A Cost-Comparison Approach for Adaptive Distributed Shared Memory [*]

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

Abstract

The focus of this paper is on software implementations of Distributed Shared Memory (DSM). In recent years, many protocols for implementing DSM have been proposed. 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. Therefore, some *adaptive* schemes that automatically choose the appropriate protocol have been proposed.

This paper presents a simple approach for implementing *adaptive* DSMs. The approach is illustrated with the example of an adaptive DSM based on the *invalidate* and *competitive update* protocols. The objective of the adaptive scheme is to minimize a pre-defined "cost" function. The cost functions considered here are *number of messages* and *amount of data transfer*.

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

1 Introduction

Software distributed shared memory (DSM) systems have many advantages over message passing systems [21, 29]. 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 on a software DSM system without significant modifications.

Many approaches have been proposed to implement distributed shared memory [6, 7, 14, 15, 19, 26, 29]. The DSM implementations are based on variations of *write-invalidate* and/or *write-update* protocols. Recent implementations of software 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. Many *adaptive* schemes have also been proposed (e.g.,[8, 25, 26, 28]), as summarized in the next section.

This paper considers a simple but effective approach for implementing adaptive DSM. This approach is similar to adaptive mechanisms used to solve many other problems [1], 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 minimizes the "cost" for each page P.
3. Use the minimum *cost* protocol for each page P to maintain consistency of page P 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 be accurate, provided that the memory access patterns change relatively infrequently. To demonstrate our approach, we present an adaptive scheme that chooses between the *invalidate* protocol and the *competitive update* protocol [1, 9, 10, 13]. The *competitive update* protocol is defined by a "threshold" parameter; we will rename the threshold as the "limit". Experimental results show that our adaptive scheme performs well because memory access patterns do not change frequently in many applications.

This paper is organized as follows. Related work is discussed in Section 2. The proposed *adaptive* protocol is presented in Section 3. Section 4 presents the performance evaluation of the proposed scheme. Section 5 concludes the paper.

2 Related Work

Many schemes have been proposed to reduce overhead by adapting to memory access patterns. Veenstra and Fowler [30] evaluate

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

[1]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 [23].

the performance of *off-line* algorithms for *bus-based* systems, that choose invalidate or update protocol based on off-line analysis. On the other hand, this paper considers adaptive (on-line) algorithms that are applicable to distributed systems. [31] examines the performance of on-line hybrid protocols for *bus-based* cache-coherent multiprocessors. Our scheme deals with distributed implementations of shared memory. [22] also describes a hardware implementation of a hybrid scheme. Ramachandran et al. [25, 27] present new mechanisms for explicit communication in shared memory multiprocessors which allows selectively updating a set of processors, or requesting a stream of data ahead of its intended use (prefetch). The basic difference between our approach and [25] is that our scheme does not need to know whether a particular synchronization controls access to a given shared memory page or not. [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 in [2]. Competitive update scheme [1, 9, 10, 13] invalidates a page if the number of remote updates to the page (between local accesses) exceeds a "threshold" or a "limit" parameter. Quarks [15] uses a variation of the competitive update scheme. Protocols presented in [8, 9, 20, 28] dynamically identify migratory shared data and switch to migratory protocol in order to reduce the overhead. Tempest [3, 26] allows programmers and compilers to use user-level mechanism to implement shared memory "policies" that are appropriate to a particular program or data structure. Multiple consistency protocol was proposed in [7] where several categories of shared data objects are identified: *conventional, read-only, migratory, write-shared, and synchronization*. But, with their approach, the programmer needs to know the memory access behaviors on each shared variable to specify a protocol used for the variable. [5, 11, 18] also present other schemes to reduce coherency overhead. IRG (Inter-Reference Gap) model for the time interval between successive references to the same address was presented in [24]. It estimates the future IRG values by using prediction based algorithm and can be used for memory replacement algorithm, etc.

3 *Adaptive* Protocol

Our objective is to implement an *adaptive* DSM that can adapt to the time-varying memory access patterns of an application. Our initial goal was to design a heuristic to dynamically choose between the *invalidate* and the *update* protocols. However, for reasons that will be apparent later, the proposed adaptive scheme actually chooses between the *invalidate* and *competitive update* [9] protocols.

