# PERFORMANCE AND RECOVERABILITY OF DISTRIBUTED SHARED MEMORY SYSTEMS USING COMPETITIVE UPDATE

A Dissertation

by

JAI-HOON KIM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 1997

Major Subject: Computer Science

# PERFORMANCE AND RECOVERABILITY OF DISTRIBUTED SHARED MEMORY SYSTEMS USING COMPETITIVE UPDATE

A Dissertation

by

JAI-HOON KIM

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Approved as to style and content by:

August 1997

Major Subject: Computer Science

## ABSTRACT

Performance and Recoverability of

Distributed Shared Memory Systems

Using Competitive Update. (August 1997)

Jai-Hoon Kim, B.S., Seoul National University;

M.S., Indiana University

Chair of Advisory Committee: Dr. Nitin H. Vaidya

Software distributed shared memory (DSM) systems have many advantages over message passing systems. Since DSM provides a user a simple shared memory abstraction, the user does not have to be concerned with data movement between hosts.

This dissertation presents a simple approach for implementing *adaptive* DSM on a network of workstations. The approach is illustrated with the example of an adaptive DSM based on the *invalidate* and *competitive update* protocols. The proposed scheme allows each node to independently choose (at run-time) a different protocol for each page. This adaptive scheme is then modified to also include a *migratory* protocol. In software DSM systems, the migratory protocol is not necessarily optimal for a migratory access pattern. We define some conditions under which the migratory protocol is preferred over other candidate protocols. Experimental evaluation of the *adaptive* DSM indicates that it is able to adapt to the memory access pattern of many applications.

In the *competitive* update protocol, multiple copies of each page may be maintained at different nodes. However, it is also possible for a page to exist in only one node, as some copies of the page may be invalidated. This dissertation proposes an implementation that makes the *competitive* update protocol recoverable from a single node failure, by guaranteeing that at least two copies of each page exist. The dissertation presents evaluation of the recoverable DSM using an implementation on a network of workstations. The dissertation also compares overhead of the single fault-tolerant DSM with a consistent checkpointing scheme and a two-level recovery scheme.

Finally, this dissertation presents a new cost analysis model for competitive update protocol. Input parameter for the cost analysis model proposed here is the probability density function of the number of remote updates in a segment. Using the proposed model, we compute the cost of the competitive update protocol for each update limit. This cost function is used to determine the optimal update limit for competitive update protocol. The proposed model is validated by comparing analytical results obtained using the model to experimental results.

To My Parents, Wife, and Daughters

## ACKNOWLEDGMENTS

First of all, I am greatly indebted to my advisor Professor Nitin H. Vaidya for his guidance, encouragement, and all the support necessary to finish my Ph.D. research. He educated me all about research and how to communicate my research results with enthusiasm and understanding. Without his help, I could not finish this dissertation.

I wish to thank the advisory committee members, Fabrizio Lombardi, Jennifer Welch, Gwan S. Choi for their valuable comments and suggestions. I also wish to thank Professor Gregory D. Reinhart for serving as the Graduate Council Representative.

I also wish to thank teachers and friends of my school years, and bosses and colleagues of my industrial career. Their influence has given me the background necessary for this research.

Most of all, I would like to thank my parents, parents of my wife, brother, sister, and relatives for their encouragement and support necessary to succeed in this Ph.D. study. Last but not least, I would like to thank my wife (Joo-Hyun) and daughters (Ji-Ho and Sarah) for their patience and love during this study.

# TABLE OF CONTENTS

LIST OF TABLES

# LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

# CHAPTER I

# INTRODUCTION

## A. Distributed Shared Memory

Communication between nodes in a multiprocessor system can be performed either using messages or shared memory. In contrast to message-passing, shared memory provides processes in a system with a shared address space. For distributed systems, no physically shared memory exists to support a shared memory abstraction. However, a software layer can be implemented to provide a shared memory abstraction. Shared memory implemented on loosely coupled systems is called *distributed shared memory* [60].

Distributed shared memory (DSM) systems have many advantages over message passing systems [47, 60]. Since DSM provides a user a simple shared memory abstraction, the user does not have to be concerned with data movement between hosts. Users can use the DSM as if the shared memory is available locally. Many applications programmed for a multiprocessor system with shared memory can be executed on a DSM system without significant modifications. In this dissertation, we consider DSM implementation achieved using a software layer (without adding special hardware). Such an implementation is often called *software DSM.* Software DSMs have been implemented on cluster of workstations.

Figure 1 shows a system configuration using DSM system. Each node has processor, memory, and connection to a network. Memory is divided into pages, and a page can have multiple copies in different nodes. DSM maintains memory consistency across the nodes by using a message passing mechanism. Each application process

The journal model is *IEEE Transactions on Automatic Control.*

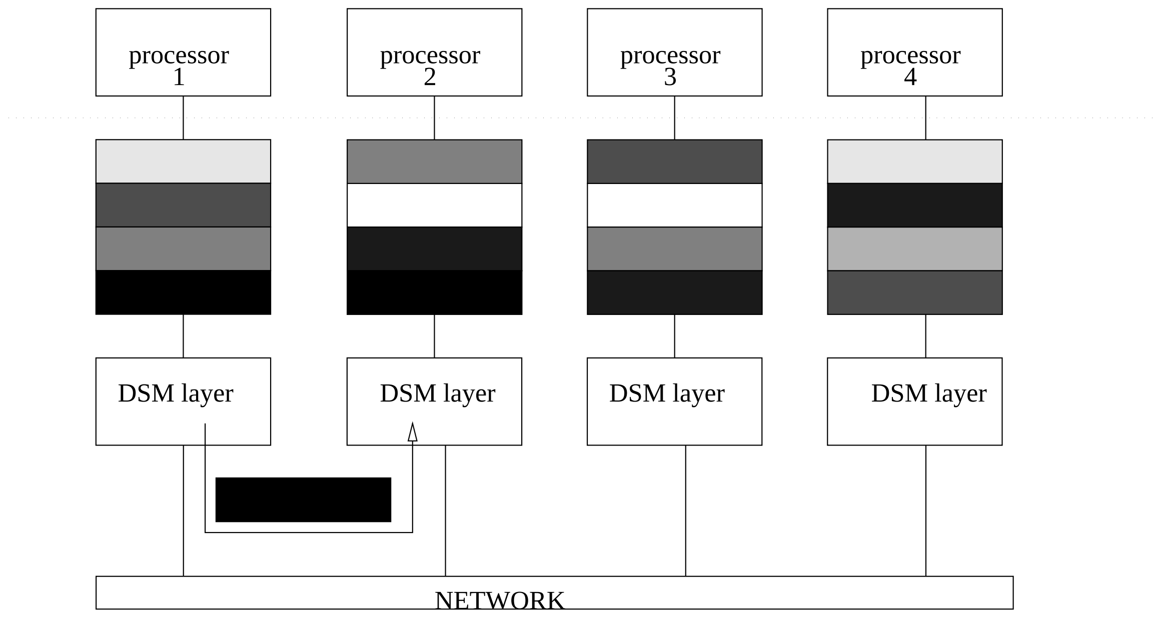


Fig. 1. Distributed Shared Memory

can transparently access the distributed shared memory in the same node as if it is local memory.

In a protocol that performs *write-update*, when a node accesses a page for the first time, a copy of the page is brought into the local memory of the node. This copy of the page is updated whenever another node modifies the page. In contrast, in protocols based on *write-invalidate*, whenever a remote node modifies a page, the local copy is invalidated.

A disadvantage of the update protocol is that, over the course of the execution, many nodes may obtain a copy of the page in their local memory. Whenever any node modifies the page, an update message must be sent to all these nodes, incurring significant overhead. Two approaches have been used to mitigate this overhead. First, a relaxed consistency model such as *release* consistency [12] is used in recent implementations. Second, some copies of a page are invalidated if they are not likely to be used in the near future (some heuristic may be used to determine which copies can be invalidated, e.g., *competitive* update protocol [23]). Now, we summarize each of these approaches.

## Release Consistency

The *release* consistency protocol is based on the observation that, in a typical program, accesses to shared variables are separated by *synchronization* operations – in release consistency [12], these operations are termed *acquire* and *release.* If an access by a process to some shared data is likely to cause a race condition, then the process first performs an *acquire* operation. When the process has completed its accesses to the shared data, it performs a *release* operation. If one process has already performed an *acquire*, another process' *acquire* will block until the first process performs a *release.* This ensures that while one process is modifying some shared data, another process will not attempt to access the data. Implementations of *release* consistency can take advantage of this observation to improve performance, as follows. Consider a process on node A that has performed an *acquire*, subsequently performed multiple writes to shared data, and is now performing a *release* operation. Because of release consistency, it is adequate if node A sends a *single* update message (to all nodes that have a copy of the modified pages) corresponding to *all* the writes performed by the process since its most recent *acquire* [12]. In implementations that use sequential consistency (instead of release consistency), it is necessary to send one update message for every *write* performed by node A. Due to release consistency, it is necessary to perform at most one update for every *release* performed by a process. This implementation of release consistency reduces the number of messages, potentially improving performance. Note that in the implementation under consideration here, the *release* operation blocks until the updates are propagated to all relevant nodes and acknowledgments are received from them.

## *Competitive* Update Protocol

The basic idea of the *competitive* update protocol [28, 23] is to update those copies of a page that are expected to be used in the near future, while selectively invalidating other copies. The *competitive* update protocol is defined using a "threshold" parameter – in this dissertation, we will refer to the threshold as "update limit" or just "limit". When using the competitive update protocol with limit $L$ ($L \geq 0$), a node A invalidates the local copy of a page P if and when the $(L+1)$-th *update* to the page by *other* nodes occurs since the previous access of page P by node A. The basic idea of the *competitive* update protocol [28, 23] is to update those copies of a page that are expected to be used in the near future, while selectively invalidating other copies. The traditional *update* protocol can be obtained by choosing $L = \infty$. The protocol obtained when $L = 0$ is similar to the traditional *invalidate* protocol.

## B. Adaptive Distributed Shared Memory

Many approaches have been proposed to implement distributed shared memory [10, 12, 30, 33, 44, 55, 60]. Most DSM implementations are based on variations of *write-invalidate* and/or *write-update* protocols. As no single protocol is optimal for all applications, researchers have proposed DSM implementations that provide a choice of multiple consistency protocols (e.g., [12]). 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. We consider 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:

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 will be quite accurate, provided that 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 [28, 17, 18, 23]. Experimental results show that our adaptive scheme performs well because memory access patterns do not change frequently in many applications.

## C. Adaptive Migratory Distributed Shared Memory

In migratory sharing, a page is accessed by a single node at any given instance. A page is modified within a critical section to maintain mutual exclusion. Every access

[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 [49].

for a page is ordered by a sequence of *acquire*, shared memory *access*, and *release*. We present an adaptive *migratory* scheme for software Distributed Shared Memory (DSM). The proposed DSM system allows each node to *independently* choose one of the following three protocols: *migratory*, *invalidate*, and *competitive update*. In software DSM systems the migratory protocol is not necessarily better than other protocols for a migratory access pattern. (For an example, if two nodes access a page of migratory sharing, then the competitive update protocol is better than the migratory protocol.) Additionally, it is not always possible to detect a migratory access pattern. We define some conditions under which the migratory protocol is to be preferred over other candidate protocols. Experimental results show that this new scheme is often able to improve performance by choosing the migratory protocol when appropriate.

## D. Single Fault-Tolerant Distributed Shared Memory Using Competitive Update

In the *competitive* update protocol, multiple copies of each page may be maintained at different nodes. However, it is also possible for a page to exist in only one node, as some copies of the page may be invalidated. We propose an implementation that makes the *competitive* update protocol recoverable from a single node failure, by guaranteeing that at least two copies of each page exist. We also present a mechanism that maintains consistency between shared data and process local state after recovery, by updating shared data and process local state atomically. The dissertation presents evaluation of the recoverable DSM using an implementation. It is shown that the overhead of making the DSM recoverable measured in terms of the number of messages and the amount of data transferred is small in many applications.

## E. Analysis of Failure Recovery Schemes

When a process rolls back and re-executes from the last checkpoint, the time required to *re-do* the lost computation is $t$ time units (excluding time units required to roll-back) when the node fails after $t$ time units from the last checkpoint. However, the *cost* (loss) occurred by re-doing the lost computation may be larger than that to execute the original computation in time critical applications (e.g., real-time systems). This dissertation analyzes how *re-do overhead* affects cost for recoverable DSM using consistent checkpoint, and analyzes optimal checkpoint interval by varying the *re-do overhead factor* $(k)$. The proposed single fault-tolerant DSM can be combined with a checkpointing scheme to recover from single and multiple-node failure. The dissertation presents an analysis of this two-level scheme as well.

## F. Cost Model for Distributed Shared Memory Using Competitive Update

We present a new "cost" analysis model for a distributed shared memory (DSM) using the competitive update protocol. The cost metric of interest here is the overhead of message passing necessary to implement DSM. This approach is based on the segment model proposed previously [39] – a segment is defined as a sequence of remote updates between two consecutive local accesses by a node. Input parameter for the cost analysis model proposed here is the probability density function of the number of remote updates in a segment. The proposed model is validated by comparing analytical results obtained using the model to experimental results. Using the proposed model, we compute the cost of the competitive update protocol for each update limit. This cost function is used to determine the optimal update limit for competitive update protocol.

## G. Dissertation Organization

In summary, this dissertation deals with issues related to performance and recoverability of a DSM using competitive update protocol. For performance improvement, we present an on-line algorithm using *competitive* update protocol, that automatically chooses the appropriate update limit (at run-time) for each shared memory page (Chapter II). This algorithm is then modified to include a *migratory* protocol as a protocol choice (Chapter III). We also present an off-line algorithm to determine the optimal update limit for competitive update protocol (Chapter VI). For recoverability, we propose a single fault-tolerant scheme by guaranteeing that at least two copies of each page exist (Chapter IV), and analytically compare the performance of the single fault-tolerant scheme to those of other recovery schemes (Chapter V).

# CHAPTER II

# ADAPTIVE DISTRIBUTED SHARED MEMORY

The performance of DSM depends on chosen consistency protocols and application behavior. This chapter presents a simple approach for implementing *adaptive* DSMs that can choose appropriate protocol at a run-time. 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*, *amount of data transfer*, and *execution time.* The proposed scheme allows each node to independently choose a different protocol for each page at run-time [37, 39].

## A. Related Work

Many schemes have been proposed to reduce overhead by adapting to memory access patterns for cache-coherent multiprocessors and DSM systems, as summarized below.

- Anderson and Karlin [3], and Raynaud et al. [54] present adaptive schemes. The scheme in [3] varies the invalidate threshold for each block by using *write-run* model [19]. Write-run is a sequence of local writes between two consecutive remote accesses. They use write-run lengths collected during the run time in order to determine the invalidation threshold for the block in the future.

  The scheme in [54] predicts *update-distance* for a block. *Update-distance* is the number of updates received between two consecutive local accesses. The "segment" model used in this dissertation is similar to *update-distance.* A directory records the update patterns observed and then uses them to selectively send updates and invalidations to processors. In our scheme, each node independently decides to update or invalidate a local copy of a page.

- Munin [11, 12] 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 (*competitive update* and *timeout*) have similar goals, they do not behave identically. 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.

- ThreadMarks [2] uses *lazy release consistency* [29] to reduce communication overhead. In *lazy release consistency*, the update message from a node $A$ is delayed until some other node $B$ performs *acquire*. Acquiring node $B$ determines the modifications it needs to receive to satisfy *release consistency*. This scheme can reduce the amount of communication, because update message is sent to the next acquiring node only (while update message is sent to all nodes that have a copy of associated page in *eager release consistency*, such as in Munin).

- Veenstra and Fowler [66] evaluate 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 [11, 12, 33], and (iii) is similar to our *adaptive* protocols which can choose the appropriate protocol at run-time. However, [66] considers *off-line* algorithms, for a bus-based system. On the other hand, this dissertation considers adaptive (on-line) algorithms that are applicable to distributed systems. Also, in [66], the chosen protocol is used for *all* copies of a cache block, whereas in our scheme, the update *limit* used for each copy of a page may be different.

- Veenstra and Fowler [67] examine 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.

- Lebeck and Wood [43] present 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, and the cache controller self-invalidates the blocks. In our scheme, each node decides invalidation of local copy.

- Optimizations for migratory sharing have also been proposed [16, 17, 46, 59]. These protocols dynamically identify migratory shared data and switch to migratory protocol in order to reduce the overhead. [16, 59] are based on invalidate protocol, and [17, 46] are based on competitive update protocol.

- Ramachandran et al. [53] and Shah et al. [57] 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). 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 [53] 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 [53] makes use of such information to determine whether a copy of the page should be updated or invalidated.

- Tempest [21, 55] 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).

- Multiple consistency protocol was proposed in [11, 12]. 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.

- 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 [48], 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). Our software DSM system (many other software DSMs also) is implemented on a workstation cluster which does not requires change of hardware or operating system.

- Yang et al. [69] presents an adaptive cache coherence protocol 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.

Our adaptive DSM is based on cost comparison using *segment* model. Let us focus on the shared memory 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. Other similar models have been proposed previously for analyzing shared memory. Eggers [19] presents a *write-run* model to predict the cache coherency overhead for the bus based multiprocessor system. The *write-run* is a sequence of local writes between two consecutive remote accesses. Anderson and Karlin [3] vary the invalidate threshold for each block by using *write-run* model. Bennett et al. [6] present a *no-synch run* model. The *no-synch run* is a sequence of accesses to a single object by any thread between two synchronization points in a particular thread. Stumm and Zhou [60] present an analysis of DSM based on many parameters such as read-write ratio, page fault ratio, and cost of sending/receiving a page.

## B. *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* [17] protocols.

The *competitive update* protocol is defined using an "update limit" or just "limit" $L$. 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 $\infty$) – 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.

## 1. Cost Analysis

The objective of our *adaptive* protocol is to minimize the "cost" metric of interest. Three cost metrics considered here are: (i) number of messages, (ii) amount of data transferred, and (iii) execution time. In this section, we evaluate the above cost metrics for the consistency protocols of interest. Our analysis assumes that the DSM uses release consistency and dynamic distributed ownership (no fixed page owner exist, which maintains information about the page) analogous to Munin [11, 12]. In dynamic distributed ownership mechanism, page owner that has information for the page changes dynamically.

### Minimizing the Number of Messages

We now consider *number of messages* as the cost metric. Let us focus on the

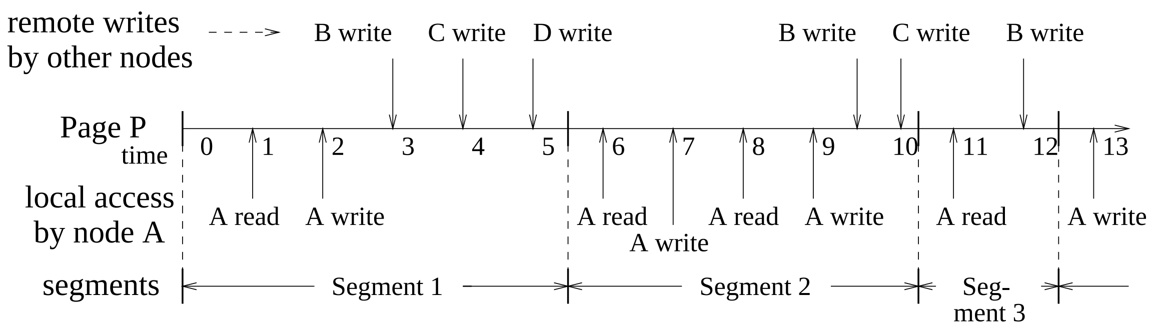


Fig. 2. Segments

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. Segments are defined from the point of view of each node. Therefore, for the same page, different nodes may observe different segments. Figure 2 illustrates *segments* observed at a node A with an example: (a) segment 1 for page $P$ at node $A$ 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.

Now we evaluate the number of messages sent during each segment for invalidate protocol (i.e., competitive update protocol with limit $L = 0$) and update protocol (i.e., 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. Therefore, the 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 (refer Figure 3).

- **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[1] – 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. Note that node $A$ sending update message does not know whether node $B$ receiving update message will invalidate or update a copy of the page in node $B$. Thus, node $A$ always sends update message instead of control message for invalidation. Ideally, once a page is invalidated, no more update messages will be sent to the node during the segment. (In reality, however,

[1]This analysis and implementation of adaptive DSM are based on another DSM, called Quarks, from University of Utah. In original Quarks, a request for the page is forwarded before reaching a node that has a copy of the page. However, we modify this scheme for the owner to maintain a copyset that is close to the "real" copyset. This scheme can reduce the chance of sending update message to a node whose local copy has been already invalidated ("false update").

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 4, $F + 4$ messages are needed in a segment. Note that the actual value of $F$ may be different in each segment.

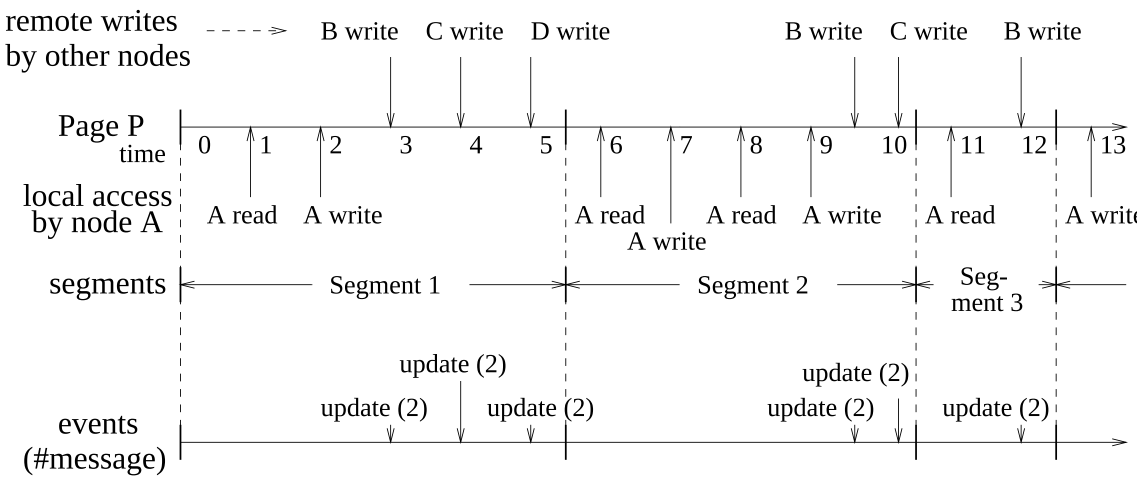


Fig. 3. Illustrations for memory access and cost (update protocol)

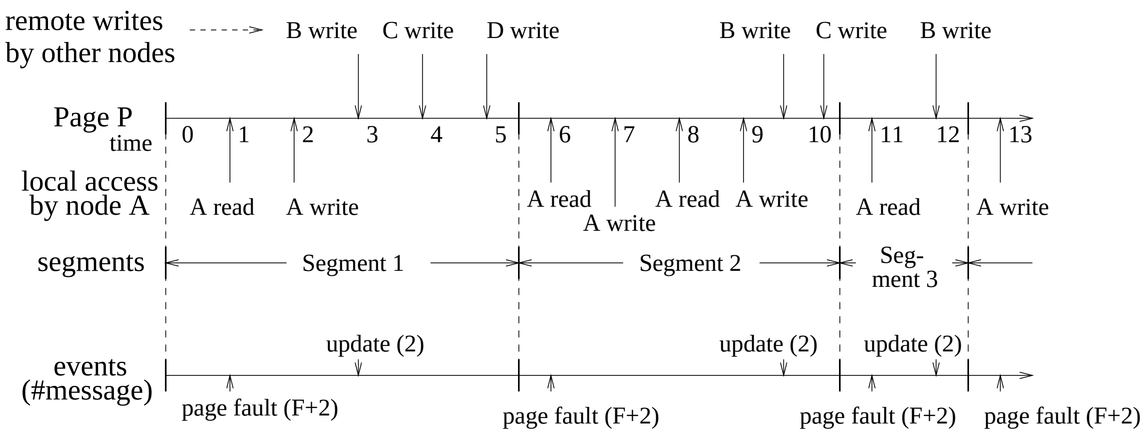


Fig. 4. Illustrations for memory access and cost (invalidate protocol)

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:

$$\begin{aligned} M_{update} &= M_{invalidate} \\ \Rightarrow 2\,U_{critical} &= F + 4 \\ \Rightarrow U_{critical} &= \frac{F+4}{2}. \end{aligned}$$

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 for better performance).

- 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 $U \geq \frac{F+4}{2}$ 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.

1. It is hard to estimate $F$ accurately (without additional message overhead) when the *limit* $L$ is non-zero. Therefore, we assume a constant value for $F$. In our experiments with up to 10 nodes, we assume $F = 4$. Clearly, $F$ must depend on the application and on the number of nodes (processors) used. Thus, $F = 4$ is not likely to be always accurate (e.g., when the number of nodes is less than 5). 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 less 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 guess turns out to be incorrect, the adaptive scheme ends up having made a wrong choice. Therefore, instead of choosing $L = \infty$ when $U < U_{critical}$, 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 $U \geq U_{critical}$ 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 in our experiments with up to 10 nodes.

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 $U \leq L$, then $M_{competitive}$ is $2\,U$, similar to $M_{update}$. Else, however, $M_{competitive} = 2\,(L+1) + (F+2) = 2\,L + M_{invalidate}$. ($2(L+1)$ 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.)

## Minimizing the 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 (assuming $\overline{F} = 4$):

$$\begin{aligned} D_{update} &= D_{invalidate} \\ \Rightarrow (\overline{p_{update}} + p_{control})\, U_{critical} &= \overline{p_{update}} + (\overline{F} + 2)\, p_{control} + p_{page} \\ \Rightarrow U_{critical} &= \frac{\overline{p_{update}} + (\overline{F} + 2)\, p_{control} + p_{page}}{\overline{p_{update}} + p_{control}} \\ \Rightarrow U_{critical} &= \frac{\overline{p_{update}} + 6\, p_{control} + p_{page}}{\overline{p_{update}} + p_{control}} \end{aligned}$$

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 equal to or 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 $U < U_{critical}$. Instead, when $U < U_{critical}$, 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.

### Minimizing the General Cost

The cost of a message of size $m$ is denoted as $c(m)$. For instance, $c(m)$ may be 1 – this means that the cost metric simply counts the number of messages. Another possibility is $c(m) = m$, which would mean that the total *amount* of data sent by messages is used as the cost metric. In general, any suitable function of $m$ may be used as the cost. For instance, $c(m) = K_1 + K_2\, m$, where $K_1$ and $K_2$ are some constants. 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 below:

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

where $\overline{c(p_{update})}$ denotes the average cost of an update message.

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

$$\begin{aligned} C_{update} &= C_{invalidate} \\ \Rightarrow (\overline{c(p_{update})} + c(p_{control}))\, U_{critical} &= \overline{c(p_{update})} + (\overline{F} + 2)\, c(p_{control}) + c(p_{page}) \\ \Rightarrow U_{critical} &= \frac{\overline{c(p_{update})} + (\overline{F} + 2)\, c(p_{control}) + c(p_{page})}{\overline{c(p_{update})} + c(p_{control})} \\ \Rightarrow U_{critical} &= \frac{\overline{c(p_{update})} + 6\, c(p_{control}) + c(p_{page})}{\overline{c(p_{update})} + c(p_{control})} \quad \text{assuming } \overline{F} = 4. \end{aligned}$$

Appropriate limit can be chosen at run-time, as in minimizing the amount of data transferred.

For adaptive DSM minimizing *execution time*, we compute critical value of $U$ ($U_{critical}$) and update limit ($L$) for *competitive* update protocol (if it is chosen) by using the similar cost analysis for minimizing the general cost. By experiment on 8-node workstation cluster connected via ethernet, the time required to request a page and receiving the page of size 4,096 bytes ($t_f$) is 30 $msec$, and the time required to send update message of size $m$ and receive response ($t_u(m)$) is approximately $t_u(m) = C_1 + C_2 \frac{m}{p_{page}}$, where $C_1 = 3.8$, $C_2 = 8$, and $p_{page} = 4,096$ (size of message for page sending) Total cost, $T$, is computed below:

- $T_{update} = t_u(\overline{p_{update}})\, U$
- $T_{invalidate} = t_u(\overline{p_{update}}) + t_f$

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

$$\begin{aligned}
T_{update} &= T_{invalidate} \\
\Rightarrow t_u(\overline{p_{update}})\, U_{critical} &= t_u(\overline{p_{update}}) + t_f \\
\Rightarrow U_{critical} &= \frac{t_u(\overline{p_{update}}) + t_f}{t_u(\overline{p_{update}})} \\
\Rightarrow U_{critical} &= 1 + \frac{t_f}{t_u(\overline{p_{update}})} \\
\Rightarrow U_{critical} &= 1 + \frac{t_f}{C_1 + C_2\,\frac{\overline{p_{update}}}{p_{page}}} \\
\Rightarrow U_{critical} &= 1 + \frac{30}{3.8 + 8\,\frac{\overline{p_{update}}}{4{,}096}}.
\end{aligned}$$

## 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 5 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 $i + 1$.

Each node independently 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.

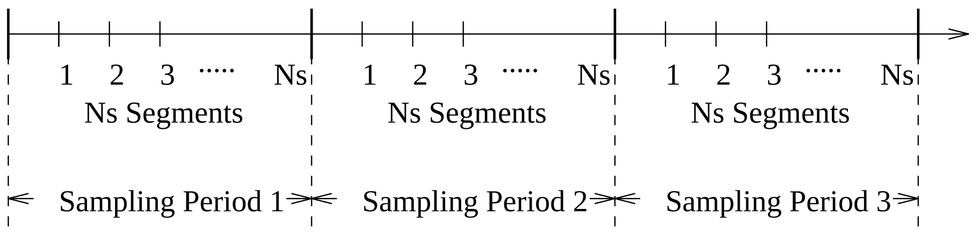


Fig. 5. Segments and Sampling Periods

- *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.

- *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. New segment start at the first local access following updates by other nodes. There are two cases at that time: (1) if node $A$ does not have a copy of page $P$, page fault occurs, (2) if node $A$ has a copy of page $P$, page fault occurs due to protected access permission (because the page is protected to detect the first local access following updates by other nodes). One of the following procedures is performed in each case:

- If node $A$ does not have a copy of page $P$: 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.

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

  Also, *dynamic_version* and *dynamic_xdata* in the local page table entry are compared to *version* and *xdata*, respectively, received with the page.

- If node $A$ has a copy of page $P$: In this case, access protection causes a page fault. In the page fault handler, *dynamic_version* and *dynamic_xdata* are compared to *version* and *xdata*, respectively, in the local page table entry.

The comparison in the above step is followed by the following procedures in both cases. 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. That is, if $d > 0$, then:

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

At this point, a new segment begins. Therefore, the *dynamic_version* is set equal to *version* and *dynamic_xdata* is set equal 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, $U$ 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 $U \geq U_{critical}$, invalidate protocol ($L = 0$) is selected; else, competitive update protocol with appropriate limit is selected (as described in subsection 1). The chosen $L$ is used for page P at node A during the next sampling period. Due to the distributed nature of the protocol, and possible differences in access patterns of different nodes, different nodes may simultaneously use different limits for the same page.

**Correctness and Cost for Protocol Switch**

Our scheme is designed for software DSM using release consistency (such as [11, 12]), and each node independently chooses appropriate protocol by dynamically deciding whether to invalidate or update local copy of a page. This does not cause any consistency problem. The change in the choice of limit only determines when a copy of a page is invalidated. Thus, the adaptive protocol is much like the competitive update protocol, but the decision-rule for page invalidation may change over time.

No extra messages are required for the adaptive protocol (or protocol switch) because each node independently estimates the average number of updates by other nodes in one segment ($U$) and the average amount of data for update message ($\overline{p_{update}}$) to select an appropriate protocol, without sending additional messages. To keep track of statistics, some message sizes may be larger by a few bytes.

## C. 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) [10, 33]. This section presents the experimental results.

We evaluated the adaptive scheme using synthetic applications (*qtest*, *ProdCons*, and *Reader/Writer*) as well as other applications (*Floyd-Warshall*, *SOR*, *Isort*, *Matmult*, and *Jacobi*). *qtest* is a simple shared memory application based on a program available with the Quarks release [33]: 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. *Floyd-Warshall*, *Isort*, and *Jacobi* applications used in the experiments were written at Texas A&M University. *SOR* and *Matmult* are available with the Quarks release [33]. *ProdCons* and *Reader/Writer* are based on *qtest*. Sampling period ($N_s$) is chosen to be 2 for all applications. We use *Limit* $L = 3$ for a competitive update protocol in all experiments.

### 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)
              shmem[n]++;  /* increment shared memory location */
            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 6 and 7 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 6 is plotted using the heuristic for minimizing the number of messages. The curve in Figure 7 is plotted using the heuristic for minimizing the amount of data transferred. (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.) Costs required for *acquire* are included in the experimental results. (We did not consider the costs required for *acquire* in our cost-comparison analysis, as the cost required for *acquire* is independent of the protocol used.)

In Figure 6, 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

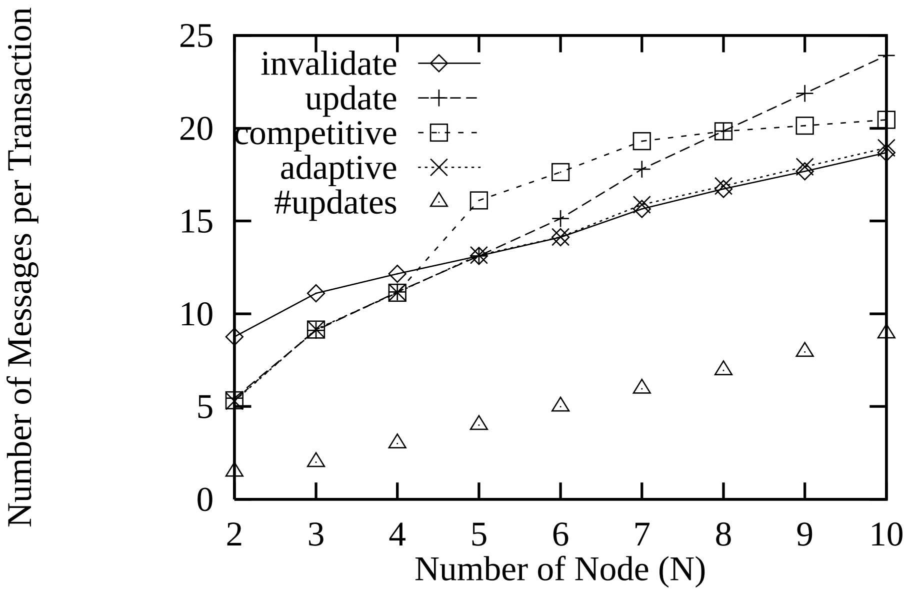


Fig. 6. qtest1: Average Number of Updates (U) and Messages per Transaction

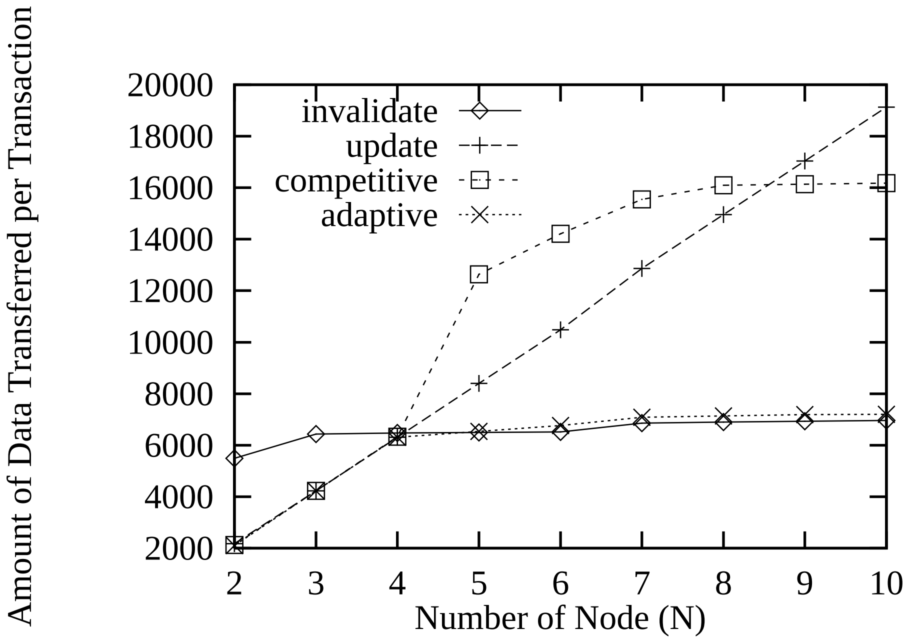


Fig. 7. qtest1: Amount of Data (Bytes) Transferred per Transaction

of the approximate estimate of $U_{critical}$ used in our analysis, the *adaptive* protocol performs well. 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 $N$). We assume $F = 4$ in our cost-comparison analysis. This assumption is incorrect for small $N$ ($N < 5$). But, near the critical value $U_{critical}$, it is roughly correct. Thus, our adaptive algorithm can choose an appropriate protocol in spite of the fixed value of $F$ in our experiments with up to 10 nodes. 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.

Figure 7 shows the comparison of the amount of data transferred per transaction. Since *qtest1* application modifies large amount of data ($NSIZE = 2,048$ 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.

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)
                   /* 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 8 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 9 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 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.

