A, the sender of the page also sends its *xdata* and *version* along with the page. On receiving the page, *xdata* and *version* in the local page table entry (for page P) at node A are set equal to those received with the page.

$$\begin{aligned} xdata &\longleftarrow xdata \text{ received with the page} \\ version &\longleftarrow version \text{ received with the page} \end{aligned}$$

Also, *dynamic_version* in the local page table entry is compared to *version* received with the page. If there is a difference (let $d = version - dynamic\_version$), then the *update* variable for page P (at node A) is increased by d, *d_update* is increased by $(xdata - dynamic\_xdata)$, and the *counter* increases by one. That is, if $d > 0$, then:

$$\begin{aligned} update &\longleftarrow update + (version \text{ - } dynamic\_version) \\ d\_update &\longleftarrow d\_update + (xdata \text{ - } dynamic\_xdata) \\ counter &\longleftarrow counter + 1 \end{aligned}$$

At this point, a new segment begins. Therefore, the *dynamic_version* is set equal to *version* and *dynamic_xdata* is set to *xdata*.

$$\begin{aligned} dynamic\_version &\longleftarrow version \\ dynamic\_xdata &\longleftarrow xdata \end{aligned}$$

3. When *counter* becomes N_s, a sampling period is completed. Now, U is estimated as

$$U = \frac{update}{N_s},$$

p_{update} is estimated as

$$p_{update} = \frac{d\_update}{update},$$

and *update*, *d_update*, and *counter* are cleared to zero.

The estimated value U (or p_{update}) is used to decide which protocol is better – this protocol is used during the next sampling period. The protocol is selected as follows [11]:

[11]Note that invalidate protocol is selected at the beginning of execution in the example below. Alternatively, $L = \infty$ may be chosen as the initial value.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Memory Access	1R	1W	2R	2W	1R	2R	2W	1R	2W	3W	1R	2W	3W	2W	3W	2W	3W	2W	1R	1W	2R	2W	3R	3W	2W	3W	1R	3R	3W	1R	1W
Version: 1	0	1	1	2	2	2	3	3	4	5	5	6	7	8	9	10	11	12	12	13	13	14	14	15	16	17	17	17	18	18	19
Dynamic_Version: 1	0	1	1	1	2	2	2	3	3	3	5	5	5	5	5	5	5	5	12	13	13	13	13	13	13	13	17	17	17	18	19
Update: 1	0	0	0	0	1	1	1	2	2	2	4	0	0	0	0	0	0	0	7	7	7	7	7	7	7	7	11	11	11	12	0
Counter: 1	0	0	0	0	1	1	1	2	2	2	3	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	3	0
U (Avg. Updates): 1											**1.3**																			**4.0**	
Selected Protocol: 1	I	I	I	I	I	I	I	I	I	I	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	I	I
Segments	**1**				**2**			**3**			**1**								**2**								**3**			**1**	
Sampling Periods	**1**										**2**																			**3**	

Figure 6: An Example: Minimizing the Number of Messages

- To minimize number of messages: If $U > U_{critical}$, invalidate protocol is selected; else, update protocol is selected. Figure 6 shows an example of how node 1 uses the adaptive protocol to select the appropriate L for a page P. Assume that $U_{critical} = 2.5$ and $N_s = 3$. Note that information shown in this figure is maintained for page P at node 1. *Release* and *acquire* operations are not shown in the figure. A "solid" vertical line represents a *release* followed by an acquire. For example, the solid vertical line between columns 2 and 3 denotes that node 1 performed a *release* followed by an *acquire* by node 2. The identity of the node that performs the *release* (*acquire*) is identical to the node that performs the access immediately to the left (right) of the solid line. At the end of every segment (at column 5, 8, 11, 19, 27, and 30), *update* and *counter* are updated. At the end of every sampling sampling period (at column 11 and 30), U is evaluated and the appropriate protocol is selected (update protocol is selected at column 11, and invalidate protocol is selected at column 30).

- To minimize amount of data: If $p_{update} \geq p_{update(critical)}$, then invalidate protocol is selected, else update protocol is selected. (Expression for $p_{update(critical)}$ was obtained in Section 3.1.)

4 Performance Evaluation

Experiments are being performed to evaluate the performance of the *adaptive* DSM by running applications on an implementation of the adaptive protocol. We implemented the protocol by modifying another DSM, Quarks (Beta release 0.8) [16, 6]. The performance evaluation is in progress. This section presents some preliminary results.

Methodology

We executed a synthetic application *qtest* with the *adaptive* DSM. (Other benchmark applications will also be evaluated.) *qtest* is a simple shared memory access application: all nodes access the shared data concurrently. A process acquires mutual exclusion before each access and releases it after that. We measured the cost (i.e., number of messages and size of data transferred) by executing different instances of the synthetic application, as described below. Sampling period (N_s) is 30 for experiment 1 and 2, and sampling period (N_s) is 2 for experiment 3, 4, and 5.

