Implementing Fault Tolerance in a 64-bit Distributed Operating System

Timothy James Wilkinson
Systems Architecture Research Centre
City University

July 1993

This thesis is submitted as part of the requirements for a Ph.D. in Computer Science, in the Department of Computer Science of City University, London, England.

ABSTRACT

This thesis explores the potential of 64-bit processors for providing a different style of distributed operating system. Rather than providing another reworking of the UNIX model, the use of the large address space for unifying volatile memory (virtual memory), persistent memory (file systems) and distributed network access is examined and a novel operating system, ARIUS, is proposed.

The concepts behind the design of ARIUS are briefly reviewed, and then the reliability of such a system is examined in detail. The unified nature of the architecture makes it possible to use a reliable single address space to provide a completely reliable system without the addition of other mechanisms. Protocols are proposed to provide locally scalable distributed shared memory and these are then augmented to handle machine failures transparently though the use of distributed checkpoints and rollback.

The checkpointing system makes use of the caching mechanism in DSM to provide data duplication for failure recovery. By using distributed memory for checkpoints, recovery from machine faults may be handled seamlessly. To cope with more "complete" failures, persistent storage is also included in the failure mechanism.

These protocols are modelled to show their operability and to determine the cost they incur in various types of parallel and serial programs. Results are presented to demonstrate these costs.

Contents

List of Figures

List of Algorithms

This is dedicated to Judy who put up with me while I was writing and needs something to stop the kitchen table from wobbling.

Acknowledgements

This work was carried out in the Systems Architecture Research Centre at City University, beginning in 1989 and finishing in 1993, during my time there as a postgraduate research student and a research assistant. Many people have influenced my work over the years and I wish to convey my thanks to those who were particularly helpful.

Philip Winterbottom, *second hand T-shirt salesman*, who had a bigger influence on my work than I care to admit,
Peter Osmon, *long suffering supervisor*, for giving me the chance to do this and providing the supervision and advice to finish it,
Andy Whitcroft, *best man elect*, for telling me when I wasn't making sense and putting up with me when I did,
Matt Sillitoe, *haircut 100um*, for his inspiration to make ARIUS more than just another operating system,
Pete Sealey, *lecturer of old*, for giving me the best systems architecture grounding I could have ever wanted and far too many Sunday dinners,
Aarron Gull, *Fish without the money*, for his ideas and out of hours access to Okapi,
Ashley Saulsbury, *Swedish holiday destination*, for his input to chapter 3, and
Nick Williams, *automounters to the gentry*, for his C++ advice (and books!).

Declaration

Chapter 1

Introduction

Computers have undergone many fundamental changes in their evolution since the emergence of commercial systems in the 1940s. However, at every point in this evolution designers have attempted to link two or more similar machines together to boost the performance provided to their users. Ultimately, however, the difficulties in doing so have always resulted in parallel machines being produced a generation behind single processor machines and, consequently, offering worse not better performance.

Parallel processor technology has matured extensively over the past ten years, as has the ability to loosely couple single machines. This has increased interest in multiprocessors and multi-computers. However, such machines still exhibit a number of problems compared with their single processor counterparts—how to write applications for them, what the operating system should provide in a parallel machine, how parallel programs should communicate, and how the machine should be made reliable.

In this thesis the last of these problems is studied in some depth and an answer proposed. Naturally, this aspect of parallel machines cannot be examined out of context and the other problems listed above are also examined. Briefly, this thesis attempts to answer the question, "How can fault tolerance be provided efficiently in a distributed, parallel operating system?"

1.1 Current system organisation

The physical composition of computers and computer systems has evolved extensively over the last two decades. Now a typical installation consists of various different machine types; such as uni-processor Sparc stations and PCs together with multi-processor Sequents. However, these machine are no longer stand-alone systems but are linked by a standard networking technology, most often Ethernet. This enables them to share and exchange information efficiently without the movement of any physical media (eg. floppy disks or cartridge tapes).

The use of networking extends beyond the interconnection of machines within the same installation, to interconnecting installations across the world. This allows facilities such as electronic mail (e-mail) to be provided where data is passed among many individuals on different machines.

1.1.1 Popular operating systems and networks

The popular operating systems now in use were designed before networking began to proliferate and consequently their handling of it is crude and limiting. Generally, networks are used in two ways; either to share a common file system or to provide remote terminal access to another machine. Both these facilities are simple modifications of already existing systems and do not attempt to fully exploit the potential of networks. Such a limited use of networking was reasonable when networks were expensive and bandwidth limited. This is no longer the case following the appearance of high-bandwidth, low cost networking, based on fibre-optics.

The inclusion of these elements into the current generation of computer systems presents a number of interesting problems and has led to an increased interest in parallel and distributed computers. For small self contained systems, high-speed networks allow many processors to act as a single computer. Over larger distances, networking allows efficient and extensive information sharing; so providing facilities such as teleconferencing and multi-media mail. Unfortunately, current computer operating systems are not in a position to exploit this technology efficiently.

1.1.2 The problem

Ultimately there are many reasons why current operating systems have problems with networks. Basic to most of them is the "stand-alone" concept inherent in their design. Although various systems were designed to be used by one, or more than one person at a time, there is no understanding of one system "using" another; and so no notion of remote machines. Such a concept has been incorporated after the fact (eg. remote login facilities, FTP services and remote file systems.); but underneath, the system still does not recognise the need to share with another.

1.2 The "sharing" solution

If high-bandwidth networking is to be embraced and used efficiently, it must be incorporated into the design of systems rather than added as a "bolt-on" extra. One method would be to use networks for message passing—allowing explicit exchange of data between entities; or alternatively, a network could be used to provide the illusion of a large machine when in fact there are really many small machines—a solution encapsulated in distributed shared memory. Adopting the latter model has many advantages. Firstly, it allows the distinction between uniprocessor, multiprocessor and multi-computer to be blurred so allowing an identical environment to be provided on all platforms. Secondly, it allows the sharing of data through common memory to become the norm for the exchange of information rather than the exception. Thirdly, by providing a single mechanism for data exchange, the actual location of the data can be hidden. Fourthly, the system can be exploited on any combination of general purpose machines.

The remainder of this chapter examines the various elements required to allow a truly unifying networked operating system to be developed.

1.3 Distributed shared memory

Distributed shared memory (DSM) is a technique whereby a number of machine connected by a network may maintain a coherent shared memory space. By doing so, an application operating on the network of machines appears to be using a physically shared memory system. DSM provides a means for connecting an arbitrary number of machines together in an arbitrary network so as to provide a simple, consistent, usable environment; that of shared memory.

The use of such a distributed system incurs a latency cost when compared to normal bus based shared memory systems. In an attempt to overcome this delay, DSM uses local caching of frequently accessed data (this is similar to the operation of a conventional processor cache). Caching brings with it the problem of data coherency; when one copy of data is changed on one processor, all other copies must be invalidated to maintain coherency—this can cause unnecessary performance degradation. This problem has been investigated by many groups and various solutions have been proposed.

Despite the coherency problems, DSM provides an efficient and versatile means for sharing information in parallel or distributed computers and the potential to eliminate the differences between the two.

1.4 Simplifying the environment

Whenever someone uses a computer, they are presented with an environment within which to work. In some case this environment is rather basic and relies on the person entering obscure typed commands; in another the person is expected to use a mouse to point, click, drag and generally manipulate images on the screen. In fact, every computer presents a variety of environments, one for the applications user, one for the application, and another for the application developer; each of which may be tailored and customised as required.

The environment provided by an application to the user is not, for the most part, important to the operating system since the application will present its services as its developer desired. However, the environment provided by the operating system for the developer or

the application is important. If this environment is ill suited to the requirements of the application, the application writer will be forced to provide the services they require over what is given.

Most operating systems provide similar facilities to execute programs and store data for later retrieval. However, the facilities for communication vary dramatically between systems. Furthermore, the facilities to share information between distinctly separate machines, although essentially similar in all operating systems, do not provide a convenient model for programs to use. Such barriers have three repercussions; they make programs difficult to port between dissimilar platforms, they force programmers to limit the type of communication used, and they make parallel applications rare.

Any new operating system architecture should aim to solve these problems. This may be done by combining many traditionally separate services into a single simplified one. The use of the network to provide shared memory has already been discussed. This may be further enhanced by making this shared memory persistent—so removing any need for an explicit file system.

1.5 64-bit architectures

The system so far discussed could be supported on any current processor design. However, processors are already taking a new evolutionary step; from 32-bit addresses to 64-bit addresses. This dramatic increase in address space may also be utilised to simplify general operating system design.

With 32-bit processors the maximum data directly addressable by any process is 4 gigabytes. This may appear large but many commercial databases far exceed this limit, and so this data is accessed indirectly, usually via a file system. However, with 64-bit addresses any process may access 16 million terabytes of data. This can easily accommodate the largest databases. Unfortunately, a conventional operating system would still rely on indirect access through a file system when a simpler and more flexible solution would be to access the database directly as conventional memory.

Some Unix based operating systems now address this problem, at least in part. However, the second attractive property of large address ranges is the potential to provide a single large address space rather than many address spaces.

1.6 Single address space architectures

A single address space architecture is a radical departure from the accepted norm of multiple address spaces, one per process. However, such a space can: provide a means for communicating among a network of machines via DSM; provide persistent data storage by absorbing the file system; and remove the aliasing problems associated with multiple address spaces.

To consider a network of machines as a single address space has many advantages. Firstly, all data is fixed for all observers. This property is of great advantage to a parallel program which may be running on several different machines yet interacting as if on the the same one. Secondly, even when programs are not sharing data in parallel, by being able to pass addresses of data from a client to a server and have the server understand it without translation or intermediate packaging thereby providing greater transparency and flexibility. Thirdly, for hiding a network of machines behind an address space to allow applications to be written more regularly without the need to provide purpose built protocols for handling networks.

All these advantages enable applications to be written more generally, a problem acute with parallel programs which tend to be optimised specifically for one architecture. Further, if the single address space is enhanced by the addition of persistence, then this removes many of the associated problems of transferring data from file to memory and back again.

The major problem in previous operating system architecture with these features is incompatibility — no common computer architecture in use today provides such an environment, but then the common operating systems were designed for single processor machines with small disks and no networks. Continuing conformance with such a model of computing is limiting and costly.