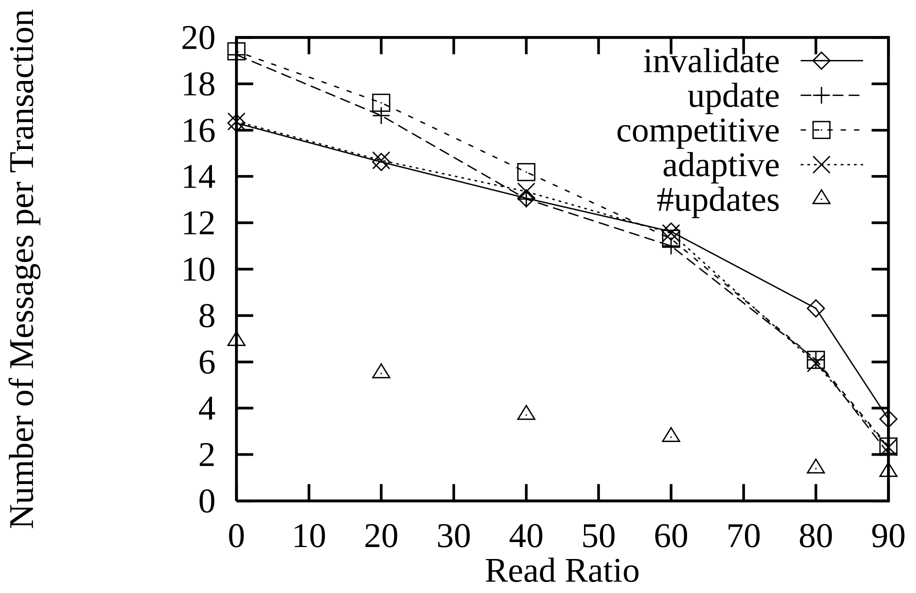


Fig. 8. qtest2: Average Number of Updates (U) and Messages per Transaction

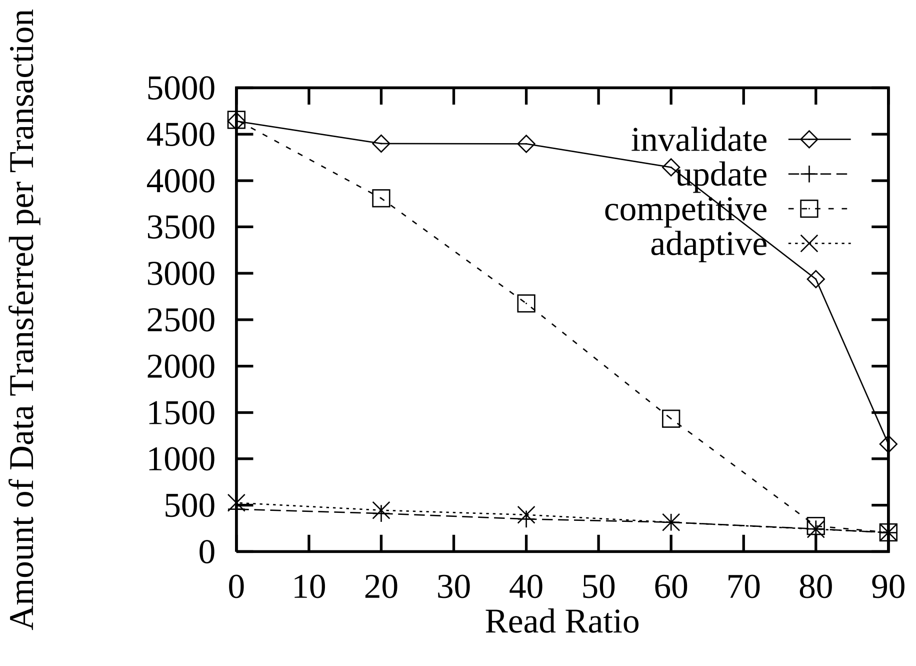


Fig. 9. qtest2: Amount of Data (Bytes) Transferred per Transaction

**Results for Other Applications**

We now evaluate our adaptive scheme by executing seven additional applications (Floyd-Warshall, SOR, ProdCons, Isort, Reader/Writer, Matmult, and Jacobi) 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 $512 \times 512$ 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*.) Isort is Integer Sorting algorithm. (We use 3,200 keys of 100 range.) Reader/Writer is implemented by modifying the *qtest* to evaluate performance in time-varying memory 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 1,920 transactions are executed.) Matmult is a matrix multiplication program which compute $A^n$. (We compute $A^{10}$, where $A$ is a $128 \times 128$ matrix.) Jacobi is a linear system solver by using iteration method. (We solve a linear system of size 128.)

We execute at least 10 times for each application and for each protocol. Table I shows experimental results: average number of messages (*Messages*), amount of data transferred (*Data (KBytes)*), execution time (*Time (seconds)*), and standard deviations of these values (*S.D.*).

Floyd-Warshall, SOR, Matmult, and Jacobi use barriers for synchronization. Floyd-Warshall and SOR have small value of $U$. However, as shown in Figures 10 and 11, update protocol unexpectedly shows bad performance (except for the amount of data transferred for Floyd-Warshall). Recall that we use a DSM implementation

Table I. Performance Comparison (I) (other applications)

| *Application* | *Messages* | (S.D.) | *Data (KB)* | (S.D.) | *Time (sec.)* | (S.D.) |
|---|---|---|---|---|---|---|
| **Floyd-Warshall** | | | | | | |
| Invalidate | 9676 | (1143) | 4674 | (65) | 25.4 | (4.29) |
| Update | 27873 | (340) | 2392 | (2) | 26.5 | (3.39) |
| Competitive | 8633 | (605) | 1562 | (2) | 26.5 | (3.39) |
| Adaptive | 8360 | (53) | 1910 | (12.8) | 14.6 | (1.66) |
| **SOR** | | | | | | |
| Invalidate | 16436 | (994) | 12204 | (87) | 46.0 | (8.92) |
| Update | 101172 | (715) | 58518 | (3) | 237.7 | (5.37) |
| Competitive | 13753 | (679) | 4588 | (3) | 30.6 | (2.45) |
| Adaptive | 13877 | (746) | 4589 | (3) | 32.2 | (2.12) |
| **ProdCons** | | | | | | |
| Invalidate | 65428 | (2221) | 17790 | (222) | 155.7 | (32.15) |
| Update | 76636 | (517) | 1387 | (8) | 112.9 | (9.96) |
| Competitive | 76527 | (739) | 18124 | (154) | 148.5 | (14.94) |
| Adaptive | 64730 | (1224) | 1451 | (5) | 114.0 | (8.17) |
| **Isort** | | | | | | |
| Invalidate | 51979 | (1551) | 14294 | (176) | 124.5 | (22.67) |
| Update | 61449 | (331) | 993 | (6) | 97.6 | (6.48) |
| Competitive | 60753 | (1240) | 14282 | (300) | 124.3 | (11.65) |
| Adaptive | 51824 | (224) | 1048 | (9) | 94.8 | (7.72) |
| **Reader/Writer** | | | | | | |
| Invalidate | 74443 | (804) | 41255 | (470) | 244.4 | (38.78) |
| Update | 91652 | (1110) | 42694 | (5) | 225.5 | (6.58) |
| Competitive | 90458 | (504) | 57702 | (60) | 253.6 | (15.73) |
| Adaptive | 68587 | (1168) | 22639 | (294) | 227.1 | (16.05) |
| **Matmult** | | | | | | |
| Invalidate | 2388 | (1345) | 864 | (5) | 22.7 | (6.39) |
| Update | 3256 | (483) | 3458 | (2) | 26.4 | (1.59) |
| Competitive | 2156 | (287) | 1381 | (1) | 20.5 | (1.25) |
| Adaptive | 2204 | (425) | 1382 | (2) | 21.2 | (1.62) |
| **Jacobi** | | | | | | |
| Invalidate | 2462 | (629) | 646 | (19) | 29.9 | (4.49) |
| Update | 2237 | (202) | 327 | (1) | 1.0 | (0.08) |
| Competitive | 2427 | (618) | 380 | (15) | 4.6 | (1.49) |
| Adaptive | 2349 | (489) | 343 | (6) | 3.3 | (0.74) |

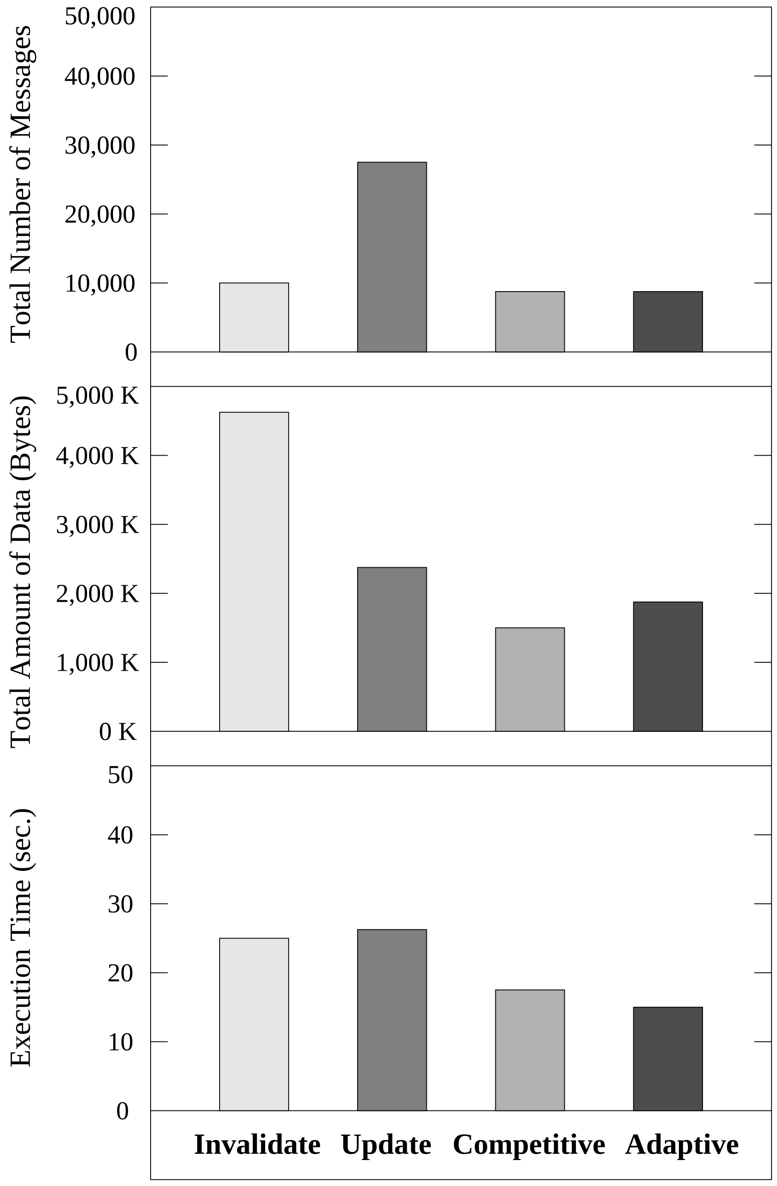


Fig. 10. Floyd-Warshall

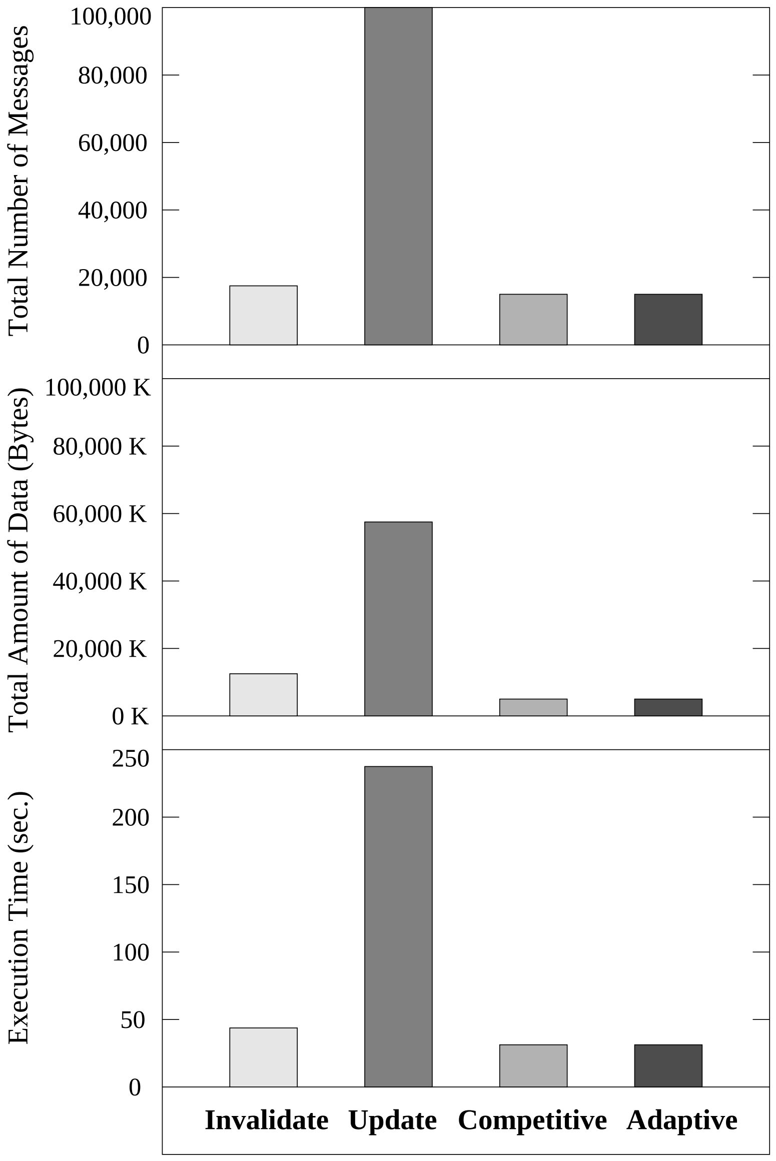

Total Number of Messages
100,000
80,000
60,000
40,000
20,000
0
Total Amount of Data (Bytes)
100,000 K
80,000 K
60,000 K
40,000 K
20,000 K
0 K
Execution Time (sec.)
250
200
150
100
50
0
Invalidate
Update
Competitive
Adaptive

Fig. 11. SOR

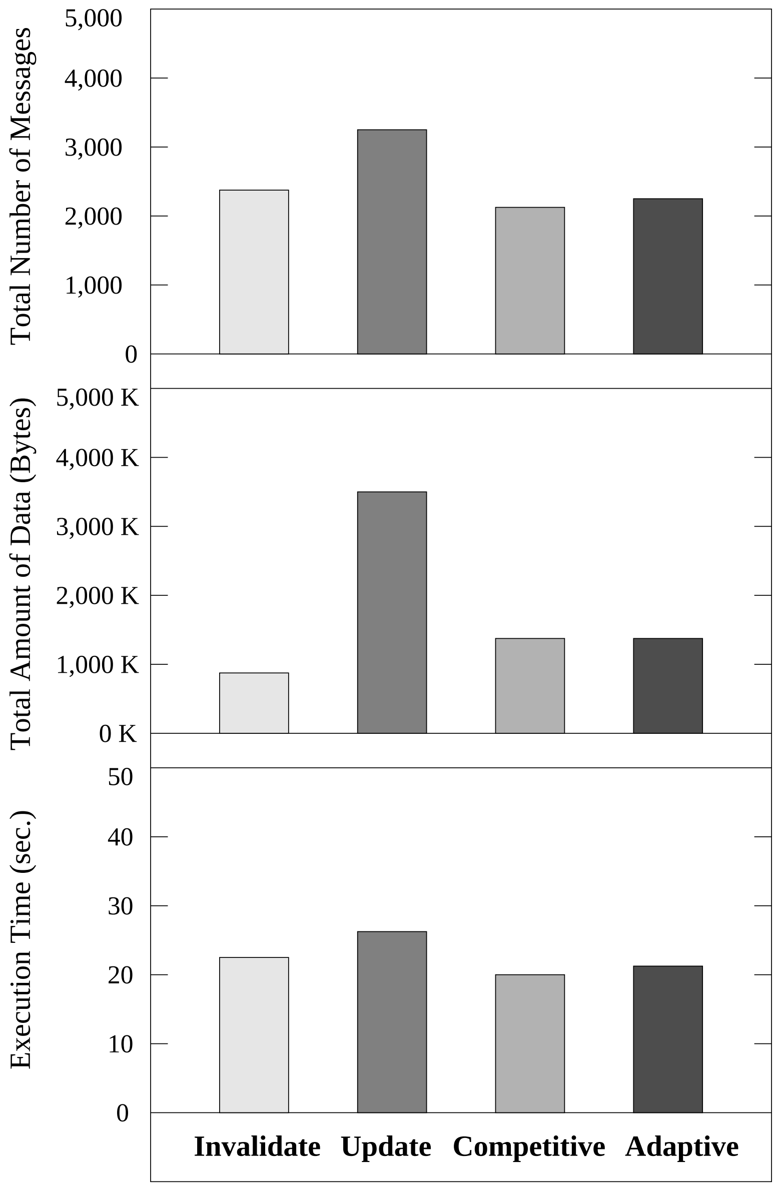


Fig. 12. Matmult

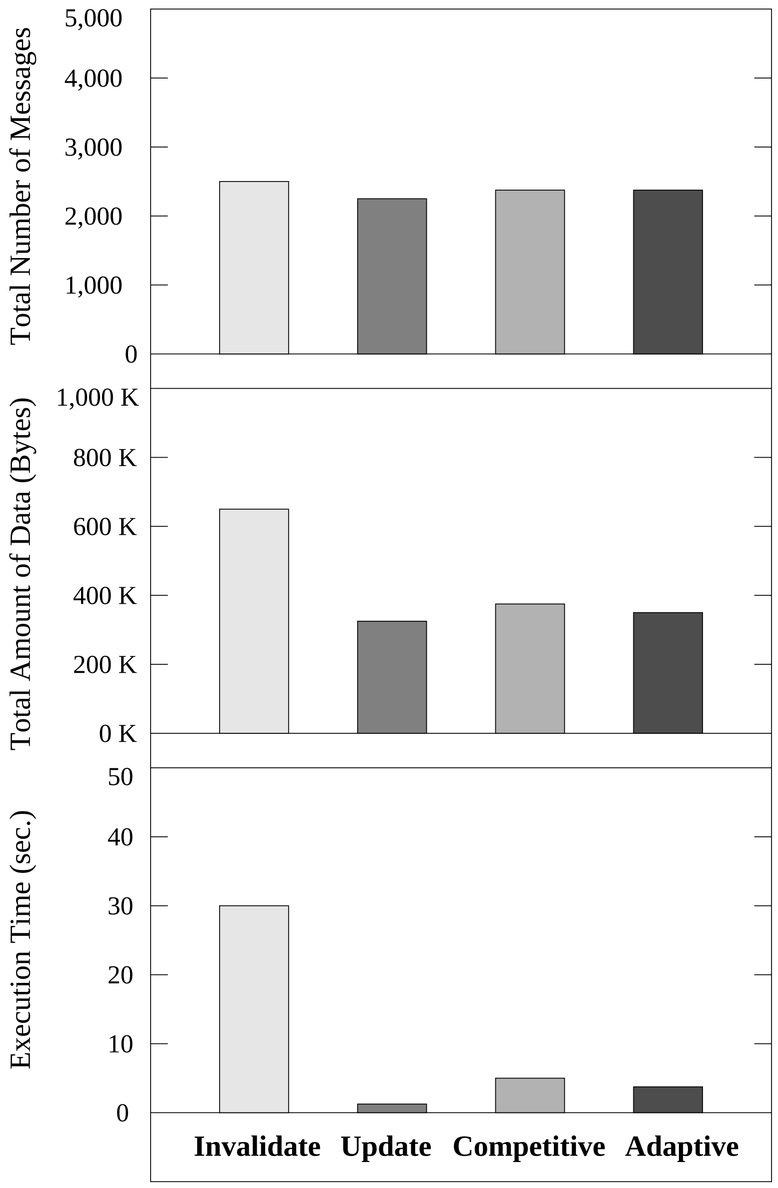


Fig. 13. Jacobi

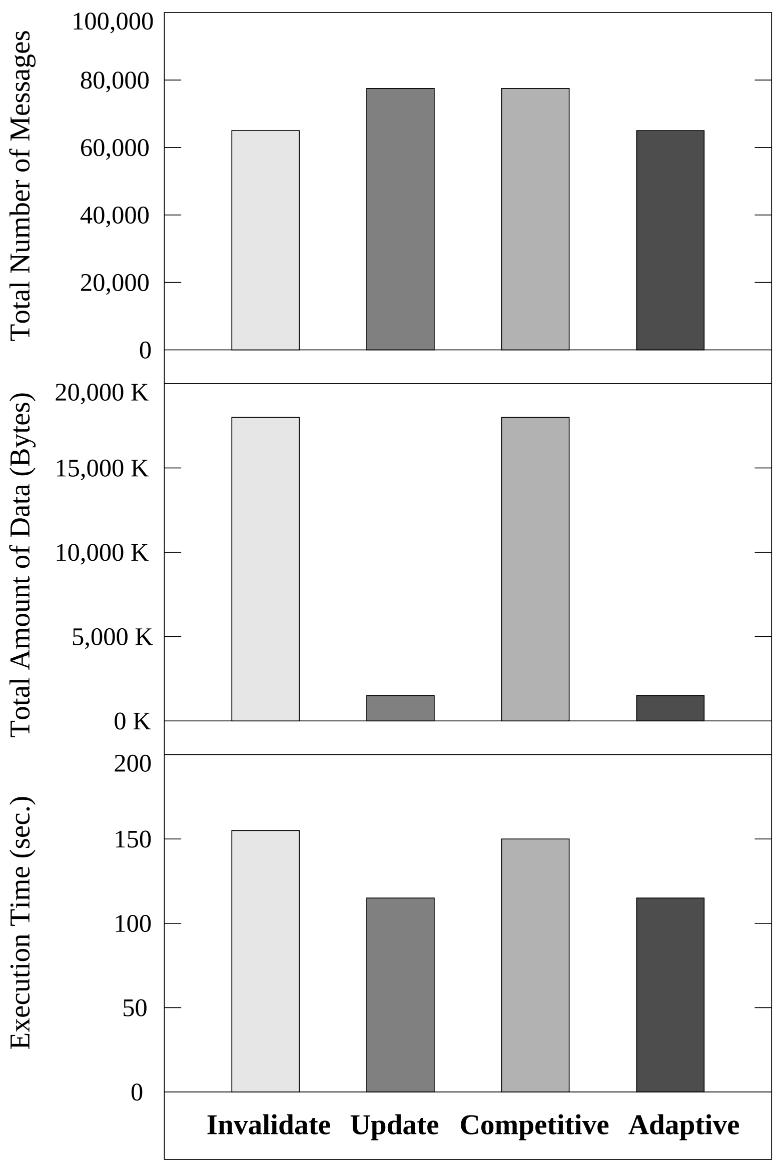
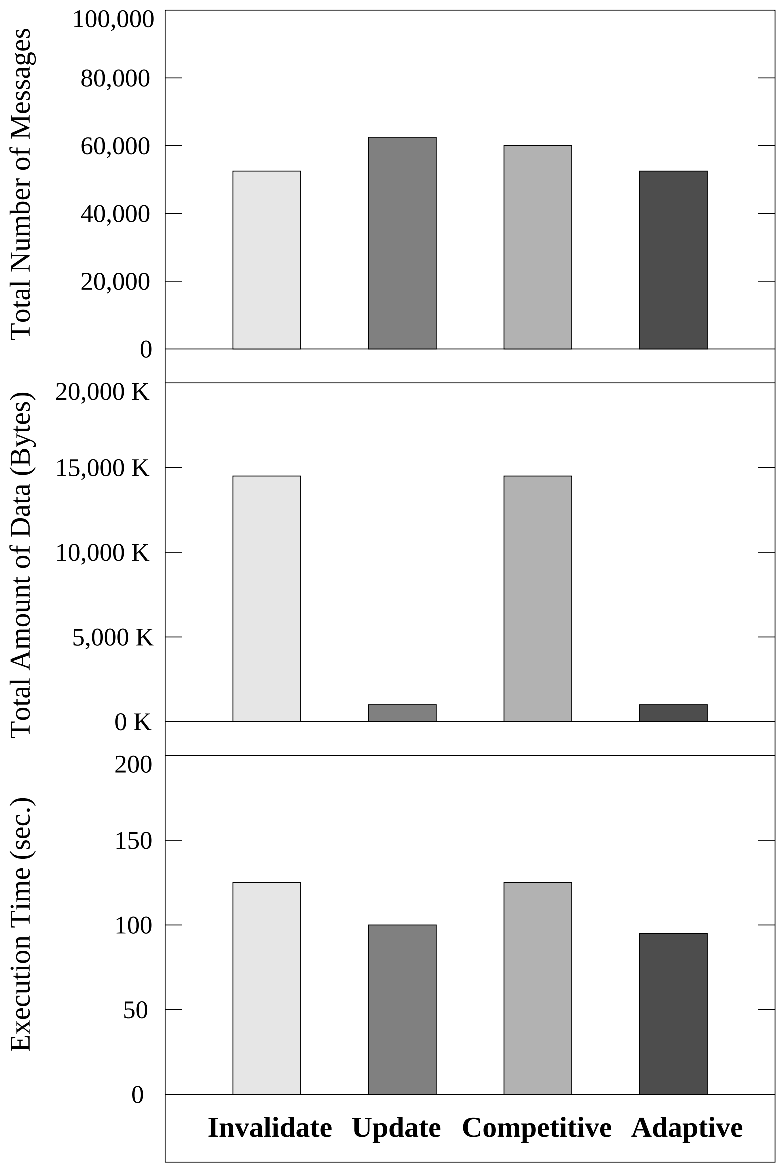


Fig. 14. ProdCons



Total Number of Messages
100,000
80,000
60,000
40,000
20,000
0
Total Amount of Data (Bytes)
20,000 K
15,000 K
10,000 K
5,000 K
0 K
Execution Time (sec.)
200
150
100
50
0
Invalidate
Update
Competitive
Adaptive

Fig. 15. Isort

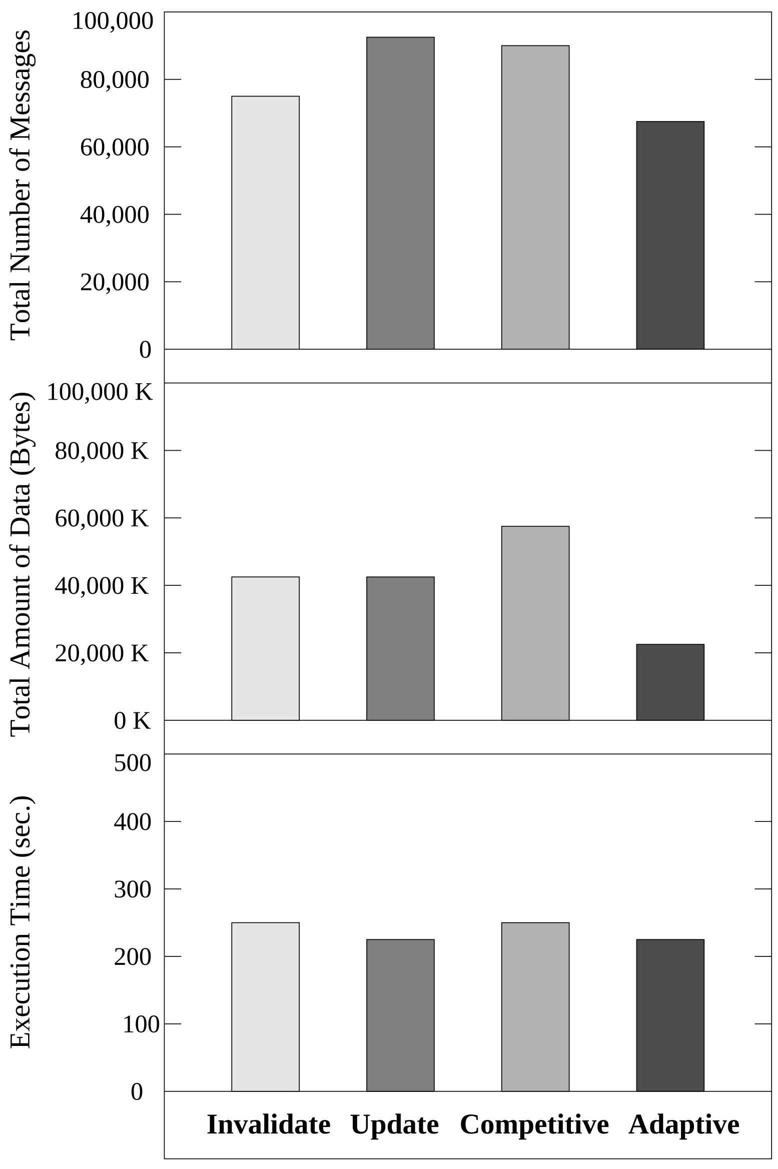


Fig. 16. Reader/Writer

based on Quarks [33] 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 four applications (except for competitive update protocol in Jacobi) as shown in Figures 10 through 13.

ProdCons uses lock/unlock for a task queue, Isort uses lock/unlock for ranking, and Read/Writer uses lock/unlock for exclusive object access. These four applications have large value of $U$, and invalidate protocol requires small number of messages (please refer Figures 14, 15, 16). However, for the amount of data in ProdCons and Isort, update protocol is better because the amount of data in an update message is much smaller than the size of a page. Competitive protocol does not show good performance for the amount of data as well as the number of messages. Adaptive protocol shows good performance for the amount of data as well as the number of messages as shown in Figures 14 through 16.

We evaluated the performance of our adaptive protocol on a synthetic Reader/Writer application (see Figure 16) where memory access patterns (read to write ratio, access period, amount of data written in each transaction, etc.) are *time-varying*. Each node access pages 1 through 4 in different patterns. As an example, nodes 1 and 2 repeatedly execute the following main loop in Reader/Writer application.

```
# Reader/Writer (at nodes 1 and 2):

    size of data = 4 bytes

    Stage 1:  Do (i = 2,3,4) {
                read page i
                write page 1
                write page 5 and 6
              }

    Stage 2:  Do (i = 3,4,1) {
                read page i
                write page 2
                write page 5 and 6
              }


    size of data = 2,048 bytes

    Stage 3:  Do (i = 4,1,2) {
                read page i
                write page 3
                write page 5 and 6
              }

    Stage 4:  Do (i = 1,2,3) {
                read page i
                write page 4
                write page 5 and 6
              }
```

Results show that the adaptive protocol performs well by adapting to time-varying memory access patterns. Observe that adaptive protocol performs *better* than any of the other protocols for Reader/Writer application. The reason is that no single protocol is optimal for all stages as the access patterns change for each stage in Reader/Writer application.

From above measurements, observe that, in most cases, the adaptive algorithm achieves performance comparable with the optimal protocol (among invalidate, update, and competitive update protocols). This suggests that the adaptive scheme is able to predict the optimal protocol accurately.

## D. Summary

This chapter presents an *adaptive* scheme for DSM that can adapt to time-varying pattern of accesses to the shared memory. The adaptive DSM automatically choose the appropriate consistency protocol (without any input from the programmer). 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 dissertation are *number* and *size* of messages, and *time* required for executing an application using the DSM implementation. A generalization to minimize arbitrary cost metrics is also possible.

Experimental evaluation of the *adaptive* DSM using an implementation based on Quarks DSM [33] is presented. Experimental results from the implementation suggest that the proposed adaptive approach can indeed reduce the *cost*.

# CHAPTER III

# ADAPTIVE MIGRATORY DISTRIBUTED SHARED MEMORY

With migratory sharing, a node that has a page fault, soon writes to the page and sends an update to other copies of the page. When using invalidate protocol, the remote copies of the page will be invalidated on this update. A message for sending a copy of a page to a remote node, on which a page fault occurs, is directly followed by an update request from the remote node. This chapter presents an adaptive *migratory* scheme that tries to detect the migratory sharing and to eliminate the overhead of receiving an update message and sending a negative acknowledgement [38, 40]. This scheme os obtained by adding migratory protocol as another choice in the scheme in Chapter II. The reason for evaluating the two schemes separately is that, sometimes addition of a protocol choice may reduce performance. By evaluating the adaptive scheme with and without migratory protocol separately, it is possible to determine how effective the additional protocol choice is.

## A. Related Work

Other researchers have also proposed adaptive schemes for migratory sharing. Our adaptive migratory scheme is implemented in a software DSM and is different from others as follows:

- Design domain: The schemes in [16, 46, 59, 43] are based on bus-based or directory-based cache coherent multiprocessors. In a bus-based multiprocessor, requests (for read miss, write miss and invalidate) can be detected by all nodes via the bus. In a directory-based cache coherent multiprocessor, a home node maintains directory entries. In these architectures, global state (number of cached copies, last invalidator of a block) can be known by some or all nodes. However, these schemes can not be used directly in software DSM such as Munin [12] or Quarks [33] where no node may have global knowledge. Our scheme can be incorporated into a software DSM in which memory coherency is maintained in a distributed manner. Each node tries to determine the best protocol using locally available information.

Schemes in [15, 42] are proposed for dynamic page placement in NUMA architecture. Their dynamic page placement policy can not be applied to DSM due to architectural differences. On page fault, in NUMA architecture, a local node can access remote memory without page allocation in local memory. However, in most DSMs (e.g., Munin [12]), remote memory access is not allowed.

- Protocol Switch: [16, 59, 46] select a migratory protocol whenever memory access pattern is migratory sharing.

  Our adaptive scheme requires each node to periodically estimate the "cost" of using each candidate protocol for each page in its local memory; the protocol with lowest estimated cost is used. Therefore, the proposed scheme uses the migratory protocol only when it is deemed optimal. If another protocol is deemed optimal, even if the access pattern is migratory, the other protocol is chosen.

  In a DSM, it is possible that migratory protocol may not be optimal for migratory access pattern. Choosing migratory pattern may save the cost of performing some updates. However, migratory protocol may add the cost of processing a page fault on a page that has been migrated to another node (this cost may be avoided by using competitive update protocol). In a software DSM, a page-request may have to be forwarded several times before it is served (a page-request is issued on a page fault). The proposed scheme chooses the migratory protocol only when its cost is expected to be lower than the other candidate protocols.

- Protocol Choices: [16, 59, 43] allow invalidate and migratory protocols, and [46] allows competitive update and migratory protocols. Our scheme allows invalidate, competitive update and migratory protocols.
- Hybrid Protocol: In [16, 46, 59], all copies of a block enter migratory mode or exit from migratory mode. In our scheme, each node independently chooses the appropriate protocol. Therefore, some nodes can use a migratory protocol while the other nodes use another protocol (*invalidate* or *competitive update* protocol) for the *same* page. What this means is that some nodes may invalidate their local copy of a page when servicing a page-request for that page (migratory protocol), while some other nodes may not invalidate the page when servicing a page-request (competitive update or invalidate protocols).

Table II summarizes the above discussion.

## B. Adaptive Migratory Scheme

The adaptive scheme presented in Chapter II is now modified to include the migratory protocol as one of the protocol choices. Doing this requires two new features:

- A heuristic to determine when the migratory protocol is likely to be optimal.
- A mechanism that will allow a node to detect the migratory access pattern.

The proposed scheme chooses the migratory protocol if: (i) the access pattern seems to be migratory, and (ii) assuming that the access pattern is migratory, the cost of migratory protocol is estimated to be the least.

Table II. Adaptive Protocols

| Scheme | Design domain | Protocols (Schemes) | Features |
|---|---|---|---|
| [16] | Dir or Bus | Inv + Mig | |
| [59] | Dir | Inv + Mig | |
| [46] | Dir | Comp + Mig | |
| [43] | Dir | Inv + Self-Inv | |
| [15, 42] | MM-NUMA | Remote + Replicate + Mig | CC |
| [39] | SDSM | Inv + Comp | CC |
| Proposed | SDSM | Inv + Comp + Mig | CC + TD |

- Bus = bus-based cache coherence multiprocessor
- Dir = directory-based cache coherence multiprocessor
- MM-NUMA = memory management system for NUMA multiprocessor
- SDSM = software Distributed Shared Memory
- Inv = invalidate protocol
- Mig = migratory protocol (scheme)
- Remote = remote memory access
- Replicate = page replication
- Comp = competitive update protocol
- CC = cost comparison
- TD = totally distributed

We now present cost analysis for the three protocols. The cost analysis of the *migratory* protocol presented below is valid only if the access pattern is *migratory sharing.*

## Cost Analysis (Number of Messages)

We first consider *number of messages* as the cost metric. As before, $\overline{F}$ denotes the average number of times a page-request is forwarded.

When using the migratory protocol, at the beginning of each segment, a page fault occurs. If memory access pattern is migratory sharing, the number of messages required for a migratory protocol in one segment ($M_{migratory}$) is computed as:

$$M_{migratory} = \overline{F} + 2$$

$\overline{F}$ messages above are required for forwarding page-request. In addition, one message is required to receive the page, and one message to acknowledge receipt of the page. Invalidate, update, and *competitive* update protocols were analyzed in Chapter II.