Results

The body of the first instance of the *qtest* program is as follows:

```
repeat NLOOP times {
    acquire(lock_id);
    for (n = 1 to NSIZE)
       shmem[n]++;        /* increment */
    release(lock_id);
}
```

Each node performs the above task. All the shared data accessed in this application is confined to a single page. For this application, Figure 7 and 8 show the measured cost by increasing the number of nodes (N). The costs are plotted per "transaction" basis. A *transaction* denotes the set of operations – *acquire*, *shared memory access*, and *release* – in one loop of the *qtest* main routine. The curve for the *adaptive* scheme in Figure 7 is plotted using the heuristic for minimizing the number of messages, the curve in Figure 8 is plotted using the heuristic for minimizing the amount of data transferred.

Each node executes the `repeat` loop 300 times ($NLOOP = 300$). The size of shared data ($NSIZE$) is 2048 bytes – all in one page – page size being 4096 bytes. *Adaptive* protocol starts with an update protocol (i.e., L is initialized to ∞ for each page at each node). At the end of each sampling period ($N_s = 30$), each node evaluates U (or p_{update}) for the page and selects the appropriate L – this L is used during the next sampling period.

Note that the adaptive DSM does *not* minimize number of messages and amount of data transferred *simultaneously* – either one of them can be minimized at any time by the choice of appropriate heuristic.

Figure 7 shows the comparisons of the number of messages transferred per transaction. (A *transaction* is defined above.) In this figure, "message:protocol" denotes the number of messages required by the specified protocol, and "#update" denotes the average number of updates (U) calculated over the entire application. As N increases, the average number of updates (U) increases proportionally. The number of messages for an update protocol increases as U increases. Previous analysis showed that ($M_{update} = 2U$). However, since this does not include messages for *acquire*, total number of messages required is greater than $2U$. Based on the previous analysis, the number of messages for an invalidate protocol may be expected to be independent of the average number of updates (U) (recall that $M_{invalidate} = F + 3$). However, the number of messages for an invalidate protocol increases a little bit (as N increases) probably because F and the number of messages for *acquire* increase as N increases.

From our approximate analysis, $U_{critical}$ was expected to be 2.5, however, for the above application, $U_{critical}$ is just over 3. This is probably because our analysis assumed the ideal situation where a node that has invalidated a page does not receive any updates for the page. However, in the implementation, this is not the case - the node may still receive some updates after invalidating the page. These updates are, of course, unnecessary. These unnecessary messages can be avoided if (1) the "updater" node first sends the update to the *last_updater* (the node which updated the page most recently), and obtains from that node the most

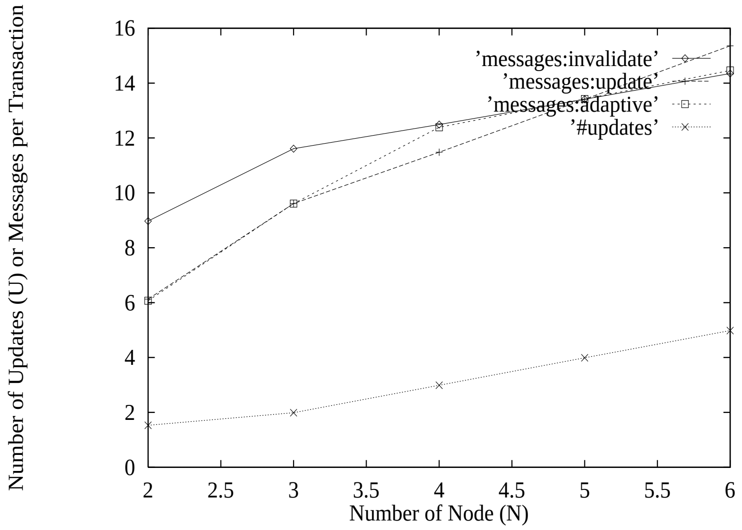

Figure 7: Experiment 1: The Average Number of Updates (U) and Messages per Transaction

recent information for the "copyset" (defined in section 2), and (2) "newborn" copyset[12] is maintained. In spite of the approximate estimate of $U_{critical}$ used in the experiment, the *adaptive* protocol performed well (except at $N = 4$). The number of messages required by the *adaptive* protocol is near the minimum of invalidate ($L = 1$) and update ($L = \infty$) protocols.

Figure 8 shows the comparison of the amount of data transferred per transaction. Since *qtest* application modifies large amount of data ($NSIZE = 2048$ bytes), an update protocol requires large amount of data transfer as the number of nodes (N) increases. However, an invalidate protocol requires nearly same amount of data for all N. *Adaptive* scheme chooses the appropriate protocol for each value of N, thereby minimizing the amount of data transferred.

The second experiment was performed with the main loop shown below:

```
repeat NLOOP times {
    acquire(lock_id);
    for (n = 1 to NSIZE) {
```

[12] A node in the "newborn" copyset has a page which has not been updated since the node obtained the page from some other node.