Of course, discarding twenty years of computer environment is a difficult step to take. Fortunately there is no reason, with modern processors and modern compiler technology, why multiple address space architecture applications cannot be run, without modification, on a single address space architecture.

1.7 Fault tolerance

If it is accepted that a new style of operating system is required (one which provides a unified machine interface to parallel and distributed systems) then, for once, the problem of fault tolerance should not be neglected. In fact, as parallel system sizes grow, fault tolerance can no longer be considered a desirable but expensive extra, it becomes an essential. If any large system is to operate for a useful period of time, it must tolerate faults since the frequency at which they occur increases dramatically.

Providing resilience to failures in a single persistent address space system can be done very simply—there is only one allocatable resource: persistent address space. If this is provided in a fault tolerant manner, then everything operating within the system cannot help but be fault tolerant also. Even the operating system can use this fault tolerant address space to preserve its state in the case of failure.

Naturally, it is never quite so simple and there are many difficulties in providing fault tolerance in a parallel or distributed machine in an efficient manner. However, solving such a problem for a machine architecture which provides a shared memory interface, rather than a message passing one, opens up the possibilities for which large scale parallel architectures may be used, since reliability is no longer provided at a design cost to the programmer or within a limited number of applications.

1.8 Original Work

This thesis addresses a number of issues regarding the next generation of operating system designs. These may be broadly split into three categories:

- The design of 64-bit single persistent address space operating system so as to unify the appearance of single, parallel and distributed machines,
- The design and implementation of a flexible distributed shared memory system for operating system and application use, and
- The design and implementation of a reliability system which uses other machines and persistent disk storage to guard against machine failures.

In addressing these issues, a number of others are raised and examined. However, the chief aim of this thesis is to examine how an efficient reliable system can be designed based on a 64-bit operating system where data sharing, in both client/server and peer-structured applications, is the common case rather than the exception.

1.9 Structure

The structure of this thesis is as follows. Chapter 2 examines the background behind the design of a distributed address space on which this work is based. Chapter 3 describes the design of the single address space operating system ARIUS. Chapter 4 describes a mechanism to support multi-policy distributed shared memory on a large network of machines. Chapter 5 examines the methods for providing fault tolerance. Chapter 6 describes volatile reliability, the means by which ARIUS handles the failure of machines. Chapter 7 expands on this to provide persistent reliability. Chapter 8 describes the experiments and results obtained during the modelling of ARIUS's reliable store AMOS. Chapter 9 presents the conclusions and possible future work.

Chapter 2

Design of a distributed address space and store

The aim of this thesis is to describe the design of a new operating system, ARIUS, and examine how fault-tolerance can be included in it. To do this we must first examine the current state of knowledge about three aspects of systems research:

- Distributed operating systems,
- Distributed shared memory, and
- Fault tolerant operating systems.

In the first, emphasis is on the interfaces used to provide both volatile and persistent storage to the user. In the second, scalability of distributed shared memory (DSM) is emphasised. In the third, the emphasis is on the methods and associated cost of providing various degrees of fault-tolerance.

This chapter is split into six sections. The first outlines the properties of the data store considered desirable, the second and third describe various operating systems developed over recent years and their relationship to their data stores. The fourth examines DSM systems. The fifth examines various fault-tolerant systems. The chapter concludes with a summary.

2.1 Requirements of a distributed store

A distributed store should satisfy a number of general requirements:

1. **Access transparency** - The store should be accessed in as near a transparent fashion as possible. The physical location of the requester should not be a problem for the programmer. A good measure of this quality is the ease with which data may be accessed from local and remote locations without adverse effect on application programs—must the access method be different?

2. **Sharing transparency** - It is necessary to provide some means of sharing data between multiple programs. Often, this is only read-sharing where personal copies are taken from the store and manipulated independently. Parallel applications require read/write-sharing where modifications to the data are seen by all those accessing it.

3. **Location transparency** - The physical location of data should be hidden from the programmer. Within restrictions imposed by security, this should not limit the actual position of the data (eg. it must be physically stored on one machine only) nor its mobility.

4. **Distributed management** - Is should not be necessary to impose a single management structure upon the store. Although this is not a problem in small systems, sharing data between different organisations could prove problematic if there are management conflicts. The management of data by each organisation must be separated and not complicated by the distributed nature of the store.

5. **Security** - Networks are insecure. Although it may be possible to guarantee their integrity in small systems, large systems using publicly accessible networks are open to attack. Consequently, it is necessary to protect secure data against snooping, and protect against the impersonation of users and machines.

6. **Resilience and reliability** - A store should be resilient to failure of machines in the network. As networks become larger, the chances of their entirety being active reduces dramatically. Failures must be tolerated without shared data being

corrupted. Reliability is also desirable so that machine failures may be handled dynamically.

The following two sections examine various operating system and storage architectures from the viewpoint of these requirements.

2.2 Conventional distributed operating system stores

Distributed stores, most usually in the form of file systems, have become increasingly important with the success of the workstation+network architecture. In this model, instead of computing resources being provided by a central machine accessed by many remote terminals, many workstations replace the terminals and machine to provide a computing service. A workstation cooperates with others by communicating over some form of network[1].

If each workstation requires only its local store, then data exchange is minimal. However, this is not the case in a distributed system. Firstly, each machine will hold copies of utilities such as text processors, compilers and windowing software. If new versions of these programs are released, each machine must be updated; this is not inconvenient for two machines perhaps but is unacceptable for a hundred. Secondly, if each user uses only one particular machine and has minimal interaction with others there are no problems. Once users are allowed to login at any machine and exchange files with each other freely, and expect their environment to remain unaltered, then a simple explicit network model breaks down. The solution to both these problems has been the distributed file system.

2.2.1 NFS Network File System

NFS [Sun89] was designed by Sun Microsystems USA, to provide a distributed file system for their workstations. NFS allows programs to access remote UNIX file systems as if they were present locally. This would seem to provide good *access transparency*, all file systems

[1] Ethernet [MBC+80] for example.

being combined into a single large file system tree, but in practice it is complicated by the need to provide specific data at fixed locations in UNIX systems (each system needs different data but from the same place). Management of data in NFS is simplified for small homogeneous systems, although the use of a number of different types of machine in the network complicates this (but not impossibly). The movement of data from machine to machine may also be done in a *location transparent* manner by those administering the system. For example, a particular sub-tree of the file system may be moved from one machine to another without applications being aware of the change as long as the tree was initially named in a machine transparent manner [PW91].

Unfortunately, NFS does not offer a complete solution. One weakness is in *security*. Until recently, the exchange of information between machines took place in plain text. Consequently, data could be read by anyone. Encryption could have provided data protection as well as authorisation checking. This would make it impossible for a rogue machine to impersonate another legitimate machine on the network since it would not have the necessary encryption/decryption keys to enable successful communication.

NFS also provides poor *sharing transparency*. In order to improve efficiency and reduce the utilisation of the network, client machines (those accessing remote machines' file systems) cache sections of the files they read from servers (those providing data from their local file systems). This avoids repeated use of the network and the server when the data is requested again and allows large blocks to be transferred per request, in the hope that this will eliminate the need for subsequent transfers. This causes no problems if the data exchange is between a single client and server. When another client is introduced however, it is possible for one client to modify the data on the server which has already been read and cached by another (figure 2.1). The caching in the client therefore prevents any changes being seen until the local copy expires, which is after a fixed period of time.

During this period, the two machines' views of the distributed file system are dissimilar and there is no mechanism to force agreement other than by waiting! When a client writes to a remote file system, the write is immediately propagated to the server. This write-through strategy guarantees that once a client's cached data has expired, any newly requested copies will be up to date, but at the cost of forcing all writes to be made synchronously

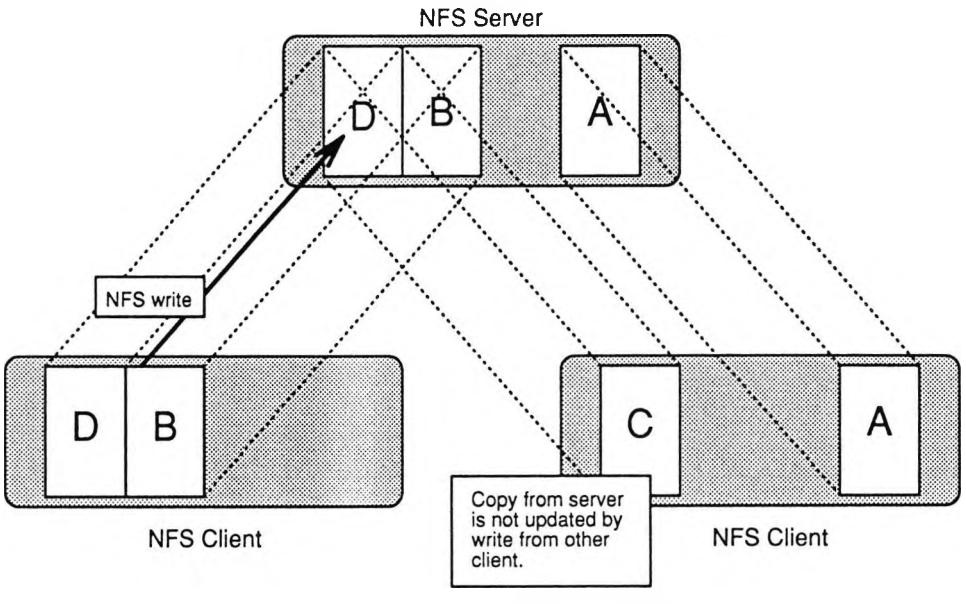


Figure 2.1: Cache inconsistency problem in NFS.

across the network.

NFS does not handle *reliability* and machine failure well. UNIX defines that once a *write()* is made to a file, and an error is not reported, the data has been correctly written. NFS already operates a write-through policy but the need to report errors correctly necessitates that each write blocks and waits for the remote machine to reply with a result. If a server becomes unavailable, then the data has been sent but never received and, consequently, there is no reply. NFS assumes the failure to be temporary and continually retries the write until successful, regardless of how long this takes (it could be forever!). A similar situation arises when a read of uncached data is made and the server is unavailable; the client retries the read until it succeeds. NFS servers are stateless [Tan88]. After the crashed machine has been returned to an operational state, the pending reads and writes initiated by clients will proceed correctly. A mode is provided to allow a blocked write to timeout after a given period but it does this by simply abandoning the write and continuing as if nothing had gone wrong; a poor solution to the problem.