The *competitive update* protocol is defined using a "threshold" parameter – in this paper, we will refer to the threshold as "update limit" or just "limit". When using the competitive update protocol with limit L, a node A invalidates the local copy of a page P if and when $(L+1)$-th update to the page by other nodes occurs since the previous access of page P by node A. The traditional *update* protocol can be obtained by choosing $L = \infty$. The protocol obtained when $L = 0$ is similar to the traditional *invalidate* protocol. Thus, the competitive update protocol is convenient for designing an adaptive scheme – the problem of choosing appropriate protocol (invalidate or update) is now reduced to the problem of choosing the appropriate *limit* (0 or ∞) – the proposed adaptive scheme actually chooses 0 or a non-zero finite limit, as explained later.

The proposed adaptive scheme collects run-time data on number and size of messages; the data is used to periodically determine the new value of *limit* for each copy of a page. The protocol is completely distributed in that each node independently determines the limit to be used for each page it has in its local memory. (Thus, different nodes may choose different limits for the same page.) Now, we present a *cost analysis* to motivate our heuristics for choosing the appropriate limit.

3.1 Cost Analysis

The objective of our *adaptive* protocol is to minimize the "cost" metric of interest. Two cost metrics considered here are: (i) number of messages, and (ii) amount of data transferred. In this section, we evaluate the above cost metrics for consistency protocols of interest. [12, 29, 30] also present cost analysis for coherency overhead. [29] presents an analysis based on many parameters such as read-write ratio, page fault ratio, and cost of sending/receiving a page. Eggers [12] presents a *write-run* model to predict the cache coherency overhead for the bus based multiprocessor system. [12, 29] both do not consider the problem of implementing adaptive schemes. [30] associates different costs with different events (such as cache hit, invalidate, update, and cache load) and presents an *off-line* algorithm to choose invalidate or update protocols at each write. We present an "on-line" (or adaptive) approach based on the number (or size) of updates by other nodes between consecutive local accesses, as explained next.

Our analysis assumes that the DSM uses release consistency and dynamic distributed ownership analogous to Munin [7] and Quarks [15].

Minimizing the Number of Messages

We now consider *number of messages* as the cost metric. 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. Thus, a segment is a sequence of remote updates between two consecutive local accesses.[2] (Segments are defined from the point of view of each node. Therefore, for the same page, different nodes may observe different segments.) Figure 1 illustrates *segments* observed at node A with an example: (a) segment 1 for page P starts at time 1 when node A reads page P, (b) copy of page P on node A is then updated by nodes B, C, and D. After that, (c) node A starts segment 2 by a 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.

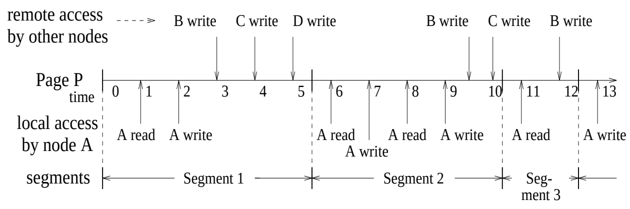

Figure 1: Segments

Now we evaluate the number of messages sent during each segment for invalidate protocol (or competitive update protocol with limit $L = 0$) and update protocol (or competitive update protocol with limit $L = \infty$). For simplicity, in the present discussion, we do not consider the messages required to perform an *acquire*. (The number of messages for an *acquire* is same for both protocols.

• **update protocol (limit $L = \infty$)**: When $L = \infty$, a copy of the page P 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 number of such updates to the local copy of page P during a segment. An acknowledgement is sent for each update message received.

[2]*Segment* is a sequence of remote updates between two consecutive local accesses. *Write-run* [12] and *no-synch run* [4] models are introduced by others. A *write-run* is a sequence of local writes between two consecutive remote accesses [12]. A *no-synch run* is a sequence of accesses to a single object by any thread between two synchronization points in a particular thread [4].

Therefore, the number of messages needed in one segment, denoted by M_{update}, is $2U$. As shown in Figure 2, 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.

- **invalidate protocol (limit $L = 0$)**: From the definition of a segment, it is clear that, when $L = 0$, each segment begins with a page fault. On a page fault, $F + 2$ messages are required to obtain the page, where F is the number of times the request for the page is forwarded (due to dynamic distributed ownership) before reaching the owner – one additional message is required to send the page, and one message to acknowledge receipt of the page. With $L = 0$, 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. (In reality, however, a node that has invalidated local copy of a page P may sometime receive an update for page P.) Therefore, when $L = 0$, (ideally) the number of messages needed in one segment (denoted by $M_{invalidate}$), is $F + 4$. As shown in Figure 3, $F + 4$ messages are needed in a segment. Note that the actual value of F may be different in each segment.