Figure 17 shows an analytical comparison of required number of messages for one *segment* as a function of $U$, assuming *migratory* memory access pattern (assuming $\overline{F} = 4$). For *competitive* update protocol in Figure 17, we have $L = 3$. Note that only the cost for memory access (read, write and page fault) is considered (cost for synchronization, *acquire*, is not considered). Under migratory sharing, the migratory protocol requires two messages less (per segment) as compared to the invalidate protocol (by eliminating an update message and corresponding acknowledgment). This figure suggests that for the migratory access pattern the migratory protocol is the best choice **if** $U \geq 4$. However, even with migratory memory access pattern, update and competitive update protocols are better choices if $U \leq 2$. When $U = 3$, and the access pattern is migratory, then migratory, update, and competitive update

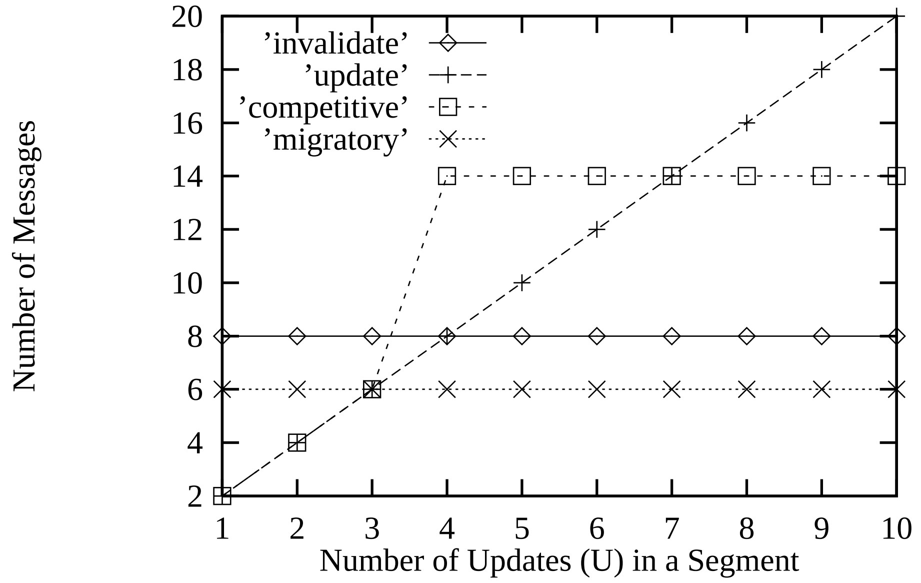


Fig. 17. Number of Messages per Segment (for Migratory Memory Access Pattern)

protocols require comparable number of messages.

**Cost Analysis (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. If memory access pattern is migratory, the amount of data transferred in one segment ($D_{migratory}$), when using the migratory protocol, is computed as follows:

$$D_{migratory} = (\overline{F} + 1)\, p_{control} + p_{page},$$

where $p_{control}$ is the size of a control message (page request, acknowledgment, etc.), and $p_{page}$ is the size of a message that is required to send a page from one node to another.

**Cost Analysis (General Cost Functions)**

In general, the cost may be an arbitrary function of the message size. Let the cost of sending or receiving a message of size $m$ be $c(m)$. If memory access pattern is migratory sharing, the cost required for a migratory protocol in one segment ($C_{migratory}$) is computed as:

$$C_{migratory} = (\overline{F} + 1)\, c(p_{control}) + c(p_{page}).$$

We also compute the execution time in Section II. By experiment on 8-node workstation cluster connected via ethernet, the time required to request and receive a page of size $4,096$ bytes ($t_f$) is 30 $msec$ on average. Thus, the time required for a migratory protocol in one segment ($T_{migratory}$) is computed as:

$$T_{migratory} = t_f = 30\ msec.$$

The implementation of adaptive migratory protocol evaluated in this dissertation chooses the appropriate limit to minimize the number of messages, the amount of data transferred, or execution time. Any one of the three may be minimized at any time, not all of these. Note that $U_{critical}$ for choosing appropriate limit is different for each cost metric.

## 1. Implementation

Based on the above analysis, we add two features to the proposed adaptive scheme:

1. *Select migratory memory access pattern when appropriate*: Node $A$ collects statistics over a *sampling period* to determine if the access pattern is migra-

tory, and whether the migratory protocol is optimal.

2. *Self-Invalidation*: Node $A$ performs *self-invalidation* of a local copy of page $P$ when sending page $P$ to any other node, if node $A$ selects migratory memory access pattern for page $P$ (as described in item 1 above).

As discussed in Chapter II, it is possible for each node to estimate $U$ and $\overline{p_{update}}$ independently, without sending additional messages. Note that the value of $U$ determined by each node (for the same page) may be different, as segments observed by each node are different. Therefore, each node needs to be able to estimate $U$ independently.

The specific heuristic that we used for selecting migratory protocol requires that the two conditions below must be true during a given sampling interval. If the conditions hold, then the migratory protocol is used in the next sampling interval. Consider page P and node A. Node A may use migratory protocol for page P during the next sampling period, if:

1. During each segment in the current sampling period, node A responds to page-request for page P. Also, after node A sends page P to another node, node A does not access page P again before a remote update to page P occurs. (These two conditions together are used to conclude that the access pattern is migratory.)

2. Number of remote updates to page P in each segment is at least $U_{critical}$. (This condition is used to determine if migratory protocol is likely to incur least cost.)

The resulting *adaptive migratory scheme* (referred as `adapt+` or `adaptive+`) can be summarized as follows:

1. If estimated $U < U_{critical}$, choose competitive update protocol with limit $L = U_{critical} - 1$.

2. Else, choose a migratory protocol if the two conditions stated above for selecting migratory protocol are satisfied.

3. Else, choose invalidate protocol.

As noted earlier, because $U_{critical}$ is different for minimizing number of messages, amount of data, and execution time, the three cost metrics cannot be minimized simultaneously.

Figure 18 shows examples of how the above procedure is used to choose appropriate protocol, according to the memory access patterns for page $P$ observed at node $A$. Assume that the sampling period consists of 2 segments (i.e., $N_s = 2$) and that $U_{critical}$ is 4.

In the first scenario (Figure 18 (a)), a competitive protocol is chosen at the end of the sampling period, because the average the number of updates $U$ per segment, denoted $U_{avg}$, is less than $U_{critical}$. Observe that, in Figure 18 (a), $U_{avg} = \frac{2+1}{2} = 1.5$ which is less than $U_{critical} = 4$.

In the third scenario (Figure 18 (c)), a migratory protocol is chosen because the conditions stated earlier for choosing migratory protocol are satisfied. Note that, in this case, the number of updates $U$ in each segment in the sampling period is $\geq U_{critical}$.

In the second scenario (Figure 18 (b)), the invalidate protocol is chosen because: (i) in segment 1, a local access (read) is performed by node $A$, after node $A$ sends page P in response to a page-request but before getting an update message from another node, and (ii) node $A$ does not send page $P$ to any other node in segment 2 (either condition would suggest that the access pattern may not be migratory).

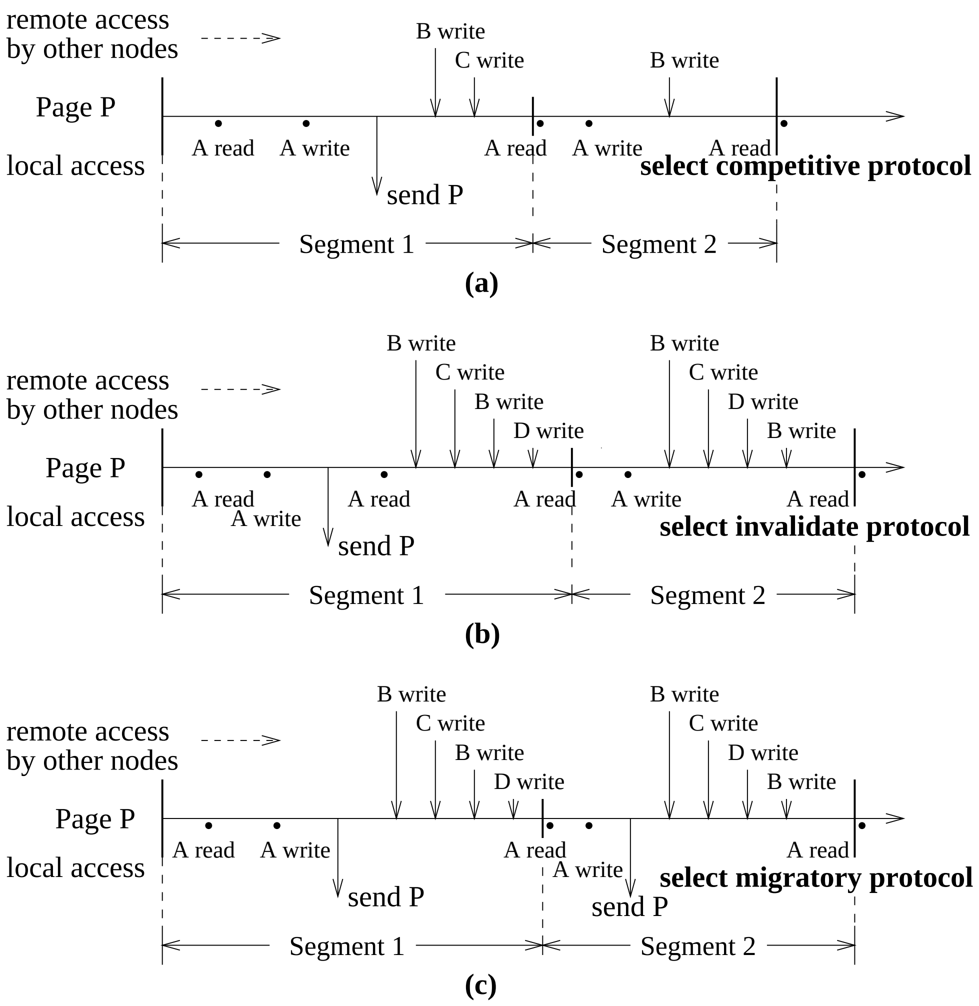


Fig. 18. Protocol Selection ($N_S = 2$, $U_{critical} = 4$)

## C. Performance Evaluation

Experiments are performed to evaluate the performance of proposed `adaptive+` protocol, by running applications on an implementation of the protocol. We implemented the adaptive protocol by modifying another DSM, named Quarks (Beta release 0.8) [10, 33]. This section presents the experimental results. We evaluated the adaptive scheme using the same applications used in the Chapter II. We use limit $L = 3$ for a competitive update protocol in all experiments.

### Results for `qtest` Application

For this application, Figures 19 and 20 show the measured cost as a function of number of nodes ($N$) executing the application. 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 schemes in Figure 19 is plotted using the heuristic for minimizing the number of messages; the curve in Figure 20 is plotted using the heuristic for minimizing the amount of data transferred.

In Figure 19, the curve named "protocol" denotes the number of messages required by the specified protocol, and "#update" denotes the average number of updates per segment ($U_{avg}$) calculated over the entire application. `adaptive` denotes the scheme in Chapter II. `adaptive+` denotes the proposed adaptive migratory protocol. As number of nodes $N$ increases, the average number of updates per segment ($U$) increases proportionally. For $N \geq 5$, adaptive migratory protocol (`adaptive+`) performs best, because `qtest1` shows the migratory memory access pattern. `Adaptive+` requires approximately 2 less messages per transaction than the `adaptive` protocol (because `adaptive+` chooses migratory protocol, while `adaptive` chooses the invali-

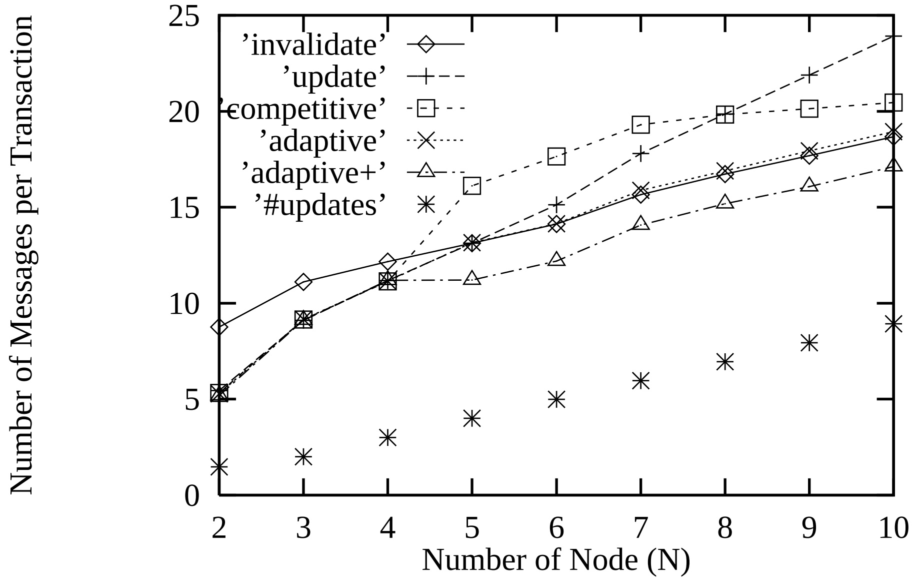


Fig. 19. qtest1: Average Number of Updates (U) and Messages per Transaction

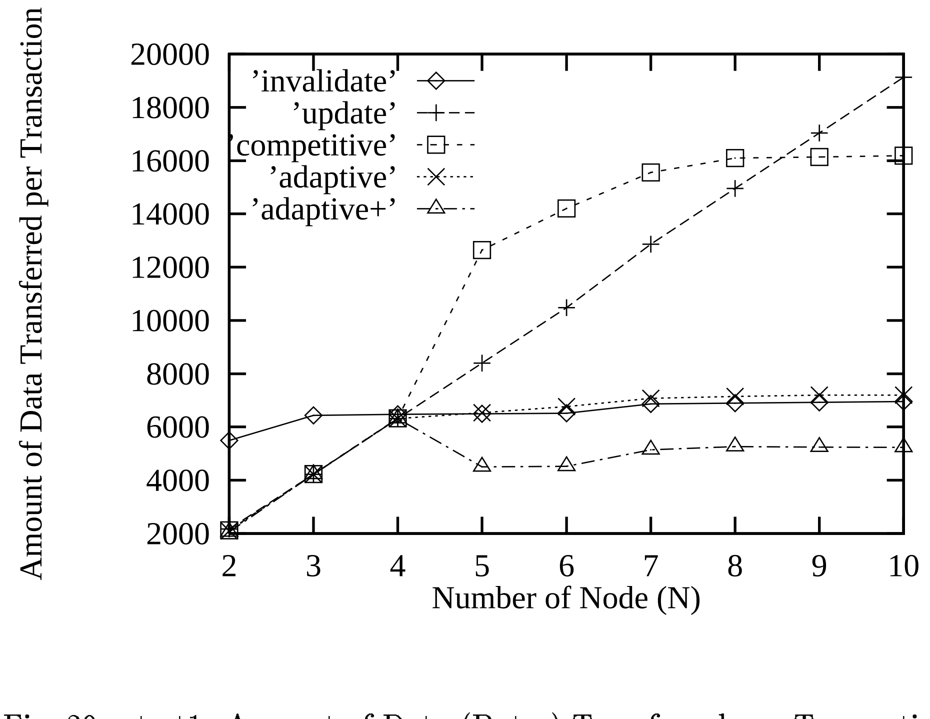


Fig. 20. qtest1: Amount of Data (Bytes) Transferred per Transaction

date protocol). However, `adaptive+` protocol requires the same number of messages as the `adaptive` protocol when $N \leq 4$, because both protocols choose competitive update protocol.

The cost graph for the invalidate protocol is not flat, while it was flat as per the cost analysis shown in Figure 17. The reason is that the cost of synchronization (*acquire*) increases as the number of nodes ($N$) increases (the cost for synchronization is not included in Figure 17, while it is taken into account in our measurements).

Figure 20 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 $N$. Adaptive *migratory* protocol chooses the appropriate protocol, thereby minimizing the amount of data transferred.

The second experiment was performed using `qtest2`. Figure 21 presents the number of messages per transaction (i.e., acquire, shared memory access, and release). Adaptive *migratory* protocol requires less number of messages than the adaptive protocol when read ratio is less than 20 % because `qtest2` tends to show migratory memory access pattern at low read ratios.

Figure 22 shows the comparison of the amount of data transferred per transaction. Since *qtest2* application modifies small amount of data ($NSIZE = 4$ bytes), both `adaptive` protocol and `adaptive+` (adaptive *migratory*) protocol choose a competitive protocol with large update limit ($L$). Therefore, both adaptive protocols require small amount of data transfer.

## Results for Other Applications

We now evaluate our adaptive scheme by executing seven additional applications

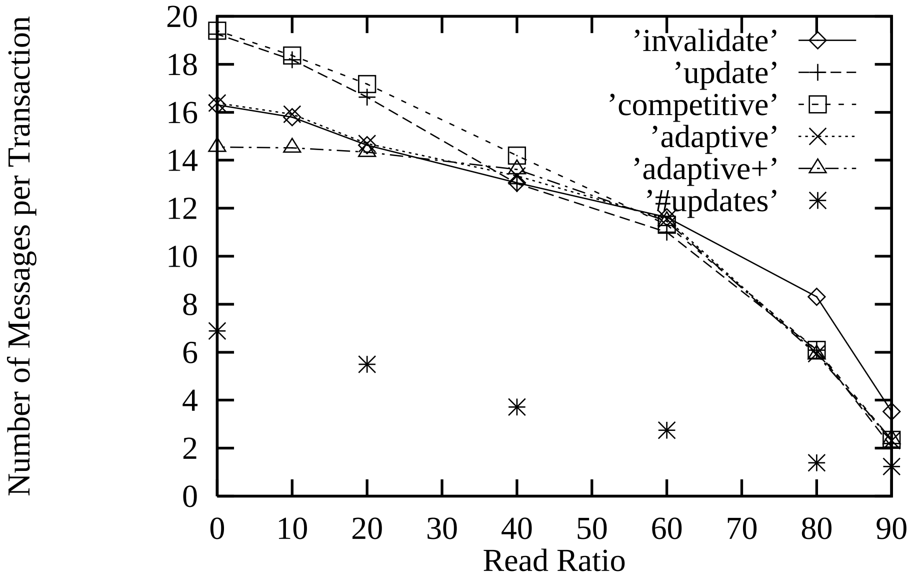


Fig. 21. qtest2: Average Number of Updates (U) and Messages per Transaction

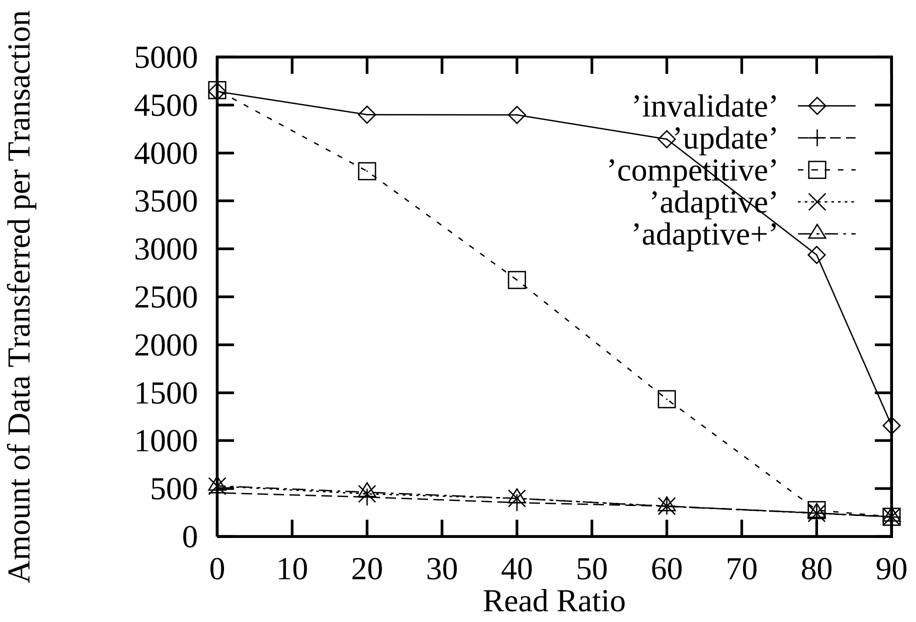


Fig. 22. qtest2: Amount of Data (Bytes) Transferred per Transaction

used in Chapter II (Floyd-Warshall, SOR, ProdCons, Isort, Reader/Writer, Matmult, and Jacobi) on 8-node workstation cluster.

We execute at least 10 times for each application and for each protocol. Table III shows experimental results: average number of messages (*Messages*), amount of data transferred (*Data (KBytes)*), execution time (*Time (seconds)*), and standard deviations of these values (*S.D.*).

Figures 23, 24, and 25 show performance comparisons for each cost metric (the number of messages, the amount of data transferred, or execution time). The figures plot costs for five protocols: invalidate (`INV`), update (`UPD`), competitive update with $L = 3$ (`COMP`), the adaptive scheme from Chapter II (`ADAPT`) and the adaptive migratory scheme (`ADAPT+`). The costs are normalized using the protocol with maximum cost for each application as the base.

Floyd-Warshall, SOR, Matmult, and Jacobi use barriers for synchronization. In these types of applications, the proposed adaptive migratory protocol (`ADAPT+`) does not show performance improvement over `ADAPT`, because Floyd-Warshall, SOR, Matmult, and Jacobi do not show the migratory memory access pattern. In Floyd-Warshall, `ADAPT+` performs worse than `ADAPT` due to false detection of migratory sharing, i.e., our heuristic uses migratory protocol when the access pattern is not actually migratory.

ProdCons uses lock/unlock for a task queue, Isort uses lock/unlock for ranking, and Reader/Writer uses lock/unlock for exclusive object access. These applications show migratory memory access patterns. (In Reader/Writer, some pages show migratory memory access patterns.) In three applications (ProdCons, Isort, and Reader/Writer), adaptive migratory protocol (`ADAPT+`) requires the least number of messages. However, in Reader/Writer application only, adaptive migratory protocol (`ADAPT+`) requires the least amount of data because the size of update message is

Table III. Performance Comparison (II) (other applications)

| *Application* | *Messages* | (S.D.) | *Data (KB)* | (S.D.) | *Time (sec.)* | (S.D.) |
|---|---|---|---|---|---|---|
| **Floyd-Warshall** | | | | | | |
| Invalidate | 9676 | (1143) | 4674 | (65) | 25.4 | (4.29) |
| Update | 27873 | (340) | 2392 | (2) | 26.5 | (3.39) |
| Competitive | 8633 | (605) | 1562 | (2) | 26.5 | (3.39) |
| Adaptive | 8360 | (53) | 1910 | (12.8) | 14.6 | (1.66) |
| Adaptive Migratory | 10146 | (623) | 3637 | (27.6) | 18.4 | (1.18) |
| **SOR** | | | | | | |
| Invalidate | 16436 | (994) | 12204 | (87) | 46.0 | (8.92) |
| Update | 101172 | (715) | 58518 | (3) | 237.7 | (5.37) |
| Competitive | 13753 | (679) | 4588 | (3) | 30.6 | (2.45) |
| Adaptive | 13877 | (746) | 4589 | (3) | 32.2 | (2.12) |
| Adaptive Migratory | 13729 | (585) | 4587 | (2) | 43.9 | (7.67) |
| **ProdCons** | | | | | | |
| Invalidate | 65428 | (2221) | 17790 | (222) | 155.7 | (32.15) |
| Update | 76636 | (517) | 1387 | (8) | 112.9 | (9.96) |
| Competitive | 76527 | (739) | 18124 | (154) | 148.5 | (14.94) |
| Adaptive | 64730 | (1224) | 1451 | (5) | 114.0 | (8.17) |
| Adaptive Migratory | 55825 | (1279) | 1441 | (20) | 112.9 | (9.55) |
| **Isort** | | | | | | |
| Invalidate | 51979 | (1551) | 14294 | (176) | 124.5 | (22.67) |
| Update | 61449 | (331) | 993 | (6) | 97.6 | (6.48) |
| Competitive | 60753 | (1240) | 14282 | (300) | 124.3 | (11.65) |
| Adaptive | 51824 | (224) | 1048 | (9) | 94.8 | (7.72) |
| Adaptive Migratory | 45413 | (879) | 1047 | (20) | 90.1 | (3.10) |
| **Reader/Writer** | | | | | | |
| Invalidate | 74443 | (804) | 41255 | (470) | 244.4 | (38.78) |
| Update | 91652 | (1110) | 42694 | (5) | 225.5 | (6.58) |
| Competitive | 90458 | (504) | 57702 | (60) | 253.6 | (15.73) |
| Adaptive | 68587 | (1168) | 22639 | (294) | 227.1 | (16.05) |
| Adaptive Migratory | 60742 | (649) | 18703 | (263) | 191.5 | (28.87) |
| **Matmult** | | | | | | |
| Invalidate | 2388 | (1345) | 864 | (5) | 22.7 | (6.39) |
| Update | 3256 | (483) | 3458 | (2) | 26.4 | (1.59) |
| Competitive | 2156 | (287) | 1381 | (1) | 20.5 | (1.25) |
| Adaptive | 2204 | (425) | 1382 | (2) | 21.2 | (1.62) |
| Adaptive Migratory | 2399 | (901) | 1383 | (3) | 20.9 | (1.37) |
| **Jacobi** | | | | | | |
| Invalidate | 2462 | (629) | 646 | (19) | 29.9 | (4.49) |
| Update | 2237 | (202) | 327 | (1) | 1.0 | (0.08) |
| Competitive | 2427 | (618) | 380 | (15) | 4.6 | (1.49) |
| Adaptive | 2349 | (489) | 343 | (6) | 3.3 | (0.74) |
| Adaptive Migratory | 2175 | (123) | 344 | (6) | 5.3 | (1.17) |

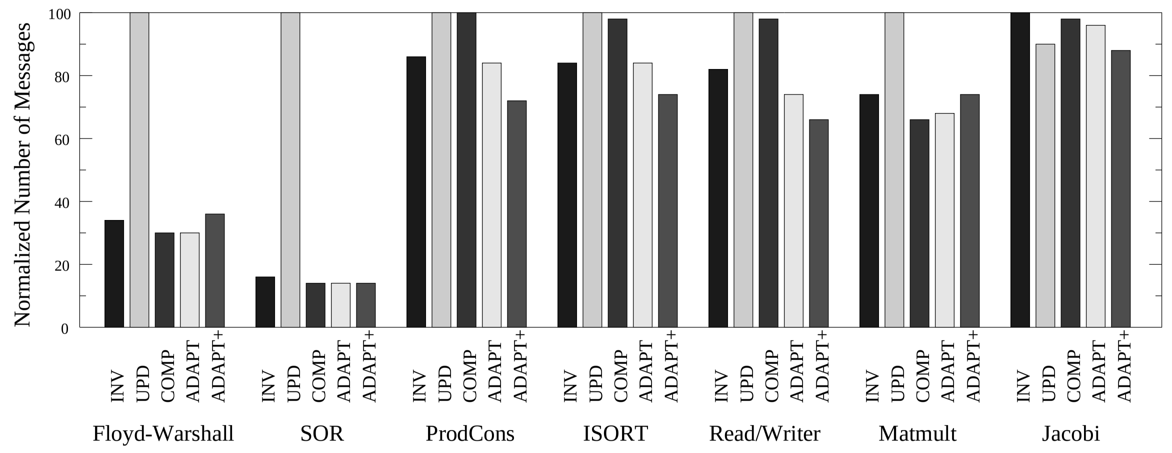


Fig. 23. Cost Comparisons (Number of Messages)

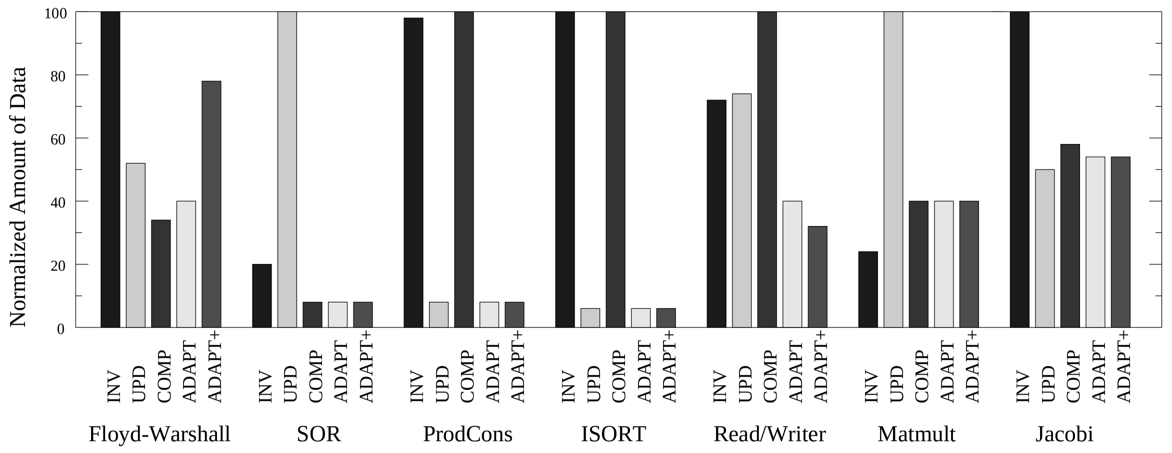


Fig. 24. Cost Comparisons (Amount of Data Transferred)

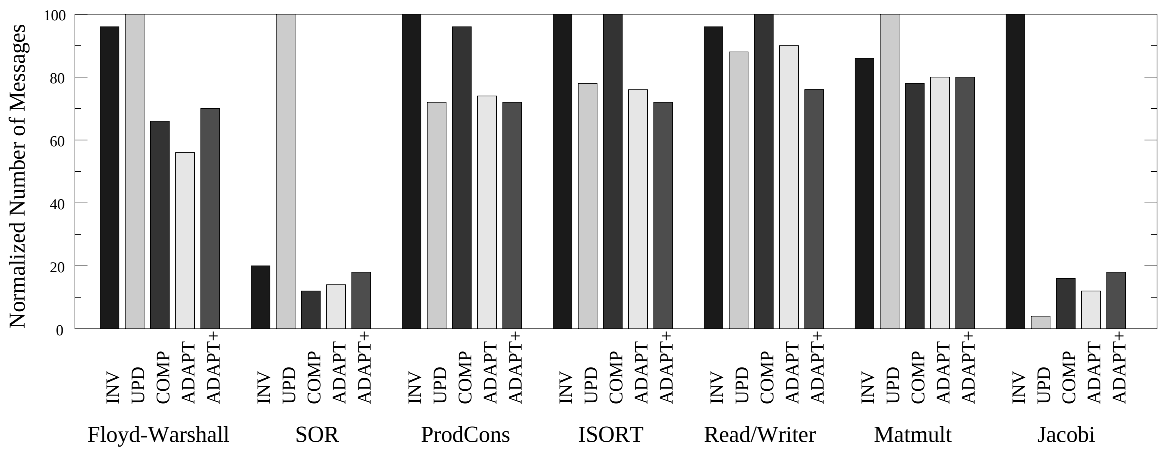


Fig. 25. Cost Comparisons (Execution Time)

small in other applications (Isort and Reader/Writer). For a similar reason, adaptive migratory protocol achieves performance improvement by 15 % in Reader/Writer application, while only 1 % and 5 % in ProdCons and Isort application, respectively, in terms of execution time.

## D. Summary

This chapter presents a new adaptive DSM that allows each node to *independently* choose any one of the following protocols for each page: migratory, invalidate, and competitive update. This protocol improves on our previous scheme in Chapter II by detecting migratory patterns. The adaptive protocol attempts to detect migratory access pattern, and chooses the migratory protocol when it is deemed most cost-effective.

We present experimental evaluation of the proposed adaptive *migratory* scheme using an implementation based on Quarks DSM [33]. Experimental results from the implementation suggest that the proposed adaptive approach can usually reduce the *cost*. Specifically, the proposed scheme can typically reduce the number of messages as compared to the adaptive scheme presented in Chapter II, as well as invalidate and competitive update protocols. However, in a few application, the adaptive migratory scheme performs worse than the adaptive scheme due to false detection of migratory sharing. In this case, the adaptive scheme in Chapter II is recommended instead of the adaptive migratory scheme.

# CHAPTER IV

# SINGLE FAULT-TOLERANT DISTRIBUTED SHARED MEMORY USING COMPETITIVE UPDATE

This chapter presents a single fault-tolerant distributed shared memory (DSM) that uses the *competitive* update protocol. In *competitive* update protocol, multiple copies of each page may be maintained at different nodes. However, it is also possible for a page to exist in only one node, as some copies of the page may be invalidated. We propose an implementation that makes the *competitive* update protocol recoverable from a single node failure, by guaranteeing that at least two copies of each page exist. This chapter also presents a mechanism that maintains consistency between shared data and process local state after recovery, by updating shared data and process local state atomically [35, 36, 34].

## A. Related Work

Many recoverable DSM schemes have been presented in the literature. Some of them use stable storage (disk) to save recovery data [24, 25, 56, 68], and others use main memory for checkpointing, replicating shared memory or logging the shared memory accesses [4, 9, 22, 27, 31, 45, 61, 63]. Proposed recoverable DSM belongs to the second category (uses main memory). [61, 63] are based on update (full-replication) protocol, while [4, 9, 22, 31, 45] are based on invalidate (read-replication) protocol.

Stumm and Zhou extend four DSM algorithms to tolerate single node failures [61]. One of their algorithms is for an update protocol. However, implementations of our algorithm is different because their algorithm is based on update protocol where all copies of a page are updated, whereas our scheme is based on the *competitive* update protocol (some copies are invalidated to reduce overhead). Additionally, our scheme supports *release* consistency.

Theel and Fleisch present a coherence protocol [63] that is highly available. Their scheme has an upper bound (to reduce overhead) as well as a lower bound (for availability) on the number of copies of each shared memory page. Unlike [63], our scheme is based on the *competitive* update protocol.

Janssens and Fuchs [25] present a recoverable DSM that exploits release consistency to reduce the number of checkpoints, as compared to communication-induced checkpointing schemes for sequential consistency. Their scheme requires a process to take a checkpoint either when performing a write on a synchronization variable, or when another process performs a read on the synchronization variable. The checkpoints are stored on a storage not subject to failures. Our single fault tolerance scheme handles the non-shared data similar to [25]; our scheme "checkpoints" non-shared data in the *volatile* memory of another processor. However, the shared data is not explicitly checkpointed – instead the shared data is duplicated as a part of the update protocol (if multiple copies already exist, no additional overhead is incurred). When compared to [25], the proposed scheme trades degree of fault tolerance to reduce the performance overhead. Janssens and Fuchs [26] also present an approach to reduce interprocessor dependencies in recoverable DSM.

Brown and Wu present recoverable DSM, based on an *invalidate* protocol, that can tolerate single point failure [9]. A dynamic *snooper* keeps a back-up copy of each page and takes over if the page owner fails. The snooper keeps track of the page contents, location of page replicas, and the identity of the page owner. The snooper can respond on behalf of a failed owner. Our scheme also maintains at least two copies of a page, however, the proposed scheme is based on an *update* protocol, unlike [9].

Neves et al. present a checkpoint protocol for a multi-threaded distributed shared memory system based on the entry consistency memory model [45]. Their algorithm needs to maintain log of shared data accesses in the volatile memory. Fuchi and Tokoro propose a mechanism for recoverable shared virtual memory [22]. Their scheme maintains back-up process for every primary process. When the primary process sends/receives a message to/from another process (or writes/reads a shared memory), the primary process sends this information to back-up process so that the back-up process can log the events of the primary process.

Richard and Singhal [56] present an invalidate-based scheme for recovery of failed processors in DSM systems. Their scheme is based on asynchronous checkpointing of application processes and logging of pages accessed via read operations on the shared address space. They use volatile logs and stable logs. Every read content is stored in volatile logs, and flushed to the stable log on a page transfer.

Backward error recovery on a Cache Only Memory Architecture is implemented using invalidate protocol by Banatre et al. [4]. (A similar scheme is implemented on an Intel Paragon by Kermarrec et al. [31].) This scheme periodically takes system-wide *consistent* checkpoints. After a node fails, all nodes need to rollback to the last checkpoint.

Plank and Li propose *parity checkpointing* [51] based on *diskless checkpointing.* A consistent checkpoint is held in $N$ processors, and bitwise exclusive-or of the checkpoints is held in a processor called *parity processor.* If any one of $N$ processors fails, the failed processor can be recovered to the consistent checkpoint by computing its checkpoint from all the other checkpoints and the parity checkpoint.

## B. Recoverable *Competitive* Update Protocol

Recoverable scheme for a DSM, based on the *competitive* update protocol [23, 28], is relatively simple. The basic idea behind the proposed scheme is to maintain, at all times, *at least two* copies of each page (at two different nodes) instead of checkpointing. This will allow the DSM to recover from a *single* node failure without significant overhead (provided the non-shared data is also recoverable, as discussed later).

When the competitive update protocol is used, it is possible that a page may be resident in *only one* node. Therefore, to tolerate a single node failure, it is necessary to modify the *competitive* update protocol, to ensure that *at least two* nodes have a copy of each page. Thus, there are two issues that must be dealt with to make the DSM fault tolerant (for single node failures).

1. Modification of the competitive update protocol to guarantee two copies of each shared memory page.

2. Some mechanism needs to be incorporated to make process local state recoverable and consistency with shared data.

We first focus on the first of the above two issues.

## 1. Recoverable Shared Data

### Maintaining at Least Two Copies of Each Page

To simplify the discussion, we assume that each page has the same fixed limit $L$. To make the DSM recoverable, we must modify the competitive update protocol, such that some copy of the page is *not* invalidated, *even if* its update counter exceeds the *limit* $L$. This is achieved by designating, for each *update*, one of the nodes as the "back-up". The copy of a page at the *back-up* node cannot be invalidated, irrespective of the value of its *update-counter*. Note that the *back-up* is specified for each *update*, and may change from one update to the next update of the *same* page. The performance of the recoverable DSM may depend on the choice of the back-up – in our approach, as described below, the node chosen as the *back-up* is the one that is expected to access the page in the near future.

Let us consider the copy of a page $P$ at a node $A$. Contents of a *back-up* field can change based on the three rules listed below.