Few of the problems of *distributed management* are addressed. For example, protection is provided by the standard UNIX *owner*, *group* and *other* attributes attached to each file. This is adequate when the network of machines is centrally administered, and each user

in the network can be given a unique network-wide user identifier. When this is not the case however, user ids may be overloaded. An access to a remote file system must not breach another user's security. NFS's solution is simple but inflexible. Machines outside the central administration's control are considered "untrusted" and any requests for file system data from them are considered to be from a special user "other". This user will normally be able to access files only available to *others*. A scheme for mapping *user A machine 1* to *user B machine 2* would be more powerful but its implementation would be costly in both time and space.

2.2.2 Sprite Network File system

The SPRITE network file system [NWO88] was implemented at Berkeley USA, to fulfil similar goals to NFS, providing similar solutions to the problems of *access transparency* and *location transparency*. Its designers also identified some of the problems inherent in NFS and attempted to correct them without sacrificing performance or flexibility. The three major problems they addressed were:

- Avoiding temporary files being written back to the server unnecessarily,
- Stopping writes from being delayed due to the speed of the network, and
- Guaranteeing that the most recently written data is seen by all clients.

To achieve these goals, SPRITE uses stateful servers which maintain information about who is accessing what and how.

When a file is opened by a client, the server is informed about the operation together with the type of open (eg. read only, write only, etc.). When a file is opened by a single client, regardless of the mode, it caches data locally. There cannot be any sharing problems since there is no one to share with. This case accounts for 95% of all file accesses. When a file is opened by more than one client, and all clients are reading the file, each one caches the data locally. However when a file is opened by more than one client, and one of the clients is writing, no one caches the file. In so doing, there can be no sharing consistency problems. This provides good *sharing transparency*.

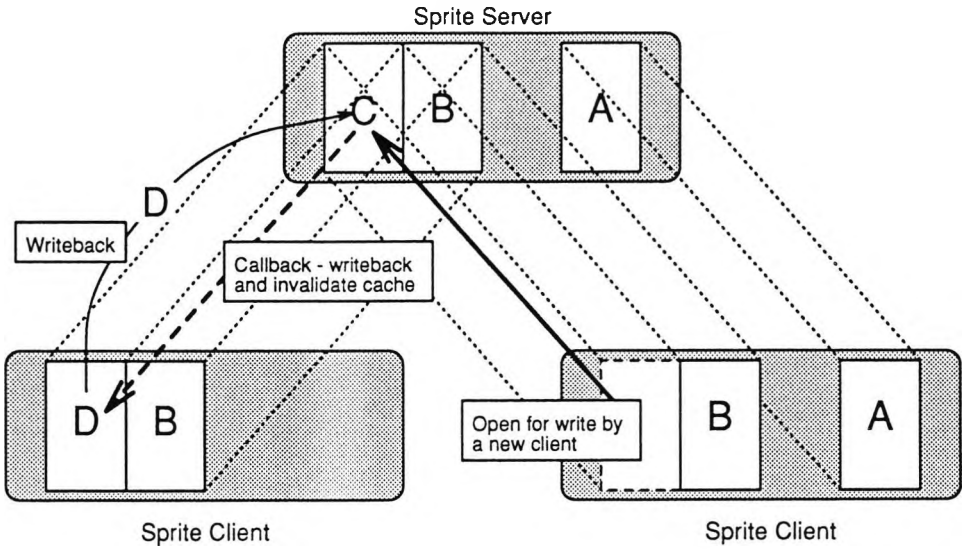


Figure 2.2: Callback mechanism in SPRITE.

SPRITE also uses a *delayed write* policy. Instead of writes to remote file systems being sent across the network immediately, they are written to local cache and only written back to the server when the file is finished with. If the file is deleted, then the data need never be written back to the server. This prevents the network bandwidth from limiting the throughput of remote file system writes.

There are occasions when SPRITE requires data to be written back from the client cache before it has finished with the file. The simplest is when the client cache becomes full and the data is flushed so the space may be reused. A more interesting situation is when a file is opened by a second client and it, or the first client, is able to write to it. The first client to open the file will already be caching the file locally. When the second client opens the file, this caching must be disabled and any modified data flushed back to the server. This cache flush is achieved by a *callback mechanism* (see figure 2.2). The server instructs the client to write back all modifications to the named file and then purge it from the local cache. Subsequent accesses to the file by the first and second client take place directly with the server. For a full description of these callback protocols see [NWO88].

SPRITE handles *resilience and reliability* far better than NFS. Crashes and subsequent recovery are handled by using a combination of a log structured file system [RO90, RO91] and distributed state held by file system clients [WBD+89, BO90]. In NFS a server may

crash and be rebooted without the knowledge of the client programs (apart from a long delay). This is simple because of the stateless operation of the servers. SPRITE uses stateful servers, which hold information pertaining to each client accessing them, and it is necessary to recover this state, after a crash, before operation can continue.

Server state may be reconstructed, following a crash, from the state held in the clients making use of it. When a client detects a reboot by a server, it helps the server to recover by informing it of any pertinent file system state information. Once all interested clients have accomplished this, the server and clients are again data consistent and may proceed. A considerable effort has been made in SPRITE to allow this procedure to happen quickly. It takes only 80 milliseconds per client to re-synchronise a server.

Another common problem with the re-synchronising of a UNIX file system with clients after a server crash, is the time to check file system consistency. On a Sun 4 with 2 Gigabytes of disk space this procedure can take 15 minutes or longer, depending on the file system usage. SPRITE attempts to remedy this problem by using a *log-structured file system* rather than the conventional update-based file system.

File system data is written to disk as a linear sequence of changes rather than a number of small writes to various places on the disk. This log, together with periodic checkpoints, forms a file system with similar or better performance for UNIX but with a significantly reduced crash recovery time. As clients modify a file on a server, rather than maintaining a traditional block cache, these changes are arranged into a *change log* which is periodically flushed to disk. These log records are marked with additional information to enable the consistency and age to be determined. Periodically, a checkpoint is also written to a fixed position on the disk (in fact, two checkpoint areas are maintained and written alternately in case of failure whilst a checkpoint is being written). This contains sufficient information to allow the log to be indexed in a read efficient manner and also provides fixed points where the file system is guaranteed consistent. Without this information, reading a file from disk would necessitate the examination of the entire log, as would recovery of the file system after a crash.

When a server is rebooted after a crash, the checkpoints are examined and the most recent is used as a base consistency point for the file system. This indicates the head of the log

when the checkpoint was committed. Any further log information can then be processed to modify this image and bring the file system completely up to date. It would be possible to ignore this log information but the file system would then only guarantee to retain data up to the last checkpoint. Of course, this is what UNIX does. The use of checkpoints in conjunction with a change log allows the checkpoints to be made infrequently (they are relatively expensive) without losing all changes since the previous one.

The problems of *security* and *distributed management* are not addressed. SPRITE provides a standard UNIX model of protection. It relies on a single naming authority to issue user and group ids, of which there are only thirty-two thousand of each. This is adequate for a single site installation. There is also no attempt to authenticate machines or users nor to restrict off site, unauthorised, access.

The SPRITE file system does solve many of the problems inherent with NFS. It does not address the problems of authentication [Ous92], nor even mapping of untrusted machine accesses to a bogus local user, or namespace management which is necessary for a global system. The use of a log-structured file system does solve some of the problems with server failures, but a catastrophic failure, where rebooting is impossible, will have much the same implications as with NFS.

2.2.3 DEcorum File System

The DECORUM file system [KLA+90] was developed by Hewlett-Packard, IBM, Locus Computing and Transarc, based on the Andrew distributed file system designed at Carnegie Mellon University, USA. DECORUM uses similar principles to SPRITE to enable it to provide single UNIX file semantics on a distributed machine. The interface provided is richer, and has provision for much finer grain file sharing, and locking.

A virtual file system (VFS) [Kle86] interface is provided to the operating system. This is similar to that used in NFS but provides extensions to allow better management of large numbers of clients and servers. The interface also allows many different physical file systems to be shared by each client which may view them in a different manner. A major advantage of this is the immediate availability of NFS file systems which may be used by

DECORUM clients without modification.

Access transparency and *location transparency* are similar to SPRITE. However, file system consistency (*data sharing*) is provided by the use of "tokens". These tokens are requested by clients from servers and, when held, indicate what operations the client is permitted to perform on the associated data. Several types of tokens are provided:

- **Data token** - allows the client to read or write (depending on the token subtype) a range of bytes in the file,
- **Status token** - allows the client to read or write (depending on the token subtype) the status information associated with a file,
- **Lock token** - allows the client to set read or write (depending on the token subtype) locks on a range of bytes in the file, and
- **Open token** - grants the client the right to open a file in a particular mode (depending on the token subtype).

Tokens of these different types do not conflict with one another since each grants access to different elements of a file. There could be conflicts between similar tokens held by different clients. These conflicts are not allowed to exist in the system and it the responsibility of a server's token manager to prevent them from happening. It does this by revoking client tokens before issuing new tokens which would otherwise conflict.

The granting and revoking of tokens is similar to the callback mechanism used in SPRITE. However, SPRITE only supports a single type of token indicating whether a file may or may not be cached. DECORUM allows a much finer control over the caching used. For example, many clients may be reading and writing the same file but in different areas. With SPRITE, no client caching is possible and all reads and writes must be made to the server. DECORUM allows each section of the file to be cached on the client using it since there are no conflicts.

Simple file protection is provided by standard UNIX permissions. In addition, access control lists (ACLs) are provided. These lists are associated with files and describe a number of users or groups which may access the file together with the operations they

may perform (read only, execute only, etc.). *Security* and user authentication are provided by a Kerberos [SNS88] service which is performed on all RPC calls between clients and servers. This prevents impersonation to gain access to their files by either compromising a machine's integrity or tapping the network. The ownership of files in a DECORUM file system is implicitly limited by the implementation of the remotely accessed file system. For a UNIX file system this is limiting. However, the use of Kerberos prevents unauthenticated users from accessing remote clients, thereby providing security without the need for a central naming service. This greatly aids *distributed management.*

Data *resilience and reliability* is maintained though the use of replication servers. These are responsible for keeping a permanent replica of a file system volume by using a *lazy replication policy*, which will guarantee to replicate data within a maximum period of time. If this time is very small, replication is necessary on every file system modification. Fortunately, this is not usually necessary. In a further effort to reduce the cost of replication, only modifications are replicated.