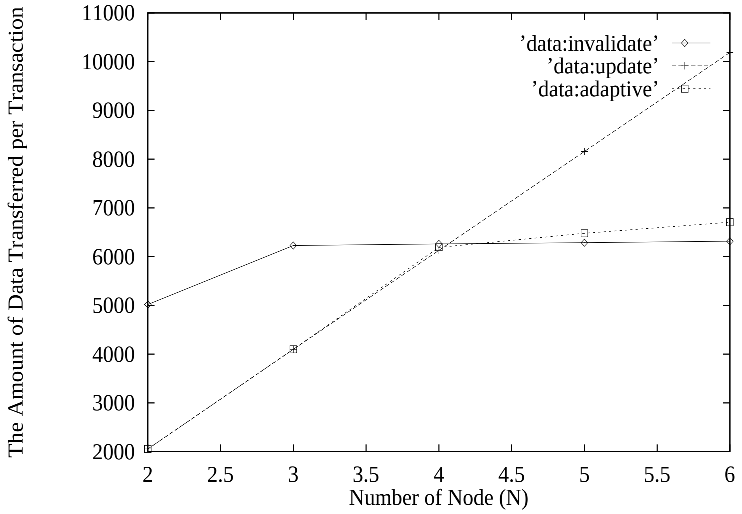

Figure 8: Experiment 1: The Amount of Data Transferred per Transaction

```
        if (random() < read_ratio) /* 0 <= random <= 1 */
          r_value = shmem[n];      /* read */
        else
          shmem[n] = w_value;      /* write */
      }
      release(lock_id);
  }
```

All the shared data accessed in this application is confined to a single page. For this experiment, we assume a small amount of shared data access per iteration of the `repeat` loop (NSIZE = 4). Additionally, each iteration of the `repeat` loop either reads or writes the shared data depending on whether a random number (`random()`) is smaller than the read ratio or not. This allows us to control the frequency of write accesses to the shared data. Each node accesses the shared data 100 times ($NLOOP$ = 100). (We observed that the results converge quite quickly.) Figure 9 presents the number of messages per transaction. As shown, the adaptive scheme performs well for all read ratios. We also measured the

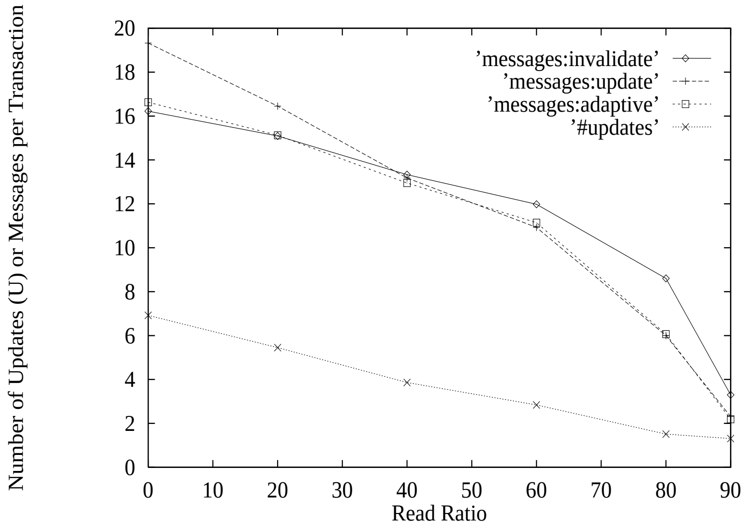

Figure 9: Experiment 2: The Average Number of Updates (U) and Messages per Transaction

amount of data transferred. Measurements (omitted here) also show that, when the amount of data transfer is the cost criterion, update protocol is better than invalidate protocol for all read ratios, because a small amount of shared data is accessed per iteration of the `repeat` loop (NSIZE = 4). Therefore, in this case, the adaptive protocol performs similar to the update protocol. (As the *qtest* application used here is simple, more experiments are needed to evaluate the full impact of read ratio on adaptive DSM performance.)

The third experiment was performed with another instance of the *qtest* application. This experiment was performed with the main loop shown below:

```
for (k = 1 to 3) {               /* three stages */
    for (l = 1 to NLOOP/3) {     /* NLOOP/3 iterations per stage */
        acquire(lock_id);
        for (j = 1 to 3)         /* j = page number */
            if(((node_id + j + k) mod 3 != 0) or (l mod 10 == 0))
                for (n = 1 to NSIZE)
                    shmem[n]++; /* increment */
        release(lock_id);
    }
```

```
}
```

We executed this applications with three nodes ($N = 3$) which share 3 pages (numbered 1, 2 and 3). Page size is 4096 bytes and each loop iteration *may* increment (not always, as described next) 2048 bytes ($NSIZE = 2048$) in each shared page. Total of 300 iterations ($NLOOP = 300$) in each node are divided into three *stages* (numbered 1, 2 and 3). In each stage, one of three pages is accessed less frequently than the other two pages. Specifically, let i, j and k be the node number, page number, and the stage number. Then, node i accesses page j during each iteration of stage k **if** $(i + j + k) \bmod 3 \neq 0$. If $(i + j + k) \bmod 3 = 0$, then node i accesses page j only during each 10-th iteration in stage k. This access pattern is time-varying – pattern of accesses by each node to each page changes with time.

To cope with this, the adaptive protocol is used in all its generality: (i) possibly different protocol is used for each page ("multiple" protocol), i.e., different limit for each page, (ii) different nodes may use different values of limit for the *same* page ("hybrid" protocol), and (iii) adapt to the time-varying access pattern by periodically computing the average number of updates (U) and the average amount of data for updates p_{update}, and modifying the limits accordingly.