1. When a node $A$ obtains a copy of a page $P$ from some other node $B$, node $B$ also sends identifier of the *last-updater* of page $P$. Node $A$, on receiving the page, sets its *last-updater* as well as *back-up* equal to the *last-updater* received from node $B$.

2. Node $A$ receives an update message for page $P$ from some other node, say $C$: In this case, the *back-up* field at node $A$ is set equal to $C$. The node $C$ is used as *back-up* when node $A$ updates other nodes.

   The motivation behind this rule is to identify a node as the *back-up* only if it has accessed the page recently (this, in turn, is motivated by the principle of locality).

3. Node $A$ performs a *release* and sends update messages, for page $P$, to other nodes: The update messages are sent to the other nodes in the order of their identifiers. When the other nodes receive these update messages, they respond to the update message. Specifically, if *update-counter* is less than or equal to *limit* $L$ or the node is the back-up, the node incorporates update message and sends an *ack* along with its *update-counter*; otherwise, it invalidates local copy and replies *negative-ack*. Node $A$ designates a node that replies *ack* with the smallest *update-counter* as the *back-up* for future updates of page $P$ (ties may be broken arbitrarily).

   In the above procedure, the back-up node, say $C$, is forced to retain the page even if its update-counter exceeds the limit. If some node, say $B$, also has a copy of the page, then there is no need to force the node $C$ to retain its copy. To reduce the situations where a back-up node is forced to retain its copy of a page, even if its update counter exceeds $L$, we modify the above procedure, as follows.

Assume that the back-up node for page $P$ at node $A$ is $C$. If node $A$ receives an *ack* from some node, say $B$, before sending the update to the node $C$, then node $A$ temporarily designates $B$ as the back-up for page $P$. Now, when the update is sent to node $C$, it is not designated as the *back-up*. After updating all nodes that have a copy of page $P$, node $A$ designates a node, say $D$, that replies *ack* with the smallest *update-counter* as the *back-up*. (Note that the original back-up node $C$ may potentially reply negative-ack if its update-counter exceeds the limit.)

Note that, for a given page, the *back-up* at different nodes may be different.

**Proposed Recoverable *Competitive* Update Protocol [36]**

The proposed scheme assumes that programs are *data-race-free*[1]. The modified protocol is mostly identical to the original competitive update protocol with one difference: A node that is designated as the *back-up* for an update does *not* invalidate the local copy of the page *even if* the *update-counter* exceeds $L$. (Update message sent to the *back-up* node is tagged by a special *marker*.) Any other node whose update-counter exceeds $L$ invalidates its local copy of the page. This procedure ensures that, at any time, at least two copies of a page are in existence.

The *back-up* for an update is always a node that has accessed the page in the recent past. Therefore, from the locality principle, this node is likely to access the

| | 0 | 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory Access | | 2L | 2R | 2U | 0L | 0W | 0W | 0U | 1L | 1R | 1W | 1U | 0L | 0W | 0U | 0L | 0W | 0U | 0L | 0W | 0U | 2L | 2R | 2U | 2L | 2W | 2U |
| Update-counter: 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Update-counter: 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | $\cancel{4}$ |
| Update-counter: 2 | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | $\cancel{3}$ | | | | | | | | 0 | 0 | 0 | 0 | 0 |
| Last-updater | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| Back-up: 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| Back-up: 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| Back-up: 2 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | | | | | | | | | 0 | 0 | 0 | 0 | 0 |

Fig. 26. Update Counter for Recoverable DSM

page in the near future as well. The modified update protocol forces this node to retain a copy of the page. This protocol may be viewed as incorporating a "pre-fetch" mechanism. As the page copy is likely to be used in the near future, the overhead of updating the copy is often compensated by a reduction in the number of page faults.

Note that "cost" (e.g., number of messages) of the recoverable protocol can be larger than that of the non-recoverable protocol, only when the non-recoverable protocol would result in a page having only one copy. Whenever, the non-recoverable protocol results in multiple copies of a page, the recoverable protocol does not result in any additional cost. Thus, the *difference* between the costs of the recoverable and non-recoverable protocols is expected to be greatest when limit is 0, and reduces as *limit* becomes larger.

## Example

Figure 26 illustrates how the *back-up* is maintained. For this example, assume that the *limit* $L$ is 2. The system is assumed to contain three nodes, 0, 1 and 2. In the figure, $iL$ and $iU$ denote *acquire* and *release* operations by node $i$. (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 necessarily equivalent to *lock* and *unlock*.) Also, $iR$ and $iW$ denote *read* and *write* operations performed on this page by node $i$. Initially, the page is loaded in the local memory of two nodes (0 and 1 in our example), and one of them (node 0) is considered to be the *last-updater*. *Back-up* at nodes 0 and 1 is initialized to 1 and 0, respectively. The *memory access* row in Figure 26 presents a total ordering on the accesses to the page under consideration. The next three rows present values of the update-counters at the three nodes at various times, e.g., the *update-counter:0* row corresponds to node 0. (The values in column $i$ correspond to the update-counters *after* the memory access in column $i$ is performed.) The next row of the table lists the *last-updater* variable at each node (it is identical at all nodes). The last three rows list the value of the *back-up* variable for the page at each node. Note that *last-updater* and *back-up* change only when a *release* is performed, whereas, *update-counter* at a node $A$ changes when either (i) node $A$ performs a local access to the node, or (ii) another node performs an update to the page. A "blank" in the table implies that the corresponding node does not have a copy of the page at that time, and an $X$ in the figure denotes an invalidation.

The initial state is illustrated in column 0 of the table. The first *acquire* is performed by node 2, followed by a *read* and a *release* (columns 1-3). As shown in column 2, a copy of the page is brought to node 2 when it reads the page, its update-counter is set to 0, and the back-up is set to 0 (the last-updater for the page).

Next, node 0 performs a *acquire-write-write-release* sequence (columns 4-7). When node 0 performs a release (column 7), it sends update messages to other nodes. As the back-up at node 0, immediately before the release is performed, is node 1, the update message sent to node 1 is tagged by a marker to inform node 1 that it is the back-up. When the acknowledgements for the update messages are received, node 0 determines its new back-up by finding the minimum of the update-counters received with the acknowledgement. As both nodes 1 and 2 return update-counter 1, node 0 arbitrarily chooses node 2 to be the back-up for its next update. The new back-up is shown in column 7 of the "back-up:0" row in the table. Nodes 1 and 2 set their *back-up* variable to 0, because they received an update from node 0 (column 7).

Next, node 1 performs *acquire-read-write-release* sequence (columns 8-11). When node 1 performs a release (column 11), the update message sent to node 0 is tagged with a marker, as node 0 is the back-up for this access (as shown in column 10, row "back-up:1"). When all the acknowledgements and update-counters are received, node 1 determines the new back-up as the node whose update-counter is the smallest, namely node 0. (The new back-up is shown in column 11, row "back-up:1"). Nodes 0 and 2 change their back-ups to 1, as they received an update from node 1 (column 11).

At this point (column 11), the update-counters for nodes 0, 1 and 2 are 1, 0 and 2, respectively. Next, node 0 performs an *acquire-write-release* sequence (columns 12-14). At the release by node 0 (column 14), the update message sent to node 1 is tagged by a marker, whereas that sent to node 2 is not tagged, as node 1 is the back-up for this update (see column 13, row back-up:0). When the update message is received by node 2, it performs the update and increments its update-counter to 3. Now, node 2 invalidates the local copy of the page because, (a) its update-counter exceeds *limit* 2, and (b) the update message sent to node 2 was not tagged by a marker (which means that node 2 is not the back-up for the update). When node 0 receives the acknowledgements, it determines that node 1 is its new *back-up*. Also, node 1 sets its *back-up* to 0, when it receives the update-message.

Now, node 0 again performs *acquire-write-release* (columns 15-17) followed by another *acquire-write-release* (columns 18-20). At the second release (column 20), update-counter for node 1 becomes equal to 3. At each of the release, node 0 sends an update message to node 1 tagged with the marker. Therefore, node 1 cannot invalidate its copy of the page. Note that the update-counter at node 1 exceeds 2 (column 20), but the page is not invalidated.

Node 2 now performs *acquire-read-release* (column 21-23), therefore, it receives a copy of the page. Along with the page, it also receives identifier of the last-updater for the page. On receiving the page, its update-counter is set to 0, and back-up set equal to the last-updater. As node 2 did not write to the page, no update is necessary at the *release* (column 23).

Subsequently, node 2 performs *acquire-write-release* (columns 24-26). At the release, node 2 sends update messages to nodes 0 and 1, the message sent to node 0 being tagged with a marker. When node 1 receives the update, its update-counter becomes 4. Node 1 invalidates the page, as the update message was not tagged with a marker, and the update-counter is larger than the *limit*.

## 2. Recoverable *Process Local State* Consistent with Shared Data

Two implementations of distributed shared memory can be conceived. In one approach, the application is written such that, to survive a failure, a consistent state of the distributed shared memory must be available (after recovery). In this case, local state of the application processes is not necessary for recovery from failure. For such systems, the scheme in Section 1 is adequate.

The other approach requires that, after a failure, the *local* state of processes scheduled on the faulty node be recovered, in addition to the distributed shared memory. In this case, additional steps must be taken to ensure that the local state is recoverable. We achieve this by maintaining two copies of the local state of each process, as described below. The *process local state* includes non-shared local data, contents of registers and stack, etc.

When a node writes shared data and updates other copies of the data, the *process local state* at the node can be sent, along with the update message, to any one node. Although no additional messages are required, the size of one of the messages will be larger. (This procedure effectively "checkpoints" the local state at another node.) The size of *process local state* to be transferred can be reduced by sending only the *modifications* to the local state since the most recent update performed by the node. This *incremental* approach [50] can reduce the overhead of saving the local state.

The above approaches are application-transparent, in that entire local state of a process is saved on another node. In many applications, it is possible to identify a small set of local variables that are sufficient to recover the local state of the process (a typical example is a *loop counter*). Consider the application below.

```
// Typical application  //
main()     // executed by master node
{
  initialize();          // application initialize
  init_shm();            // shared memory initialize
  fork_threads(compute);// fork remote processes to execute compute()
  compute();             // compute
}


compute()  // executed by all processes
{
  while (not finish) {  // repeat until FINISH
    update(local_vars); // update local variables (e.g., a counter)
    read_shm();          // read shared memory
    calculation();       // calculation
    write_shm();         // write shared memory
    synch();             // synchronization (e.g., barrier, lock-unlock)
    // take checkpoint of local state at the first iteration
    if (first iteration)
      checkpoint_local_state()
  }
}
```

In the above application, each process repeats a computation loop in `compute()` several times. A process may have some local state that does not change after it is initialized. This local state can be checkpointed during the first iteration of the computation loop. During each iteration, although many local variables may potentially be modified, only a small set of modified local variables constitute the critical state of the process. Only these critical variables need be saved to recover the process from a failure. (Similar techniques have been proposed in [5, 50].) When a process sends an update message to other nodes, a copy of the critical local variables should be sent to any one node. On a failure, a copy of the local variables is obtained from another node. These local variables, in addition to the state checkpointed during first iteration, can be used to recover the process state. The distributed shared memory state is recoverable by the algorithm in the previous subsection.

## 3. Recovery

The proposed DSM system is recoverable from single node failures (fail-stop), because all shared pages have at least two copies, and process local state of each faulty process can be reconstructed (if necessary). The recovery is straightforward. After a single node failure, the shared memory remains available. If the faulty node is to be recovered, then its *process local state* is obtained from saved process image and local variable. Two issues need further elaboration.

### Atomic Updates

Since failure can occurs at any time, contents of the copies of the same page may be different (if the failure occurs while an update is in progress). In this case, some copies are out-of-date. This problem can be resolved by searching the most up-to-date copy – to facilitate this, a *version* number is attached to each page to count the number of updates performed to the page from the beginning of execution. The copy with the largest version number is the most up-to-date copy (this is similar to [61]). If a node fails after it has written to a page, but before it has performed a *release* then the modifications made by the node are lost when the node fails. This is acceptable, as the system state will still be consistent after the failure. However, if a node fails after the node sent update messages only for the part of pages to be updated on a *release*, the node may not restart from the previous consistent state because old *version* of the updated pages may not exist. This problem can be solved by new mechanisms. One possibility is for the updating node to send all updates (for all pages) atomically to another node, then to send update messages to the other nodes (copiers) in sequence of nodes for each page. By this *atomic* update mechanism, all pages can be updated atomically in spite of a single node failure. If the node receiving *atomic* update have copies of associated pages, no additional overhead for sending update message is necessary for *atomic* update mechanism.

### Maintaining at Least Two Copies after Recovery

It is necessary to ensure that, after recovery, each shared memory page has at least two copies. Therefore, after failure, if only one node has a copy of a page, then another copy is created on any other node. Now we assume that two copies of each page exist. The recovery algorithm must also ensure that all the *last-updater* and *back-up* fields are correct. We now illustrate how this can be achieved. Consider a page $P$. Two cases are possible.

(a) If the *last-updater* for page $P$ fails, then any other node having the page is designated as the *last-updater*, and its update-counter is cleared to 0. All relevant nodes are informed of the new *last-updater*. These nodes set their *last-updater* as well as the *back-up* fields to point to the new *last-updater*. The new *last-updater* sets its *back-up* field to point to any other node that has a copy of the page.

(b) If some node other then the *last-updater* is faulty, then it is possible that the *back-up* field at the last-updater may be pointing to the faulty node. It is only necessary to set the back-up to point to any other node that has a copy of the page.

## C. Performance Evaluation

Experiments are performed to evaluate the overhead for maintaining recoverable process local state as well as shared data, by comparing the "cost" for non-recoverable protocol and recoverable protocol. The "cost" metrics used here are (i) number of messages and (ii) amount of information transferred between the nodes. We implemented the recoverable DSM by modifying Quarks (Beta release 0.8) [10, 33].

We evaluated the recoverable DSM scheme using the same applications used in Chapters II and III.

## Results for `qtest` Application

This application is the same as `qtest2` used in the Chapter II. We execute at least 5 times for each read ratio and for each update limit. Appendix A shows experimental results: average number of messages (*Messages*), amount of data transferred (*Data (KBytes)*), standard deviations of these values (*S.D.*), and overhead ratio percentage of recoverable scheme.

Figures 27 through 38 plot costs (the number of messages and the amount data transferred) for non-recoverable scheme and recoverable scheme for each read ratio with different update limits. The **qtest*x*.msg** curves show the number of messages required for non-recoverable scheme with read ratio ***x*** percent. The **rqtest*x*.msg** curves correspond to recoverable scheme. The **qtest*x*.dat** and **rqtest*x*.dat** curves correspond to the amount of data transferred. Figures 39 and 40 plot overhead ratio percentage for recoverable schemes. Overhead ratio percentage ($r$) is computed as: $r = \frac{C_r}{C_n} \times 100$, where $C_n$ is the cost for non-recoverable scheme and $C_r$ is the cost for recoverable scheme. The **qtest*x*.msg** and **qtest*x*.dat** curves show the overhead for recoverable scheme with read ratio ***x***, in terms of the number of messages and the amount of data transferred, respectively. Overhead (the number of messages and the amount of data) for recoverable scheme is reasonably small for many read ratios and update limits. Overhead of the number of messages for recoverable scheme converges to zero as update limit increases because multiple copies exist for each page, thus no extra messages are required to maintain at least two copies for each page. For small update limit (1 to 3), the overhead of the number of messages for recoverable scheme tends to decrease as read ratio increases because back-up node sometimes avoids a page fault. (Back-up node probably uses the page again without or with small number of updates by other nodes.) In many cases, the amount of data transferred is reduced for recoverable scheme. For example, maintaining back-up reduces the page faults at

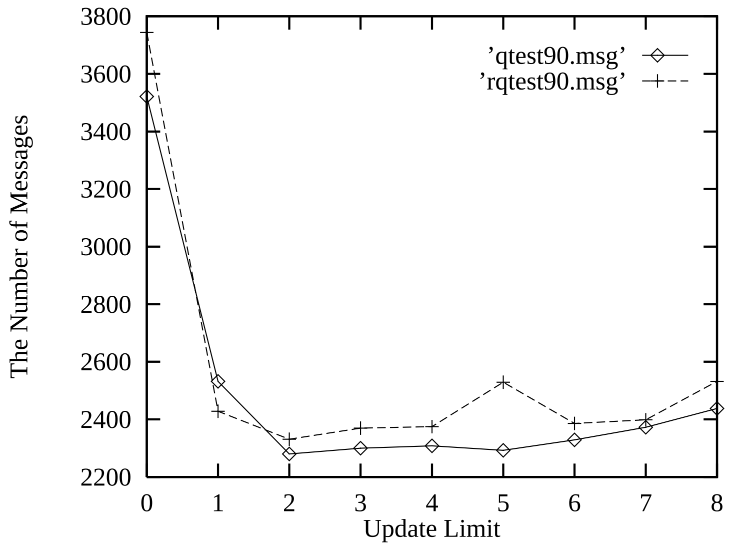


Fig. 27. qtest (Read Ratio = 90 %): The Number of Messages

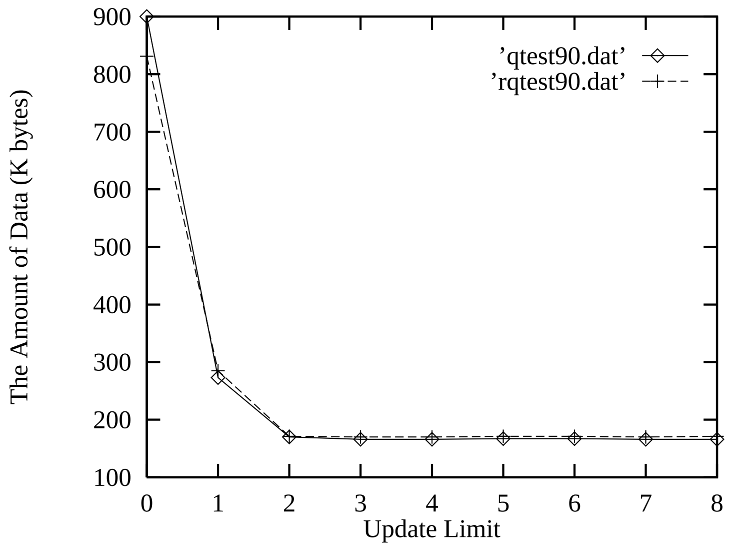


Fig. 28. qtest (Read Ratio = 90 %): The Amount of Data Transferred

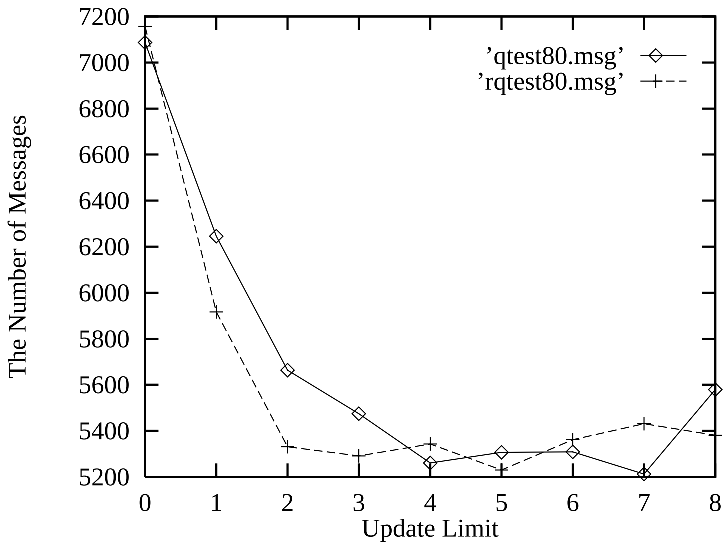


Fig. 29. qtest (Read Ratio = 80 %): The Number of Messages

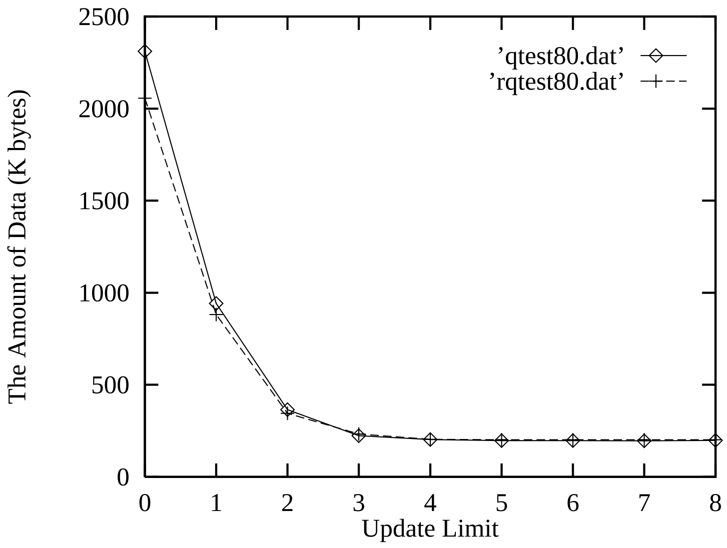


Fig. 30. qtest (Read Ratio = 80 %): The Amount of Data Transferred

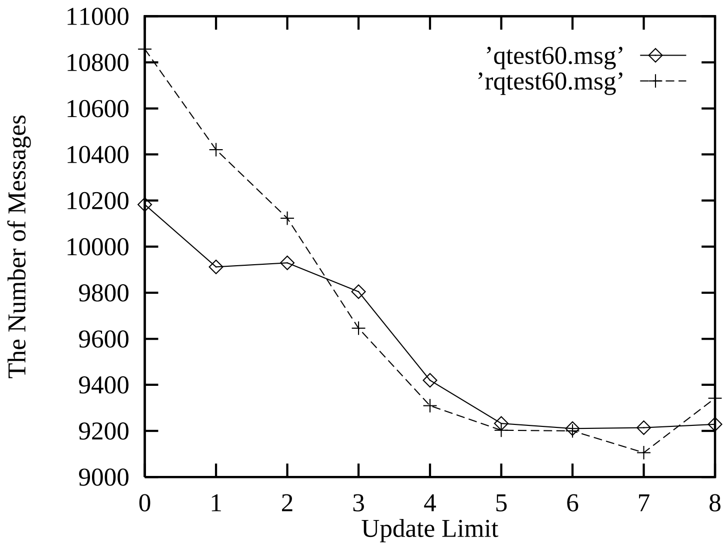


Fig. 31. qtest (Read Ratio = 60 %): The Number of Messages

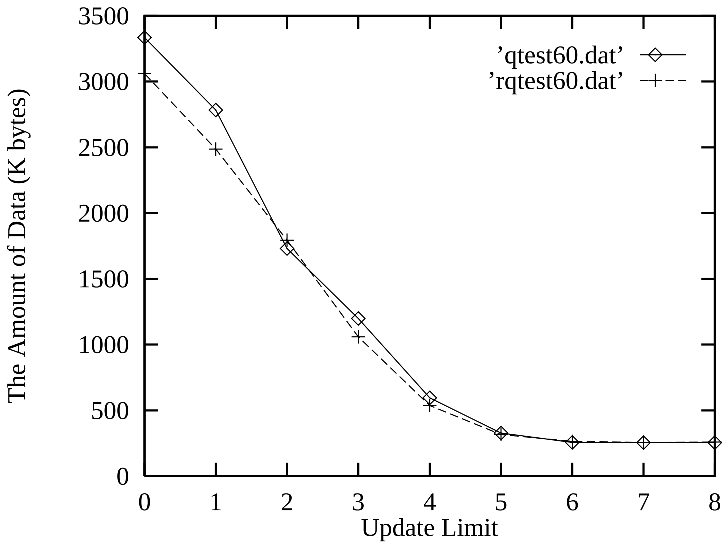


Fig. 32. qtest (Read Ratio = 60 %): The Amount of Data Transferred

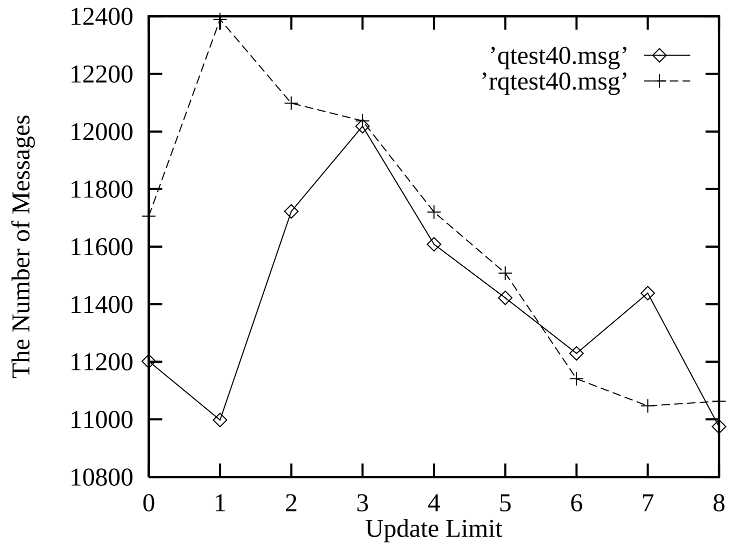


Fig. 33. qtest (Read Ratio = 40 %): The Number of Messages

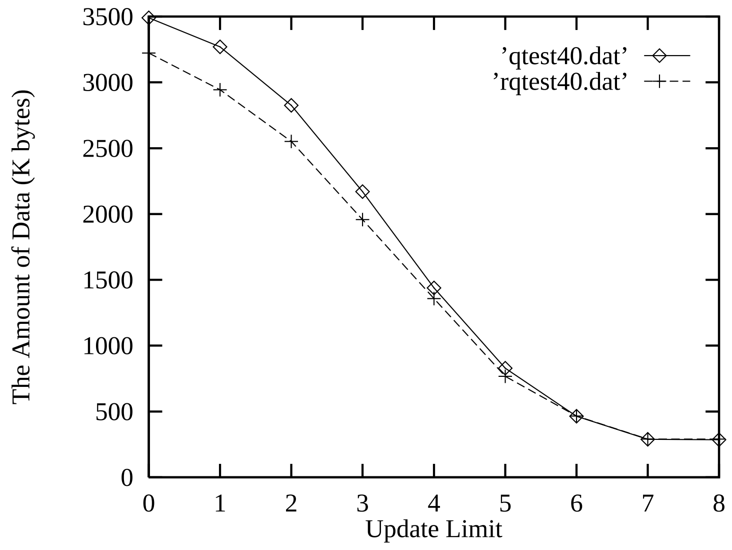


Fig. 34. qtest (Read Ratio = 40 %): The Amount of Data Transferred

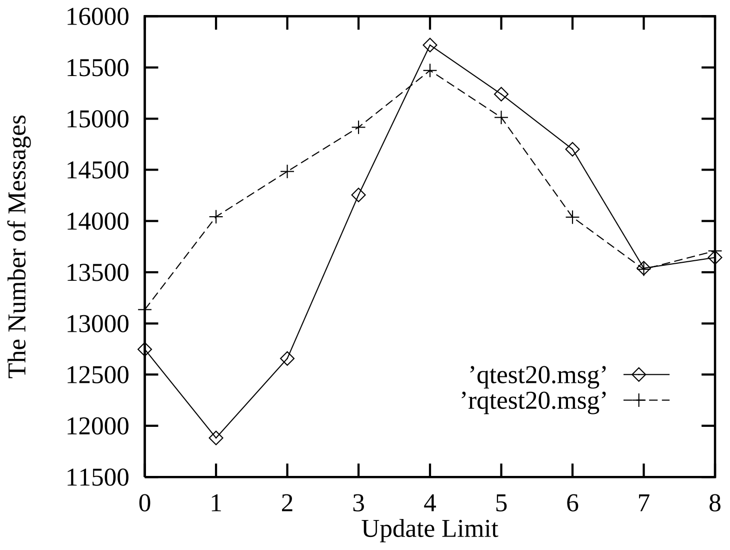


Fig. 35. qtest (Read Ratio = 20 %): The Number of Messages

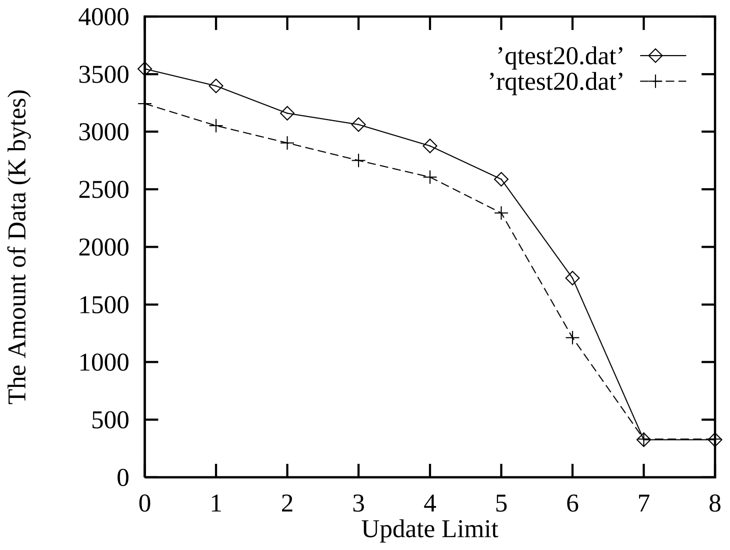


Fig. 36. qtest (Read Ratio = 20 %): The Amount of Data Transferred

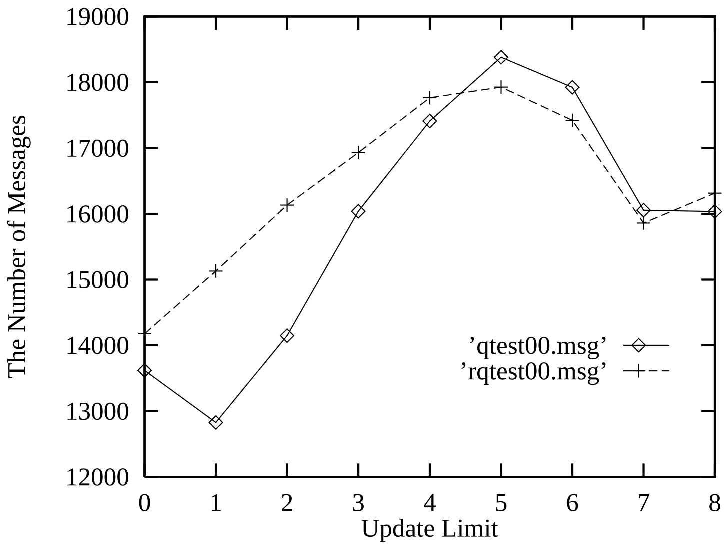


Fig. 37. qtest (Read Ratio = 0 %): The Number of Messages

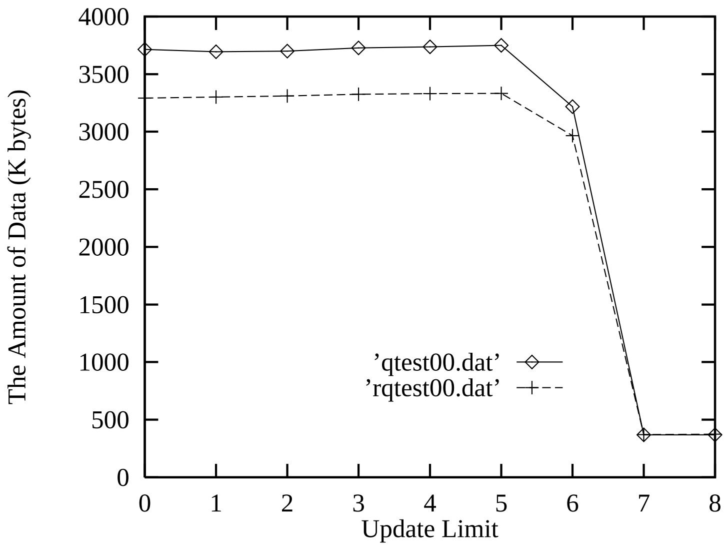


Fig. 38. qtest (Read Ratio = 0 %): The Amount of Data Transferred

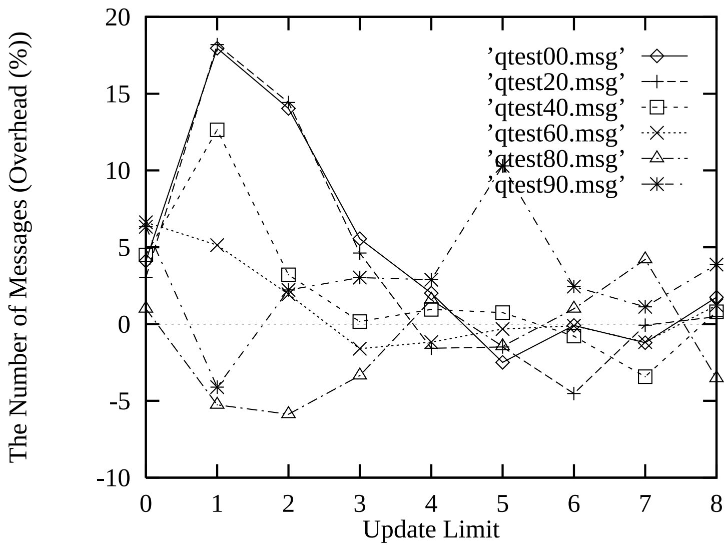


Fig. 39. Overhead for qtest: The Number of Messages

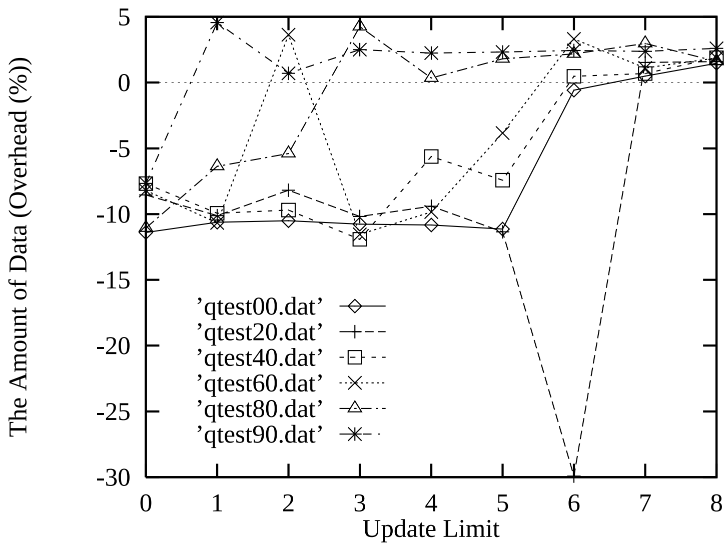


Fig. 40. Overhead for qtest: The Amount of Data Transferred

the back-up node which avoids sending page at the cost of sending update message(s) of small size. The number of messages required for non-recoverable scheme of read ratio 40%, 20%, and 0% (Figures 33, 35, and 37) at update limit 0 is larger than that at update limit 1, because the number of messages for forwarding the page request is larger.

## Results for Other Applications

We now evaluate our recoverable scheme by executing seven additional applications (Floyd-Warshall, SOR, ProdCons, Isort, Reader/Writer, Matmult, and Jacobi) on 8-node workstation cluster. These applications are the same as those used in Chapters II and III. We execute at least 5 times for each application and for each update limit. Appendix B shows experimental results: average number of messages (*Messages*), amount of data transferred (*Data (KBytes)*), standard deviations of these values (*S.D.*), and overhead ratio percentage of recoverable scheme.

Figure 41 through 54 plot costs (the number of messages and the amount data transferred) for non-recoverable scheme and recoverable scheme for each application with different update limits. The ***appl*.msg** curves show the number of messages required for non-recoverable scheme for application ***appl***. The **r*appl*.msg** curves correspond to recoverable scheme. The ***appl*.dat** and **r*appl*.dat** curves correspond to the amount of data transferred. Figures 55 and 56 plot overhead ratio (%) for recoverable schemes. The ***appl*.msg** and ***appl*.dat** curves show the overhead for recoverable scheme in application ***appl*** in terms of the number of messages and the amount of data transferred, respectively. For *Jacobi*, overhead of recoverable scheme is small because all of updated shared data is used in other nodes. For *ProdCons Isort*, and *Reader/Writer*, overhead of the recoverable scheme is relatively small because the size of shared data updated in each node is relatively small in many cases. For

*SOR* and *Matmult*, only a small part (or no part) of updated shared data is used in other nodes, which makes cost for non-recoverable scheme small, however, cost for recoverable scheme large. In particular, the overhead for recoverable scheme in *SOR* is very high: up to 1,300 % for the number of message and 2,000 % for the amount data transferred (not shown in Figure 55 or 56). For *Floyd-Warshall*, updated shared data is not used (immediately) in other nodes after the modification, which makes overhead of recoverable scheme large.

## D. Summary

This chapter presents a scheme to implement a software DSM that is recoverable in the presence of a single node failure. Our scheme differs from the previous work in that the proposed scheme is based on the *competitive* update protocol, which combines the advantages of invalidate as well as traditional update protocols. Our approach is based on the simple observation that, to make the DSM recoverable from a single failure, it is adequate to ensure that each page has at least two copies at all times. To achieve this we suggest a modification to the basic *competitive* update protocol.

We implemented recoverable DSM by modifying Quarks [10, 33] on a network of workstations. Experimental results indicate that the overhead for the proposed scheme is low for some applications in which a large portion of memory is updated by one node while not used by other nodes.