DECORUM addresses most of the requirements for a distributed storage system. The ability to replicate file systems "at will" is particularly powerful in preventing a server failure from halting a network of machines (if it were serving X-windows for example). Its protection of servers prevents unauthorised access and requires only specific authentication servers to be trusted by the server. This is usually adequate but means a user may still have to use several identities when working in a system, one obtained from each relevant Kerberos server.

2.2.4 Plan 9 Network File System

PLAN 9 [PPTT89, PPTT91] is a distributed operating system developed at Bell Labs., USA by the designers of UNIX. Although not UNIX compatible, it is designed to be "culturally" compatible.

Three major types of networked machines are supported:

- **Diskless CPU servers** - provide processing power for users,

- **File servers** - provide persistent data storage, and
- **Dedicated bitmap terminals** - provide a windowing environment for users.

The separation of functionality is intended to provide both a cost efficient means of providing resources for many users and simplify the resource management problems. For the purposes of this chapter, only the file services will be considered in any detail.

PLAN 9 does not provide coherent *data transparency* and has no complicated data coherency protocol. In many instances, communication between file server and CPU server is assumed to be fast enough not to warrant caching the data on the CPU server. Without a cache, there are no coherency problems. However, PLAN 9 is designed to operate over a variety of interconnection systems including slow serial lines. Consequently, failure to provide some form of caching on a CPU server or bitmap terminal would result in a very poor performance. Caching is therefore provided when the network, between file server and client, has insufficient bandwidth and latency to support uncached operations. The policy operated is very simple.

When a file on a server is opened by a client, either as a data file or as an executable, a 64 bit key is returned to the client. Part of this key identifies the file and the rest identifies a file version. This key is compared with the keys held by any cached sections of the file held by the client. If the keys are similar, the cached data may be reused. If the keys are different, the cached data is purged and any data must be re-read from the server. To further reduce utilisation of a low bandwidth network, especially when it may be necessary to load binaries across it, data may not only be cached in memory but may also be cached on local disk. This disk does not provide a file system but merely acts as a persistent cache of previously requested data. Data held on it is invalidated in a similar way to that held in memory.

Cache coherency is maintained only when a file is opened; once this has happened, coherency is never checked again. This makes it difficult to use files for communicating between different clients. To force coherency, it is necessary to re-open files at the required times. This will flush the entire file from local caches if it has changed, even if the changes are small. The whole file must then be re-read from the server. An additional

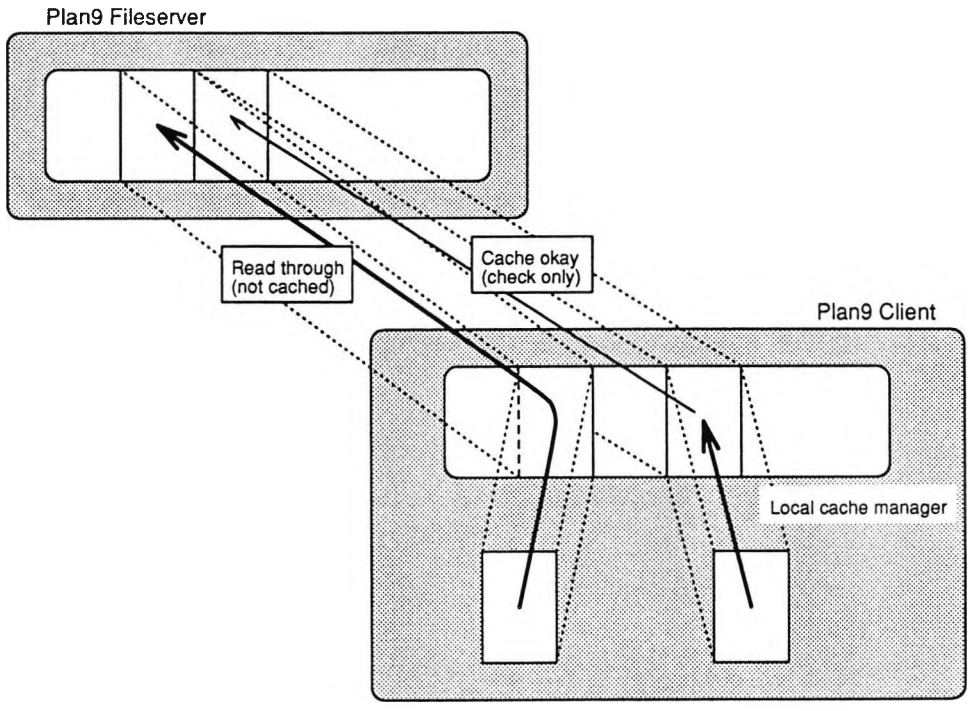


Figure 2.3: File caching in Plan 9

problem is associated with writes. The client cache is write through, in much the same way as in NFS. Even if writes are operated asynchronously, a large amount of network bandwidth is consumed, even for temporary files which will be deleted eventually.

PLAN 9's solution to *location and access transparency* is to provide every process group with its own namespace which it is allowed to construct and modify as it wishes. It is therefore possible for any user or application to construct a similar environment no matter where they access the system from or where that data resides. The ability to construct individual namespaces aids *distributed management* since each application can construct the file system it requires regardless of what is provided by the servers.

Protection and authentication is provided using DES encrypted character strings [Win92] in a challenge response system. User ids are passed as strings when a file server is first contacted. This string is examined by the file server to determine if a user is allowed to mount the requested file system. A special user id "none" is also provided which allows any file system to be mounted. Any access to such a system by a user only confers permissions on files available to everyone (world permissions). This scheme is not unlike that in NFS,

where an untrusted machine's request is mapped to a user "other". However, the use of textual names and a specified user naming policy allows simple *distributed management* of users without problems.

PLAN 9 addresses the problems of large distributed systems by relaxing many of the traditional UNIX semantics whilst providing a similar programming model. The solutions to authentication are adequate but not as flexible as might be desirable. The major problem lies in the lack of data coherency in the file systems. Whilst this may not be a problem in a traditional UNIX environment, it makes it difficult for PLAN 9 to be used as a parallel programming environment. The literature is not clear on some aspects of the system. For example, *reliability* is not mentioned and the treatment of faults is not discussed, although the use of optical disk for backups is examined.

2.2.5 Conclusions

The four architectures described attempt to provide data sharing in distributed machines through their file systems. Each succeeds in some respects (SPRITE provides a good data coherency policy for example) but fail in others (it does not address the problems of distributed management). Any complete distributed file system could be designed by combining these various ideas.

But the treatment of all data as a file is not the most efficient way of handling sharing, since every access requires an explicit `read/write` system call. A better solution, if finer grain sharing is required, is to present files as memory mapped objects, so allowing sharing to be handled at the instruction level. This approach is taken in the following systems.

2.3 Novel distributed stores

Implementing a distributed store in the shape of a file system is, although the most obvious, not the only solution. Alternatively, data may be stored without files by simply defining that the structure in which it is held will persist beyond the lifetime of the process manipulating it. This blurring of the boundary between volatile, memory resident data

and persistent, file system data has been implemented in a number of systems and has important advantages.

Firstly, it removes the artificial difference between data "inside" a program and data "outside" a program. This problem is historic and results from the structuring of the hardware storage as "persistent" disk and "volatile" memory. Secondly, once a persistent address space is realised, it is no longer difficult to store and manipulate complex data structures, since the problems of transferring to and from files are removed. Thirdly, the sharing of data within parallel programs is greatly simplified.

Three such systems are MONADS, CLOUDS and PSYCHE. These are briefly examined below, with attention centred on the the object store, the principles used to maintain it's integrity and its protection mechanisms.

2.3.1 MONADS

The MONADS system [RHB+90] was developed at the University of St. Andrews. It makes use of a large persistent virtual memory in which data persists until deleted rather than when the creating process terminates. This removes any need for a file system in the conventional sense.

The MONADS system provides a single 60 bit persistent namespace. This is divided into 28 bit address spaces, each of which may contain code and data segments. Access to these address spaces is in an object-oriented fashion; the methods in the code segments within the address space being the only means to access the data within that space. These address spaces, or *modules*, are protected by capabilities which define who may invoke which methods within the module.

Capabilities are manufactured and protected by the kernel and it is not possible for an arbitrary program to circumvent this protection. Additionally, MONADS guarantees that an old capability can never be used to obtain access to new data. This is done by never reusing deleted address space. Although this places a maximum limit on the amount of data which may ever exist within the system, a million terabytes is accepted as a reasonable limit.

This system also implements a data integrity policy. This guarantees that, if the system should crash, the persistent store will be returned to a "correct" state. This means that the store will be causally correct after recovery; something which is very important since such a store is implicitly cross referenced and has no predefined structure (unlike a UNIX file system which is treelike) so any error could result in usurping of protection rights or the unintended loss of information.

2.3.2 Clouds

The CLOUDS operating system [DLAR91, DCM+91, BAHK+88] was developed at the Georgia Institute of Technology, USA. It implements a single level persistent object store similar to that found in MONADS, and is currently implemented on a network of Sun 3 workstations. Three types of machines exist in the network:

- Compute servers,
- Data servers, and
- User workstations[2].

CLOUDS uses a passive object and active thread model. Passive objects are memory structures containing code and data which have no computing element associated with them. Active threads form the computing resource. These move from object to object, executing code and manipulating data.

Objects are uniquely named and consist of a group of memory segments. Each segment resides at a different address within the object and contains a different type of data. Typically, an object contains four segments:

- Persistent code,
- Persistent data,

[2]These workstations run UNIX with a windowing environment providing remote access to the CLOUDS environment.