Figure 10 presents the *total* number of messages sent over the execution of the application, and the *total* amount of data transferred for update, invalidate and adaptive protocols. Even though access patterns are different in each node as well as in each page, and vary during the execution, *adaptive* protocol can adapt to this. As shown in Figure 10, the *adaptive* protocol requires less number of messages and less amount of data than the other two protocols. (As noted earlier, either the number of messages or the data size can be minimized at any time, not both.)

The fourth experiment was performed with an instance of the *qtest* application similar to the third experiment. We executed this applications with eight nodes ($N = 8$) which share 4 pages (numbered 1 through 4). Page size is 4096 bytes and each loop iteration *may* increment 1024 bytes ($NSIZE = 1024$) in each shared page. Total of 400 iterations ($NLOOP = 400$) in each node are divided into four stages (numbered 1 through 4). In each

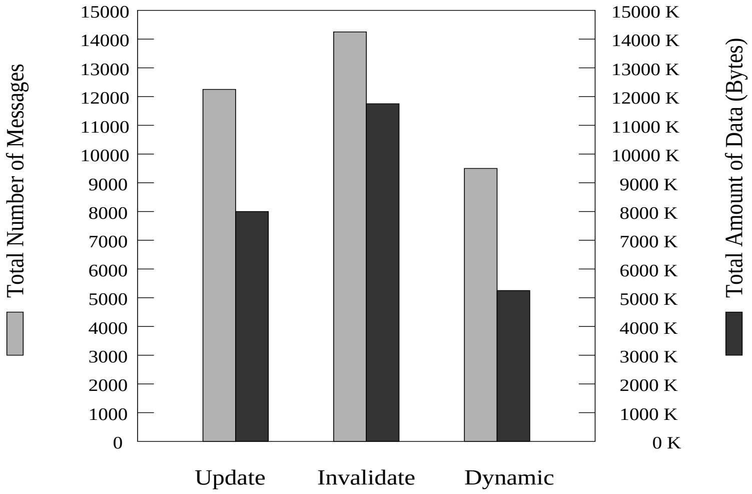

Figure 10: Experiment 3: Performance Comparison: Total costs over the entire application

stage, one of four pages is accessed more frequently than the other three pages. Specifically, let i, j and k be the node number, page number, and the stage number. Then, node i accesses page j during each iteration of stage k **if** $(i + j + k)$ mod $4 = 0$. If $(i + j + k)$ mod $4 \neq 0$, then node i accesses page j only during each 10-th iteration in stage k. This access pattern is also time-varying – pattern of accesses by each node to each page changes with time.

As shown in Figure 11, the *adaptive* protocol requires less number of messages and less amount of data than the other two protocols. Observe that invalidate protocol performs better than update protocol, whereas update protocol performs better than invalidate protocol in the third experiment since the number of nodes is small.

The fifth experiment was performed with an instance of the *qtest* application similar to experiment 4, except for memory access frequency. In each stage, two of four pages are accessed more frequently than the other two pages. Specifically, let i, j and k be the node number, page number, and the stage number. Then, node i accesses page j during each iteration of stage k **if** $(i + j + k)$ mod $2 = 0$. If $(i + j + k)$ mod $2 \neq 0$, then node i accesses page j only during each 10-th iteration in stage k.

As shown in Figure 12, update protocol performs worst because four nodes access each page frequently which causes U to be large much of the time. When $U > U_{critical}$, invalidate protocol is better than update protocol. Therefore, the *adaptive* protocol behaves like the

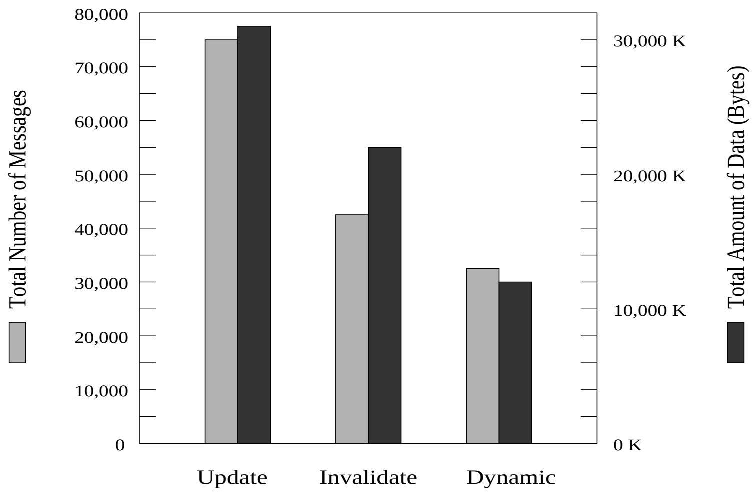

Figure 11: Experiment 4: Performance Comparison: Total costs over the entire application

invalidate protocol, and results in costs comparable with the invalidate protocol.

The preliminary results presented above suggest that the adaptive scheme can perform well. More work is needed to evaluate the performance using other benchmark applications, and to explore other optimizations to the adaptive scheme.