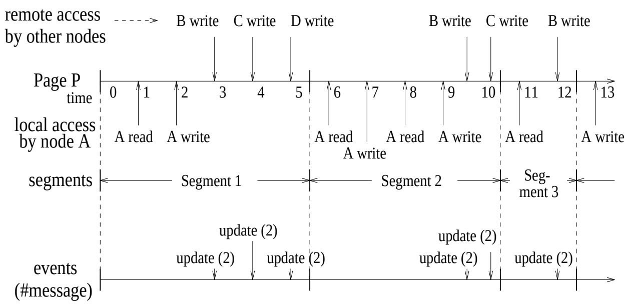

Figure 2: Illustrations for memory access and cost (update protocol)

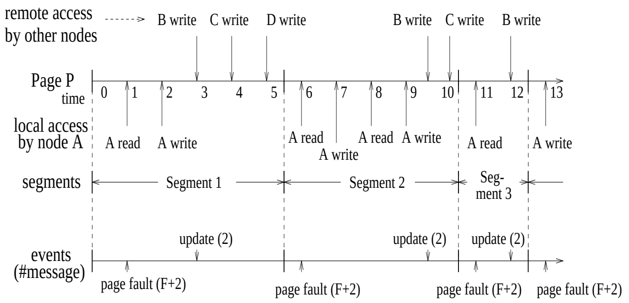

Figure 3: Illustrations for memory access and cost (invalidate protocol)

Critical value of the number of updates, $U_{critical}$, where $L = 0$ and $L = \infty$ require the same number of messages, is computed as follows: $M_{update} = M_{invalidate} \Rightarrow 2U_{critical} = F + 4 \Rightarrow U_{critical} = \frac{F+4}{2}$.
Therefore, if $U > \frac{F+4}{2}$, invalidate protocol has a lower cost. If $U < \frac{F+4}{2}$, update protocol performs better. Based on this observation, the following adaptive scheme is derived (this scheme will be modified soon).

- As the value of U may be different in each segment, each node collects data for a few consecutive segments (termed "sampling period") and estimates average value of U and F.
- At the end of the sampling period, if $U > \frac{F+4}{2}$ then the invalidate protocol $(L = 0)$ is chosen for the next sampling period, otherwise, the update protocol ($L = \infty$) is chosen.

The above protocol is modified in two ways as described next. We will describe an implementation of the final adaptive scheme later.

1. It is hard to estimate F accurately (without additional message overhead) when the *limit* is non-zero. Therefore, we assume a constant value for F. In the following, we assume $F = 4$. Clearly, F must depend on the application and on the number of nodes (processors) used. Thus, assuming $F = 4$ is not likely to be always accurate. This assumption could cause the adaptive scheme to achieve worse performance than it potentially can. Yet, as shown here, the approximate heuristic performs reasonably well for the applications and number of nodes considered here. With the above assumption, $U_{critical} = 4$.

2. The above adaptive scheme chooses $L = \infty$ when estimated U is no larger than $U_{critical}$. The motivation for this choice is the following: if U was small in the recent past, it is expected to be small in the near future. However, when this assumption turns out to be incorrect, the adaptive scheme ends up having made a wrong choice. Therefore, instead of choosing $L = \infty$ when $U \le U_{critical}$, we choose $L = U_{critical} - 1 = 3$. When $L = 3$, a local copy of a page is invalidated if the page is updated 4 times by other nodes within one segment. (The adaptive scheme will perform comparably if L were chosen to be $U_{critical}$ instead of $U_{critical} - 1$.)

With the above modifications, the adaptive scheme that attempts to minimize the *number of messages* can be summarized as follows:

- *Each* node collects data over a "sampling period" for *each* local page, and estimates the average value of U.
- At the end of the sampling period, if $U > U_{critical}$ then the invalidate protocol $(L = 0)$ is chosen for the next sampling period for that page, otherwise, the competitive update protocol (with $L = 3$) is chosen. $U_{critical}$ is assumed to be 4.

As a reference, the number of messages required in a segment when using a competitive update protocol (with limit L, $0 < L < \infty$) is computed below:

- competitive update protocol ($0 < L < \infty$): A copy of the page is updated until it receives L update messages from other nodes (between two consecutive local accesses). Upon receiving $(L + 1)$-th update message, local copy of the page is invalidated. If the number of update messages (U) received during the segment is at most L, then the page is not invalidated. In the case of competitive update protocol, it is convenient to include the messages required to bring a page from a remote node when counting the number of messages for the segment in which the page was invalidated (rather than when counting the number of messages for the next segment). Thus, if $U \le L$, then $M_{competitive}$ is $2U$, similar to M_{update}. Else, however, $M_{competitive} = 2(L + 1) + (F + 2) = 2L + M_{invalidate}$. ($2(L + 1)$ messages for $L + 1$ updates and their acknowledgements, and $F + 2$ for bringing a page on the page fault when the next local access is attempted.)