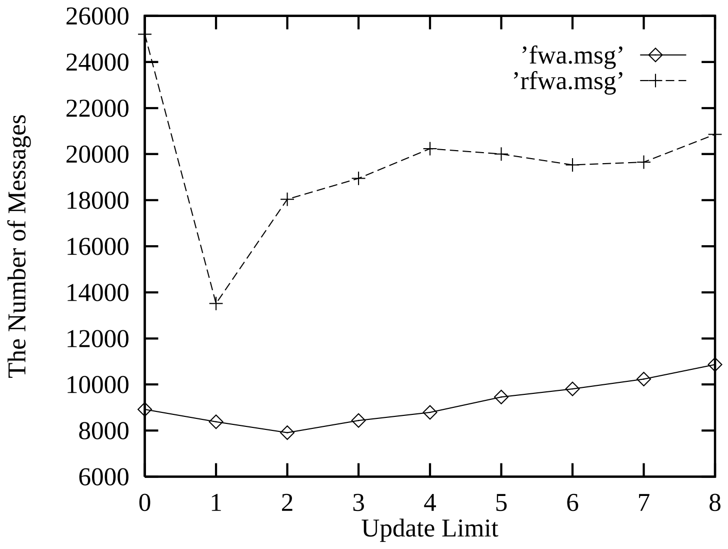


Fig. 41. Floyd-Warshall: The Number of Messages

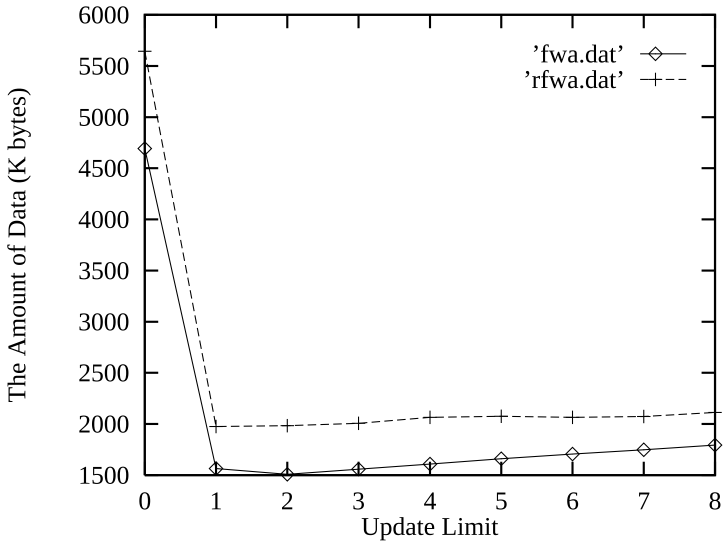


Fig. 42. Floyd-Warshall: The Amount of Data Transferred

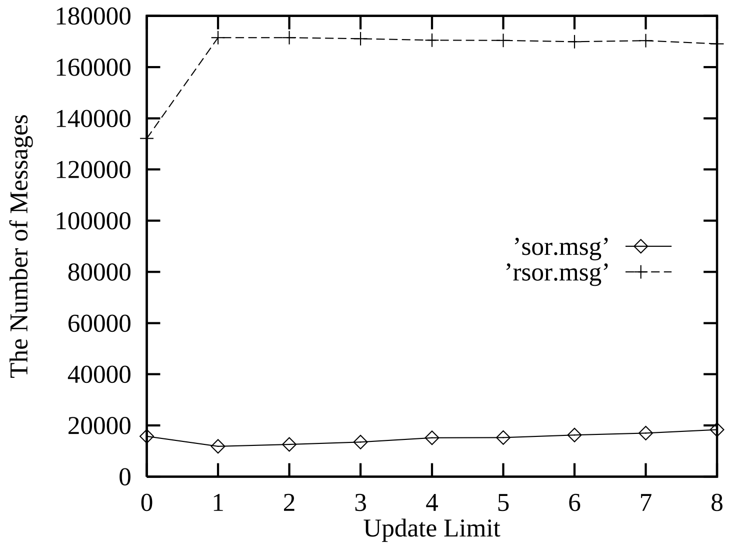


Fig. 43. SOR: The Number of Messages

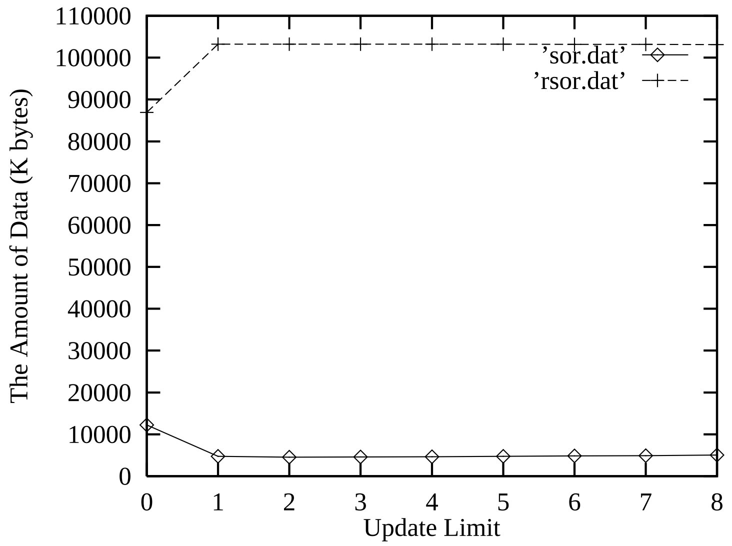


Fig. 44. SOR: The Amount of Data Transferred

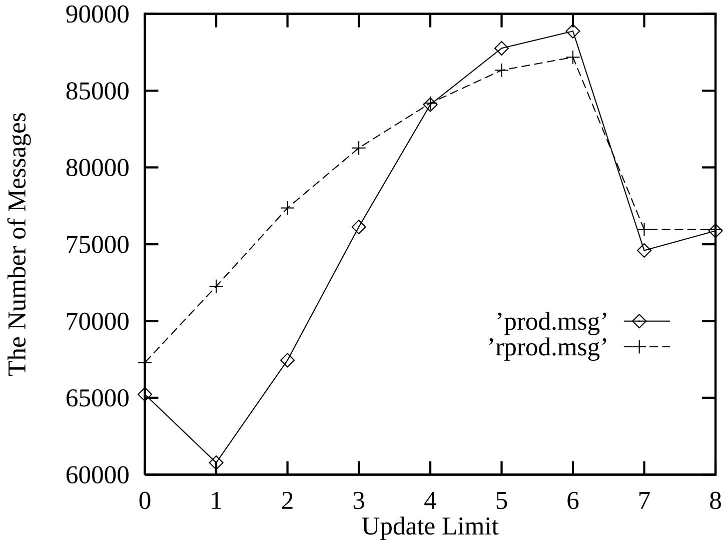


Fig. 45. ProdCons: The Number of Messages

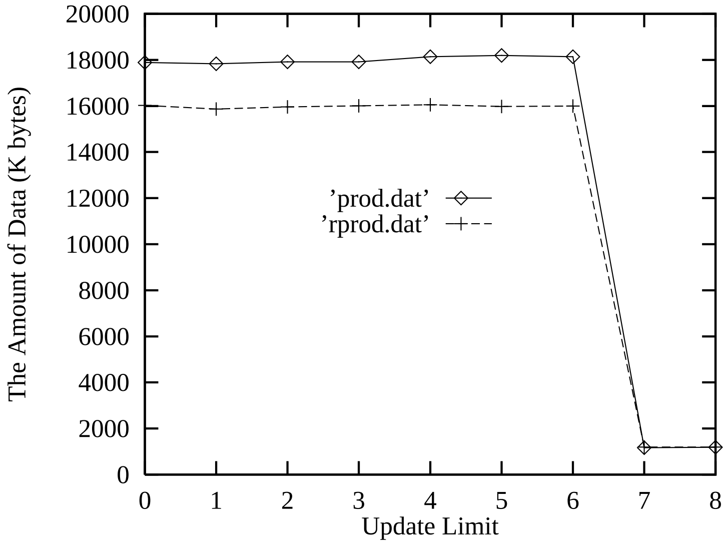


Fig. 46. ProdCons: The Amount of Data Transferred

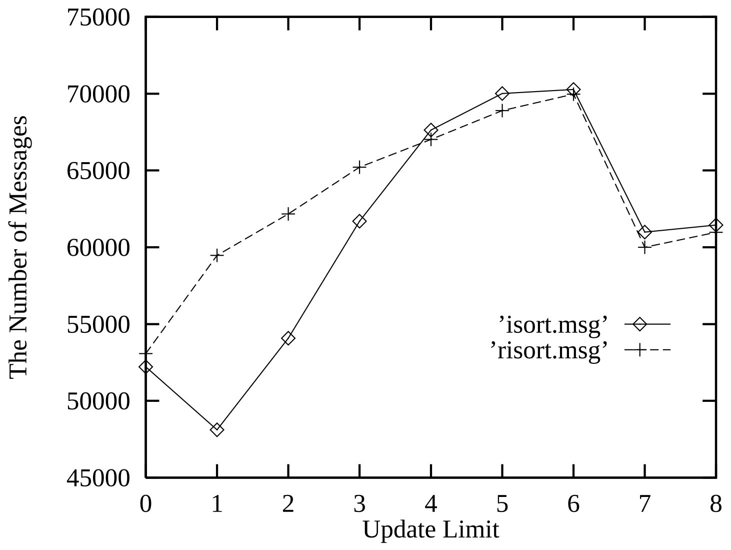


Fig. 47. Isort: The Number of Messages

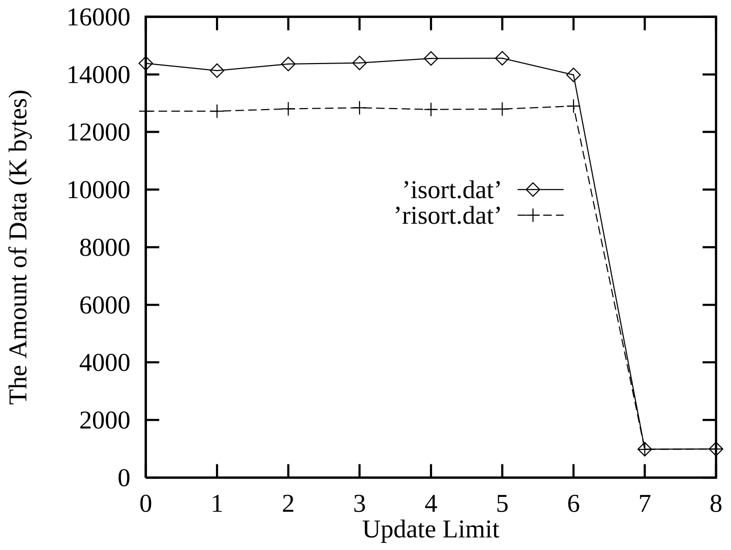


Fig. 48. Isort: The Amount of Data Transferred

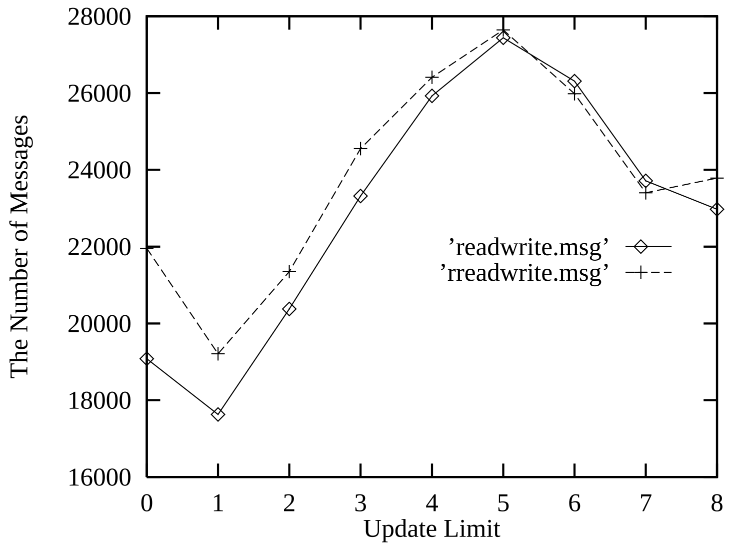


Fig. 49. Reader/Writer: The Number of Messages

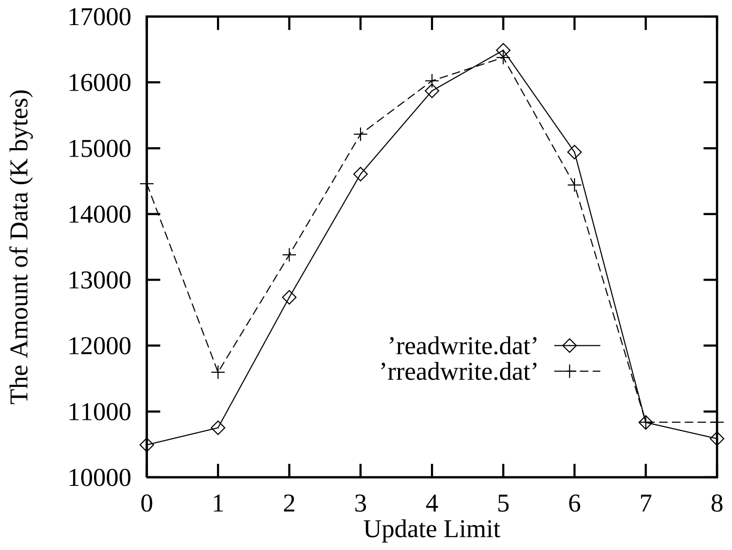


Fig. 50. Reader/Writer: The Amount of Data Transferred

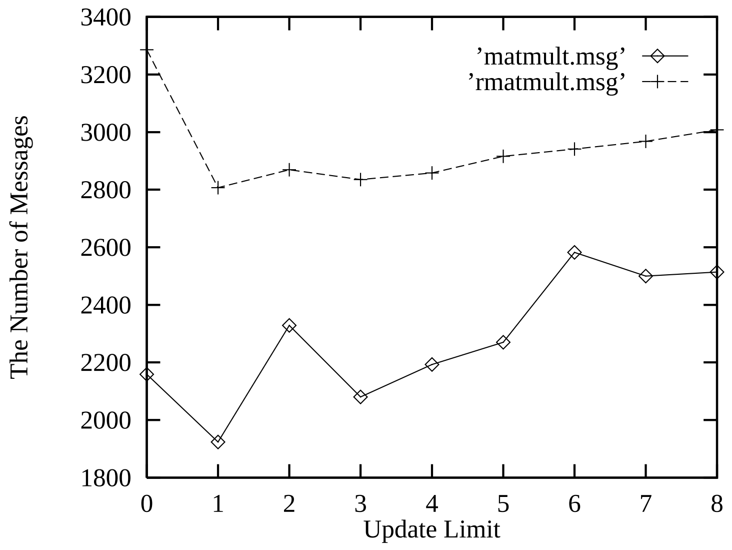


Fig. 51. Matmult: The Number of Messages

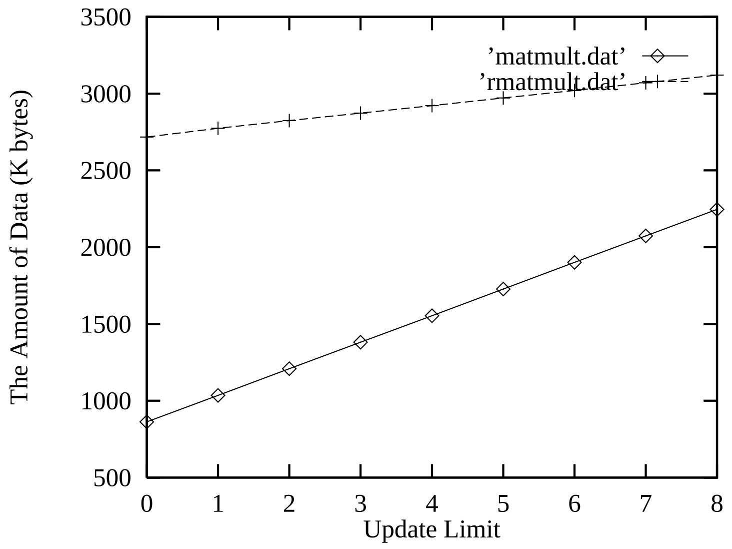


Fig. 52. Matmult: The Amount of Data Transferred

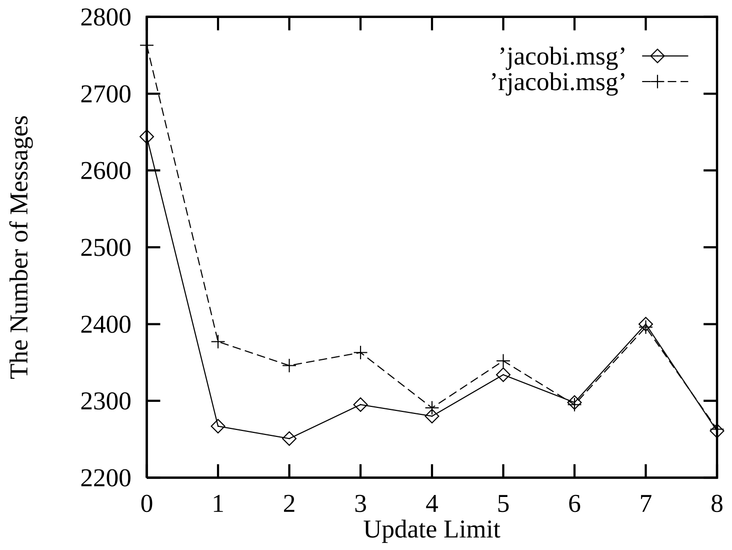


Fig. 53. Jacobi: The Number of Messages

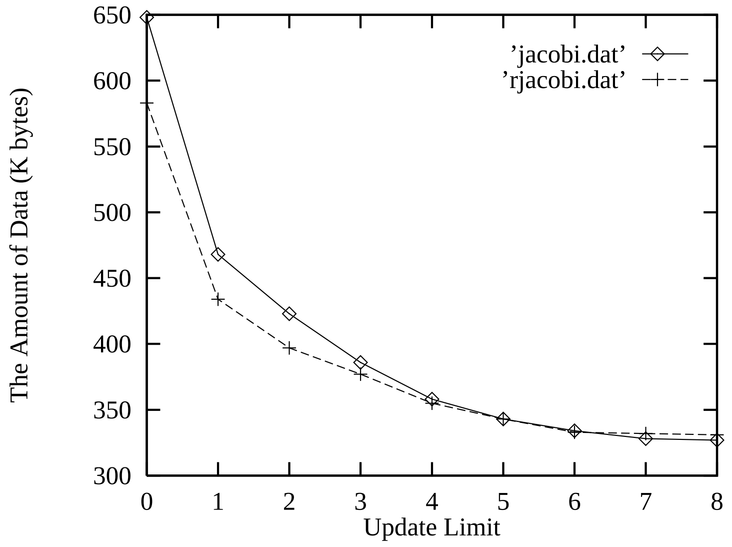


Fig. 54. Jacobi: The Amount of Data Transferred

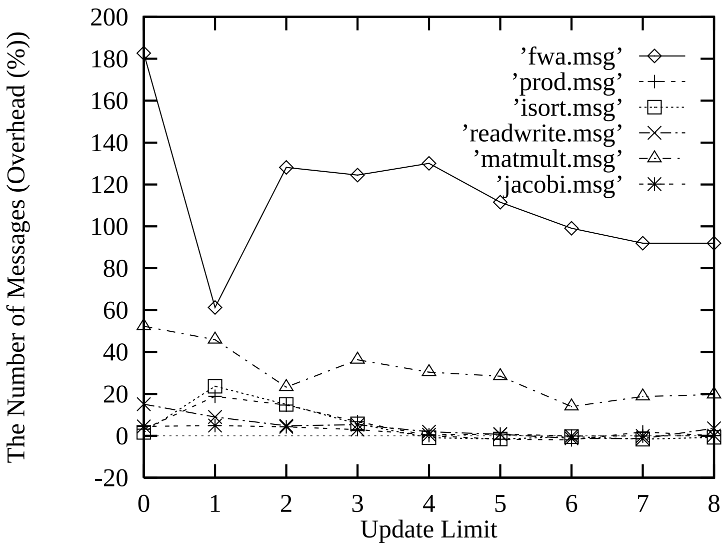


Fig. 55. Overhead for Other Applications: The Number of Messages

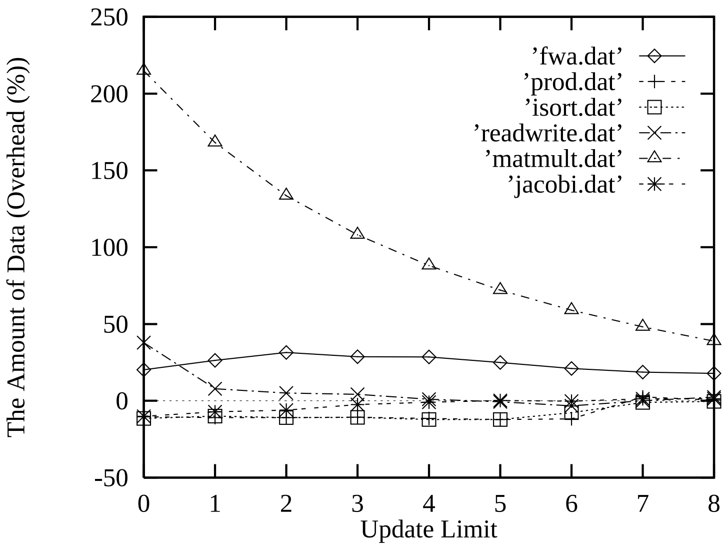


Fig. 56. Overhead for Other Applications: The Amount of Data Transferred

# CHAPTER V

# ANALYSIS OF FAILURE RECOVERY SCHEMES

Checkpoint and rollback recovery is a technique used to minimize the loss of computation when failures occur. A checkpoint is a state of application stored on a stable storage. The application periodically saves checkpoints, and can recover from a failure by rolling back to the checkpoint [65]. When a process rolls back and re-executes from the last checkpoint, the *cost* (loss) incurred by re-doing the lost computation may be larger than that to execute the original computation. In addition to completion time delay, other performance metrics (e.g., user's satisfaction in real-time or on-line transaction applications) may also degrade by unexpected failure and recovery. This chapter determines how *re-do* overhead factor for unexpected execution overhead affects the performance of recoverable DSM.

This chapter analyzes the performance of three recoverable schemes (incorporating *re-do* overhead factor): (1) multiple fault-tolerant scheme using checkpointing and rollback recovery, (2) single fault-tolerant scheme presented in chapter IV, and (3) a two-level scheme [64].

## A. Related Work

For a transaction oriented system, Chandy et al. [13] measure the time to *re-do* the transactions arrived during $t$ time units by using *compression factor*. The *compression factor* $c$ is given by:

$$c = \frac{\mu}{b},$$

where $\mu$ is arrival rate of transactions and $b$ is re-doing rate. Therefore, time required for re-doing the lost transactions during $t$ time units is $ct$. $c$ is assumed be less than

1 (order of 1/10) because $\mu$ is much smaller than $b$. They assume under-loaded transaction system and do not consider the cost of delaying the newly arrived transactions after failure.

Many researchers present cost analysis for the roll-back recovery scheme [13, 64, 65, 70], roll-forward scheme [52], and replicating data [63]. Some papers analyze cost for recovery schemes using volatile memory [63, 62, 64], while many other papers present recovery scheme using volatile memory [4, 9, 22, 31, 36, 61] without any analysis. Chandy et al. [13], Young [70], and Vaidya [65] present methods to compute optimal checkpoint interval to minimize expected (average) execution time.

Theel and Fleisch analyze the costs for read and write operations, and availability for the boundary-restricted protocol [63]. They also present dynamic boundary-restricted protocol that can change the range (boundaries) of the number of cached copies to reduce operation cost while maintaining desired data availability.

Vaidya [64] presents a two-level recovery scheme that tolerates the more probable failures with low performance overhead, while the less probable failures may possibly incur a higher overhead. By minimizing overhead for the more frequently occurring failure scenarios, the two-level approach can achieve lower performance overhead (on average) as compared to single-level recovery schemes.

## B. Recoverable DSM Schemes

We analyze three recoverable schemes incorporating *re-do* overhead factor. The *cost* required to *re-do* the lost computation after a process rolls back to the last checkpoint or restart may be different from that to execute the original computation. For an example, consider an airline reservation system. If system fails temporarily, then transactions executed during $t$ time units will re-execute. An airline company will lose a lot of money by suspending reservation during the $t$ time units. The monetary loss by stopping reservation for $t$ time units may be much more serious than the computational loss of $t$ time units. The three schemes analyzed here are summarized below.

### 1. Multiple Fault-Tolerant Scheme

Many applications require long execution time to finish tasks. Such applications may lose computation if a failure occurs during the execution. Checkpoint and rollback recovery scheme can be used to reduce the loss of computation upon failure. When the application executes on multiple processors, a consistent checkpointing scheme is used to save a global consistent state of the multi-process application [14, 20, 41]

Arbitrary number of failure can be tolerated by rolling the application back to the most recent consistent checkpoint.

### 2. Single Fault-Tolerant Scheme

Our recoverable DSM scheme presented in the Chapter IV is an example of a single fault-tolerant scheme. This scheme maintains at least two copies for each page in a DSM to recover from single-node failure. Additional cost is incurred to maintain at least two copies for each page when a node executes *release* operation. If a single node failure occurs, then the application can recover from the failure without re-executing previously committed computation. However, all processes may have to restart from the initial point of the task if multiple-failure occurs.

### 3. Two-level Recovery Scheme

We consider a two-level recovery scheme obtained by combining *multiple fault-tolerant checkpointing scheme* and *single fault-tolerant scheme* from Chapter IV. With this two

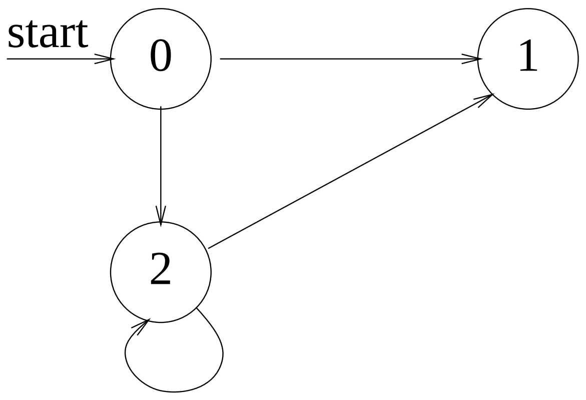


Fig. 57. Markov Chain for a Checkpoint Interval

level scheme, a single-node failure (more probable) can be recovered with the single fault-tolerant scheme, while multiple-node failure (less probable) can be recovered using a global consistent checkpoint.

## C. Performance Analysis

This section presents performance analysis for the three recovery scheme described in the previous section. For this analysis, we incorporate a *re-do* overhead factor ($k$) that is defined as the relative cost of additional computation needed due to failure.

### 1. Multiple Fault-Tolerant Scheme

Expected (average) execution time (denoted as $\Gamma$) of a single checkpoint interval with *re-do* overhead factor $k = 1$ is evaluated in [65, 70]. We consider expected cost $\Gamma$ with *re-do* overhead factor $k > 1$, and analyze optimal checkpoint interval by varying *re-do* overhead factor $k$. Figure 57 shows the same 3-state discrete Markov chain as that presented in [65].

State 0 is the initial state at the start of a checkpoint interval. A transition from state 0 to state 1 occurs if the interval is completed without failure. If a failure occurs, then a transition is made from state 0 to state 2. After state 2 is entered, a transition occurs to state 1 if no further failures before the next checkpoint is taken. If another failure occurs after entering state 2 and before the next checkpoint, then a transition back to state 2 occurs.

Each transition $(X, Y)$, from state $X$ to state $Y$ in the Markov chain, has an associated transition probability $P_{XY}$ and a cost $K_{XY}$ [65]. Cost $K_{XY}$ of a transition $(X, Y)$ is the expected cost spent in state $X$ before making the transition to state $Y$. In this analysis, $T$ is the amount of *useful* computation per checkpoint interval, $C$ is a checkpoint overhead, $R$ is a overhead of a rollback to the checkpoint, and $\lambda$ is the aggregate failure rate of all nodes in the system. (*Useful* computation excludes the cost spent on checkpointing and rollback recovery.) Failures are assumed to be governed by a Poisson process. Note that *re-do* overhead factor $k$ is only included for the computation overhead due to failure(s). Refer to Figure 58 for an illustration – the figure shows the time required for different operations and also their costs.

$$
\begin{aligned}
P_{01} &= e^{-\lambda(T+C)} \\
K_{01} &= T + C \\
P_{02} &= 1 - P_{01} = 1 - e^{-\lambda(T+C)} \\
K_{02} &= \int_0^{T+C} (kt) \frac{\lambda\, e^{-\lambda t}}{1 - e^{-\lambda(T+C)}} dt = k\left(\lambda^{-1} - \frac{(T+C)e^{-\lambda(T+C)}}{1 - e^{-\lambda(T+C)}}\right) \\
P_{21} &= e^{-\lambda(R+T+C)} \\
K_{21} &= k\,R + T + C \\
P_{22} &= 1 - P_{21} = 1 - e^{-\lambda(R+T+C)} \\
K_{22} &= \int_0^{R+T+C} (kt) \frac{\lambda\, e^{-\lambda t}}{1 - e^{-\lambda(R+T+C)}} dt = k\left(\lambda^{-1} - \frac{(R+T+C)e^{-\lambda(R+T+C)}}{1 - e^{-\lambda(R+T+C)}}\right)
\end{aligned}
$$

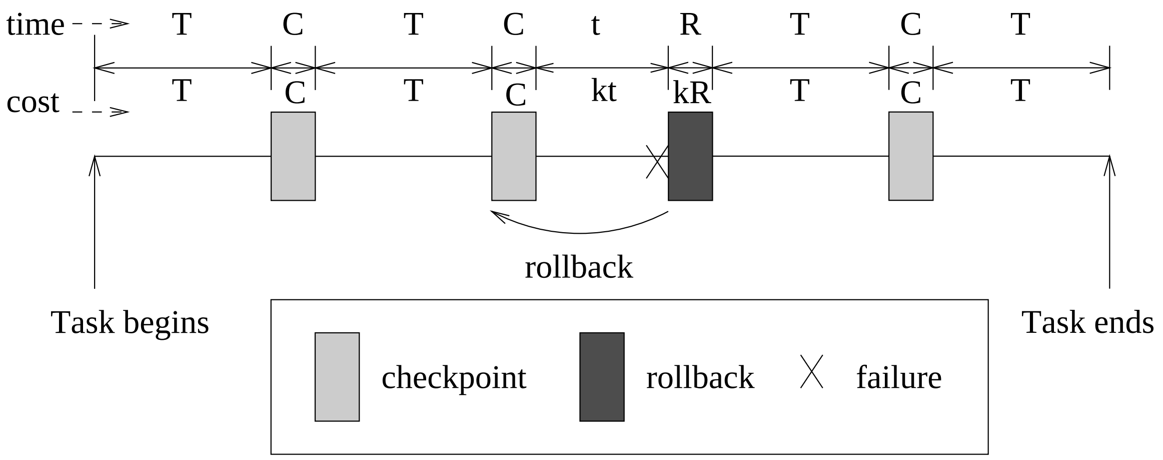


Fig. 58. Checkpoint and Rollback Recovery Scheme

The expected cost, Γ, required to execute one checkpoint interval is the expected cost of a path from state 0 to state 1.

$$\Gamma = P_{01}\,K_{01} + P_{02}\left(K_{02} + \frac{P_{22}}{1-P_{22}}\,K_{22} + K_{21}\right)$$

By substituting and simplification:

$$\Gamma = (1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)} - 1)$$

Let $G(t)$ denote the expected cost required to perform $t$ units of *useful* computation. Then, we define *overhead ratio* $(r)$ as [65]:

$$r = \lim_{t\to\infty} \frac{G(t)-t}{t} = \lim_{t\to\infty} \frac{G(t)}{t} - 1.$$

In this analysis, overhead ratio $(r)$ is given by:

$$r = \frac{\Gamma}{T} - 1$$

To choose an appropriate value of $T$ so as to minimize the overhead ratio $r$, the optimal value of $T$ ($T_{opt}$) must satisfy the following equation:

$$\begin{aligned} \frac{\partial r}{\partial T} &= 0 \\ \Rightarrow \frac{\partial}{\partial T}\left(\frac{(1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)}-1)}{T} - 1\right) &= 0 \end{aligned}$$

By simplification and approximation (as shown in Appendix C):

$$T_{opt} \approx \sqrt{\frac{2C}{\lambda k}}$$

We now present numerical examples:

1. Assume the following parameters: checkpoint overhead ($C$) is 2 time units, recovery overhead ($R$) is 2 time units, failure rate ($\lambda$) is 0.01 per time unit. Figure 59 shows overhead ratio $r$ by varying checkpoint period $T$. Overhead ratio is minimum at $T$ = 18.7, 13.6, 10.0 for $k$ = 1, 2, 4, respectively. The checkpoint period ($T$) that minimizes the overhead ratio $r$ is close to computed approximation value ($T_{opt}$): $T_{opt}$ = 20.0, 14.1, 10.0 for $k$ = 1, 2, 4, respectively.

2. As another example, we assume that failure rate ($\lambda$) is 0.001 per time unit, and the other parameters are the same as the previous example. Figure 60 shows overhead ratio $r$ by varying checkpoint period $T$. Overhead ratio is minimum at $T$ = 61.9, 44.1, 31.4 for $k$ = 1, 2, 4, respectively. The checkpoint period ($T$) that minimizes the overhead ratio $r$ is also close to computed approximation value ($T_{opt}$): $T_{opt}$ = 63.2, 44.7, 31.6 for $k$ = 1, 2, 4, respectively.

We can observe that the optimal checkpoint interval ($T_{opt}$) decreases as *re-do* overhead factor ($k$) increases. This is intuitive, because larger $k$ implies that cost incurred by a failure is high, if checkpoint interval is large.

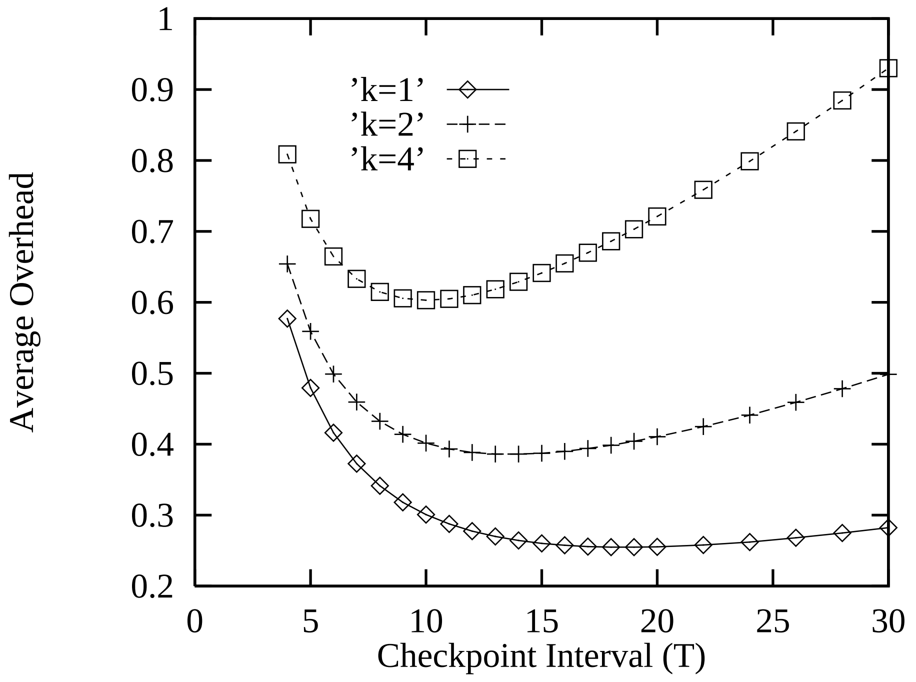


Fig. 59. Re-Do Overhead ($\lambda = 0.01$)

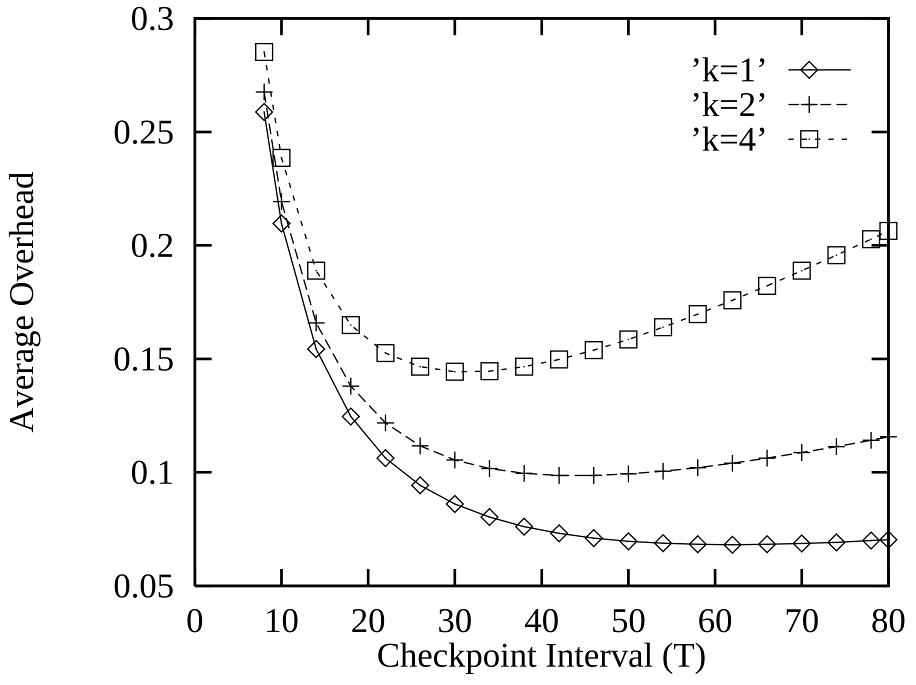


Fig. 60. Re-Do Overhead ($\lambda = 0.001$)

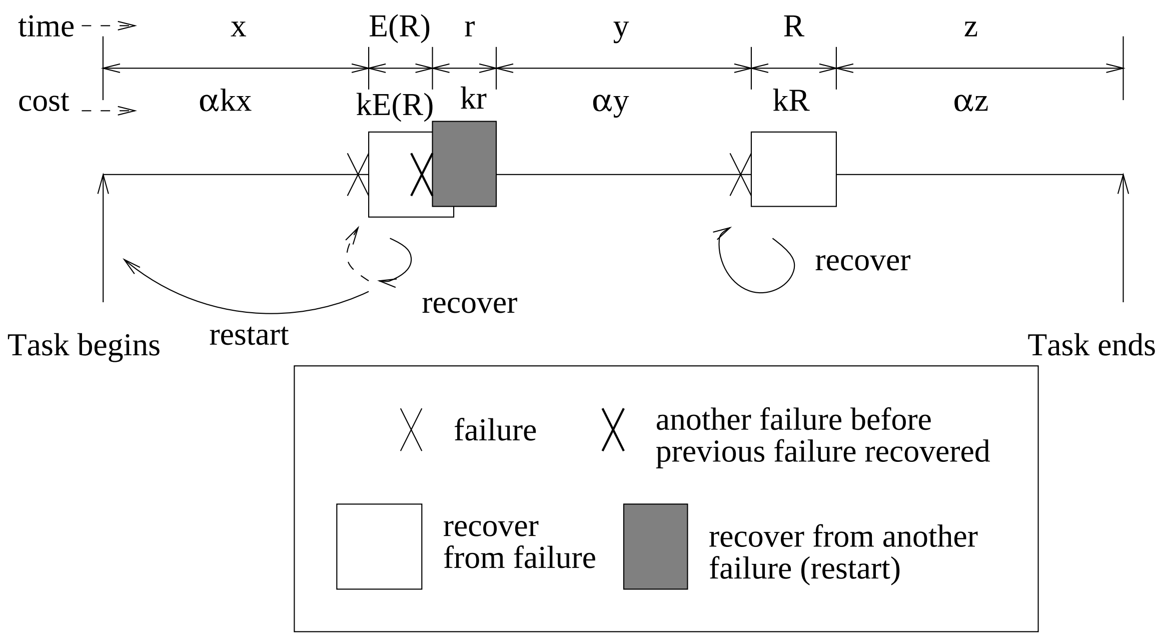


Fig. 61. Single Fault-Tolerant Scheme

## 2. Single Fault-Tolerant Scheme

Our single fault-tolerant scheme presented in the Chapter IV maintains at least two copies for each page to recover from single-node failure. Additional cost is incurred to maintain at least two copies for each page when a node executes *release* operation. To simplify analysis, we assume that the failure-free overhead of the single fault-tolerant scheme to maintain at least two copies for each page is a multiplicative factor $\alpha$. If the failure-free execution time of a task without using a recovery scheme is $\gamma$, then the failure-free execution time of the task using the single fault-tolerant scheme will be $\alpha\gamma$. We also assume that recovery overhead to recover from a single fault is a constant $R$. We assume that if another failure occurs before system recovers from a failure, a restart is required. (This assumption is somewhat pessimistic.) Single fault-tolerant scheme is illustrated in Figure 61 – the figure shows the time and cost required for different operations.