5 Comparison with *Competitive* Update Protocol

As shown in section 4, *adaptive* protocol performs better than the update and invalidate protocols. The question that arises is, how does the *adaptive* protocol compare with a *competitive* update protocol for which $1 < L < \infty$. The *adaptive* protocol presented above chooses between $L = 1$ and $L = \infty$. Therefore, it is unlikely that it will perform well for *all* applications when compared to *competitive* update protocol with a fixed L in the range $1 < L < \infty$, say $L = 4$. To perform better that such a protocol, the *adaptive* protocol must also be given the choice of using values of limit L other than 1 and ∞. We are presently developing and evaluating heuristics that will allow the *adaptive* protocol to choose such values of L. We expect that, with such heuristics, the *adaptive* protocol will be able to perform better than the *competitive* update protocol with a fixed limit.

In the rest of this section, we compare the above *adaptive* protocol with *competitive* update protocol and present a simple modification to the *adaptive protocol.* We use a fixed limit

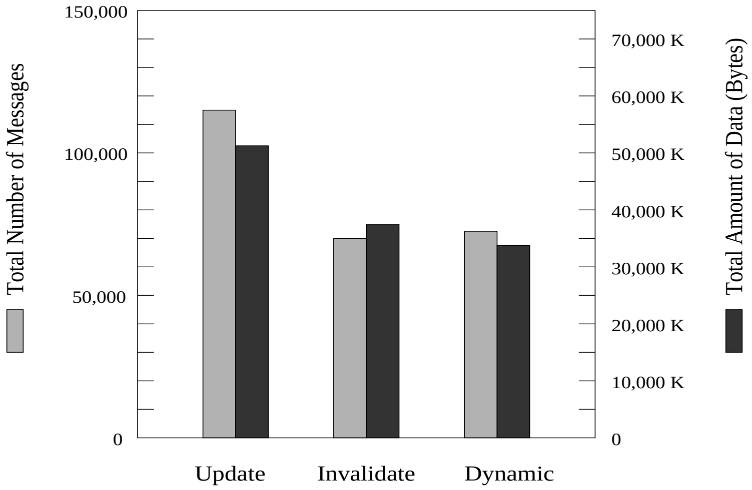

Figure 12: Experiment 5: Performance Comparison: Total costs over the entire application

$L = 4$ for the *competitive* update protocol. We use the same applications as in experiments 3, 4, and 5 – let us number the new experiments 6, 7, and 8, respectively.

Figures 13, 14, and 15 present results for the above *adaptive* protocol (bars marked as **`adaptive`**) and the *competitive* update protocol. Observe that *adaptive* protocol *does not always* achieve a lower cost than *competitive* update protocol.

To improve the performance of *adaptive* protocol, we modified it as follows:

- Adjust the critical value of the number of updates ($U_{critical}$): As observed in the previous experiments, $U_{critical}$ is actually larger than 2.5 that we obtained by an approximate analysis. For the modified protocol we use $U_{critical} = 4$.

- The previous adaptive protocol is modified such that, when $U < U_{critical}$, L is set equal to 4 (instead of ∞). The protocol with the above two modification is named **`adaptive2`**. In Figures 13, 14, and 15, observe that **`adaptive2`** outperforms *competitive* update protocol with $L = 4$ as well as the previous *adaptive* protocol.

The above observation suggests that more work is needed to obtain an appropriate heuristic to choose arbitrary values of L. As the range of possible values of L is large ($1 \leq L \leq \infty$), one alternative may be restrict L to a small set of values, for instance $\{1,2,4,6,\infty\}$ – of course, the appropriate set of values needs to be determined first.

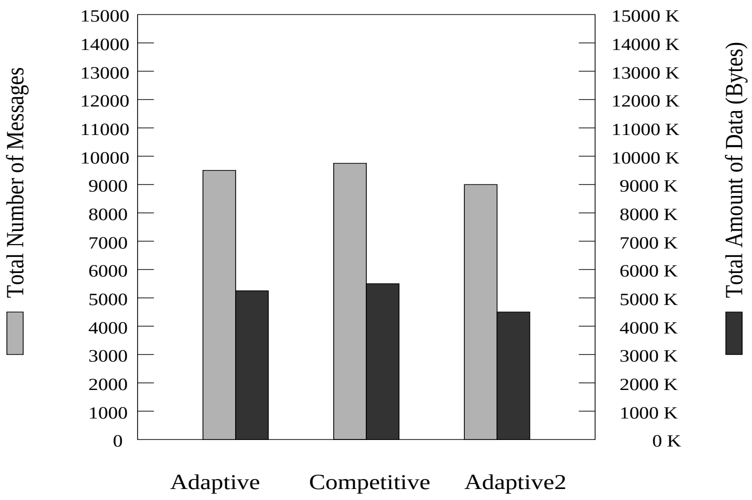

Figure 13: Experiment 6: Performance Comparison: Total costs over the entire application