Amount of Data Transferred

In the above analysis, we consider the number of messages as the cost. Now, we consider the amount of data transferred as the cost metric. The *average* amount of data transferred per segment is evaluated below.

- Let $D_{invalidate}$ denote the *average* amount of data transferred per segment when using the invalidate protocol ($L = 0$). Then, $D_{invalidate} = \overline{p_{update}} + (\overline{F}+2)\, p_{control} + p_{page}$, where $\overline{p_{update}}$ is the *average* size of an update message that causes the local copy of the page to be invalidated, $p_{control}$ is the size of a control message (page request, acknowledgment of update, etc.), p_{page} is the size of a message that is required to send a page from one node to another, and $\overline{F}$ is the average number of times a page request is forwarded.

- Let D_{update} denote the *average* amount of data transferred in one segment for the update protocol ($L = \infty$). Then, it follows that, $D_{update} = (\overline{p_{update}} + p_{control})\, U$ where U now denotes the average number of remote updates in a segment.

Critical value of U ($U_{critical}$), where the two protocols require the same amount of data transfer, is computed as follows:

$$D_{update} = D_{invalidate}$$
$$\Rightarrow (\overline{p_{update}} + p_{control})\, U_{critical} = \overline{p_{update}} + (\overline{F}+2)\, p_{control} + p_{page}$$
$$\Rightarrow U_{critical} = \frac{\overline{p_{update}} + (\overline{F}+2)\, p_{control} + p_{page}}{\overline{p_{update}} + p_{control}}$$
$$\Rightarrow U_{critical} = \frac{\overline{p_{update}} + 6\, p_{control} + p_{page}}{\overline{p_{update}} + p_{control}} \quad \text{assuming } \overline{F} = 4.$$

Note that $U_{critical}$ is different when minimizing *amount of data* as compared to when minimizing *number of messages*.

Having determined $U_{critical}$, $L = 0$ is chosen if U measured at run-time is greater than $U_{critical}$. To evaluate $U_{critical}$, $\overline{p_{update}}$ is also estimated at run-time. For a reason similar to that described previously when minimizing the number of messages, we do not choose $L = \infty$ when $U \leq U_{critical}$. Instead, when $U \leq U_{critical}$, we choose the competitive update protocol with limit = $U_{critical}$. Choosing limit = $U_{critical} - 1$ would also result in similar cost. Because we chose limit = $U_{critical} - 1$ for minimizing the number of messages, as an illustration, we decided to use limit = $U_{critical}$ for minimizing amount of data.

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:

- $C_{update} = (\overline{c(p_{update\_msg})} + c(p_{control}))\, U$
- $C_{invalidate} = \overline{c(p_{update\_msg})} + (2 + F)\, c(p_{control}) + c(p_{page})$

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.

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. When both need to be small, a cost function of the form $K_1 + K_2\, m$ should be used, where m is message size.

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 ($\overline{p_{update}}$) are important factors to decide which protocol is better. Our *adaptive* protocol estimates these values over consecutive N_s segments (let us call it a "sampling period") and selects appropriate protocol for the next sampling period. Figure 4 illustrates segments and sampling periods. The U and $\overline{p_{update}}$ values estimated during sampling period i are used to determine the value of limit L to be used during sampling period $i+1$.

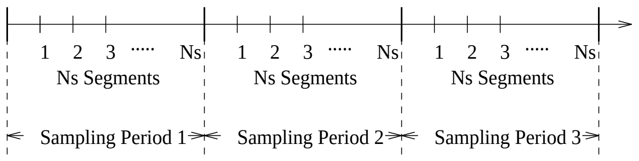

Figure 4: Segments and Sampling Periods

Each node <u>independently</u> estimates U and $\overline{p_{update}}$ for each page. To facilitate estimation of U and $\overline{p_{update}}$ at run-time, each node maintains the following information for each page.

- *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 <u>not</u> 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. In addition, when node A sends update messages, *dynamic_version* is set equal to *version* and *dynamic_xdata* is set equal to *xdata*.

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. Also, *dynamic_version* in the local page table entry is compared to *version* received with the page. Let $d = version - dynamic\_version$. Then the *update* variable for page P (at node A) is incremented by d, *d_update* is incremented by $(xdata - dynamic\_xdata)$, and the *counter* incremented by one. At this point, a new segment begins. Therefore, the *dynamic_version* is set equal to *version* and *dynamic_xdata* is set to *xdata*.