Let $f(\alpha\gamma)$ denote the expected *time* (not cost) required to perform $\gamma$ units of *useful* computation. (We will determine the expected cost later.) Then, $f(t+\varepsilon)-f(t)$ is the time required to perform $\varepsilon$ time units of computation starting from time $t$. Let $E(x)$ denote the amount of time, during an interval of length $x$, before a failure occurs, given that a failure occurs sometime during the interval. Then,

$$E(x) = \int_0^x t\,\frac{\lambda\, e^{-\lambda t}}{1-e^{-\lambda x}}\,dt = \lambda^{-1} - \frac{x\, e^{-\lambda x}}{1-e^{-\lambda x}}$$

Now, consider an interval of length $\varepsilon$, starting at time $t$. There are three cases:

1. No fault occur during $\varepsilon$ (probability is $e^{-\lambda\varepsilon}$): $\varepsilon$ time units are required to complete the interval.

2. A fault occurs during $\varepsilon$, say at $t+\varepsilon_1$ ($\varepsilon_1 < \varepsilon$), but the fault is recovered without another fault during the recovery time $R$ (probability is $(1-e^{-\lambda\varepsilon})\,e^{-\lambda R}$): After that, (1) if the task proceeds after the recovery without any fault until $t+\varepsilon$ (probability is $e^{-\lambda\,(\varepsilon-\varepsilon_1)}$), then $R+\varepsilon$ time units are required to complete the interval; (2) if another fault occurs after recovery (probability is $1-e^{-\lambda(\varepsilon-\varepsilon_1)}$), some additional time required to complete the interval – let this time be denoted as $h(\varepsilon-\varepsilon_1)$. Clearly, $h(\varepsilon-\varepsilon_1)$ will approach 0 as $\varepsilon$ approaches 0.

3. A fault occurs during $\varepsilon$, at $t+\varepsilon_1$ ($\varepsilon_1 < \varepsilon$), and another fault occurs at $t+\varepsilon_1+\varepsilon_2$ ($\varepsilon_2 \leq R$) before recovering from the first failure (probability is $(1-e^{-\lambda\varepsilon})\,(1-e^{-\lambda R})$): In this case, $E(\varepsilon)+E(R)+r+f(t)+(f(t+\varepsilon)-f(t))$ time units are required to complete the interval ($E(\varepsilon)+f(t)$ is required for *re-doing* the lost computation, $E(R)$ for recovering from the first failure, $r$ for recovering from the second failure (restart) – we assume that $r=0$, and $f(t+\varepsilon)-f(t)$ for original computation).

Thus, $f(t+\varepsilon) - f(t)$ is obtained as:

$$\begin{aligned}
f(t+\varepsilon) - f(t) &= e^{-\lambda\varepsilon}\varepsilon \\
&+ (1-e^{-\lambda\varepsilon})\, e^{-\lambda R}\Big[e^{-\lambda(\varepsilon-\varepsilon_1)}\,(R+\varepsilon) \\
&\quad +(1-e^{-\lambda(\varepsilon-\varepsilon_1)})\,(R+\varepsilon+h(\varepsilon-\varepsilon_1))\Big] \\
&+ (1-e^{-\lambda\varepsilon})\,(1-e^{-\lambda R})\;[E(\varepsilon)+E(R)+f(t)+f(t+\varepsilon)-f(t)]
\end{aligned}$$

On simplification and taking a limit as $\varepsilon$ approaches 0, we obtain:

$$\begin{aligned}
\lim_{\varepsilon\to 0}\left[\frac{f(t+\varepsilon)-f(t)}{\varepsilon}\right] &= 1+\lambda\, e^{-\lambda R}\,R+\lambda\,(1-e^{-\lambda R})\,E(R)+\lambda\,(1-e^{-\lambda R})\,f(t) \\
\Rightarrow \frac{\partial f(t)}{\partial t} &= A+B\,f(t),
\end{aligned}$$

where $A = 1+\lambda\, e^{-\lambda R}\,R+\lambda\,(1-e^{-\lambda R})\,E(R)$ and $B = \lambda\,(1-e^{-\lambda R})$.

By calculus:

$$f(t) = \frac{A}{B}\,e^{B\,t} - \frac{A}{B}.$$

Now, we determine the average cost of executing $t$ units of useful computation. Observe that, out of $f(\alpha\,\gamma)$, average time spent on unexpected execution (due to failure) is $f(\alpha\,\gamma)-\alpha\,\gamma$. Thus, the average cost due to *re-do* is $k\,(f(\alpha\,\gamma)-\alpha\,\gamma)$. Therefore, the average cost required to perform $\gamma$ units of *useful* computation, denoted as $g(\alpha\,\gamma)$, is:

$$\begin{aligned}
g(\alpha\gamma) &= \alpha\gamma + k\,(f(\alpha\,\gamma)-\alpha\,\gamma) \\
\Rightarrow g(\alpha\gamma) &= \alpha\gamma\,(1-k)+k\,f(\alpha\,\gamma) \\
\Rightarrow g(\alpha\gamma) &= \alpha\gamma\,(1-k)+k\,\left(\frac{A}{B}\,e^{B\,\alpha\,\gamma}-\frac{A}{B}\right)
\end{aligned}$$

The *average overhead* is evaluated as a fraction of $\gamma$ (task length).

$$r = \frac{g(\alpha\gamma)}{\gamma} - 1$$

$$\Rightarrow r \;=\; \alpha(1-k) + \frac{k}{\gamma}\left(\frac{A}{B}e^{B\alpha\gamma} - \frac{A}{B}\right) - 1$$

Figure 62 shows overhead ratio $r$ by varying the failure-free overhead factor ($\alpha$) for the single fault-tolerant scheme. We use short task length ($\gamma = 80$) for this analysis. We assume that recovery overhead ($R$) is 0.6 time units, failure rate ($\lambda$) is 0.01 per time unit. The overhead of single fault-tolerant scheme is lower than that of the checkpoint scheme (shown in the Figure 59) using optimal checkpoint interval, when the failure-free overhead factor ($\alpha$) for single fault-tolerant scheme is less than 1.25 (at $k = 1$), 1.36 (at $k = 2$), and 1.55 (at $k = 4$). Our previous research in Chapter IV shows that the failure-free overhead ($\alpha$) for single fault-tolerant scheme is less than 1.25 in many applications. Another observation is that the single fault-tolerant DSM scheme is not too sensitive to *re-do* overhead factor ($k$), if the task length ($\gamma$) is short and the failure rate ($\lambda$) is not high, while checkpoint scheme is more sensitive to *re-do* overhead factor ($k$). Thus, in particular, our single fault-tolerant scheme is better for high *re-do* overhead factor ($k$), for short tasks.

However, our single fault-tolerant DSM scheme is not good when the task length ($\gamma$) is very long. As the task length ($\gamma$) increases, the probability of rolling back to start point of the task becomes large due to multiple-failure, while the average overhead of checkpoint scheme is essentially independent of the task length ($\gamma$). Figure 63 shows the average overhead by varying task length. We use fixed failure-free overhead ($\alpha = 1.1$). As task length increases, the average overhead increases. *Re-do* overhead factor ($k$) affects the average overhead more, as the task length increases.

Figure 64 shows the average overhead for single fault-tolerant DSM scheme and checkpoint scheme by varying failure rate ($\lambda$). We use $k = 1$, $\alpha = 1.1$, and $\gamma = 80$ for this analysis. For the single fault-tolerant scheme at low failure rate, the average overhead is approximately equal to $\alpha - 1$, because the single fault-tolerant scheme

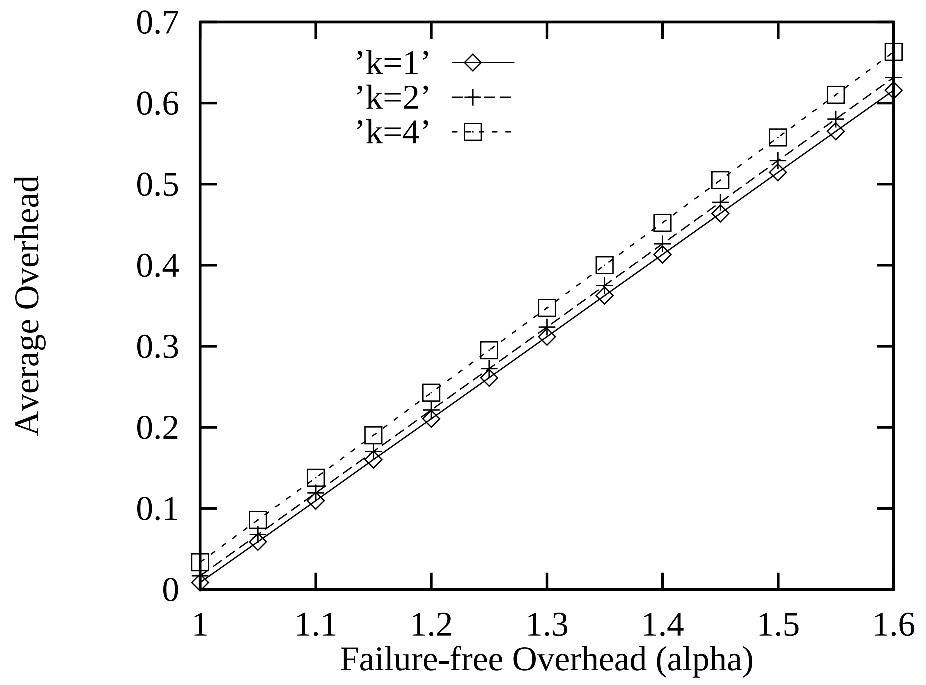


Fig. 62. Overhead by Varying Failure-Free Overhead

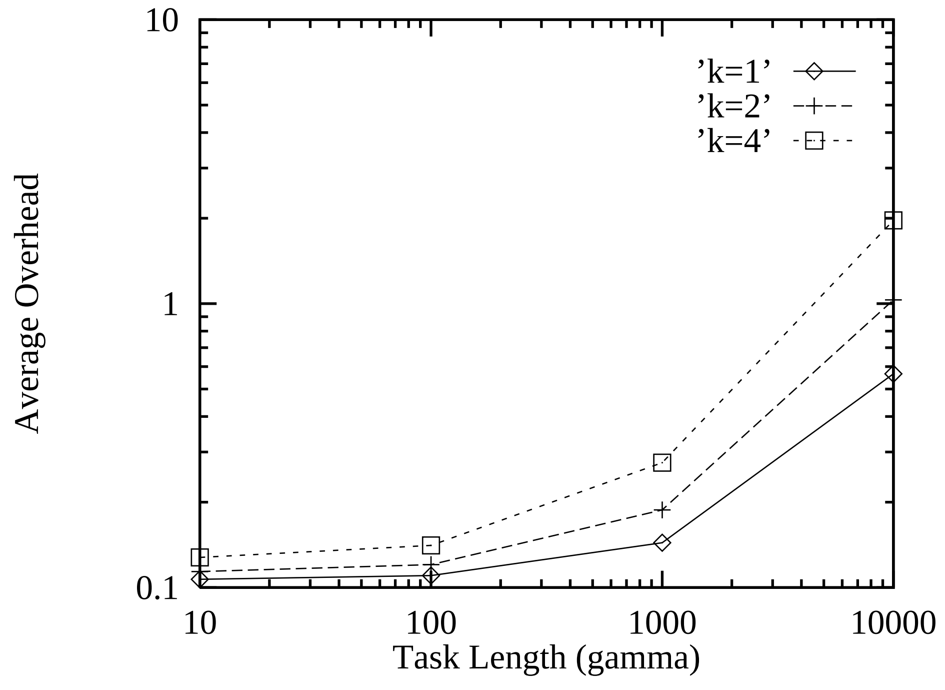


Fig. 63. Overhead by Varying Task Length

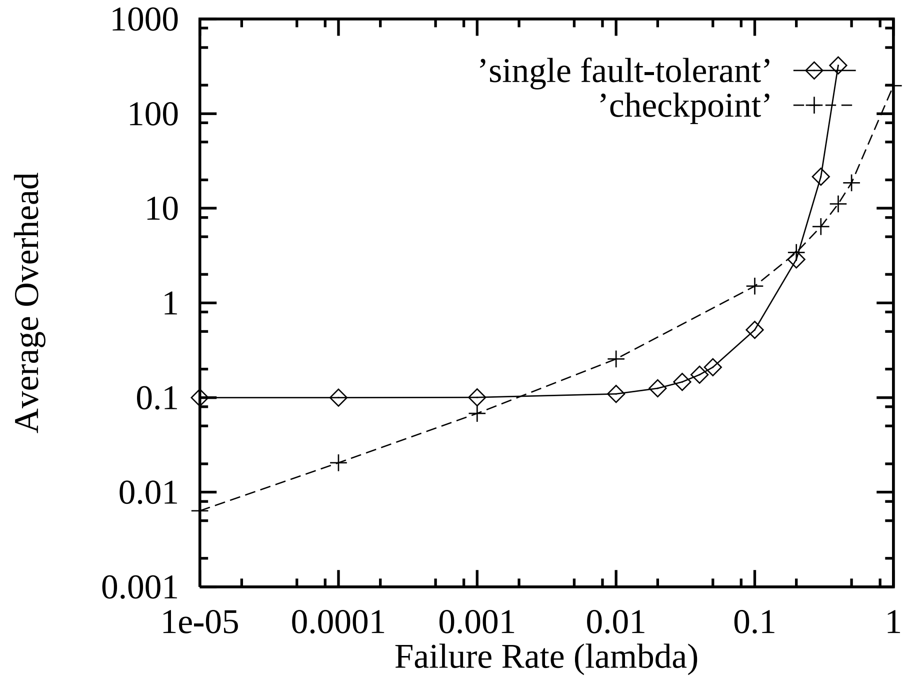


Fig. 64. Single Fault-Tolerant vs. Checkpoint Scheme

is enough to recover most faults. However, the average overhead increases rapidly when the failure rate is high. Overhead of the checkpoint scheme is lower than the single fault-tolerant scheme for small $\lambda$. As the failure rate becomes moderately large, single fault-tolerant scheme performs better. The overhead of the single fault-tolerant scheme is still near $\alpha - 1$ because many of failures can be recovered by the single fault-tolerant scheme, while the overhead of the checkpoint scheme increases more rapidly. At higher failure rate, again checkpoint scheme performs better, because the single fault-tolerant scheme suffers from frequent restarting due to multiple near-simultaneous failures.

In general, the single fault-tolerant scheme performs better for low failure-free overhead ($\alpha$), short task length ($\gamma$), and/or low failure rate ($\lambda$). However, if task length ($\gamma$) is long and/or failure rate ($\lambda$) is very high, then it is highly possible that another failure will occur before recovering from the previous failure. When more

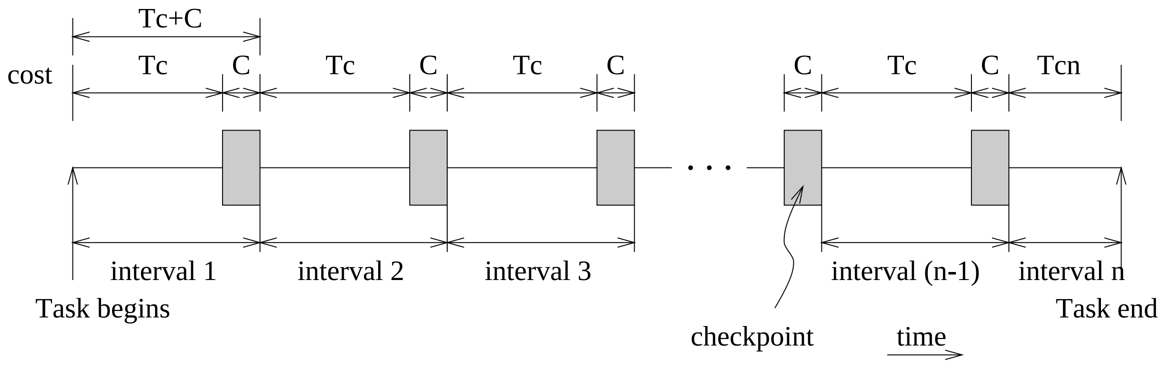


Fig. 65. Failure-free Execution

than one failure occurs, task has to restart from the initial point, because single fault-tolerant scheme can recover single failure only. To solve this problem, we consider a two-level scheme in the next section.

## 3. Two-Level Scheme

The two-level scheme periodically takes checkpoints to allow recovery from arbitrary number of failures. Between checkpoints, the single fault-tolerant scheme is used to allow quick recovery from single failures. Figure 65 illustrates a failure-free execution of two-level scheme. To evaluate expected execution time, we assume that the amount of useful computation in a checkpoint interval is $T_c$, and checkpoint overhead is $C$. Each checkpoint interval ends with a checkpoint except the last interval. There are $\lceil \frac{\gamma}{T_c} - 1 \rceil$ intervals of length $T_c + C$ that requires the cost of $g(T_c + C)$ per interval, and the last interval of length $\gamma - \lceil \frac{\gamma}{T_c} - 1 \rceil T_c$ ($T_{cn}$ in Figure 65) that requires the cost of $g(\gamma - \lceil \frac{\gamma}{T_c} - 1 \rceil T_c)$. Thus, the expected task completion cost ($E(\Gamma)$) is:

$$g(T_c + C) \left\lceil \frac{\gamma}{T_c} - 1 \right\rceil + g\left(\gamma - \left\lceil \frac{\gamma}{T_c} - 1 \right\rceil T_c\right),$$

$g(t)$ is defined in the previous subsection.

As an example, Figure 66 shows the average overhead by using the following

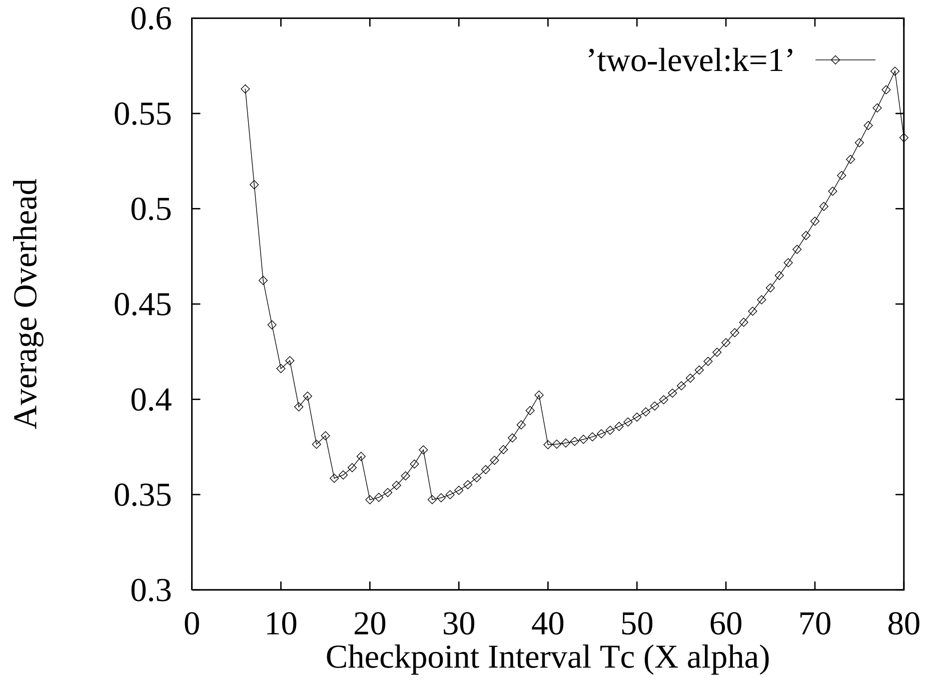


Fig. 66. Minimum Achieved When $T_c = 20 \times 1.1$

parameters: failure-free overhead for single fault-tolerant scheme ($\alpha$) is 1.1, *re-do* overhead factor ($k$) is 1.0, overhead of checkpoint ($C$) is 2.0, rollback overhead by single fault-tolerant scheme ($R$) is 0.6, rollback overhead to checkpoint ($R_c$) is 2.0, failure rate ($\lambda$) is 0.1, length of task ($\gamma$) is 80. From Figure 66, observe that the average overhead is minimized when the checkpoint interval ($T_c$) is $20 \times \alpha = 20 \times 1.1 = 22$.

To compare the performance of two-level scheme with one-level checkpoint scheme, we use very long length of task ($\gamma = 1,000,000$) (other parameter are the same as the previous example). As shown in Figure 67 (**`checkpoint`** denotes checkpointing scheme, and **`two-level:alpha=`**$\alpha$ denotes two-level scheme with the failure-free overhead of $\alpha$ for single fault-tolerant scheme), the minimum overhead of two-level scheme with $\alpha = 2.0$ is comparable with that of one-level checkpoint scheme. In this example, the two-level scheme with $\alpha < 2.0$ is better than one-level checkpoint scheme. The $\alpha_{critical}$ where two schemes require same overhead will decrease as $\lambda$ decreases,

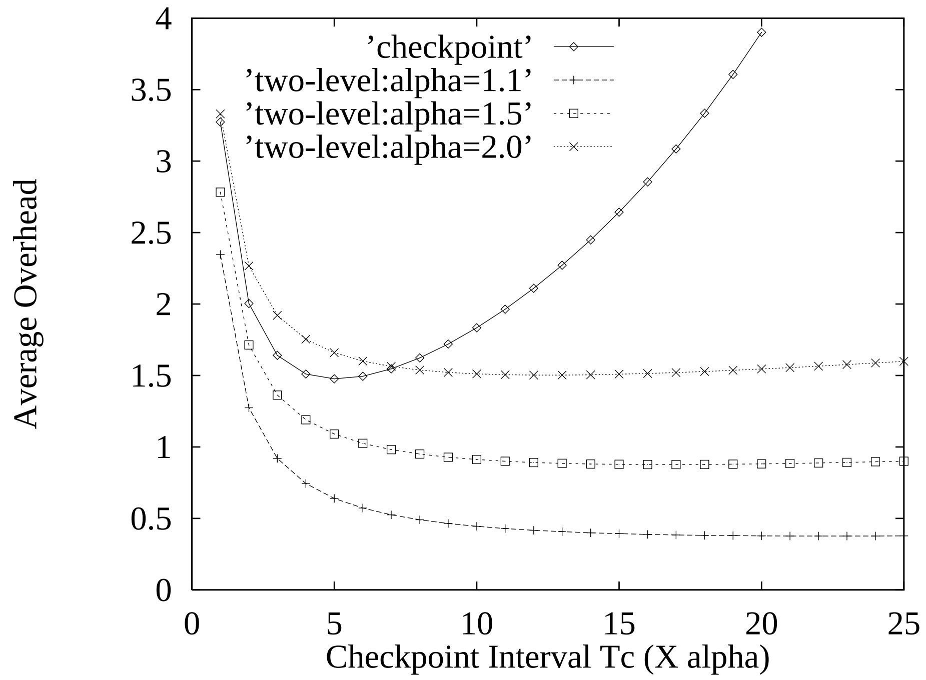


Fig. 67. Minimum Achieved When $T_c = 24.9$

because the overhead of checkpoint scheme decreases. (Figures 59 and 60 imply this.)

Two-level recovery scheme includes single fault-tolerant (one-level) scheme as a special case. When $T_c = \gamma$, the two-level scheme is identical to single fault-tolerant scheme. Two-level scheme can be used when $\gamma$ is long, $\lambda$ is high, and/or $R$ is long, because in these cases it is highly possible that processes using the single fault-tolerant scheme may restart due to multiple-failure (another failure occurring before recovering from a previous failure).

## Optimal Checkpoint Interval

Now, we compute the optimal checkpoint interval of two-level scheme approximately. In our case, the *first-level* recovery scheme is the single fault-tolerant scheme, and the *second-level* scheme is the checkpointing scheme. The failure from the point of view of the checkpointing scheme is the failure that can not be recovered by the *first-level* recovery scheme. If another failure occurs before a failure is recovered, then this is considered as "failure" in the *second-level.* Thus failure rate ($\lambda_2$) of the *second-level* is computed as follows:

$$\begin{aligned} \lambda_2 &= \lim_{\varepsilon \to 0} \tfrac{(1-e^{-\lambda\,\varepsilon})\,(1-e^{-\lambda\,R})}{\varepsilon} \\ \Rightarrow \lambda_2 &= \lim_{\varepsilon \to 0} \tfrac{\lambda\,\varepsilon\,(1-e^{-\lambda\,R})}{\varepsilon} \\ \Rightarrow \lambda_2 &= \lambda\,(1 - e^{-\lambda R}). \end{aligned}$$

This is an approximation because the second-level failures are not exponentially distributed. Thus, an approximation of the optimal checkpoint interval for the two-level scheme is (obtained using equation for $T_{opt}$, and assuming that second level failures as exponentially distributed with mean interval $\frac{1}{\lambda_2}$:

$$T_{opt2} = \sqrt{\frac{2\,C}{\lambda_2\,k}} = \sqrt{\frac{2\,C}{\lambda\,(1 - e^{-\lambda R})\,k}}$$

As an example, in Figure 67 the overhead of two-level scheme is minimum at $T_c = 24.9$ for all $\alpha$. The computed optimal checkpoint interval ($T_{opt2}$) for two-level scheme is:

$$T_{opt2} = \sqrt{\frac{2\,C}{\lambda_2\,k}} = \sqrt{\frac{2\,C}{\lambda\,(1 - e^{-\lambda R})\,k}} = \sqrt{\frac{2 \times 2}{0.1 \times (1 - e^{-0.1 \times 0.6})}} = 26.2$$

$T_{opt2} = 26.2$ is close to the actual optimal checkpoint interval (24.9).

## D. Summary

This chapter evaluates how *re-do* overhead factor ($k$) affects the cost of recoverable DSM. We also analyze optimal checkpoint interval by varying the *re-do* overhead factor ($k$). We analyze and compare the performance of three recoverable schemes (multiple fault-tolerant scheme, single fault-tolerant scheme, and two-level scheme) incorporating the *re-do* factor.

In general, single fault-tolerant scheme presented in Chapter IV is better for low failure-free overhead ($\alpha$), short task length ($\gamma$), and/or moderate failure rate ($\lambda$). However, if task length ($\gamma$) is long and/or failure rate ($\lambda$) is high, then it is highly possible that another failure will occur before recovering from a previous failure. When more than one failure occurs, task has to restart from the initial point, because single fault-tolerant scheme can recover single failure only. To solve this problem, we can use the two-level recovery scheme. The two-level scheme tends to perform better than the checkpointing scheme unless the failure-free overhead ($\alpha$) is large and/or the failure rate ($\lambda$) is very small.

# CHAPTER VI

# A COST MODEL FOR DISTRIBUTED SHARED MEMORY USING COMPETITIVE UPDATE

Selecting appropriate update limit $L$ is important for the *competitive* update protocol. However, no study has been done for *analytically* determining appropriate update limit. This chapter presents a new cost analysis model for distributed shared memory (DSM) using competitive update protocol. Using the proposed model, we compute the cost of the *competitive* update protocol for each update limit. This cost function is used to determine the optimal update limit for the *competitive* update protocol. The proposed model is validated by comparing analytical results obtained using the model to experimental results.

## A. Related Work

Many cost analysis models are presented by other researchers. Most existing models use read/write ratio and/or memory access fault ratio as input parameters [8, 7, 32, 60, 63, 58]. References [8, 7] present modeling to predict the number of cache misses.

Stumm and Zhou [60] compare the performance of four basic protocols. They compute the average memory access cost based on parameters: the cost of sending packet, the cost of sending page, the number of nodes, read/write ratio, and probability of fault. Srbljic et al. [58] present similar performance analysis based on data access type: single/multiple-reader single/multiple-writer. Kessler and Livny [32] compare the performance DSM algorithms by varying a synthetic memory access pattern characterized by several parameters. Brorsson and Stenstrom [7] classify shared memory access pattern as read-only/read-write exclusive/shared-by-few/shared-by-many. They characterize producer-consumer and migratory access pattern using this classification, and compute cache miss ratio. They also apply similar method to stationary access pattern [8]. Other related works on cost models is discussed in Chapter II.

Selecting appropriate update limit $L$ is important for the *competitive update* protocol. However, no study has been done for *analytically* determining appropriate update limit. Previous approaches include selecting update limit by guesswork, or experimental evaluation of the application for many different update limits. [23] presents simulation results by varying the update limit for the competitive update protocol. They conclude that the best update limit is difficult or impossible to predict.

## B. Cost Analysis

The cost of a message of size $m$ is denoted as $c(m)$. At first, we consider message cost in a segment in which the number of updates is $U$. 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.

The *average* cost, denoted $C(L,U)$, of a segment with $U$ updates when using competitive update protocol with limit $L$ is:

$$C(L,U) = \begin{cases} U\left(\overline{c(p_{update})} + c(p_{control})\right) & \text{when } U \le L \\ \\ (L+1)\left(\overline{c(p_{update})} + c(p_{control})\right) & \\ +(\overline{F}+1)\, c(p_{control}) + c(p_{page}) & \text{when } U > L. \end{cases}$$

Note that $c(p_{control})$ and $c(p_{page})$ are constants, while $c(p_{update})$ depends on size of the update message, and $\overline{F}$ is the average number of hops a request message takes before reaching a node that has the requested page. Therefore, the above expression uses average value of $c(p_{update})$.

Now, let $p(x)$ denote the probability density function (pdf) of the number of updates $x$ in a segment for a given application. Based on the above cost analysis for a segment with $U$ updates and limit $L$, average cost per segment for the application, denoted $C_{avg}(L, p(x))$, can be obtained as:

$$\begin{aligned} C_{avg}(L, p(x)) &= \left[\overline{c(p_{update})} + c(p_{control})\right] \textstyle\sum_{x=1}^{L} x\, p(x) \\ &+ \left[(L+1)\left(\overline{c(p_{update})} + c(p_{control})\right)\right. \\ &\quad \left. + (\overline{F}+1)\, c(p_{control}) + c(p_{page})\right] \textstyle\sum_{x=L+1}^{\infty} p(x) \end{aligned}$$

Using the average cost per segment for a particular limit $L$, optimal update limit, $L_{optimal}$, can be obtained as the value of $L$ that minimizes the above expression for $C_{avg}(L, p(x))$. More formally,

$$L_{optimal} = \{l \mid min_{L \geq 0}\, C_{avg}(L, p(x)) = C_{avg}(l, p(x))\}$$

Now, in Section C, we illustrate how the above model can be used to determine optimal $L$ for many different probability density functions $p(x)$. Section D compares analytical results with experimental measurements to validate the proposed model.

## C. Application of the Cost Model

We consider several different probability density functions $p(x)$ and plot the average cost $C_{avg}(L, p(x))$ for different values of $L$. These plots can then be used to: (a) observe the impact of $p(x)$ on the $C_{avg}(L, p(x))$ curve, and (b) to determine optimal $L$ for a given $p(x)$. For the illustration, we assume that $c(p_{control}) = 1$, $\overline{c(p_{update})} = 3$, $c(p_{page}) = 10$, and $\overline{F} = 3$.

The figures in this section may be somewhat confusing to read. Therefore, we now provide an explanation to read these figures.

Consider Figure 68. This figure plots two curves: One curve corresponds to a particular probability density function $p(x)$ named `pdf1`. For the `pdf1` curve, the horizontal axis corresponds to the number of remote updates $x$, and vertical axis corresponds to $p(x) \times 100$. For instance, with `pdf1`, there is a 30% chance that the number of updates in a segment is 1.

The second curve (named `cost1`) in Figure 68 plots average cost $C_{avg}(L, p(x))$ as a function of update limit $L$. For the `cost1` curve, the horizontal axis corresponds to update limit $L$, and vertical axis corresponds to the average cost per segment $C_{avg}(L, p(x))$. For instance, in Figure 68, average cost per segment with limit $L = 3$ is 10.

Similarly, Figures 69 through 73 plot five more probability density functions (named `pdf2` through `pdf6`), and the corresponding curves for average cost (named `cost2` through `cost6`, respectively).

These figures provide an interesting way to view the correlation between the update probability density function and the cost function. As noted before, these curves can also be used to determine the optimal update limit. For instance, for `pdf6`, $L = 1$ is optimal. In general, more than one value of $L$ may yield optimal performance – for instance, for `pdf1`, all values of $L$ greater than 4 yield the same cost.

The greatest advantage of using the proposed cost model is that it is only necessary to simulate or execute the application once to estimate the probability density function $p(x)$ – once $p(x)$ is known, the optimal $L$ can be determined analytically. Without this approach, determining optimal $L$ will require multiple simulations (or executions) for different values of $L$.