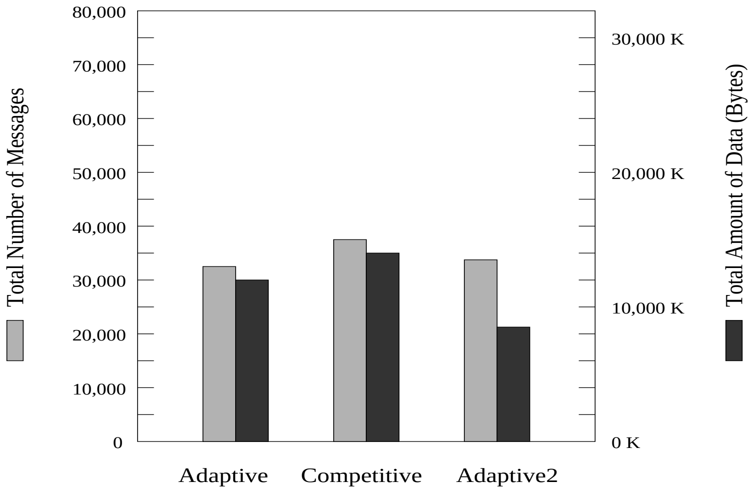

Figure 14: Experiment 7: Performance Comparison: Total costs over the entire application

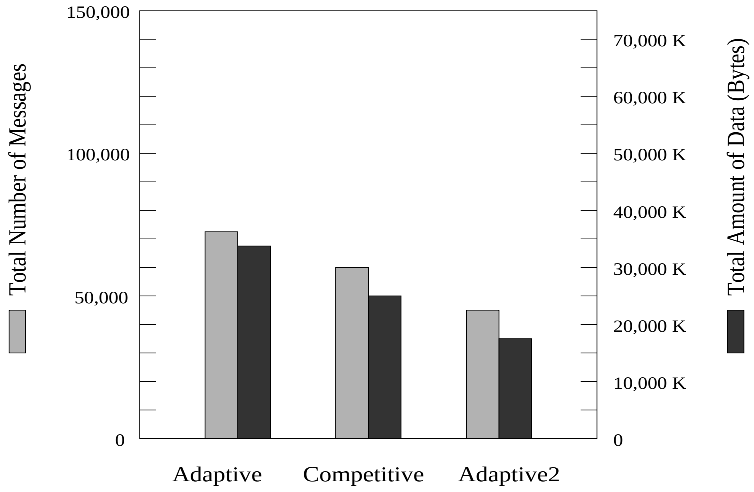

Figure 15: Experiment 8: Performance Comparison: Total costs over the entire application

6 Related Work

Many schemes have been proposed to reduce overhead by adapting to memory access patterns [2, 33, 11, 8, 29, 24, 9, 1, 14, 10, 4, 7, 20, 5, 26, 27, 3, 21, 16, 6]:

- The approach proposed in this paper is related to the work by Veenstra and Fowler [31]. [31] evaluates the performance of three types of *off-line* algorithms: (i) an algorithm that chooses statically, at the beginning of the program, either invalidate or update protocols on a per-page basis, (ii) an algorithm that chooses statically either invalidate or update protocols for each cache block, and (iii) an algorithm that can choose invalidate or update protocols at each write. Algorithms (i) and (ii) are similar to *multiple* protocols in [7, 16], and (iii) is similar to our *adaptive* protocols which can choose the appropriate protocol at run-time. However, in [31], the chosen protocol is applicable to all copies of a cache block, whereas in our scheme, the protocol used for each copy of a page may be different. [31] considers *off-line* algorithms, for a bus-based system. On the other hand, this report considers adaptive (on-line) algorithms that are applicable to distributed systems.

- [32] examines the performance of on-line hybrid protocols that combine the best aspects of several protocols (invalidate protocol, update protocol, migratory protocol, etc.), on bus-based cache-coherent multiprocessors. The results shows that the hybrid protocols outperform any single pure protocol in most applications.

- Ramachandran et al. [26, 28] present new mechanisms for explicit communication in shared memory multiprocessors. They propose explicit communication primitives which allows selectively updating a set of processors, or requesting a stream of data ahead of its intended use (prefetch). Their scheme can also adapt to time-varying sharing pattern by dynamically changing the set of nodes to be updated (or invalidated). The basic difference between our approach and [26] is that our scheme does not need to know whether a particular synchronization controls access to a given shared memory page or not. The scheme in [26] makes use of such information to determine whether a copy of the page should be updated or invalidated.

- Dynamic cache coherence approach presented by Archibald [2] dynamically chooses to update or invalidate copies of a shared data object. If there are three writes by a single processor without intervening references by any other processor, all other cached copies are invalidated.

- Optimizations for migratory sharing have also been proposed [8, 29, 9, 22]. These protocols dynamically identify migratory shared data and switch to migratory protocol in order to reduce the overhead. [8, 29] are based on invalidate protocol, and [22, 9] are based on competitive update protocol.

- Quarks [16] limits the number of updates independently for each page and each copy. If a copy of the page is updated more than *limit* times without local access, the copy is invalidated. From the source code for the Beta release 0.8 of Quarks, we understand that Quarks detects all local accesses for write, however, it can not detect local accesses for read if the copy exists locally (page hit). The counter for the number of updates is cleared on detected local accesses. The initial *limit* is set when the page table is initialized, and is doubled at a page fault, until it reaches a predefined maximum. The *limit* in Quarks does not seem to decrease once it has increased.

The *adaptive* protocol proposed in this report can detect all relevant[13] local accesses (read as well as write), and the *limit* is changed based on access patterns in the previous sampling period (the limit may decrease or increase).