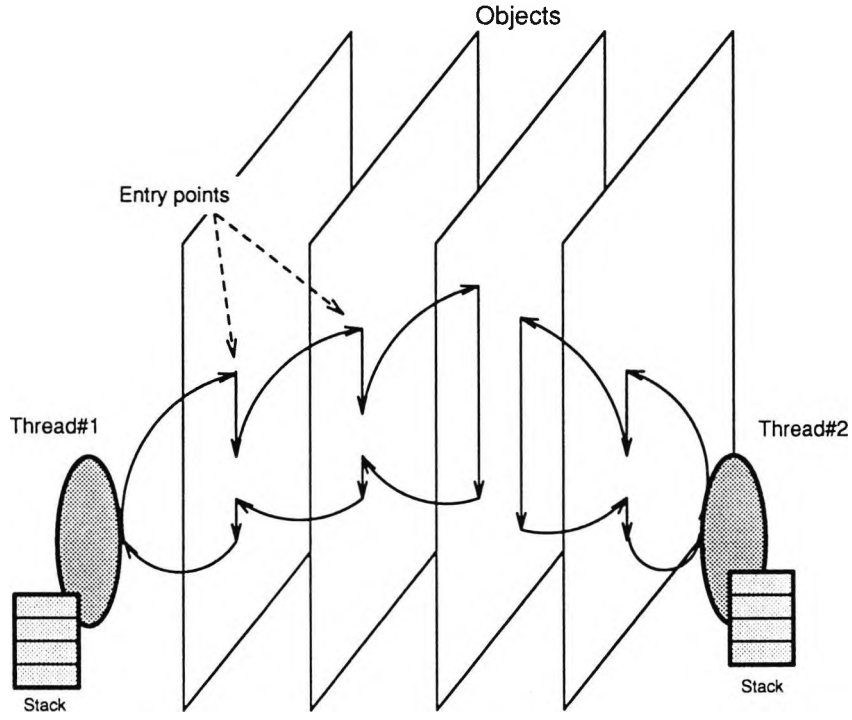


Figure 2.4: Passive objects and active threads in Clouds

- Volatile heap, and
- Persistent heap.

The code segment provides a procedural interface to the object. Data may only be accessed using these procedures by a thread moving into the object and executing them. Data is moved between objects, as arguments and return values, using a thread's associated stack segment. All segments within an object are shared by all invoking threads and this stack provides the only independent storage medium.

Each object contains two persistent data areas. Any data stored here will persist from one invocation of the object to the next. Volatile data is not preserved. This persistent data forms the only available storage system in CLOUDS and consequently, there is no need to support any kind of file system. Object flexibility is enhanced by the use of distributed shared memory techniques. This enables the same object to be invoked on physically separate machines but still maintains the illusion of physical data sharing. This has many advantages for the construction of large parallel shared memory applications. Of course,

it might be more efficient for all threads to invoke a given object in the same physical location. This is possible too.

A CLOUDS object provides *sharing transparency* through distributed shared memory, *access transparency* by use of unique names, and *location transparency* by divorcing any association between names and locations, an approach made simpler by the enforcement of a procedural object interface. In addition to this, objects provide both volatile and persistent storage at the instruction level.

Currently, no *security* or protection is enforced by the CLOUDS operating system [Ram92]. The ability to invoke a CLOUDS object is issued by a nameserver and any protection is implemented by the programming language.

CLOUDS supplies a flexible parallel programming environment. It provides a good shared data model capable of using distributed machines transparently. Reliability, data integrity and fault-tolerance are all handled by the use of data commit protocols, segment locking and replicated objects and threads. No attempt has been made to handle either authentication or protection within CLOUDS except at the language level. Additionally, the environment cannot be used directly and must be accessed through a UNIX system. *Distributed management* issues are not addressed.

2.3.3 Psyche

PSYCHEwas developed at the University of Rochester, USA [SLM+88, SLM89a, SLM89b, SLM89c]. Although it does not provide a persistent distributed store, it does have many attributes worth consideration for use in a general distributed store.

PSYCHE implements a single consistent namespace. For shared data, this appears identical to all observers so providing good *access transparency* and *location transparency.* Unlike CLOUDS, this namespace maps directly onto the virtual memory system allowing it to be accessed directly rather than via object procedures, so providing simple *data sharing.* The current implementation, which is on a BBN Butterfly machine, has only a 32 bit address space which causes problems because it is insufficient to provide a complete single level store. Consequently, several modifications have been adopted to allow bigger structures

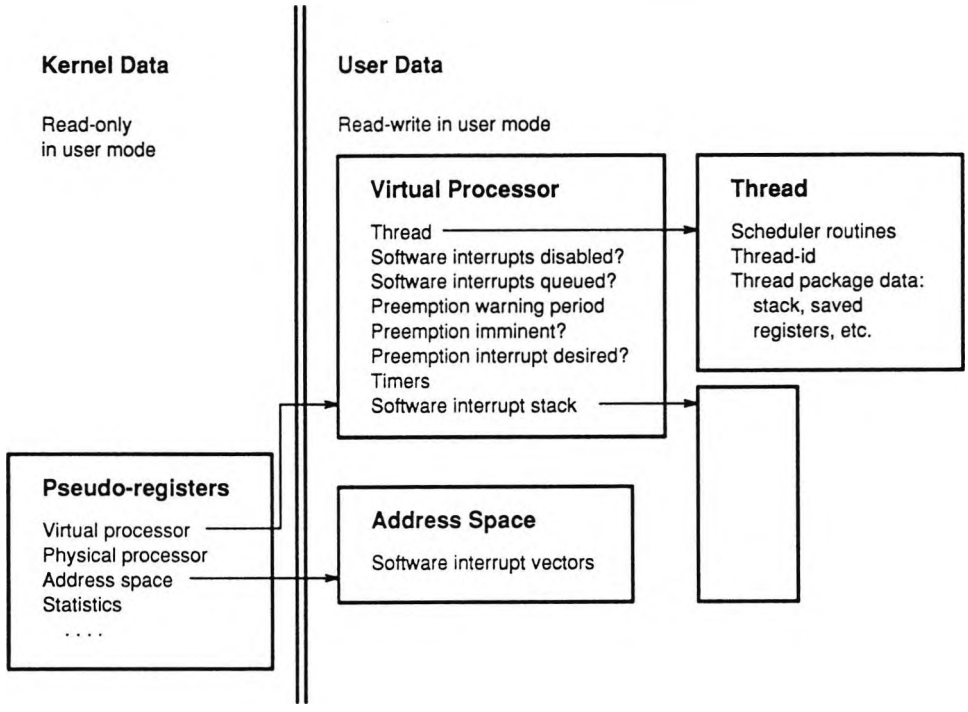


Figure 2.5: Psyche kernel/user interface

to be supported without severely compromising the simple architecture. Ultimately, this 32 bit addressing barrier will be removed by new processor architectures with larger virtual address spaces.

The virtual address space is divided into four sections; one for the kernel, one for sharable data, one for private data and one for paranoid data. Data and code are contained in *realms*. A realm in PSYCHE is synonymous with a segment in CLOUDS. Each realm may be protected from others within the system; it may be inaccessible, read only, read and written, or executed via invocation of internal protocols. An invocation-only realm is similar to a CLOUDS object.

The emphasis in PSYCHE is on providing a flexible environment where many different parallel computation models can co-exist. Rather than implementing a set of these in the kernel, a powerful set of primitives is provided over which parallel processing models may be implemented. This environment passes much of the scheduling control to the parallel applications which are run by "virtual processors". These virtual processors communicate with the kernel using a shared memory kernel/user interface. This mechanism provides

much greater control for the user without circumventing the protection imposed by the kernel (figure 2.5). Each virtual processor may run a number of processes, all sharing the same *protection domain* (a protection domain is a set of realms, each with varying permissions). This situation can be compared with a UNIX process running many threads but it is more flexible. Under PSYCHE, each of these threads is able to migrate to other protection domains, and call and block in the kernel without affecting other threads.

Protection of realms is provided by a simple capability *security* system. Each realm is associated with an access list consisting of <*key,right*> pairs. The possession of a <key,right> by a thread gives it the access rights designated by the "right" field. The "key" field range is large and capabilities are selected at random from within it. This provides probabilistic protection by making it unlikely that a correct <key,right> could be guessed. The larger the key field's range, the more time consuming it becomes to guess a capability. Access rights are only determined when a realm is first used by a thread and incur no additional costs thereafter. The flexible nature of the kernel design allows these accesses to be resolved in a lazy manner (as they are needed) rather than explicitly at the beginning of program execution. This avoids the need to resolve potential accesses which may never be used.

PSYCHE was not designed to support *reliable* operation. Data resident in the system is always volatile and must be loaded from external file servers when the machine is initialised. Consequently, there is no need to guarantee data integrity in the store; since there is none. PSYCHE was designed to provide an experimental parallel programming environment and not as a general operating system. The inclusion of a distributed persistent store is outside the aims of the project. Similarly, it operates on only a single NUMA (Non Uniform Memory Access) machine and not on a network of distributed machines. This removes the necessity to address the problems of network and machine failure as well as *distributed management.*

PSYCHE differs from other systems in three major ways. Firstly it tries to provide a flexible parallel programming environment and does not implicitly force the use of any one paradigm. Secondly, by providing a single memory space the sharing of data is greatly simplified. Thirdly, much of the control of processes is passed from the kernel to the user

so enabling the users to manage the processes as best suits the application.

PSYCHE is not a general purpose distributed operating system environment. It ignores the need and problems associated with the support of a persistent store in a distributed system. It also assumes the use of a reliable, secure network. If PSYCHE were used as a general system it would offer no authentication of protection from bogus machines on the network. It is also assumed to be a single integrated system and cannot be distributively managed.

2.3.4 Conclusions

MONADS, CLOUDS and PSYCHE are all operating system projects, the storage system is merely a part of each. In no case have all the requirements of the store been addressed; for example, distributed management issues are not addressed well in any of them. However, the systems do propose a simpler form of storage management, by combining volatile memory with persistent store to provide a single unified resource. The systems differ in the way this storage system is managed. CLOUDS and MONADS maintain a single namespace with pieces encapsulated in object where inter-object data sharing is impossible. PSYCHE provides a single flat storage system (though compromises this because of address space restrictions) in which it is possible for any data to be shared directly. PSYCHE may provide the rigid encapsulation of the other two systems but they cannot provide its simple data sharing environment. PSYCHE's store model is therefore more flexible (even if it does not, in fact, provide persistence).