In Figures 68 through 73, we used the same values of parameters $\overline{c(p_{update})}$ and $c(p_{page})$ for all cases. Next we investigate the impact of the value of these parameters

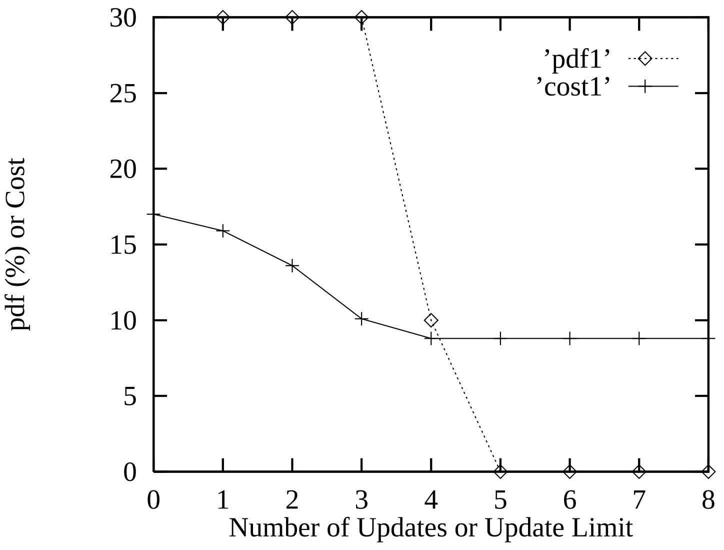


Fig. 68. Cost ($pdf1$)

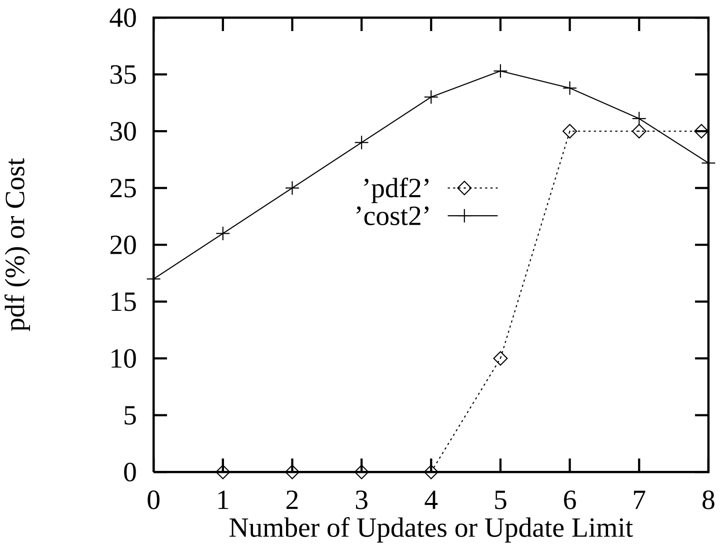


Fig. 69. Cost ($pdf2$)

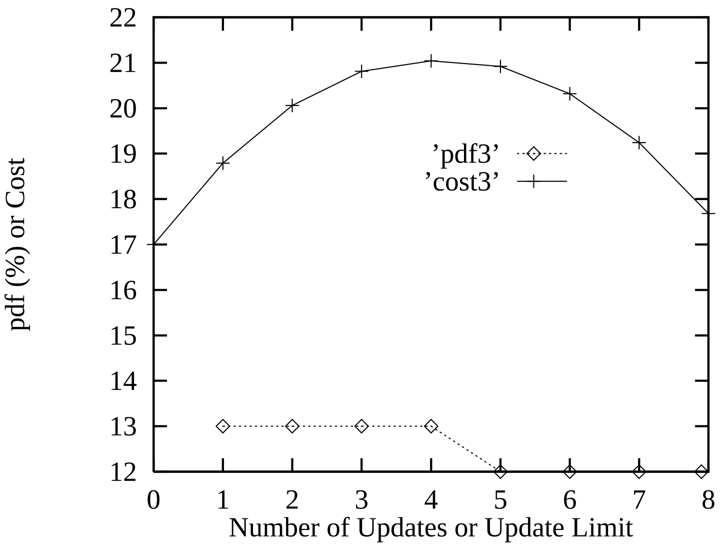


Fig. 70. Cost ($pdf3$)

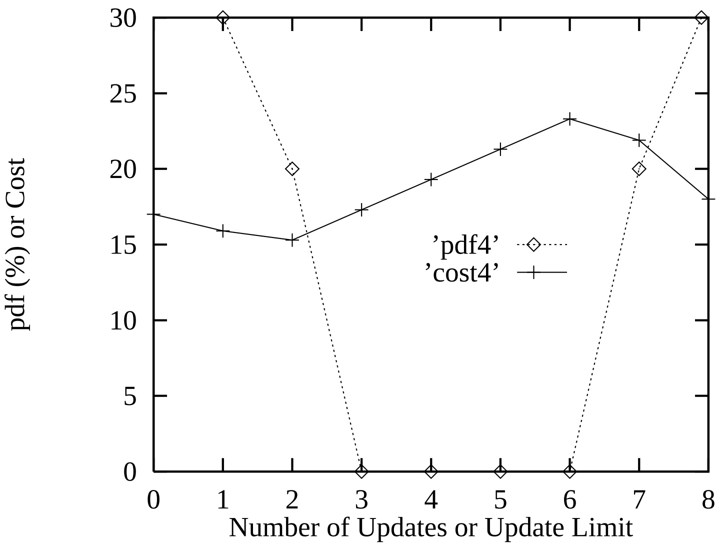


Fig. 71. Cost ($pdf4$)

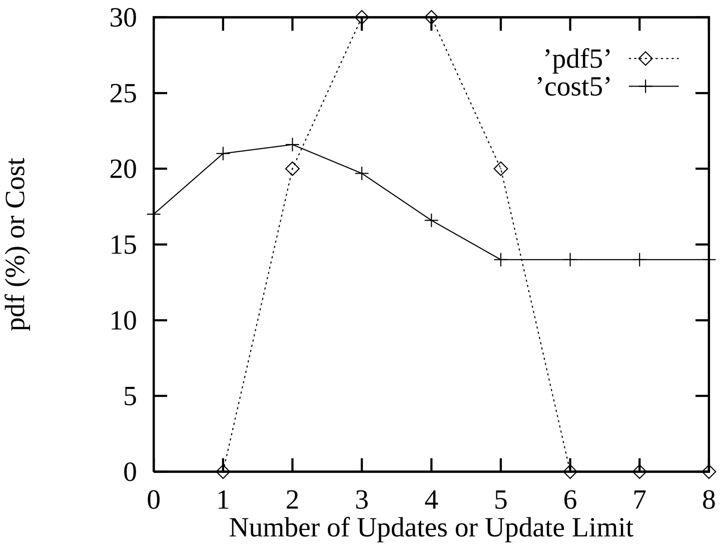


Fig. 72. Cost ($pdf5$)

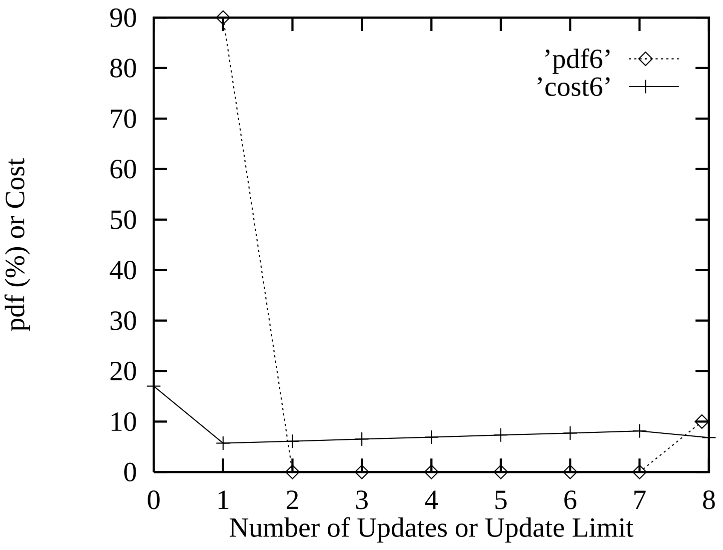


Fig. 73. Cost ($pdf6$)

on the average cost. For this analysis, we use probability density function `pdf3` in Figure 70.

Figure 74 shows how $\overline{c(p_{update})}$ affects the average cost per segment. Here we assume that $c(p_{page}) = 10$ and $\overline{F} = 3$. In Figure 74, the curve labeled `update1` corresponds to the case when $\overline{c(p_{update})} = 1$. Similarly, other curves labeled `update`*i* correspond to the case when $\overline{c(p_{update})} = i$. As the curves show, competitive update protocol with small update limit $L$ is better for large $\overline{c(p_{update})}$, and vice-versa. This is intuitive, because if cost of an update is large, then updates should be avoided (this is achieved by invalidating a page soon by using small $L$).

Figure 75 shows how $c(p_{page})$ affects the average cost per segment. In this case, we assume that $\overline{F} = 3$ and $\overline{c(p_{update})} = 1$. Here, the curve labeled `page`*i* corresponds to the case when $c(p_{page}) = i$ (thus, curve `page3` assumes that $c(p_{page}) = 3$). When $c(p_{page})$ is large, the cost of serving a page fault is high (because a page must be transferred via a message costing $c(p_{page})$). Therefore, with large $c(p_{page})$, a page should not be invalidated too often (to avoid future page faults). Therefore, as one would expect, Figure 75 shows that competitive update protocol with large update limit is better for larger $c(p_{page})$.

## D. Validation of the Proposed Model

To verify the accuracy of our cost analysis model, we compare the average cost estimated by our model with that obtained by experimental measurements. For this validation study, we assume that cost of a message of size $m$ is $m$, that is, $c(m) = m$.

We implemented the competitive update protocol by modifying Quarks DSM (Beta release 0.8) [10, 33]. For this study, we used several real applications (named *Floyd-Warshall*, *Isort*, *Jacobi*, *SOR*)) as well as synthetic applications (named *qtest50*

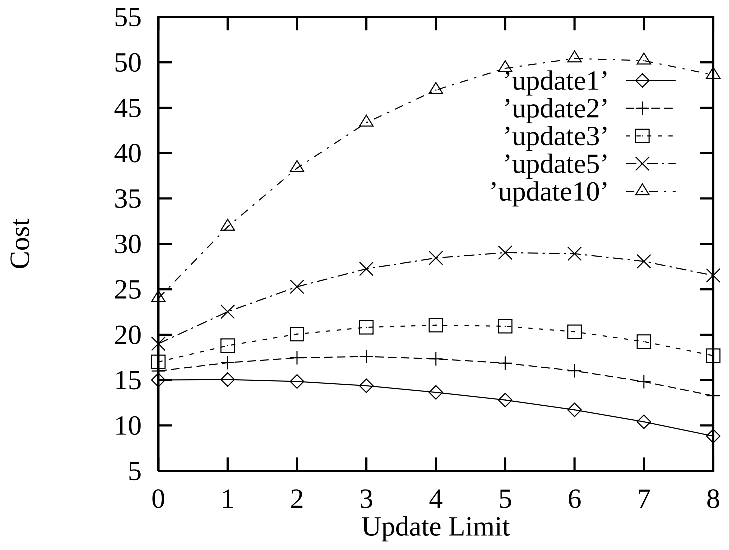


Fig. 74. Cost by Varying $\overline{c(p_{update})}$

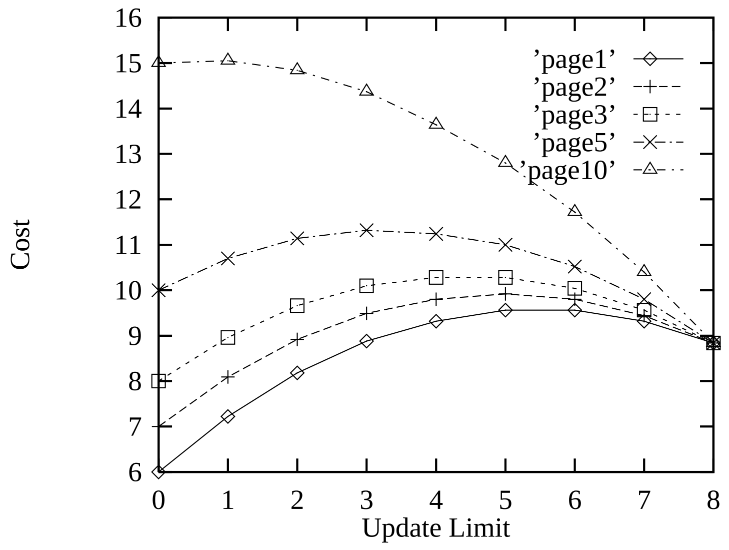


Fig. 75. Cost by Varying $c(p_{page})$

and *qtest10*, i.e., *qtest2* in Chapter II with read ratio 50% and 10%, respectively). The experiments were used to determine: (a) the experimental cost for each update limit $L$, and (b) the probability density function $p(x)$ of the number of updates in a segment. All applications are executed on a 8-node workstation cluster. Using the $p(x)$ measured by experiments, we then computed the average cost per segment analytically. (We use $\overline{F} = 4$ in this analysis.) The objective here is to compare this analytical result with the cost measured experimentally.

In Figures 76 through 81, the curve labeled `pdf` plots $p(x) \times 100$, where $p(x)$ is obtained via experimental measurements. For the `pdf`, the horizontal axis corresponds to number of updates $x$, and vertical axis corresponds to $p(x) \times 100$. $p(x)$ is obtained by keeping track of the number of updates per segment for all pages at all nodes. The `cost.analysis` curves in Figures 76 through 81 plot the analytical values of average cost per segment, as obtained by using our model. The `cost.exp` curve plots the average cost per segment as measured during the experiments. The average is taken over all segments for all pages at all nodes.

In our analysis, we did not consider the messages required for *synchronization*, initialization (when the program starts executing), and false updates (sending update message to a node whose local copy has been already invalidated). Therefore, one would expect a mismatch between `cost.exp` and corresponding `cost.analysis`. However, the difference between these two curves would be relatively independent of the value of $L$. The `cost.adj` (adjusted) curves in Figures 76 through 81 are obtained by subtracting the unaccounted costs from `cost.exp`.

Figures 76 through 81 show that cost estimated by analytical model, `cost.analysis`, is typically close to the adjusted cost obtained by experimentation results, `cost.adj`. In fact, `cost.analysis` and `cost.exp` are also typically close. However, there are significant differences between `cost.analysis` and `cost.exp`, es-

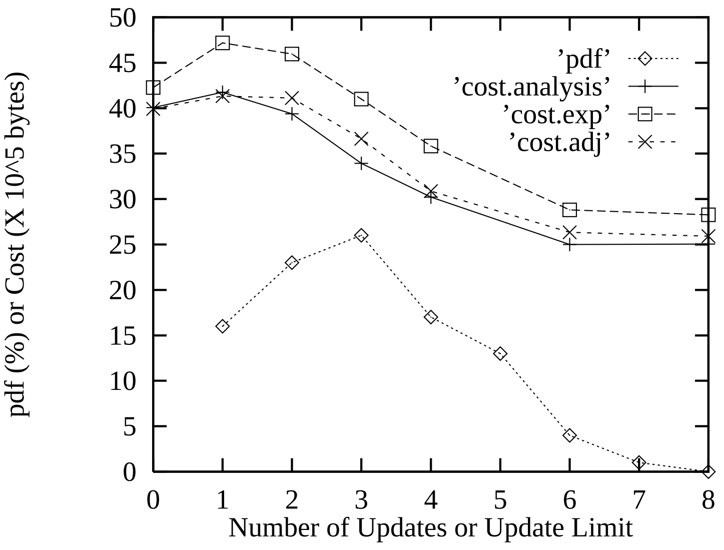


Fig. 76. qtest (read ratio = 50%)

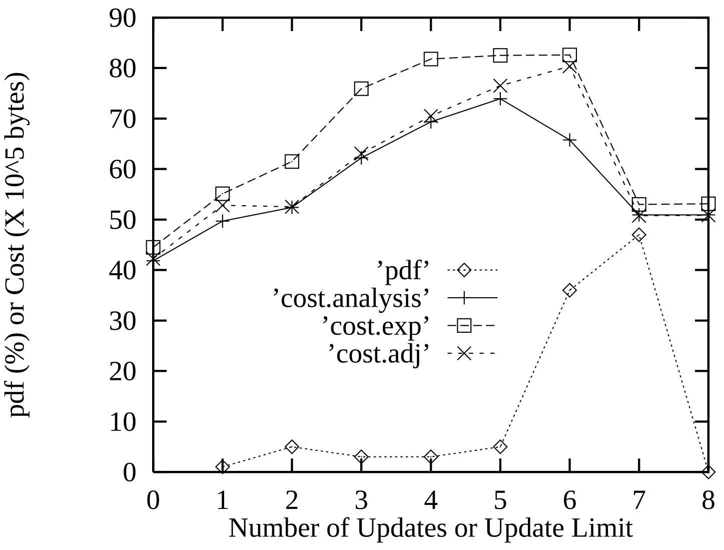


Fig. 77. qtest (read ratio = 10%)

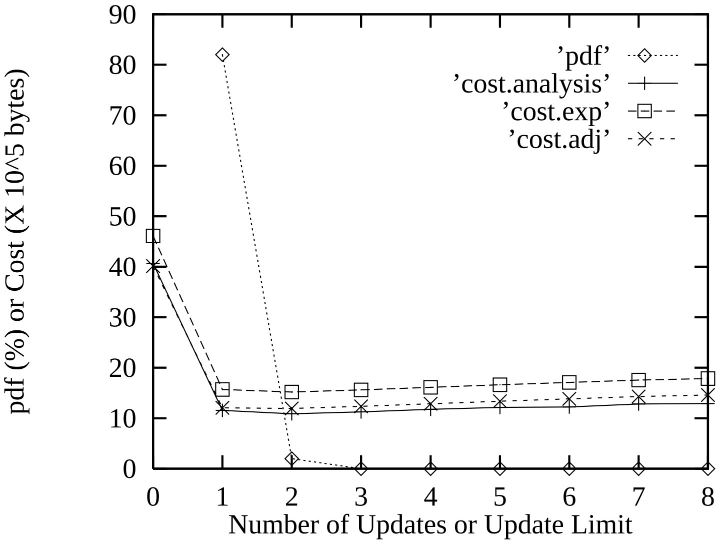


Fig. 78. Floyd-Warshall (size = 128)

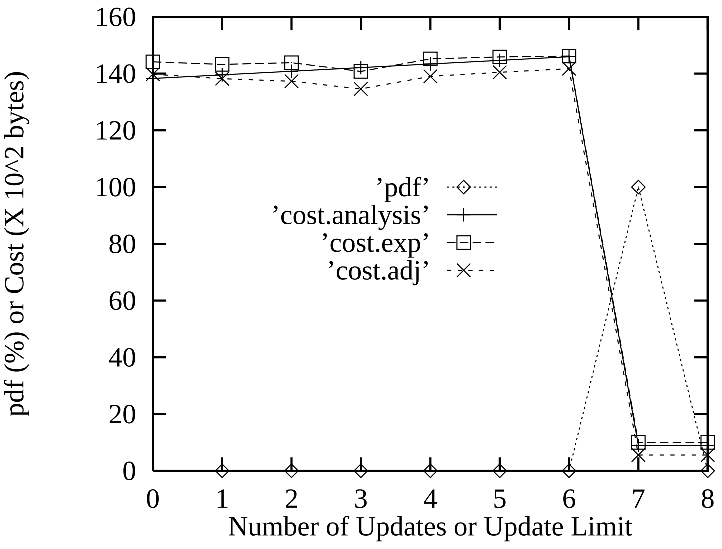


Fig. 79. Isort (size = 3200)

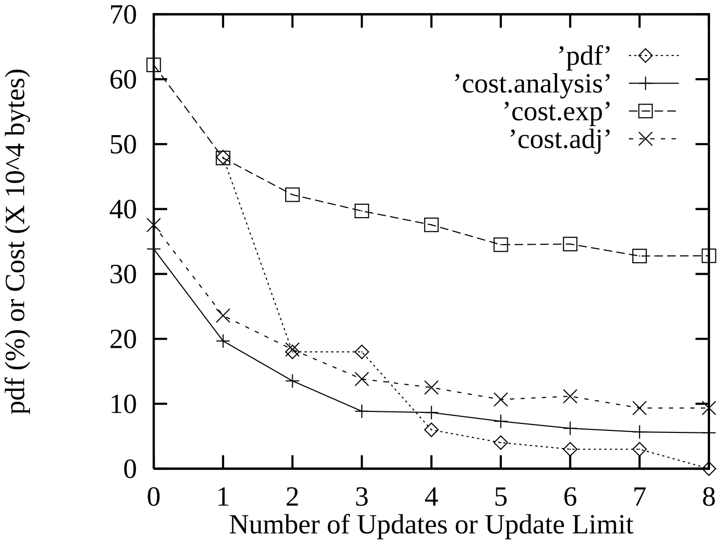


Fig. 80. Jacobi (size = 128)

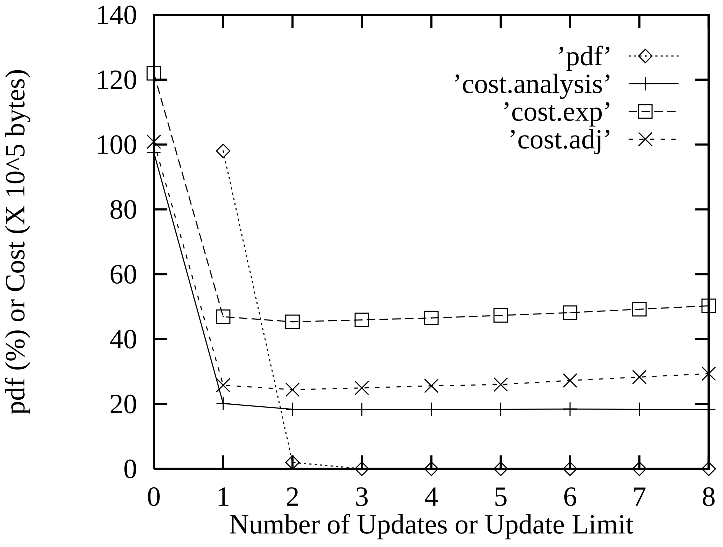


Fig. 81. SOR (size = 512)

pecially in the Figures 80 and 81. As noted before, these differences are due to unaccounted messages, for instance, for *synchronization*, *initialization*, and *false updates*. In case of Jacobi and SOR, many messages are sent initially to initialize the shared data. These messages form a large fraction of all messages sent. Therefore, the error in our model is much more pronounced for these applications, as compared to other applications.

To determine whether the probability density function of the number of updates per segment is stable in different executions, we measure $p(x)$ by executing *qtest* (10% read ratio), *Jacobi*, *Floyd-Warshall*, and *Isort* 5 times with update limit $L = 3$ and $L = 8$. Figures 82 through 89 show the experimental results. The probability density function is stable for *Floyd-Warshall* and *Isort*, relatively stable for *qtest* ($L = 3$) and *Jacobi* ($L = 8$), and less stable for *qtest* ($L = 8$) and *Jacobi* ($L = 3$), on each run.

We also determine whether the probability density function varies much with the update limit, we measured $p(x)$ by executing three applications (*qtest* with 50% read ratio, *Jacobi*, and *SOR*) for different update limits. Figure 90 through 92 show that $p(x)$ of *qtest* is relatively stable, $p(x)$ of *SOR* is very stable, and the $p(x)$ of *Jacobi* is relatively unstable (however, in general, the $p(x)$ of *Jacobi* is decreasing as update limit increases). All 8 nodes share some shared memory space in *qtest* and *Jacobi* applications. The order of shared memory access may be different for each loop in *qtest* application, and the order of nodes reaching the barrier may be different for each loop of *Jacobi* application, which causes $p(x)$ to be unstable. However, only 2 nodes share the shared memory space in *SOR* application which causes $p(x)$ to be stable.

One may expect that the probability density functions will be different when using different input data for some applications. To verify, we measured the probability density functions of the number of updates in a segments with different input data

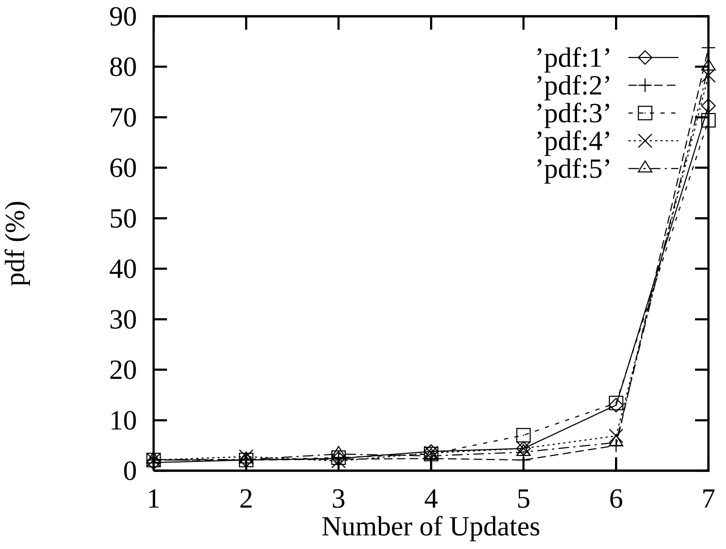


Fig. 82. pdf (qtest: $L = 3$, read ratio = 10 %)

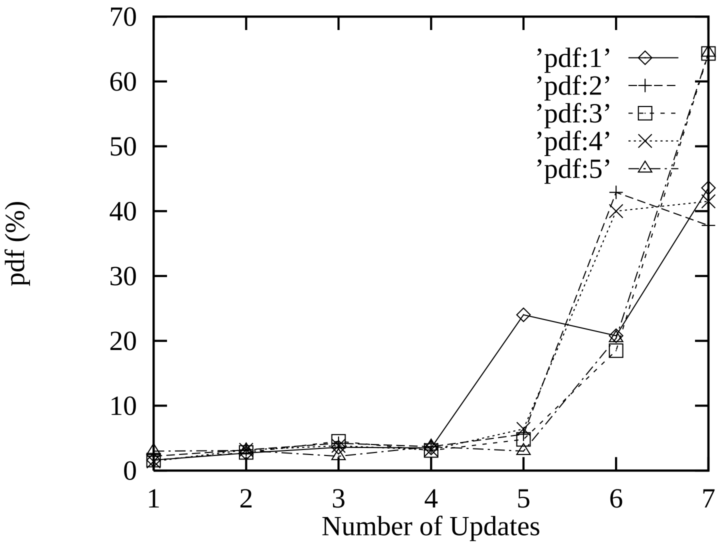


Fig. 83. pdf (qtest: $L = 8$, read ratio = 10 %)

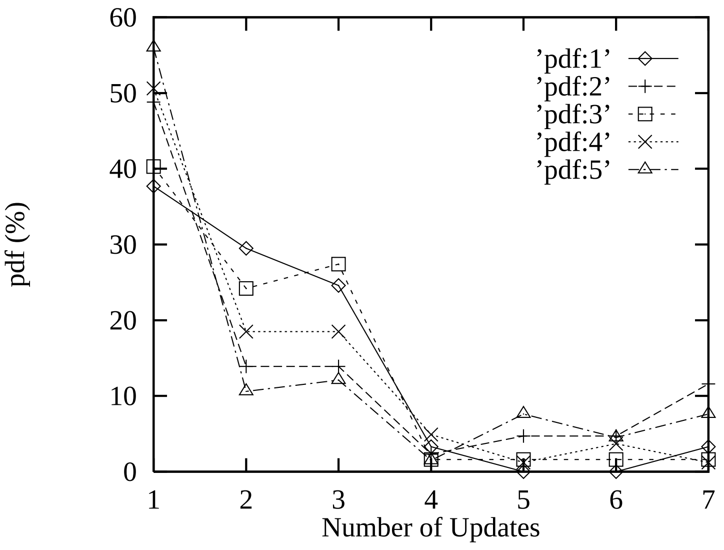


Fig. 84. pdf (Jacobi: $L = 3$, size = 128)

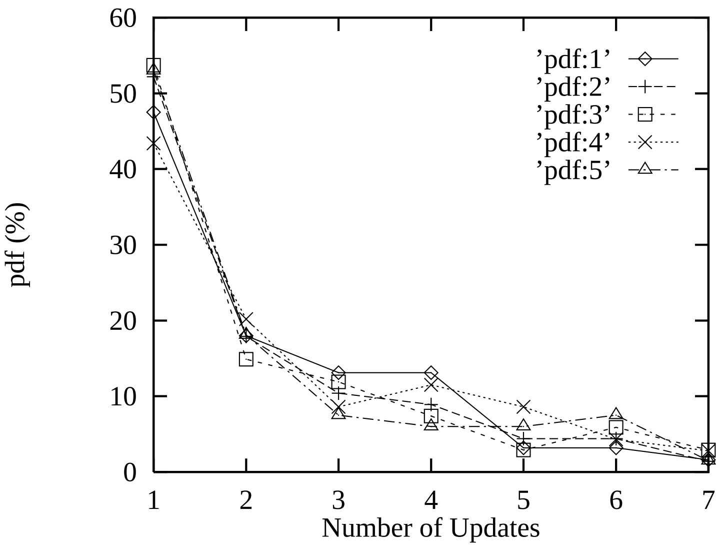


Fig. 85. pdf (Jacobi: $L = 8$, size = 128)

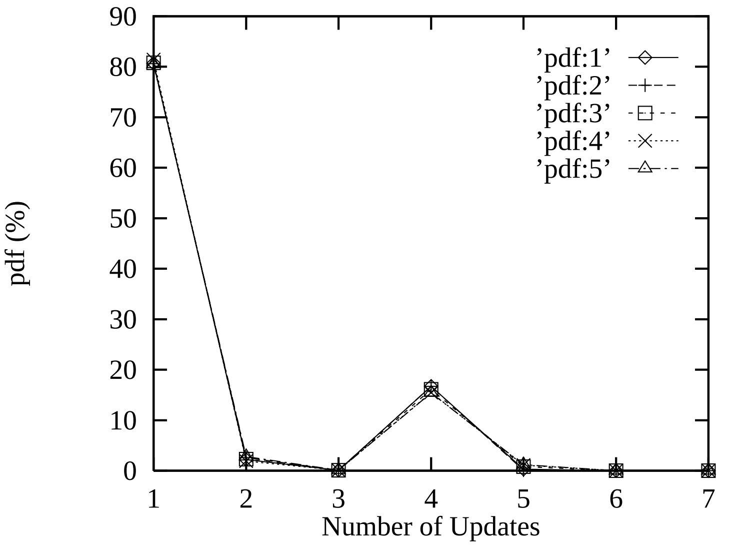


Fig. 86. pdf (Floyd-Warshall: $L = 3$, size = 128)

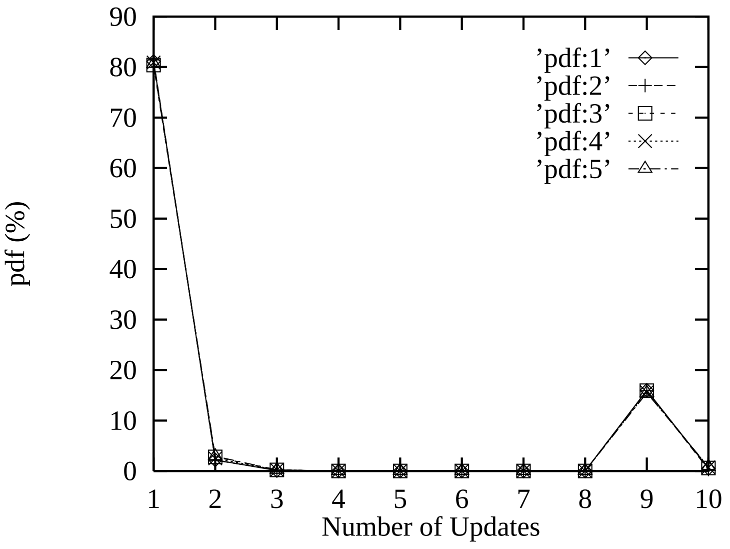


Fig. 87. pdf (Floyd-Warshall: $L = 8$, size = 128)

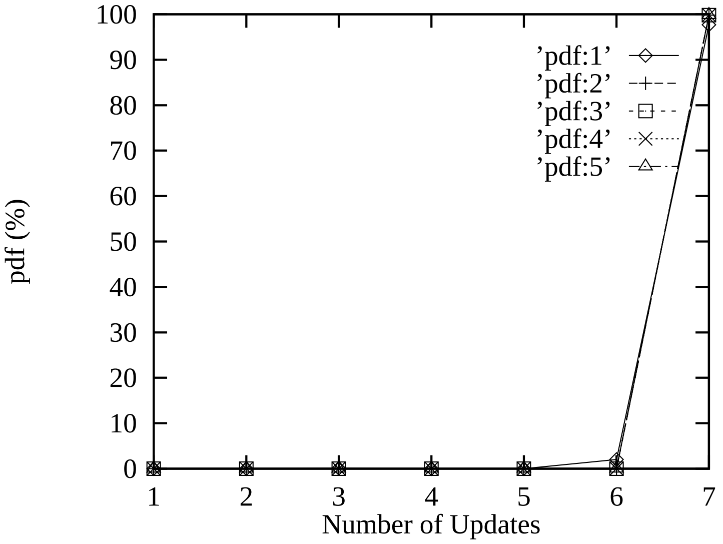


Fig. 88. pdf (Isort: $L = 3$, size = 3200)

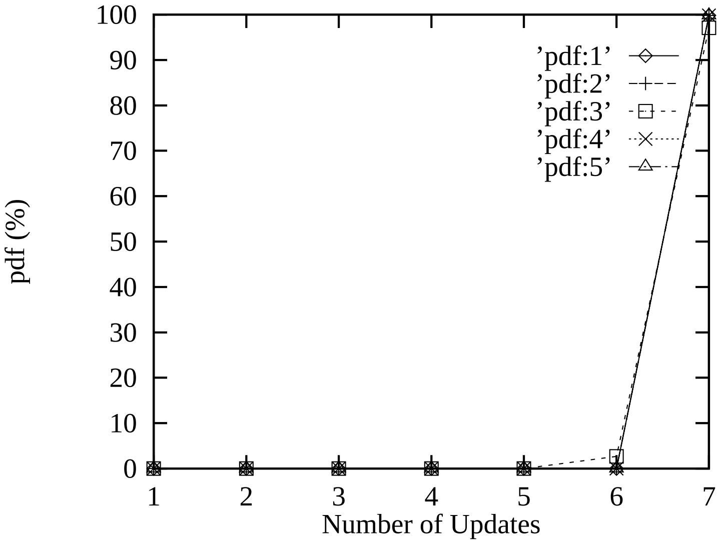


Fig. 89. pdf (Isort: $L = 8$, size = 3200)

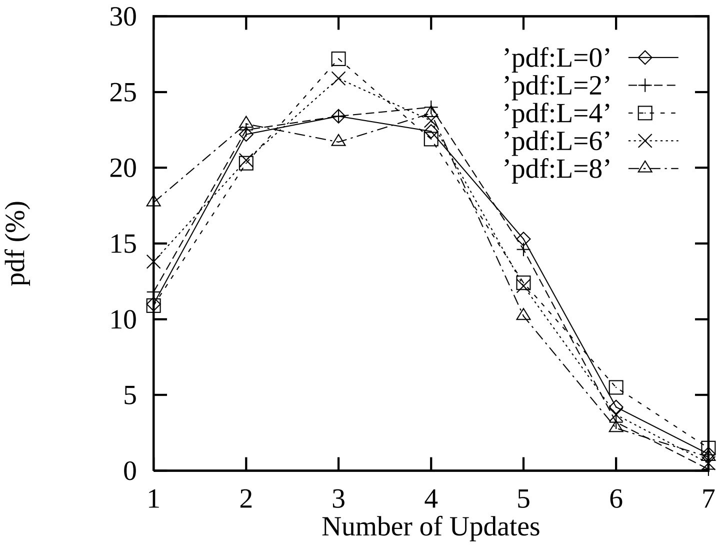


Fig. 90. pdf (qtest: read ratio = 50 %)

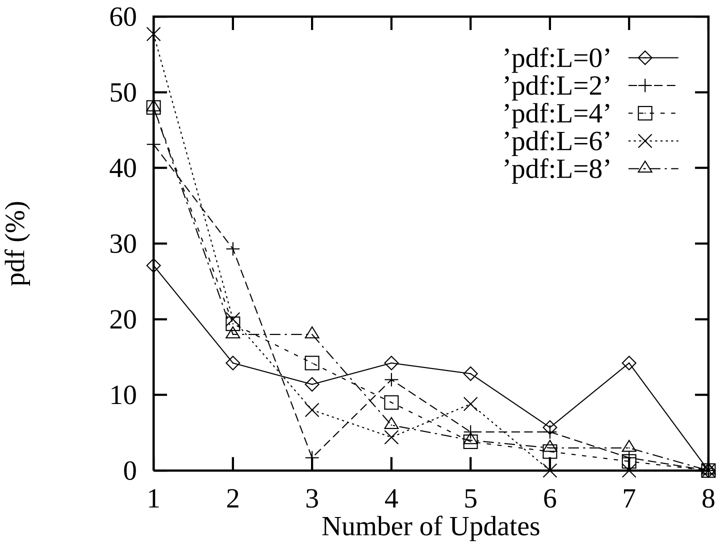


Fig. 91. pdf (Jacobi: size = 128)

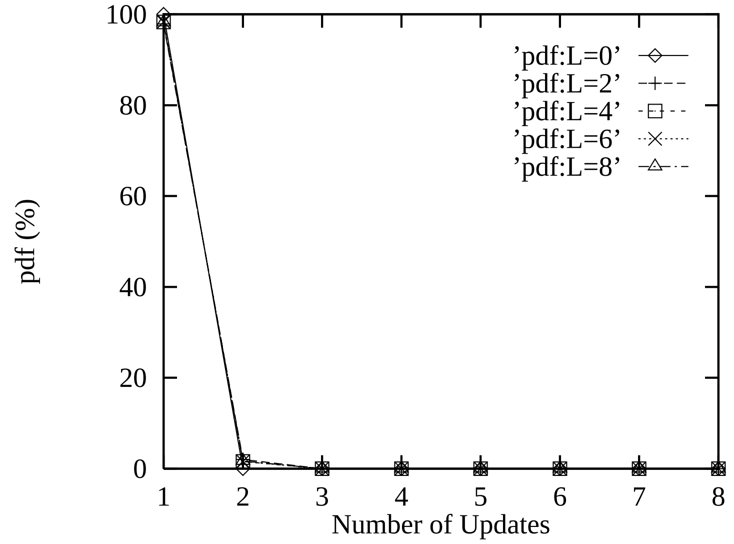


Fig. 92. pdf (SOR: size = 512)

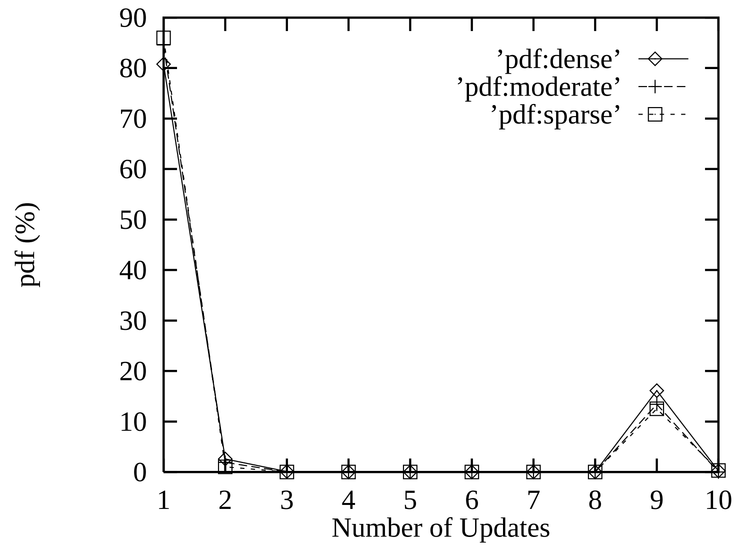


Fig. 93. pdf (Floyd-Warshall: size = 128)

for *Floyd-Warshall* application. We choose *Floyd-Warshall* because memory access pattern is different on different inputs. (The degree of sharing increases if the input matrix is more dense.) Figure 93 shows the probability density functions of different input data for *Floyd-Warshall* application. There is only a little difference for each input data.