- Tempest [27, 3] allows programmers and compilers to use user-level mechanism to implement shared memory "policies" that are appropriate to a particular program or data structure. Tempest consists of four types of mechanisms (low-overhead messaging, bulk data transfer, virtual memory management, and fine-grained memory access control).

- Munin [7] incorporates an update *timeout* mechanism. The main idea of this mechanism is to invalidate local copy of a page that has not been accessed for a certain period of time, *freeze* time, after it was last updated. Although the two approaches (*limit* and *timeout*) have similar goals, they do not behave identically [17]. Whereas the time limit, *freeze time*, is fixed in Munin, our *adaptive* protocol can adapt to time-varying memory access patterns by changing the update *limit* at run-time.

- Multiple consistency protocol was proposed in [7, 16]. Several categories of shared data objects are identified: *conventional, read-only, migratory, write-shared, and synchronization.* They developed many memory coherence techniques that perform efficiently for these categories of shared data objects. But programmer should know the memory access behaviors on each shared variable to specify a protocol used for the variable.

- Lebeck and Wood [18] introduce dynamic self-invalidation (DSI) scheme to reduce overhead in directory-based write-invalidate cache coherence protocol. The directory identifies blocks for self-invalidation. The directory conveys the self-invalidation information to the cache when responding to a cache miss. The cache controller self-invalidates the blocks.

- Lindemann and Schon [20] add LOCAL state, to SHARED, INVALID, and EXCLUSIVE states, to relax the consistency model. All memory accesses to the shared pages

[13] It is not necessary to detect all local accesses, only the first access – read or write – after a remote update of a page needs to be detected.

are performed locally at the node which has invoked the *define local* system call. The global image of the shared pages are updated with the local images by invoking the *define global* system call.

- Bianchini and LeBlanc address software caching which can adapt to changes in memory reference behavior by making a new copy of data and repartitioning the data as needed for each phase of execution [5].

- [21] presents a flexible communication mechanism. Their scheme uses a programmable node controller, called MAGIC. MAGIC is responsible for implementing the cache-coherence and message-passing protocols.

- Hybrid protocol is more appropriate than a "pure" protocol for a DSM, if the access pattern for the same page is different in each node. TOP-1 [24], a tightly coupled snoop-cache-based multiprocessor, has a hybrid coherence protocol which allows an update protocol and an invalidate protocol, which can be dynamically changed, to coexist simultaneously. However, TOP-1 needs additional hardware design, cache mode register (to specify a cache mode: update mode and invalidate mode) and CH (Cache Hit) bus line (to indicate a snoop hit).

- An adaptive cache coherence protocol is presented by Yang, Thangadurai, and Bhuyan [33]. This scheme is based on a hardware approach that handles multiple shared reads efficiently. Their protocol allows multiple copies of a shared data block in a hierarchical network with minimum cache coherence overhead by dynamically partitioning the network into sharing and nonsharing regions based on program behavior.

- Adjustable block size coherent caching scheme is proposed by C. Dubnicki and T. LeBlanc [11]. Their cache structure dynamically adjusts the cache block size according to recently observed reference behavior. Cache blocks are split across cache lines when false sharing occurs, and merged into a single cache line to exploit spatial locality.

7 Conclusion and Future Work

Our objective is to design an *adaptive* DSM that can adapt to time-varying pattern of accesses to the shared memory. Our approach continually gathers statistics, at run-time, and periodically determines the appropriate protocol for each copy of each page. The choice of the protocol is determined by the "cost" metric that needs to be minimized. The cost metrics considered in this report are number and size of messages required for executing an application using the DSM implementation. A generalization to minimize arbitrary cost metrics is also presented.

The proposed adaptive approach is illustrated by means of an adaptive DSM scheme that chooses either update or invalidate protocol for each copy of a page – the choice changes with time, as the access patterns change. The update and invalidate protocols are implemented as special cases of the *competitive* update protocol. The report presents preliminary evaluation of the *adaptive* DSM using an implementation. Preliminary results from the implementation suggest that the proposed adaptive approach can indeed reduce the *cost*.

Further work is needed to fully evaluate the effectiveness of the proposed adaptive approach. Issues being addressed include the following:

- Extensive evaluation of the adaptive scheme is necessary to determine whether it will perform well with real applications. The *qtest* application is quite simple. Therefore, We plan to evaluate the adaptive scheme with several benchmark applications.

- Another issue that needs to be addressed is the choice of N_s that determines the length of the sampling period. Instead of keeping N_s fixed, it may be possible to choose the appropriate value at run-time.

- The *cost* metrics considered in the report are *number* and *size* of messages. Other cost metrics need to be considered. In particular, impact of our heuristics on application execution time needs to be evaluated.

- The report presented a heuristic for choosing between two protocols. In general, the DSM may provide a larger set of protocols, and the appropriate protocol should be adaptively chosen from this set. For instance, the choices may include *migratory* protocol, and competitive update protocol with $L = 1, 2, 4, 6, \infty$. A heuristic for choosing between one of these, at run-time, needs to be developed to implement more efficient DSMs.

- Comparison of the proposed approach with previously proposed adaptive schemes.

- The adaptive approach (based on collection of statistics) presented here can be combined with ideas developed by other researchers (e.g., [26]) to obtain further improvement in DSM performance. As yet, we have not explored this possibility.