CLOUDSintroduces distributed shared memory as a means of providing data sharing between physically distinct machines. Such a mechanism is essential if a unified resource approach is adopted in an operating system designed to operate on a network of machines. By absorbing access to remote data into a common store, a process need not be aware of the origin of the data it is manipulating nor the location of another process it may be sharing it with—it just treats the data as random access memory. As with the persistent and volatile data unification, the location of the data is no longer the problem of the application and so further simplifies data sharing. Methods to implement distributed shared memory are examined in the following section.

2.4 Distributed Shared Memory

In the past few years there has been a great amount of interest in the subject of *Distributed Shared Memory* or *DSM*. Unlike parallel machine architectures such as physically shared memory which is not scalable, or message passing, which is difficult to program with, DSM offers the possibility of scalability without loss of programmability.

The following examines various schemes to provide DSM. These schemes only address the problem of *data sharing* and leave other issues to the supporting operating system.

2.4.1 What is distributed shared memory?

DSM is a technique for giving the illusion of physically shared memory on machines which have distributed memory and an interconnection network. Much of the work on DSM has arisen from bus based cache coherency policies [AB86] where cheap broadcasting is available—in DSM this is not the case. Various mechanisms have therefore been developed to provide the same semantics (a read always returns the last written value) as the tightly coupled broadcast facility.

The following examines various implementations of DSM. Many of them share common features in their methods for maintaining data coherency and locating data in a distributed machine. After these systems have been analysed, we conclude by discussing their appropriateness for a distributed operating system. Each DSM system is analysed with a number of requirements in mind:

1. **Coherency model** - A DSM system should provide at least one well defined coherency model. This usually includes strict causal coherency (every reader always see the last value written) although weaker methods may be available (where a read may return a recently written value but not necessarily the most recent).

2. **Locking** - A DSM system may include explicit locking of data. This may allow data to be obtained for reading and not allow any changes to be made until released.

3. **Scalability** - Not all DSM systems scale. This may be due to the method used to

manage the data distribution or due to the number of message exchanges necessary to maintain the coherency model.

4. **Software/Hardware implementation** - Some DSM system are implemented in software, some in hardware. Hardware systems are invariably quicker but the software systems support more coherency models. What compromise should be made for an efficient system?

2.4.2 Ivy

Ivy [TSF90] was implemented on the Apollo Domain. The virtual memory system on Apollos is divided into two sections. The first section is private, contains process specific information, and cannot be accessed by other processes. The second section is public and is shared by all processes in the system.

Data coherency

Data is shared at the page level. This enables the virtual memory system to be augmented to provide the necessary page faults to handle data coherency; read protection on pages allows them to be shared coherently, attempted writes allows a page fault to be taken to force invalidation of copies before the write may proceed. This protocol, called write-invalidate, is commonly used in DSM systems since being proposed in [Li86]. In order to operate this protocol without broadcast, it is necessary for every page to have a designated owner and a copyset. The owner of a page is the holder of the copyset, indicating which other machines hold a copy of the page. When a new machine requires a copy, it requests it from the owner and is added to the copyset. When the owner wishes to write to the page, invalidations are sent to each machine in the copyset. If another machine wishes to write to the page, it first requests ownership and the copyset, and then proceeds with the invalidations and writes.

Data locating

IVY has investigated various schemes to locate pages in a system. These are:

- Central server
- Distributed servers
- Probable owner chains

A central server uses a single machine as an ownership coordinator. This machine holds an entry for each active page which details the current owner. When a page must be located, the request is sent to this server and forwarded to the owner. The scheme has two major disadvantages; firstly a single server providing this service creates a bottleneck; and secondly, the server must be informed whenever an owner moves.

Distributed servers solves the first of these problems by hashing the page number onto a set of servers; each server provides the location information for a fixed set of pages. With probabilistic hashing, the request should be spread equally amongst them. The problem of informing the server of ownership movement is not addressed.

Probable owner chains discards location servers, instead using a *probable owner linked list* [Fow86]. These chains initially point at a location hashed from the page number, and are subsequently modified by additional ownership information delivered when copies of pages are requested, invalidates received, or ownership transferred. When a copy is required, the request is sent to the probable owner. If this machine owns the page then the request can be serviced, else the request is forwarded to that page's probable owner. This scheme means that for N machines, a request can take N-1 hops to reach the owner. In practice it is rarely this bad.

2.4.3 Clouds

CLOUDS [DLAR91] is a novel object based operating system, the general structure of which has already been examined in §2.3.2. It operates four different DSM policies using software drivers operating at the page level.

Data coherency

The smallest unit of memory allocation in CLOUDS is the segment. DSM policies may be applied on a segment basis, though sharing and locking is made on a page basis. These policies are briefly described below.

1. **None**

   No DSM policy is employed to keep pages coherent. Instead, the pages are moved between machines as they are requested so only a single copy is ever present in the system.

2. **Weak Read**

   Weak read coherency has two forms. In one form it is similar to IVY's write-invalidate policy except that, instead of invalidating page copies on writes, updates are sent to other page copies. These updates may occur at any time without prior notification. In the other form, a copy is supplied to a machine when explicitly requested and without any coherency being maintained. The copy must be explicitly discarded after use.

3. **Read**

   Read coherency obtains a copy of the page and locks it for read access, so preventing any writes or *Policy 1 reads* taking place. Many *Policy 3 read* copies may be taken and locked concurrently but all locks must be released before other operations are permitted.

4. **Read/Write**

   Read/write coherency obtains an exclusive copy of the page and locks it. No other copies of the page are available until the lock is released.

Note that the last two coherency policies include implicit locking. These locks allow page data and ownership to be guaranteed for the duration of the locked operation; something which can dramatically increase the performance of a DSM system by preventing page thrashing. The policies also defines that when a page lock is released, the page copy is

written back and discarded, based on the assumption that it is no longer required. This reduces the need for invalidation between machines in order to maintain data coherency.

Data locating

In CLOUDS unlike IVY, the creator of a segment is always its owner and the owner of its pages. All DSM requests are forwarded to the owner whose responsibility it is to maintain the correct coherency. This scheme greatly simplifies the problem of locating a page since the owner is fixed. However, it has the disadvantage of forcing all requests for a page of an object through the same machine and takes no account of data or machine locality.

2.4.4 MemNet

MEMNET [Del88] is one of the few hardware distributed shared memory (HDSM) implementations and was developed at the University of Delaware. It was originally conceived as a method of treating a network as a random access device rather than an I/O device. In doing so however, a device was constructed which operates according to DSM principles, but on physical memory rather than, as in software systems, on virtual memory.

Data coherency

MEMNET operates a single coherency policy, that of write-invalidate. The way in which this is done is very different from the two systems described so far. The use of hardware allows much smaller data items to be handled since it is not restricted by the virtual memory's page size or the associated overheads of manipulating coherency in software.

These data items, each 32 bytes in size, are exchanged by MEMNET nodes using a token ring network. For example, when a node requires a particular data item, it inserts a request into the ring which is passed from node to node. When a node receives such a request, it determines whether it holds a copy of the desired data and, if so, places it on the ring, marks the request as satisfied and passes it on. If a node cannot fulfil a request, it is passed on unaltered. A request eventually returns to its originator with the data within

it. The originator then places the data in the node's memory for use. Similar schemes are used to pass invalidates between machines in order to maintain data coherency when writes occur.

To avoid the situation where every node reserving space for every data item in the system, every data item has a home location. This is a guaranteed place to which an item can return if it is overflowed from another node's cache.

Data locating

Locating any data item in MEMNET is trivial; a request is placed on the ring, it circulates around the ring, is fulfilled by a node along the way, and finally returns to its initiator. If a returned request has not been fulfilled, the data does not already exist in the system and so may be created. The use of a ring structure makes this operation simple, but inefficient, since all nodes must share the same linear network and a request passes though all nodes for any operation. Such a system has scalability problems.

2.4.5 Scalable Coherent Interface

Scalable Coherent Interface (SCI) [KABJ89, SCI91] is an IEEE standard interface designed to couple up to 64,000 machines together in a cache coherent fashion. Unlike all the other systems reviewed in this section, this is the only scheme designed to scale to massively distributed systems and do so in hardware. Also, unlike the other hardware systems, the network which supports SCI need not have any regular structure.

SCI is intended to be used with many networks but fibres are particularly relevant to it. Additionally, the latencies of large scale distribution mean that the protocols, used for request and acknowledge messages in the network, must be capable of reliable communication in the presence of errors and also tolerating long delays for replies. SCI therefore uses a simple sequence number based transmission scheme to guarantee requests are received and handled in order, even when the maximum of 256 is outstanding.

Data coherency

Data coherency is maintained using a strict write-invalidate policy on pages of 64 bytes in size. In implementation it is unlike other write-invalidate systems. To allow for massive networks of machines, the copyset is kept as a doubly linked list distributed across the relevant machines, each machine which holds a copy holds a link in the list. To obtain a copy of a page, the requester adds itself to the head of the relevant list and obtains a copy of the page from the previous list head. If a machine wishes to modify a page copy, it removes itself from its position in the list, adds itself to the head of the list, and then invalidates all the copies in the list except itself. Once this is done it may make the modification.

The major advantage of this system is its scalability. The only scaling problem is the size of the addresses stored in the SCI requests and replies. However, for systems where large numbers of machines share the same data, the copyset list can become very large, so increasing the time to invalidate it when a write is made (the list means that invalidates are made sequentially from machine to machine).

Besides a write-invalidate DSM policy, SCI supports the notion of remote device access. This allows a device to be interrogated correctly, without caching, even from a remote machine.

Data locating

Each page entry in an SCI chain maintains a pointer to the current head of the chain. This is necessary for efficient removal and insertion of a list entry when a page is to be modified. This also allows new copies of a page to be located quickly and efficiently. However, no details are published on how a page is initially found when it is available in the system but unknown to the requester.

SCI is the only DSM system examined which is designed to operate in a large scale environment. However, it is unclear how well it handles the faults which will inevitably occur in such systems. Although the use of sequence numbers and retransmissions helps this problem in the case of partial network failure, the unexpected failure of a machine

would appear to result in the failure of all the copyset of which is was part. This propagates failures to other machines. In many cases there is no loss of information and such failures should not be fatal.