3. When *counter* becomes N_s, a sampling period is completed. Now, U and $\overline{p_{update}}$ are estimated as $U = \frac{update}{N_s}$, and $\overline{p_{update}} = \frac{d\_update}{update}$, and *update*, *d_update*, and *counter* are cleared to zero.

The estimated values of U and $\overline{p_{update}}$ for page P at node A are used to decide which protocol is better. If $U > U_{critical}$, invalidate protocol ($L = 0$) is selected; else, competitive update protocol with appropriate limit is selected (as described in section 3.1). The chosen L is used for page P at node A during the next sampling period.

4 Performance Evaluation

Experiments are performed to evaluate the performance of the *adaptive* DSM by running applications on an implementation of the adaptive protocol. We implemented the adaptive protocol by modifying another DSM, named Quarks (Beta release 0.8) [6, 15]. This section presents the experimental results.

We evaluated the adaptive scheme using a synthetic application (named *qtest*) as well as five other applications (Floyd-Warshall, SOR, ProdCons, Reader/Writer, and QSORT). *qtest* is a simple shared memory application based on a program available with the Quarks release [15]: 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. SOR is available with the Quarks release [15]. ProdCons and Reader/Writer are based on *qtest*. Sampling period (N_s) is chosen to be 2 for all applications.

Results for `qtest` Application

The body of the first instance of the *qtest* program (named `qtest1`) is as follows:

```
qtest1: repeat NLOOP times {
          acquire(lock_id);
            for (n = 1 to NSIZE)
              /* increment shared memory
                 location */
              shmem[n]++;
            release(lock_id);
        }
```

Each node performs the above task. All the shared data accessed in this application is confined to a single page. Each node executes the `repeat` loop 300 times, i.e., $NLOOP = 300$. 300 iterations were sufficient for the results to converge. The size of shared data ($NSIZE$) is 2048 bytes – all in one page – page size being 4096 bytes. (The next experiment considers small NSIZE.) The *adaptive* protocol initializes L to 3 for each page at each node. At the end of each sampling period ($N_s = 2$), each node estimates U and $\overline{p_{update}}$ for the page and selects the appropriate L – this L is used during the next sampling period.

For this application, Figures 5 and 6 show the measured cost by increasing the number of nodes (N). The costs are plotted per "transaction" basis. A *transaction* denotes a sequence of operations – namely, *acquire*, *shared memory access*, and *release* – in one loop of the *qtest1* main routine. The curve for the *adaptive* scheme in Figure 5 is plotted using the heuristic for minimizing the number of messages; the curve in Figure 6 is plotted using the heuristic for minimizing the amount of data transferred.

In Figure 5, the curve named "protocol" denotes the number of messages required by the specified protocol, and "#update" denotes the average number of updates per segment (U) calculated over the entire application. As number of nodes N increases, the average number of updates per segment (U) increases proportionally. In spite of the approximate estimate of $U_{critical}$ used in our analysis, the *adaptive* protocol performs well except for $N = 5$. For small N, the adaptive scheme performs similar to update schemes (which are optimal for small N), and for large N the adaptive scheme performs similar to the invalidate scheme (which is optimal for large N). (In case of $N = 5$, our adaptive protocol often chooses $L = 3$ because estimated U in most sampling periods is not greater than $U_{critical} = 4$. Therefore, for $N = 5$, the adaptive scheme performs similar to the competitive update scheme.) In summary, the number of messages required by the *adaptive* protocol is near the minimum of invalidate ($L = 0$) and competitive update ($L = 3$) protocols (except when $N = 5$).

Figure 6 shows the comparison of the amount of data transferred per transaction. Since *qtest1* application modifies large amount of data ($NSIZE = 2048$ bytes), an update protocol requires larger amount of data transfer as the number of nodes (N) increases. However, an invalidate protocol requires nearly constant amount of data transfer (per transaction) for all N. Competitive protocol requires large amount of data transfer when $N > 4$ because it cannot adapt to minimize the amount of data transferred. *Adaptive* protocol chooses the appropriate protocol for all values of N, thereby minimizing the amount of data transferred.