Acknowledgements

We thank John Carter and D. Khandekar at the University of Utah for making Quarks [16] source code available in public domain.

References

[1] A. Karlin et al., "Competitive snoopy caching," in *Proc. of the 27'th Annual Symposium on Foundations of Computer Science*, pp. 244–254, 1986.

[2] J. Archibald, "A cache coherence approach for large multiprocessor systems," in *International Conference on Supercomputing*, pp. 337–345, July 1988.

[3] B. Falsafi et al., "Application-specific protocols for user-level shared memory," in *International Conference on Supercomputing*, pp. 380–389, Nov. 1994.

[4] J. Bennett, J. Carter, and W. Zwaenepoel, "Adaptive software cache management for distributed shared memory architectures," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 125–134, May 1990.

[5] R. Bianchini and T. LeBlanc, "Software caching on cache-coherent multiprocessors," in *Proceedings of International Conference on Parallel and Distributed Processing*, pp. 521–526, 1992.

[6] J. Carter, D. Khandekar, and L. Kamb, "Distributed shared memory: Where we are and where we should be headed," in *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, pp. 119–122, May 1995.

[7] J. B. Carter, *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.* PhD thesis, Rice University, Sept. 1993.

[8] A. Cox and R. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 98–108, May 1993.

[9] F. Dahlgren, M. Dubois, and P. Stenstrom, "Combined performance gains of simple cache protocol extentions," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 187–197, Apr. 1994.

[10] F. Dahlgren and P. Stenstrom, "Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 26, pp. 193–210, Apr. 1995.

[11] C. Dubnicki and T. LeBlanc, "Adjustable block size coherent caches," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 170–180, May 1992.

[12] S. Eggers and R. Katz, "A characterization of sharing in parallel prograns and its application to coherency protocol evaluation," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373–382, May 1988.

[13] S. J. Eggers, "Simplicity versus accuracy in a model of cache coherency overhead," *IEEE Transactions on Computers*, vol. 40, pp. 893–906, Aug. 1991.

[14] H. Grahn, P. Stenstrom, and M. Dubois, "Implementation and evaluation of update-based cache protocols under relaxed memory consistency models," *Future Generation Computer Systems*, vol. 11, pp. 247–271, June 1995.

[15] P. Keleher, *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, Jan. 1995.

[16] D. Khandekar, "Quarks: Portable dsm on unix," tech. rep., University of Utah.

[17] J.-H. Kim and N. H. Vaidya, "Distributed shared memory: Recoverable and non-recoverable limited update protocols," Tech. Rep. 95-025, Texas A&M University, College Station, 1995. To appear in Proc. of 1995 Pacific Rim International Symposium on Fault-Tolerant Systems.

[18] A. Lebeck and D. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995. To appear.

[19] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

[20] C. Lindemann and F. Schon, "Performance evaluation of consistency models for multicomputers with virtually shared memory," in *System Science, 1993 Annual Hawaii International Conf.*, vol. II, pp. 154–163, 1993.

[21] M. Heinrich et al., "The performance impact of flexibility in the stanford flash multiprocessor," in *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 274–285, Oct. 1994.

[22] H. Nilson and P. Stenstrom, "An adaptive update-based cache coherence protocol for reduction of miss rate and traffic," tech. rep., Lund University. To appear in Parallel Architectures and Languages Europe, July 1994.

[23] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *IEEE Computer*, vol. 24, pp. 52–60, Aug. 1991.

[24] N. Oba, A. Moriwaki, and S. Shimizu, "Top-1: A snoop-cache-based multiprocessor," in *Proc. 1990 International Phoenix Conference on Computers and Communication*, pp. 101–108, Oct. 1990.

[25] J. Peterson and A. Silberschatz, *Operating System Concepts*, pp. 105–108. Addison-Wesley Publishing Company, Inc., 1983.

[26] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural mechanisms for explicit communication in shared memory multiproccessors," Tech. Rep. GIT-CC-94-59, Georgia Institute of Technology, Dec. 1994. To appear in Proc. of International Conference on Supercomputing 1995.

[27] S. Reinhardt, J. Larus, and D. Wood, "Tempest and typoon: User-level shared memory," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 325–336, Apr. 1994.

[28] G. Shah, A. Singla, and U. Ramachandran, "The quest for a zero overhead shared memory parallel machine," in *Proceedings of International Conference on Parallel Procesing*, vol. I, 1995.

[29] P. Stenstrom, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 109–118, May 1993.

[30] M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Computer*, pp. 54–64, May 1990.

[31] J. Veenstra and R. Fowler, "A performance evaluation of optimal hybrid cache coherency protocols," in *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 149–160, Oct. 1992.

[32] J. Veenstra and R. Fowler, "The prospects for on-line hybrid coherency protocols on bus-based multiprocessors," Tech. Rep. 490, The University of Rochester, Mar. 1994.

[33] Q. Yang, G. Thangadurai, and L. Bhuyan, "Design of an adaptive cache coherence protocol for large scale multiprocessors," *IEEE Transaction on Parallel and Distributed Systems*, vol. 3, pp. 281–293, May 1992.