2.4.6 Choices

CHOICES [MR91] is an operating system developed at the University of Illinois at Urbana-Champaign. Unlike many modern operating systems, CHOICES was written in C++ as an experiment in the use of an object oriented language for operating systems work.

In many ways, the structure of CHOICES is much like MACH except that distributed shared memory page faults are handled by the choices class *DistributedMemoryObjectCache*. This class is responsible for maintaining the necessary data coherency between the sharing machines.

Data coherency

Data coherency is maintained using IVY's distributed manager and write-invalidate protocol at a page level. Besides this scheme, CHOICES also provides two additional features. Firstly, page locking is provided to allow atomic updates to data structures. Acquiring the lock on a page guarantees the ownership and uniqueness of the page until it is released. Secondly, delta times are provided. A delta time is a fixed time period for which a page copy is retained before any pending requests pertinent to it are processed. For example, it allows read copies of pages to be held for a set time without having them invalidated by another node's write access. The write access is stalled until the delta expires and the invalidate message acknowledged. Delta timers have been shown to increase the potential efficiency of a DSM machine quite dramatically [CF89, FP89].

Data locating

Data locating is done using a distributed ownership manager. Every page in the DSM system is controlled from a preset machine, determined by a function on the page's address.

This machine is always in possession of any information regarding the number of copies in a system and the place of the current owner.

CHOICES' DSM scheme provides only a re-implementation of other work and offers no novel features other than a study of implementing such a system in C++.

2.4.7 Mach and Agora

MACH [ABG+86] is a UNIX compliant micro-kernel operating system designed at Carnegie-Mellon. Although it was not designed with DSM in mind, it provides support for such a software mechanism through the use of *external pagers*. An external pager may be assigned to each region in MACH to handle page faults. Rather than force the kernel to handle such faults, they are forwarded as messages to the relevant pager. This provides for a more flexible memory management strategy determined on a region by region basis.

Data coherency

AGORA [MR91] was written to use MACH's external pagers to provide DSM for networks of machines. The system is highly integrated with the applications using it so as to provide more flexible handling of data coherency. Instead of exchanging pages, like most software DSM schemes, data structures are exchanged. This allows only the required amount of data to be transferred by the DSM (unlike many systems where a page may be transferred when only a few bytes are of interest) but means that data structures must be carefully allocated so that the correct page faults are generated when the data is not present. For this reason two small data structures could not share the same page, since only the accessed one would be copied to the machine when a fault occurred and no fault could then occur for the other which was not present.

The coherency model is also relaxed. Instead of forcing any particular coherency policy on the application, coherency is handled lazily. When a read is made, it is always made from a locally cached copy. If this is not present, it is first obtained from a master copy. When writes are made, they are written directly to the master. No invalidates are performed so cached data can become incoherent. However, the update is propagated from the master

lazily so that at some time in the future, all copies will again be up-to-date.

The problem of stale data is handled explicitly with synchronisation primitives. The designers of AGORA considered that such synchronisation is usually present in parallel programs and this requires little additional overhead and is easily compensated for by the improved performance of a weakly coherent data model.

Data locating

This is trivial under AGORA since a master copy is always maintained and never moves. To obtain a copy, a request is always made to the same place for the same data structure. As long as the master copies of data are spread evenly throughout a system, it is unlikely to cause a bottleneck. Of course, if a bottleneck does arise due to the way in which the data was accessed, there is nothing the system can do to alleviate the problem and such a task is left the the applications writer or "clever" compiler.

Unlike other DSM systems, AGORA does not offer a strict causal coherency mechanism so cannot provide a "true" shared memory environment by default. This makes it more difficult when initially writing parallel program since no well defined data sharing is available. However, the weak coherency policy does allow the latency associated with strict policies to be hidden once the application's performance becomes important.

2.4.8 Mether

METHER [MR91] is not an operating system but a set of mechanisms for sharing memory across a network of SunOS 4.0 machines. Unlike most other DSM systems, METHER does not provide any explicit data coherency but expects the application to make the necessary request to fulfil its requirements.

Data coherency

No explicit coherency is provided. Instead, a number of system calls provide the coherency scheme. When a write is made to a page, the page must be owned. However, these writes

are not propagated to other copies of the data nor are these copies invalidated. If a machine holding a copy requires a new instance of the data, it can be obtained in three ways. Firstly the owner can explicitly propagate the changes, secondly a copy holder can explicitly invalidate the copy so a new one is requested when the data is next accessed, and thirdly a copy holder can explicitly request a new copy.

Besides these coherency protocols, METHER also supports *data driver page-faults.* When such a fault occurs, a request for the data is **not** made, instead the process is halted until the data is explicitly provided by another process. Such page faults are completely passively (ie. there is no intervention from the faulting process).

Data coherency is supported by providing two different data sizes, a full page (8192 bytes) and a short page (32 bytes). A short page is the first 32 bytes of a full page and allows small data items to be exchanged more efficiently.

By not providing any explicit coherency scheme, the application can use the available primitives as it desires. However, as with AGORA, this makes the development of program difficult even though the flexibility to change them for better performance is greater. This scheme could also allow an application to tolerate failures although any kind of recovery would be left up to the implementation.

2.4.9 Stanford DASH

The Stanford DASH machine [LLG+92] is one of the most recent efforts to implement DSM in hardware. The current implementation is based around a mesh-interconnect, the Mips R3000/R3010 and some specialised hardware developed at Stanford to support the DSM strategy. At present a 16 cluster machine (a cluster is a number of processors organised as a physically shared machine) is the largest supportable by the current DSM hardware.

Data coherency

By default, DASH supports a write-invalidate coherency policy. This operates in the same fashion as IVY with a distributed manager. Besides this, various extensions have been provided. DASH supports out-of-order memory accesses; a situation where sequential memory accesses may take place in a different order from that in which they were issued. In most cases, there are no problems with this although there are algorithms for which it is necessary and so DASH provides *fences* to enforce ordered accesses. A write-update policy is also supported. When a write is made to a data copy, the modification is sent to all other copies also. In some circumstances such accesses are more efficient. Another form of write-update, called *deliver*, is also supported. This policy explicitly delivers a copy of a uniquely held datum to a specific destination node. This closely resembles explicit message passing.

DASH also supports two forms of locks. Firstly, uncached atomic *fetch-and-increment* and *fetch-and-decrement* operations are supported. Here a request is made to a statically located lock and a single data item fetched in response. Secondly, a hardware assisted *test-and-set* operation is provided which is termed *queue-based locking.* When a lock is acquired by a cluster, all other clusters attempting to acquire the lock obtain cached copies on which they spin. When the lock is released, rather than invalidating all cached copies and allowing every node to compete for the lock again, only one copy is invalidated, so only one node attempts to acquire the lock. Timeouts are included in this mechanism so that when a lock is released and a copy invalidated, if a new copy is not quickly requested, then the process attempting to acquire the lock is considered swapped out, and so another cluster is invalidated.

DASH does not appear to support delta timers. It relies instead upon its other mechanisms, such as lock queues and explicit delivery of data, for performance enhancement.

Data locating

As mentioned above, DASH uses a distributed manager to hold the locations and copysets of pages. This makes every page easy to find but limits the size of the machine by

restricting the copyset size (currently to 16 processors per cluster) and limiting the number of pages in the system. Potentially, any given cluster could be saturated with requests for data to its manager. In practice this is unlikely because the pages are hashed across the managers in a manner likely to destroy manager locality for adjacent pages.

In general, DASH is the most comprehensive HDSM scheme so far implemented. However, it lacks the scalability of SCI. The current implementation only includes mechanisms for strict coherency policies. It will be interesting to see it the speed advantages of hardware make weak policies unnecessary.

2.4.10 Conclusions

How do these systems relate to the design of the DSM for an operating system? Nearly all the systems, except SCI, limit the size of the machine by the representation of the copyset as a table or bit vector. Additionally, only SCI supports a retry protocol necessary for highly distributed DSM systems. However, even SCI does not appear to handle failures within the copyset lists.

It is also obvious that as DSM research has progressed, the use of a single coherency policy has been shown to be insufficient. Hardware solutions are also being favoured for their ability to handle smaller shared data sizes. The DASH machine provides a good demonstration of current DSM hardware, supporting a single major policy but making a few additions to allow others to be explicitly used when desired. It is also apparent that strict coherency, whilst being provided, is not always necessary and so the ability to relax it and gain performance benefits is useful.

The inclusion of locking in DSM, rather than on top of it, allows various optimisations on data usage. The selective invalidation for locks is particularly interesting in this respect as is the explicit locking of acquired data for set periods of time using delta timers. All the examined locking mechanism aim to reduce DSM coherency traffic when a lock is being obtained and so improve both performance and network utilisation.

None of these systems addresses the consequences of unreliability in the machines taking part in a DSM exchange; only SCI does any retrying and this is only to handle intrinsic

network errors. As DSM system grow to be utilised by large number of processor in distributed environment, some form of reliability becomes essential.

2.5 Fault-tolerant active data stores

Too often fault-tolerance is disregarded when designing a new operating system and then added later as an afterthought. As the size of distributed systems increase, so does their unreliability and the need to provide fault tolerance becomes more important. Attempts to add reliability after the fact [CDG92, BG91] have spawned numerous different approaches, none of which handles all problems or copes with all situations.

In the following section we will examine the provision of fault-tolerance in active stores (such as volatile virtual memory). The issues are different from those in entities such as databases, where reliability is concerned with the persistent storage and transaction processing.

A fault tolerance system for general applications should be able to:

1. **Provide fault tolerance without modification to applications**, so allowing any application to make use of the service rather than ones specifically designed to do so;

2. **Provide a complete solution rather than support only some services**, since a solution which, for example, supports fault tolerant memory but not files is useless for many applications;

3. **Support both uni- and multi-processor applications**, so allowing parallel applications on distributed machines to be handled correctly, and

4. **Provide fault tolerance efficiently for the general, fault free, case**, since faults are rare and protection against them should not have a prohibitive cost.

The systems described below all solve these problems to some degree but no solution adequately address all these issues.