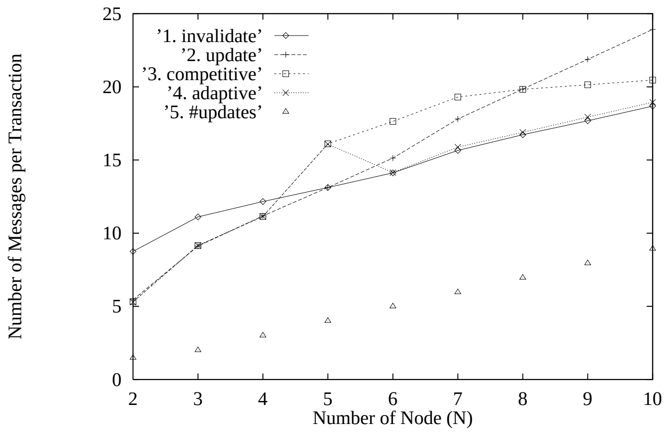

Figure 5: qtest1: Average Number of Updates (U) and Messages per Transaction

The second experiment was performed with the main loop (`qtest2`) shown below:

```
qtest2:  repeat NLOOP times {
            acquire(lock_id);
            if (random() < read_ratio)
            /* 0 <= random <= 1 */
               for (n = 1 to NSIZE)
```

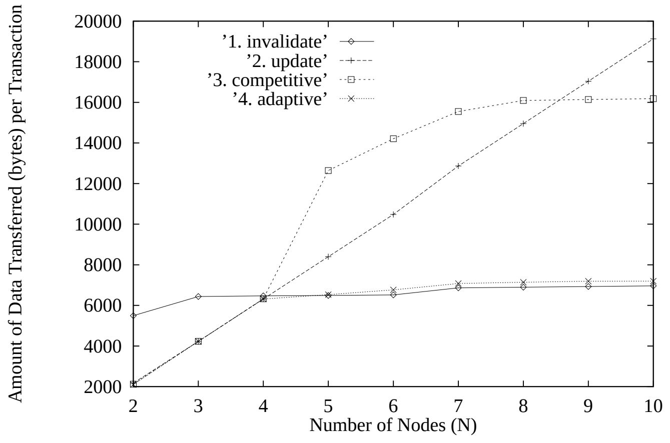

Figure 6: qtest1: Amount of Data (Bytes) Transferred per Transaction

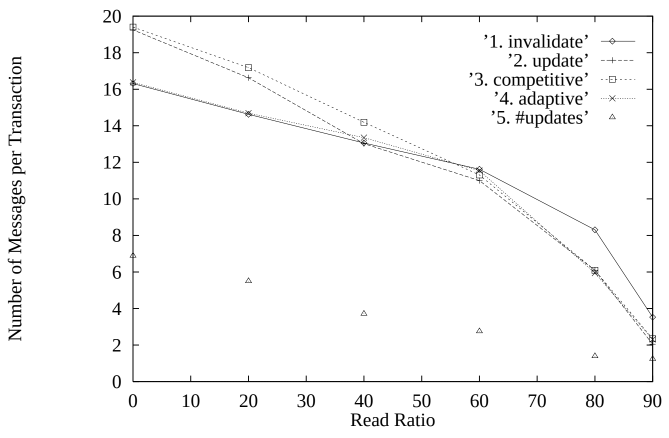

Figure 7: qtest2: Average Number of Updates (U) and Messages per Transaction

```
                /* read shared memory */
                r_value = shmem[n];
          else
             for (n = 1 to NSIZE)
                /* write shared memory */
                shmem[n] = w_value;
          release(lock_id);
       }
```

All the shared data accessed in qtest2 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. 8 nodes access the shared data 100 times each ($NLOOP = 100$). (We observed that the results converge quite quickly.) Figure 7 presents the number of messages per transaction (i.e., acquire, shared memory access, and release). As shown, the adaptive scheme performs well for all read ratios.

Figure 8 shows the comparison of the amount of data transferred per transaction. Since *qtest2* application modifies small amount of data ($NSIZE = 4$ bytes), our adaptive protocol chooses a competitive protocol with large update limit (L) (refer to Section 3.1). Therefore, the adaptive protocol requires small amount of data transfer. Competitive update protocol with limit $L = 3$ (or small L, in general) results in relatively larger amount of data transfer when the average size of an update message, p_{update}, is small.

Results for Other Applications

We now evaluate our adaptive scheme by executing five additional applications (Floyd-Warshall, SOR, ProdCons, QSORT, and Reader/Writer) on 8 nodes. Floyd-Warshall is all-pair-shortest-path algorithm. (We use 128 vertices as input.) SOR is Successive Over-Relaxation algorithm which executes simple iterative relaxation algorithm. (We use 512×512 grid.) ProdCons is implementation of a simple Producer/Consumer model. Producers make data which will be used by consumers. (We execute total 4,000 "transactions" for ProdCons. A transaction denotes a sequence of operations – namely, *acquire shared memory access* and *release* – similar to as defined in *qtest*.) QSORT is Quick sorting algorithm. (We use 65,536 elements to be sorted.) Reader/Writer is implemented by modifying the *qtest* to evaluate performance in time-varying mem-