The above measurements show that for some applications, the probability density function $p(x)$ is relatively independent of the update limit and input data. Of course, it should be emphasized that, this is not true for all applications.

## E. Summary

This chapter presents a new cost analysis model for competitive update protocol for software distributed shared memory (DSM). This model can be used to compute optimal update limit for the competitive update protocol. The optimal limit is chosen such that the cost metric is minimized for the given application (as characterized by its probability density function $p(x)$ of number of updates $x$ in a segment). We validated the proposed model by comparing analytical results from the model to experimental results obtained from an experimental DSM implementation. The analytical results often closely match the experimental results. We conclude that the proposed model can, therefore, be used to estimate the optimal update limit for an application.

# CHAPTER VII

# CONCLUSION AND FUTURE WORK

## A. Contribution

This dissertation deals with distributed shared memory using the *competitive* update protocol. The dissertation makes several contribution, as discussed below.

### 1. Adaptive Distributed Shared Memory

Our objective is to design an *adaptive* DSM that can adapt to time-varying pattern of accesses to the shared memory. The adaptive DSM automatically choose the appropriate consistency protocol (without any input from the programmer). 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 dissertation are *number* and *size* of messages required for executing an application using the DSM implementation. A generalization to minimize arbitrary cost metrics, including execution time is also discussed briefly.

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 dissertation presents experimental evaluation of the *adaptive* DSM using an implementation based on Quarks DSM [33]. Experimental results from the implementation suggest that the proposed adaptive approach can indeed reduce the *cost*.

### 2. Migratory Adaptive Distributed Shared Memory

We modify the above adaptive DSM to allow migratory protocol as a consistency protocol. The modified adaptive protocol attempts to detect migratory access pattern, and chooses the migratory protocol when it is deemed most cost-effective. Due to the dynamic distributed ownership algorithm used in many DSMs, migratory protocol is not always optimal *even if* the access pattern is migratory sharing.

The dissertation presents experimental evaluation of the proposed adaptive *migratory* scheme using an implementation based on Quarks DSM [33]. Experimental results from the implementation suggest that the proposed adaptive approach can usually reduce the *cost*. Specifically, the proposed scheme can typically reduce the number of messages as compared to the adaptive scheme in [39], as well as invalidate and competitive update protocols.

### 3. Single Fault-Tolerant Distributed Shared Memory Using Competitive Update

This dissertation presents a scheme to implement a software DSM that is recoverable in the presence of a single node failure. Our scheme differs from the previous work in that the proposed scheme is based on the *competitive* update protocol, which combines the advantages of invalidate as well as traditional update protocols. In addition, our approach is integrated with the release consistency model for maintaining memory consistency. In the basic competitive update protocol, the number of copies of a page varies dynamically – in the extreme, only one node may have a copy of the page or all nodes may have a copy of the page. Our approach is based on the simple observation that, to make the DSM recoverable from a single failure, it is adequate to ensure that each page has at least two copies at all times. To achieve this we suggest a modification to the basic *competitive* update protocol. Recovery is simple because an active back-up copy exists for each page. The proposed scheme is applicable to other updated-based protocols that incorporate mechanisms to selectively invalidate some pages. It is also applicable to generalizations of the *competitive* update protocols where the *limit* may be different for each page, and vary with time [36, 39].

We implemented recoverable DSM by modifying Quarks [10, 33] on a network of workstations. Experimental results indicate that, in many applications, the proposed scheme does not significantly increase the number or size of messages required.

## 4. Analysis of Failure Recovery Schemes

We analyzes the performance of 3 recoverable DSM schemes (incorporating with *re-do* overhead factor): (1) multiple fault-tolerant scheme by using the checkpoint and rollback recovery scheme, (2) single fault-tolerant scheme presented in chapter IV, and (3) two-level scheme [64] combining scheme (1) and (2).

In general, single fault-tolerant scheme presented in Chapter IV has advantage in the low failure-free overhead ($\alpha$), short task length ($\gamma$), and/or low failure rate ($\lambda$). However, if task length ($\gamma$) is long and/or failure rate ($\lambda$) is high, then it is highly possible that another failure will occur before recovering from the previous failure. When more than one failure occurs, task has to restart from the initial point, because single fault-tolerant scheme can recover single failure only. To solve this problem, we can use the two-level recovery scheme.

## 5. A Cost Model for Distributed Shared Memory Using Competitive Update

This dissertation presents a new cost analysis model for competitive update protocol for software DSM. This model can be used to compute optimal update limit for the competitive update protocol. The optimal limit is chosen such that the "cost" metric is minimized for the given application (as characterized by its probability density function $p(x)$ of number of updates $x$ in a segment). We validated the proposed model by comparing analytical results from the model to experimental results obtained from an experimental DSM implementation.

## B. Future Work

Three issues for future work are summarized below.

### 1. Adaptive Distributed Shared Memory

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. We use fixed $\overline{F}$ in our analysis, choosing the appropriate value of $\overline{F}$ at run-time is also useful.

The adaptive approach (based on cost-comparison) presented here may be combined with ideas developed by other researchers (e.g., [53]) to obtain further improvement in DSM performance. As yet, we have not explored this possibility.

### 2. Single Fault-Tolerant Distributed Shared Memory

We implemented recoverable DSM using *competitive* update by guaranteeing that at least two copies of each page exist, and measured the failure-free overhead. It is also applicable to generalizations of the *competitive* update protocols where the *limit* may be different for each page, and vary with time as presented in Chapter II. Implementation and evaluation for recovery procedure would be interesting.

### 3. A Cost Model for Distributed Shared Memory

We presented a new cost analysis model for competitive update protocol for software DSM. Future work includes application of this model to estimate costs of different schemes for recoverable DSM systems.

# REFERENCES

[1] S. V. Adve, *Designing Memory Consistency Models for Shared-Memory Multiprocessors*, Ph.D. dissertation, University of Wisconsin-Madison, Dec. 1993.

[2] C. Amza et al., "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, pp. 18–28, Feb. 1996.

[3] C. Anderson and A. Karlin, "Two adaptive hybrid cache coherency protocols," in *Proc. of the International Symposium on High-Performance Computer Architecture*, pp. 303–313, Feb. 1996.

[4] M. Banatre, A. Gefflaut, and C. Morin, "Tolerating node failures in cache only memory architectures," Tech. Rep. 853, INRIA, 1994.

[5] M. Banatre, P. Heng, G. Muller, N. Peyrouze, and B. Rochat, "An experience in the design of a reliable object based system," in *Proc. of the International Conference on Parallel and Distributed Information Systems*, pp. 187–190, Jan. 1993.

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

[7] M. Brorsson and P. Stenstrom, "Modeling accesses to migratory and producer-consumer characterized data in a shared memory multiprocessor," in *Proc. of IEEE Symposium on Parallel and Distributed Processing*, pp. 612–619, Oct. 1994.

[8] M. Brorsson and P. Stenstrom, "Modeling accesses to stationary data in a shared memory multiprocessor," in *Proc. of International Conference on Parallel and Distributed Computing Systems*, pp. 802–807, Oct. 1994.

[9] L. Brown and J. Wu, "Dynamic snooping in a fault-tolerant distributed shared memory," in *Proc. of the 14th International Conference on Distributed Computing Systems*, pp. 218–226, June 1994.

[10] 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.

[11] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOS P'91)*, pp. 152–164, Oct. 1991.

[12] J. B. Carter, *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*, Ph.D. dissertation, Rice University, Sept. 1993.

[13] K. Chandy, J. Browne, C. Dissly, and W. Uhrig, "Analytic models for rollback and recovery strategies in data base systems," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 100–110, Mar. 1975.

[14] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *IEEE Transactions on Computer systems*, vol. 3, pp. 63–75, Feb. 1985.

[15] A. Cox and R. Fowler, "The implementation of a coherent memory abstraction on a numa multiprocessor: Experience with platinum," in *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pp. 32–44, Dec. 1989.

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

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

[18] 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.

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

[20] E. Elnozahy, D. Johnson, and W. Zwaenepoel, "Measured performance of consistent checkpointing," in *Proc. of the Eleventh Symposium on Reliable Distributed Systems*, pp. 39–47, Oct. 1992.

[21] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood, "Application-specific protocols for user-level shared memory," in *Proc. of Supercomputing '94*, pp. 380–389, Nov. 1994.

[22] T. Fuchi and M. Tokoro, "A mechanism for recoverable shared virtual memory," manuscript, Dept. of Computer Science, University of Tokyo, 1994.

[23] 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.

[24] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputer," in *Proc. of the 13th Symposium on Reliable Distributed Systems*, pp. 42–51, Oct. 1994.

[25] B. Janssens and W. K. Fuchs, "Relaxing consistency in recoverable distributed shared memory," in *Proc. of the 23rd International Symposium on Fault-Tolerant Computing*, pp. 155–163, June 1993.

[26] B. Janssens and W. K. Fuchs, "Reducing interprocessor dependence in recoverable distributed shared memory," in *Proc. of the 13th Symposium on Reliable Distributed Systems*, pp. 34–41, Oct. 1994.

[27] S. Kanthadai and J. L. Welch, "Implementation of recoverable distributed shared memory by logging writes," in *Proc. of the 16th International Conference on Distributed Computing Systems*, May 1996.

[28] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator, "Competitive snoopy caching," in *Proc. of the 27'th Annual Symposium on Foundations of Computer Science*, pp. 244–254, Oct. 1986.

[29] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proc. of the 19th Annual International Symposium on Computer Architecture*, pp. 13–21, May 1992.

[30] P. Keleher, *Lazy Release Consistency for Distributed Shared Memory*, Ph.D. dissertation, Rice University, Jan. 1995.

[31] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A recoverable distributed shared memory integrating coherence and recoverability,"

in *Proc. of the 25th International Symposium on Fault-Tolerant Computing*, pp. 289–298, June 1995.

[32] R. Kessler and M. Livny, "An analysis of distributed shared memory algorithms," in *Proc. of the 9th International Conference on Distributed Computing Systems*, pp. 498–505, June 1989.

[33] D. Khandekar, "Quarks: Portable dsm on unix," manuscript, Dept. of Computer Science, University of Utah, 1995.

[34] J.-H. Kim and N. H. Vaidya, "Single fault-tolerant distributed shared memory using competitive update," *Microprocessors and Microsystems* (accepted).

[35] J.-H. Kim and N. H. Vaidya, "Distributed shared memory: Recoverable and non-recoverable limited update protocols," Tech. Rep. 95-025, Dept. of Computer Science, Texas A&M University, College Station, 1995.

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

[37] J.-H. Kim and N. H. Vaidya, "Towards an adaptive distributed shared memory," Tech. Rep. 95-037, Dept. of Computer Science, Texas A&M University, College Station, 1995.

[38] J.-H. Kim and N. H. Vaidya, "Adaptive migratory scheme for distributed shared memory," Tech. Rep. 96-023, Dept. of Computer Science, Texas A&M University, College Station, 1996.

[39] J.-H. Kim and N. H. Vaidya, "A cost-comparison approach for adaptive distributed shared memory," in *Proc. of 1996 International Conference on Super-*

*computing*, pp. 44–51, May 1996.

[40] J.-H. Kim and N. H. Vaidya, "Adaptive migratory scheme for distributed shared memory," in *Proc. of 1997 International Conference on Supercomputing*, pp. 325–332, July 1997.

[41] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. 13, pp. 23–31, Jan. 1987.

[42] R. LaRowe, C. Ellis, and L. Kaplan, "The robustness of numa memory management," in *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pp. 137–151, 1991.

[43] A. Lebeck and D. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pp. 48-59, 1995.

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

[45] N. Neves, M. Castro, and P. Guedes, "A checkpoint protocol for an entry consistent shared memory system," in *Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pp. 121–129, Aug. 1994.

[46] H. Nilsson and P. Stenstrom, "An adaptive update-based cache coherence protocol for reduction of miss rate and traffic," in *Proc. of the Parallel Architectures and Languages Europe Conference*, pp. 363–374, July 1994.

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

[48] 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.

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

[50] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proc. of the Winter 1995 USENIX Conference*, pp. 213–224, Jan. 1995.

[51] J. Plank and K. Li, "Faster checkpointing with n + 1 parity," in *Proc. of the 24th International Symposium on Fault-Tolerant Computing*, pp. 288–297, June 1994.

[52] D. K. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *IEEE Transactions on Computers*, pp. 1163–1174, Oct. 1994.

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

[54] A. Raynaud, Z. Zhang, and J. Torrellas, "Distance-adaptive update protocols for scalable shared-memory multiprocessors," in *Proc. of the International Symposium on High-Performance Computer Architecture*, pp. 323–334, Feb. 1996.

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

[56] G. Richard and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," in *Proc. of the 12th Symposium on Reliable Distributed Systems*, pp. 58–67, Oct. 1993.

[57] G. Shah, A. Singla, and U. Ramachandran, "The quest for a zero overhead shared memory parallel machine," in *Proc. of International Conference on Parallel Processing*, vol. I, pp. 194–201, Aug. 1995.

[58] S. Srbljic, Z. Vranesic, and L. Budin, "Performance prediction for different consistency schemes in distributed shared memory systems," in *Proc. of the International Symposium on High Performance Distributed Computing*, pp. 295–302, Apr. 1994.

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

[60] M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Computer*, vol. 23, pp. 54–64, May 1990.

[61] M. Stumm and S. Zhou, "Fault tolerant distributed shared memory algorithms," in *Proc. of the International Conference on Parallel and Distributed Processing*, pp. 719–724, Dec. 1990.

[62] O. Theel and B. Fleisch, "Analysis of a fault-tolerant coherence protocol for distributed memory systems under heavy write load," in *Proc. of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, pp. 146–151, Dec. 1995.

[63] O. Theel and B. Fleisch, "Design and analysis of highly available and scalable coherence protocols for distributed shared memory systems using stochastic modeling," in *Proc. of the International Conference on Parallel Processing*, vol. I, pp. 126–130, Aug. 1995.

[64] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *Proc. of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 64–73, May 1995.

[65] N. H. Vaidya, "On checkpoint latency," in *Proc. of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, pp. 60–65, Dec. 1995.

[66] 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.

[67] 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.

[68] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory: Memory coherence and storage structures," in *Proc. of the 19th International Symposium on Fault-Tolerant Computing*, pp. 520–527, June 1989.

[69] 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.

[70] J. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, pp. 530–531, Sept. 1974.

# APPENDIX A

Tables IV through XV show experimental results of executing `qtest` application to measure the overhead of recoverable scheme presented in Chapter IV.

**Legends**

- *Limit*: update limit
- *Messages*: the number of messages for non-recoverable scheme
- *Recovery Messages*: the number of messages for recoverable scheme
- *Data*: denotes the amount of data transferred ($KBytes$) for non-recoverable scheme
- *Recovery Data*: the amount of data transferred for recoverable scheme
- *Overhead*: the overhead percentage for recoverable scheme
- *S.D.*: standard deviation

Table IV. The Number of Messages (qtest: Read Ratio = 90 %)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 3521 | (728) | 3744 | (710) | 6.33 |
| 1 | 2532 | (311) | 2428 | (100) | -4.10 |
| 2 | 2280 | (105) | 2331 | (181) | 2.22 |
| 3 | 2300 | (188) | 2370 | (144) | 3.02 |
| 4 | 2308 | (98) | 2375 | (318) | 2.89 |
| 5 | 2293 | (129) | 2529 | (503) | 10.29 |
| 6 | 2329 | (169) | 2386 | (92) | 2.44 |
| 7 | 2373 | (202) | 2399 | (178) | 1.12 |
| 8 | 2438 | (283) | 2532 | (569) | 3.87 |

Table V. The Amount of Data (qtest: Read Ratio = 90 %)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 900 | (58) | 831 | (45) | -7.67 |
| 1 | 273 | (15) | 285 | (21) | 4.57 |
| 2 | 170 | (7) | 171 | (4) | 0.71 |
| 3 | 166 | (1) | 170 | (1) | 2.50 |
| 4 | 166 | (1) | 170 | (2) | 2.24 |
| 5 | 167 | (3) | 171 | (3) | 2.33 |
| 6 | 167 | (1) | 171 | (1) | 2.43 |
| 7 | 166 | (1) | 170 | (1) | 2.37 |
| 8 | 166 | (1) | 171 | (3) | 2.60 |

Table VI. The Number of Messages (qtest: Read Ratio = 80 %)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 7087 | (54) | 7157 | (83) | 1.00 |
| 1 | 6245 | (636) | 5916 | (92) | -5.26 |
| 2 | 5663 | (652) | 5331 | (96) | -5.87 |
| 3 | 5474 | (835) | 5291 | (184) | -3.35 |
| 4 | 5260 | (135) | 5343 | (455) | 1.58 |
| 5 | 5307 | (116) | 5230 | (401) | -1.45 |
| 6 | 5308 | (250) | 5361 | (349) | 0.99 |
| 7 | 5212 | (88) | 5431 | (649) | 4.21 |
| 8 | 5579 | (788) | 5381 | (462) | -3.54 |

Table VII. The Amount of Data (qtest: Read Ratio = 80 %)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 2311 | (18) | 2056 | (12) | -11.05 |
| 1 | 942 | (63) | 882 | (23) | -6.37 |
| 2 | 365 | (24) | 345 | (24) | -5.39 |
| 3 | 224 | (17) | 233 | (23) | 4.26 |
| 4 | 203 | (7) | 204 | (5) | 0.35 |
| 5 | 197 | (1) | 201 | (2) | 1.81 |
| 6 | 197 | (1) | 201 | (3) | 2.17 |
| 7 | 196 | (1) | 202 | (2) | 2.97 |
| 8 | 198 | (4) | 202 | (2) | 1.61 |

Table VIII. The Number of Messages (qtest: Read Ratio = 60 %)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 10183 | (849) | 10857 | (705) | 6.62 |
| 1 | 9912 | (175) | 10421 | (180) | 5.14 |
| 2 | 9930 | (1982) | 10123 | (120) | 1.94 |
| 3 | 9804 | (113) | 9646 | (91) | -1.61 |
| 4 | 9420 | (156) | 9310 | (259) | -1.17 |
| 5 | 9233 | (48) | 9203 | (56) | -0.32 |
| 6 | 9210 | (130) | 9200 | (499) | -0.12 |
| 7 | 9214 | (28) | 9106 | (373) | -1.17 |
| 8 | 9229 | (80) | 9342 | (467) | 1.22 |

Table IX. The Amount of Data (qtest: Read Ratio = 60 %)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 3334 | (43) | 3060 | (43) | -8.21 |
| 1 | 2783 | (127) | 2487 | (68) | -10.66 |
| 2 | 1731 | (634) | 1794 | (91) | 3.64 |
| 3 | 1198 | (106) | 1059 | (40) | -11.54 |
| 4 | 596 | (70) | 537 | (78) | -9.84 |
| 5 | 328 | (32) | 316 | (27) | -3.84 |
| 6 | 256 | (5) | 265 | (11) | 3.32 |
| 7 | 254 | (1) | 257 | (4) | 1.11 |
| 8 | 254 | (1) | 259 | (2) | 1.87 |

Table X. The Number of Messages (qtest: Read Ratio = 40 %)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 11202 | (708) | 11706 | (321) | 4.50 |
| 1 | 10998 | (79) | 12389 | (802) | 12.64 |
| 2 | 11722 | (92) | 12098 | (137) | 3.20 |
| 3 | 12018 | (486) | 12037 | (622) | 0.16 |
| 4 | 11608 | (160) | 11720 | (129) | 0.96 |
| 5 | 11422 | (186) | 11508 | (478) | 0.75 |
| 6 | 11229 | (139) | 11141 | (223) | -0.78 |
| 7 | 11438 | (735) | 11047 | (97) | -3.42 |
| 8 | 10975 | (444) | 11063 | (133) | 0.80 |

Table XI. The Amount of Data (qtest: Read Ratio = 40 %)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 3491 | (49) | 3223 | (85) | -7.69 |
| 1 | 3269 | (51) | 2944 | (56) | -9.94 |
| 2 | 2825 | (42) | 2551 | (53) | -9.70 |
| 3 | 2170 | (55) | 1957 | (95) | -9.79 |
| 4 | 1438 | (50) | 1357 | (59) | -5.63 |
| 5 | 829 | (81) | 767 | (105) | -7.43 |
| 6 | 464 | (49) | 466 | (14) | 0.47 |
| 7 | 289 | (3) | 291 | (2) | 0.69 |
| 8 | 285 | (4) | 291 | (2) | 1.89 |

Table XII. The Number of Messages (qtest: Read Ratio = 20 %)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 12746 | (677) | 13134 | (494) | 3.04 |
| 1 | 11881 | (58) | 14042 | (528) | 18.19 |
| 2 | 12657 | (681) | 14483 | (769) | 14.42 |
| 3 | 14254 | (111) | 14915 | (81) | 4.63 |
| 4 | 15718 | (1047) | 15471 | (805) | -1.57 |
| 5 | 15239 | (228) | 15012 | (270) | -1.49 |
| 6 | 14701 | (829) | 14037 | (1040) | -4.52 |
| 7 | 13539 | (681) | 13528 | (351) | -0.08 |
| 8 | 13643 | (746) | 13708 | (721) | 0.48 |

Table XIII. The Amount of Data (qtest: Read Ratio = 20 %)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 3546 | (3) | 3243 | (157) | -8.55 |
| 1 | 3397 | (5) | 3053 | (74) | -10.11 |
| 2 | 3161 | (272) | 2903 | (126) | -8.18 |
| 3 | 3062 | (27) | 2750 | (19) | -10.11 |
| 4 | 2876 | (87) | 2606 | (33) | -9.41 |
| 5 | 2588 | (127) | 2294 | (120) | -11.35 |
| 6 | 1730 | (520) | 1212 | (398) | -29.92 |
| 7 | 326 | (6) | 331 | (4) | 1.53 |
| 8 | 327 | (5) | 332 | (5) | 1.59 |

Table XIV. The Number of Messages (qtest: Read Ratio = 0 %)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 13618 | (47) | 14177 | (759) | 4.11 |
| 1 | 12827 | (29) | 15131 | (50) | 17.96 |
| 2 | 14146 | (73) | 16132 | (139) | 14.04 |
| 3 | 16038 | (40) | 16929 | (66) | 5.56 |
| 4 | 17412 | (69) | 17763 | (517) | 2.01 |
| 5 | 18382 | (534) | 17925 | (23) | -2.49 |
| 6 | 17923 | (2280) | 17908 | (270) | -0.09 |
| 7 | 16054 | (41) | 15860 | (455) | -1.21 |
| 8 | 16034 | (26) | 16315 | (675) | 1.75 |

Table XV. The Amount of Data (qtest: Read Ratio = 0 %)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 3715 | (0) | 3292 | (3) | -11.39 |
| 1 | 3694 | (0) | 3302 | (19) | -10.62 |
| 2 | 3700 | (14) | 3311 | (19) | -10.50 |
| 3 | 3727 | (0) | 3326 | (0) | -10.76 |
| 4 | 3736 | (17) | 3331 | (2) | -10.83 |
| 5 | 3749 | (2) | 3332 | (0) | -11.13 |
| 6 | 3218 | (1189) | 3200 | (183) | -0.58 |
| 7 | 369 | (1) | 371 | (6) | 0.49 |
| 8 | 369 | (1) | 375 | (3) | 1.46 |

# APPENDIX B

Tables XVI through XXIX show experimental results of executing other applications to measure the overhead of recoverable scheme presented in Chapter IV.

**Legends**

- *Limit*: update limit
- *Messages*: the number of messages for non-recoverable scheme
- *Recovery Messages*: the number of messages for recoverable scheme
- *Data*: denotes the amount of data transferred ($KBytes$) for non-recoverable scheme
- *Recovery Data*: the amount of data transferred for recoverable scheme
- *Overhead*: the overhead percentage for recoverable scheme
- *S.D.*: standard deviation

Table XVI. The Number of Messages (Floyd-Warshall)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 8919 | (167) | 25207 | (846) | 182.63 |
| 1 | 8385 | (2000) | 13518 | (253) | 61.21 |
| 2 | 7906 | (18) | 18037 | (1445) | 128.14 |
| 3 | 8440 | (125) | 18947 | (1982) | 124.49 |
| 4 | 8794 | (139) | 20231 | (72) | 130.05 |
| 5 | 9459 | (325) | 20001 | (780) | 111.46 |
| 6 | 9809 | (55) | 19527 | (1607) | 99.08 |
| 7 | 10236 | (58) | 19647 | (1329) | 91.94 |
| 8 | 10862 | (294) | 20853 | (1300) | 91.98 |

Table XVII. The Amount of Data (Floyd-Warshall)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 4692 | (69) | 5643 | (69) | 20.26 |
| 1 | 1565 | (21) | 1976 | (4) | 26.28 |
| 2 | 1509 | (2) | 1984 | (54) | 31.45 |
| 3 | 1559 | (2) | 2007 | (58) | 28.74 |
| 4 | 1608 | (2) | 2066 | (5) | 28.52 |
| 5 | 1661 | (3) | 2075 | (20) | 24.89 |
| 6 | 1706 | (3) | 2065 | (48) | 21.05 |
| 7 | 1748 | (1) | 2074 | (42) | 18.69 |
| 8 | 1793 | (2) | 2114 | (27) | 17.90 |

Table XVIII. The Number of Messages (SOR)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 15765 | (518) | 132104 | (12833) | 737.96 |
| 1 | 11862 | (129) | 171451 | (318) | 1345.36 |
| 2 | 12568 | (29) | 171484 | (464) | 1264.49 |
| 3 | 13504 | (116) | 171102 | (56) | 1167.01 |
| 4 | 15194 | (1844) | 170465 | (95) | 1021.94 |
| 5 | 15268 | (66) | 170441 | (16) | 1016.35 |
| 6 | 16238 | (161) | 169892 | (307) | 946.26 |
| 7 | 17040 | (11) | 170295 | (625) | 899.37 |
| 8 | 18333 | (818) | 169076 | (69) | 822.26 |

Table XIX. The Amount of Data (SOR)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 12243 | (53) | 86892 | (11947) | 609.71 |
| 1 | 4718 | (27) | 103246 | (7) | 2088.34 |
| 2 | 4533 | (2) | 103245 | (4) | 2177.73 |
| 3 | 4587 | (0) | 103244 | (4) | 2150.71 |
| 4 | 4656 | (8) | 103216 | (14) | 2116.64 |
| 5 | 4731 | (1) | 103207 | (5) | 2081.69 |
| 6 | 4819 | (1) | 103168 | (9) | 2040.85 |
| 7 | 4917 | (1) | 103172 | (18) | 1998.10 |
| 8 | 5030 | (3) | 103106 | (5) | 1949.98 |

Table XX. The Number of Messages (ProdCons)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 65225 | (1343) | 67304 | (2778) | 3.19 |
| 1 | 60781 | (19) | 72259 | (87) | 18.88 |
| 2 | 67452 | (40) | 77361 | (124) | 14.69 |
| 3 | 76129 | (949) | 81267 | (116) | 6.75 |
| 4 | 84085 | (513) | 84196 | (86) | 0.13 |
| 5 | 87762 | (819) | 86326 | (879) | -1.64 |
| 6 | 88880 | (208) | 87178 | (168) | -1.91 |
| 7 | 74602 | (3414) | 75954 | (1636) | 1.81 |
| 8 | 75874 | (1591) | 75963 | (1610) | 0.12 |

Table XXI. The Amount of Data (ProdCons)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 17888 | (29) | 16024 | (402) | -10.42 |
| 1 | 17828 | (0) | 15871 | (23) | -10.98 |
| 2 | 17914 | (0) | 15959 | (1) | -10.91 |
| 3 | 17911 | (198) | 16006 | (17) | -10.63 |
| 4 | 18135 | (23) | 16054 | (2) | -11.47 |
| 5 | 18192 | (19) | 15979 | (166) | -12.17 |
| 6 | 18136 | (173) | 15995 | (147) | -11.80 |
| 7 | 1164 | (63) | 1194 | (29) | 2.54 |
| 8 | 1188 | (28) | 1194 | (27) | 0.47 |

Table XXII. The Number of Messages (Isort)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 52213 | (992) | 53073 | (1394) | 1.65 |
| 1 | 48115 | (1063) | 59476 | (6151) | 23.61 |
| 2 | 54079 | (93) | 62170 | (649) | 14.96 |
| 3 | 61692 | (1299) | 65218 | (79) | 5.72 |
| 4 | 67632 | (616) | 67023 | (1223) | -0.90 |
| 5 | 70006 | (278) | 68899 | (611) | -1.58 |
| 6 | 70276 | (2525) | 69971 | (51) | -0.43 |
| 7 | 60983 | (1654) | 60001 | (2201) | -1.61 |
| 8 | 61448 | (149) | 60974 | (1267) | -0.77 |

Table XXIII. The Amount of Data (Isort)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 14385 | (25) | 12723 | (607) | -11.55 |
| 1 | 14134 | (252) | 12726 | (93) | -9.96 |
| 2 | 14365 | (25) | 12802 | (18) | -10.88 |
| 3 | 14402 | (190) | 12844 | (13) | -10.82 |
| 4 | 14555 | (3) | 12783 | (189) | -12.17 |
| 5 | 14564 | (41) | 12795 | (225) | -12.15 |
| 6 | 13983 | (1510) | 12903 | (0) | -7.72 |
| 7 | 13983 | (1510) | 978 | (31) | -1.02 |
| 8 | 996 | (0) | 992 | (19) | -0.36 |

Table XXIV. The Number of Messages (Reader/Writer)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 19079 | (152) | 21955 | (143) | 15.07 |
| 1 | 17629 | (278) | 19207 | (104) | 8.96 |
| 2 | 20377 | (53) | 21349 | (49) | 4.77 |
| 3 | 23319 | (32) | 24554 | (45) | 5.30 |
| 4 | 25927 | (45) | 26412 | (19) | 1.87 |
| 5 | 27439 | (721) | 27643 | (759) | 0.74 |
| 6 | 26309 | (26) | 25981 | (16) | -1.25 |
| 7 | 23712 | (677) | 23401 | (18) | -1.31 |
| 8 | 22975 | (993) | 23784 | (743) | 3.52 |

Table XXV. The Amount of Data (Reader/Writer)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 10493 | (0) | 14459 | (1) | 37.80 |
| 1 | 10751 | (165) | 11595 | (1) | 7.84 |
| 2 | 12735 | (0) | 13381 | (0) | 5.07 |
| 3 | 14606 | (0) | 15214 | (0) | 4.16 |
| 4 | 15871 | (0) | 16024 | (0) | 0.97 |
| 5 | 16489 | (39) | 16377 | (3) | -0.68 |
| 6 | 14939 | (0) | 14440 | (0) | -3.34 |
| 7 | 10833 | (3) | 10836 | (0) | 0.03 |
| 8 | 10588 | (545) | 10838 | (3) | 2.35 |

Table XXVI. The Number of Messages (Matmult)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 2159 | (427) | 3286 | (721) | 52.23 |
| 1 | 1924 | (97) | 2807 | (36) | 45.90 |
| 2 | 2329 | (625) | 2869 | (44) | 23.18 |
| 3 | 2080 | (40) | 2835 | (49) | 36.31 |
| 4 | 2193 | (59) | 2858 | (68) | 30.33 |
| 5 | 2270 | (18) | 2916 | (17) | 28.47 |
| 6 | 2582 | (542) | 2941 | (90) | 13.90 |
| 7 | 2500 | (33) | 2968 | (56) | 18.73 |
| 8 | 2514 | (76) | 3008 | (31) | 19.63 |

Table XXVII. The Amount of Data (Matmult)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 863 | (2) | 2718 | (3) | 214.85 |
| 1 | 1035 | (0) | 2774 | (0) | 168.06 |
| 2 | 1209 | (3) | 2824 | (0) | 133.52 |
| 3 | 1381 | (0) | 2873 | (0) | 108.04 |
| 4 | 1554 | (0) | 2922 | (0) | 88.01 |
| 5 | 1727 | (0) | 2972 | (0) | 72.09 |
| 6 | 1901 | (2) | 3021 | (0) | 58.93 |
| 7 | 2074 | (1) | 3071 | (0) | 48.10 |
| 8 | 2246 | (1) | 3120 | (0) | 38.89 |

Table XXVIII. The Number of Messages (Jacobi)

| *Limit* | *Messages* | *(S.D.)* | *Recovery Messages* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 2644 | (1365) | 2763 | (849) | 4.51 |
| 1 | 2267 | (97) | 2377 | (340) | 4.84 |
| 2 | 2251 | (51) | 2346 | (346) | 4.23 |
| 3 | 2295 | (364) | 2363 | (342) | 2.95 |
| 4 | 2280 | (215) | 2291 | (421) | 0.52 |
| 5 | 2334 | (400) | 2352 | (448) | 0.79 |
| 6 | 2298 | (421) | 2295 | (353) | -0.13 |
| 7 | 2400 | (521) | 2396 | (463) | -0.16 |
| 8 | 2261 | (102) | 2263 | (107) | 0.09 |

Table XXIX. The Amount of Data (Jacobi)

| *Limit* | *Data* | *(S.D.)* | *Recovery Data* | *(S.D.)* | *Overhead* |
|---|---|---|---|---|---|
| 0 | 648 | (21) | 583 | (30) | -10.06 |
| 1 | 468 | (31) | 434 | (20) | -7.20 |
| 2 | 423 | (19) | 397 | (11) | -6.09 |
| 3 | 386 | (12) | 377 | (9) | -2.50 |
| 4 | 358 | (12) | 355 | (12) | -0.86 |
| 5 | 343 | (7) | 343 | (5) | 0.13 |
| 6 | 334 | (4) | 333 | (4) | -0.26 |
| 7 | 328 | (2) | 332 | (2) | 1.29 |
| 8 | 327 | (0) | 331 | (1) | 1.26 |

# APPENDIX C

In this Appendix, we compute approximately the optimal checkpoint interval ($T_{opt}$) for the checkpointing scheme.

The expected cost, $\Gamma$, required to execute one checkpoint interval is (as obtained in Section V):

$$\Gamma = (1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)} - 1)$$

The overhead ratio of checkpointing scheme is:

$$r = \frac{\Gamma}{T} - 1$$

To compute the optimal value of $T$ ($T_{opt}$) that minimizes the overhead ratio $r$:

$$\begin{aligned}
\frac{\partial r}{\partial T} &= 0 \\
\Rightarrow \frac{\partial}{\partial T}\left[\frac{(1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)} - 1)}{T} - 1\right] &= 0 \\
\Rightarrow \frac{\partial}{\partial T}\,\left[(1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)} - 1)\right]\,T & \\
-\left[(1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)} - 1)\right] &= 0 \\
\Rightarrow \left[(1-k) + k\,e^{\lambda R}\,e^{\lambda(T+C)}\right]\,T - \left[(1-k)(T+C) + k\,\lambda^{-1}\,e^{\lambda R}(e^{\lambda(T+C)} - 1)\right] &= 0
\end{aligned}$$

Using the expansion of $e^{\lambda(T+C)}$ and $e^{\lambda R}$ as far as the second degree term:

$$\begin{aligned}
e^{\lambda(T+C)} &\approx 1 + \lambda\,(T+C) + \frac{\lambda^2\,(T+C)^2}{2} \\
e^{\lambda R} &\approx 1 + \lambda\,R + \frac{\lambda^2\,R^2}{2}
\end{aligned}$$

By approximation, ignoring $\lambda^3$ term, and simplification:

$$\begin{aligned}
\Rightarrow T &\approx \sqrt{\frac{2\,C}{\lambda\,k}\left(1+\lambda\,k\,R+\frac{\lambda\,k\,C}{2}\right)} \\
\Rightarrow T &\approx \sqrt{\frac{2\,C}{\lambda\,k}} \text{ when } \lambda\,k\,R \ll 1\text{, and } \lambda\,k\,C \ll 1
\end{aligned}$$

Thus, unique optimal checkpoint interval is:

$$T_{opt} \approx \sqrt{\frac{2\,C}{\lambda\,k}}.$$

## VITA

Jai-Hoon Kim received a B.S. in Control & Instrumentation Engineering from Seoul National University, Seoul, Korea in 1984, and a M.S. in Computer Science from Indiana University, Bloomington, Indiana, in 1993. He joined the R&D center of Daewoo Telecom, Ltd., Seoul, Korea in 1984 where he developed system software and application software for 7.5 years. Currently, he is a research assistant in the Computer Science Department at Texas A&M University, College Station, Texas. His research interests include distributed systems, fault-tolerant systems, operating systems, and computer architectures.

Permanent address: Department of Computer Science, Texas A&M University, H.R. Bright Building, College Station, Texas 77843-3112.

The typist for this dissertation was Jai-Hoon Kim.