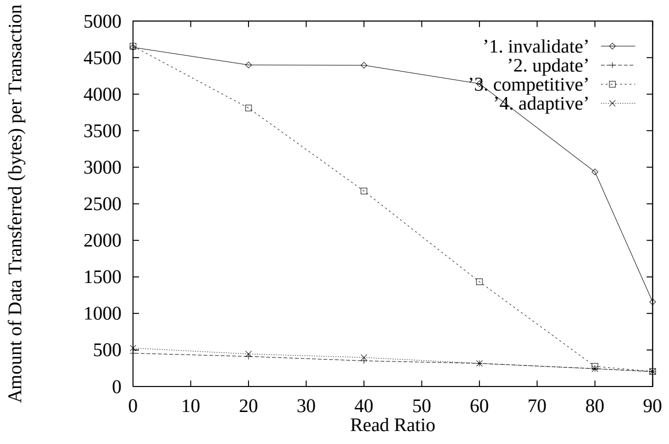

Figure 8: qtest2: Amount of Data (Bytes) Transferred per Transaction

ory access patterns. Execution time is divided into 4 stages and memory access pattern is different for each stage. A node can be either a reader or a writer for each page depending on the execution stage. The size of data for write is different for each stage. (Total 4,800 transactions are executed.)

Floyd-Warshall and SOR use barriers for synchronization. These two applications have small value of U. However, as shown in Figures 9 and 10, update protocol unexpectedly shows bad performance. Recall that we use a DSM implementation based on Quarks [15] for these experiments. In Quarks, the "Master" node initializes all shared memory and the Master node is in the copyset of all pages. Pure update protocol implementation based on Quarks performs bad due to the overhead of updating Master node for all shared memory writes. (However, this performance degradation does not happen in the original Quarks release because Quarks uses a mechanism similar to competitive update protocol.) Competitive update protocol and adaptive protocol perform well for both applications

ProdCons and QSORT use lock/unlock for a task queue. These two applications have large value of U, and invalidate protocol requires small number of messages (please refer Figures 11 and

12). However, for the amount of data, update protocol is better because the amount of data in an update message is smaller than the size of a page. Competitive protocol shows fair performance for the number of messages. However, it shows poor performance for the amount of data. Adaptive protocol shows good performance for the amount of data as well as the number of messages.

We evaluated the performance of our adaptive protocol on a synthetic Reader/Writer application (see Figure 13) where memory access patterns (read to write ratio, access period, amount of data written in each transaction, etc.) are time-varying. Results show that the adaptive protocol performs well by adapting to time-varying memory access patterns.

Experimental results show that our adaptive scheme performs well. This results suggest that our adaptive scheme can predict the optimal consistency protocol accurately when memory access patterns do not change frequently.

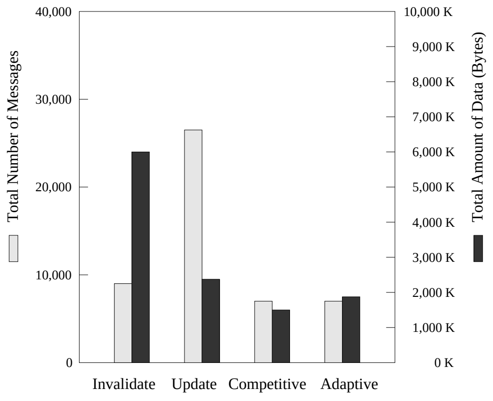

Figure 9: Cost Comparisons (Floyd-Warshall)

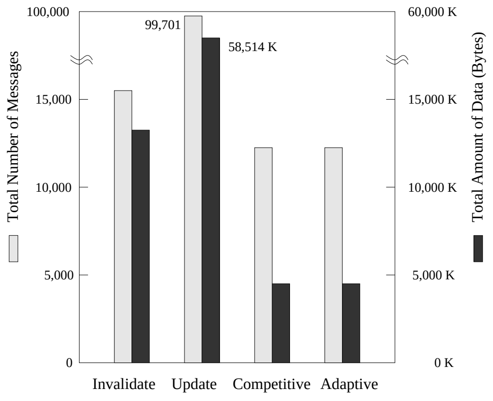

Figure 10: Cost Comparisons (SOR)

5 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 based on the "cost" metric that needs to be minimized. The cost metrics considered in this paper are *number* and *size* of messages required for executing an application using the DSM implementation. A generalization to minimize arbitrary cost metrics is also discussed briefly.

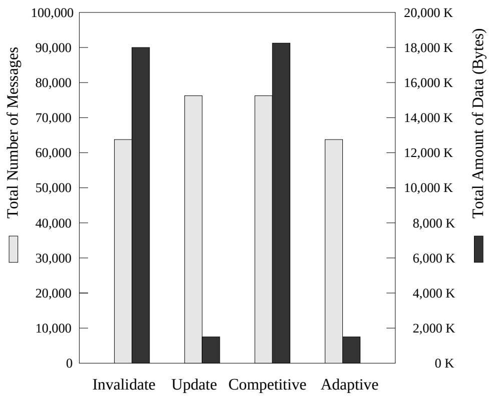

Figure 11: Cost Comparisons (ProdCons)

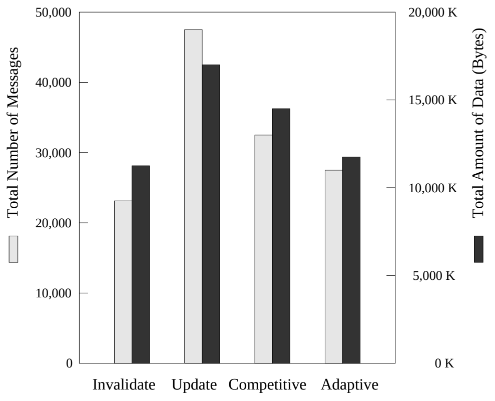

Figure 12: Cost Comparisons (QSORT)

Our adaptive approach determines, at run-time, the *cost* of each candidate consistency protocol, and uses the protocol that appears to have the smaller cost. The proposed adaptive approach is illustrated here by means of an adaptive DSM scheme that chooses either the *invalidate* or the *competitive update* protocol for each copy of a page – the choice changes with time, as the access patterns change. The paper presents experimental evaluation of the *adaptive* DSM using an implementation based on Quarks DSM [15]. Experimental results from the implementation suggest that the proposed adaptive approach can indeed reduce the *cost*.

Further work is needed to fully examine the effectiveness of the proposed approach:

• One 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 paper presented a cost-comparison based 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 $L = 0, 3, 7, \infty$. A heuristic for choosing between one of these, at run-time, needs to be developed to implement more efficient DSMs.

• The *cost* metrics considered in the paper 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 adaptive approach (based on cost-comparison) presented here can be combined with ideas developed by other researchers (e.g., [25]) to obtain further improvement in DSM performance. As

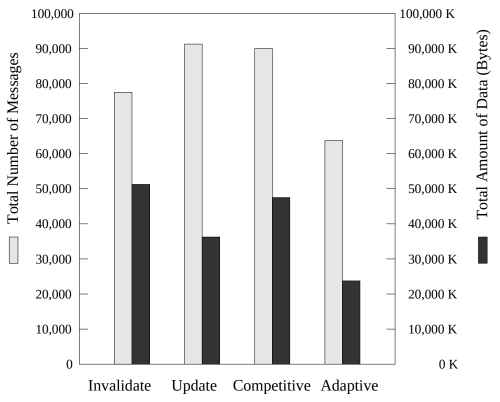

Figure 13: Cost Comparisons (Reader/Writer)

yet, we have not explored this possibility.

Acknowledgements

We thank John Carter and D. Khandekar at the University of Utah for making Quarks source code available in public domain, and Akhilesh Kumar for the Floyd-Warshall source code. We also thank the referees for their detailed comments.

References

[1] A. Karlin et al., "Competitive snoopy caching," in *Proc. of the 27'th Annual Symp. on Found. 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 Syst.*, pp. 119–122, May 1995.

[7] J. B. Carter, *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice Univ., 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. J. Eggers, "Simplicity versus accuracy in a model of cache coherency overhead," *IEEE Transactions on Computers*, vol. 40, pp. 893–906, Aug. 1991.

[13] 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.

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

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

[16] J.-H. Kim and N. H. Vaidya, "Recoverable distributed shared memory using the competitive update protocol," in *1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, pp. 152–157, 1995.

[17] J.-H. Kim and N. H. Vaidya, "Towards an adaptive distributed shared memory," Tech. Rep. 95-037, Texas A&M Univ., 1995.

[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] H. Nilson and P. Stenstrom, "An adaptive update-based cache coherence protocol for reduction of miss rate and traffic," tech. rep., Lund University, July 1994.

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

[22] 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.

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

[24] V. Phalke and B. Gopinath, "An inter-reference gap model for temporal locality in program behavior," in *1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 291–300, 1995.

[25] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural mechanisms for explicit communication in shared memory multiproccessors," in *Supercomputing '95*, Dec. 1995.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.