2.5.1 MONADS

The general organisation of MONADS has been discussed in §2.3.1, where it was mentioned that MONADS provides a data integrity policy; this is implemented using a checkpoint and rollback policy.

In MONADS a module can be checkpointed at any point in its execution. When this is done a copy-on-write version of the module is taken (this can be done quickly) and the module allowed to continue. The checkpointed image is then lazily written to disk using a two-phase commit protocol[3].

By closely linking this reliability scheme to the virtual memory system, MONADS provides a single reliability system which solves all data integrity problems. However, the solution does not address the problems encountered in a parallel machines. This is because of the self-contained module structure imposed on data and code organisation only allows well defined data exchanges between modules (via method invocation and returns) and not more general parallel programming. This makes it easy to track intermodule data dependencies but difficult to write parallel programs. The situation is further simplified since there is no means to invoke a method in the same module on different machines in parallel, hence no means to support distributed shared variable parallelism.

2.5.2 Clouds

As already described, CLOUDS and MONADS provide a similar representation of data to applications. This results in the necessity to prevent data corruption, in their persistent object stores, due to machine failure. In a distributed environment, *resilience and reliability* are a critical problem. CLOUDS address this problem in two ways; firstly by providing a checkpoint mechanism similar to MONADS and secondly by providing replicated threads.

CLOUDS assumes that maintaining the integrity of an object is not always necessary. Temporary data, for example, need not survive a machine crash. Several levels of data integrity management are therefore provided. The desired policy decision is assigned to

[3]More detail of this protocol is given in chapter 6.

the thread performing the operation; an *s-thread* provides no integrity guarantees and is the standard default, a *cp-thread* maintains data consistency for all objects modified.

When a cp-thread accesses a segment within an object, the segment is locked for reading or writing depending upon the operation performed. Other cp-threads are then prevented from accessing these segments. When the thread finishes with the locked segments, modifications are committed to disk using a standard two-phase atomic commit scheme. The segments are then unlocked and made available to any other cp-threads which may be waiting to use them. For a more complete explanation see [CD89].

Data integrity in such a system is only half the problem. Complete fault tolerance is also necessary to guarantee that a machine failure during computation does not jeopardise program completion. CLOUDS achieves this by using replicated objects, replicated threads and an atomic commit mechanism to ensure only one, correct, result is produced. To allow for faults, a number of versions of the computation are performed in fault independent domains. Each computation makes use of replicated objects and replicated threads. All computations are performed in parallel. When one completes successfully, it is chosen to commit its changes. If this commit is also successful, all replicated versions of the objects are removed and all replicated threads destroyed. If the commit is unsuccessful, then the objects and thread are discarded and the completion of another replica waited for. The process is then repeated. With sufficient initial replication and few faults, the computation will eventually complete. This mechanism is described in full in [ADL90].

2.5.3 Recoverable distributed shared memory

Providing recoverable distributed shared memory is an attractive proposition for two major reasons. Firstly, standard uni-process programs may use the system to provide recoverable virtual memory. Secondly, multi-process programs may use the system to provide not only recovery for their virtual memory but also recovery for any interactions among themselves (with message passing recovery must also be provided in the message system).

In [TH90b] and [TH90a], a scheme is proposed to support recoverable DSM. The scheme relies on three major properties. Firstly, when a cooperating machine fails, the failure is

not terminal and the machine can be restarted. Secondly, each machine supports its own checkpointing disk. Thirdly, each machine is assumed to comprise two cpu's and enough memory to hold the entire shared address space.

Two basic recovery principles are proposed, one to recover page data and the other to recover DSM state information. The first principle is simply implemented; whenever a request is made to a DSM server for a page copy, if the page has been modified by the holding machine then all modified pages are first checkpointed before a response is made. This guarantees that data which cannot be recovered is never shared and so avoids any interdependence problems when a machine fails.

Recovery of DSM state is more complex and relies on the use of a distributed database to hold the DSM state information. However, instead of operating a standard two-phase commit policy to maintain coherency in the presence of failures, a unilateral commit protocol is proposed which involves far fewer message exchanges. This aside, the state recovery operates by periodically committing checkpoints of local state. Also, any state changes are first logged to disk locally before being exported to other machines. After a failure, the most recent checkpoint is recovered and the log rolled forwards to replay any potentially lost messages (duplicates are handled by using sequence numbers).

This system works well but relies heavily on the saving to disk of small and frequent state changes. This must be done whenever an attempt is made to export modified data or when a DSM request is made. Both these events are common in parallel algorithms and the effect of this logging on performance can be dramatic.

2.5.4 Conclusions

Two method are generally used to provide fault tolerance; one is checkpointing where "snapshots" of the current application's state are taken, the other has replicas of the application running in parallel. Each of these strategies has advantages and disadvantages; checkpointing does not require multiple instances of the application but does involve time consuming disk logging, whilst replicas provide fast recovery and operation but require redundant applications running all the time. Neither solutions as described above are

particularly well adapted to a shared memory distributed machine architecture.

Recoverable distributed shared memory offers a simple way of supporting the necessary model. However, because of need to checkpoint all modified data before any communication, something a parallel application could reasonably be expected to do a lot of, recoverable distributed shared memory introduces a large performance overhead.

2.6 Summary

Some degree of unification of volatile memory storage, persistent storage and network communication systems taking some account of reliability has been attempted in MONADS, CLOUDS and PSYCHE. But each of these designs addresses only some of the problems associated with a truly unifying, reliable storage resource. Standard file system solutions (such as NFS and SPRITE) provide some degree of access transparency but do not combine with volatile memory in a coherent manner. CLOUDS and MONADS both provide a more unified storage model including some forms of fault tolerance, but do this at a large cost to the application.

The ideal solution should provide a single, unified data and name space which may be accessed efficiently by any process, regardless of its physical location, and tolerate faults without loss of service. The ARIUS operating system, detailed in the next chapter, is designed to provide such a system. It does this by providing a simple but powerful computing environment. This is viewed as a single machine when, in fact, the "machine" consists of many machines linked via busses and networks.

Chapter 3

The design of a 64-bit operating system

This chapter examines the design of the ARIUS [SW92] operating system, inspired by work on the ANGEL [WSO+92a, WSO+92b] operating system being developed jointly by City University and Imperial College London. ARIUS forms the base on which this work rests. Consequently, much of the background presented in the chapter is necessary for understanding this thesis' main research goals. Many of the ideas presented here are either original or applied in an original fashion. They will be highlighted as they are examined.

3.1 Introduction

This chapter presents the design of an operating system which aims to make better use of 64-bit addressing processors than is possible in the UNIX model. In doing this ARIUS attempts to unify various mechanisms:

- Many namespaces into one, global namespace.
- Threads with processes,

- Volatile memory with persistent file store,
- Shared with distributed memory,
- Local machine access with remote machine access.

This approach results in a cleaner kernel structure and, together with a few other design decisions, makes the provision of fault tolerance relatively simple.

ARIUS is designed as a *Cache Only Kernel.* This means the kernel contains only state which may be regenerated from the applications and the address space it supports. The design of a kernel in this way is original and the design of fault-tolerance is greatly simplified since there is no need to save any kernel state when checkpointing an application. For example, if a page fault from a process were lost, then simply restarting the process would regenerate the fault. Similarly if a number of unlock upcalls cannot be delivered due to a resource shortfall, see §3.7.2, then simply releasing the processes involved allows all locking requests to be regenerated.

To further simplify kernel design, ARIUS attempts to unify and simplify many operating system mechanisms which, for one reason or another, are conventionally separate. Often, such mechanisms are almost, but not quite, identical. ARIUS identifies these redundancies and attempts to provide a set of orthogonal primitives. Consequently ARIUS provides few resources:

- **Single Namespace** to provide uniform references to data and services,
- **Objects** in which to store data and instructions either temporarily or permanently,
- **Capabilities** to protect and validate object accesses,
- **Domains** to allow processes to access sets of objects,
- **Processes** to operate upon the objects, and
- **Synchronisation** to coordinate cooperation between processes.

With these few resources it is possible to provide a complete operating system environment without constraining users or processes to set organisational policies which are not the operating system designers' to make.

The next six sections of this chapter will examine each of these resources in turn. The final sections will deal with other aspects of the ARIUS system. These include; the need for dynamic linking in ARIUS, the structure of an ARIUS "machine", the I/O system, the services a typical system needs to provide, and UNIX compatibility.

3.2 A Single Namespace

The single namespace, or the single address space since one maps directly to the other, is the central feature of ARIUS's design. The small addressing ranges of microprocessors in the 1970's and 1980's forced operating systems to provide the multiple, independent, address spaces which are such a major feature of virtual memory operating systems today [Bac86, LMKQ89, ABG+86]. The emergence of 32 bit processors, offering large virtual memories matched by much smaller physical ones, was a step back toward the single address space, yet it is now commonplace for physical memory, including disk storage, to expand beyond the 32 bit limit. The 1990's has seen the advent of 64 bit processors, the first examples being the MIPS R4000 [MIP91] and the DEC 21064 [Dob92], both pushing the address boundaries up four billion fold.

ARIUS supposes the existence of such a massive address space. Instead of using the address space for just another UNIX like system, it takes a more radical approach; using the space to provide a single, persistent, uniform, address space shared amongst many tightly and loosely coupled machines. This use of a single address space makes the sharing of data in parallel applications almost transparent; firstly by allowing instruction level data sharing through the use of a shared memory model, and secondly by providing a single, coherent naming scheme.

3.2.1 Coherent naming

By providing a single persistent address space which maps directly to a single namespace, many multiprocessing problems are removed. The most important gain is the identity established between a datum and an address. In a multiple address scheme, an address is context sensitive, meaning something only to the process it is used within. Such schemes have proved unsatisfactory for parallel applications where independent contexts attempt to work on a common data set. To solve this problem, a variety of methods have been developed to transport data between contexts (marshalling [HL82]), to store them in files, or to pass context independent pointers (swizzling [Wil90]). Such methods involve both error prone complexity as well as loss in application performance. In a single persistent namespace none of these techniques are needed.

However, providing a single, amorphous space for sharing data is only part of the problem. As this stands it is completely unmanageable. Therefore ARIUS uses the notion of *Objects* to contain data. These objects are divided and managed by *Space and Object managers.* The management of object is considered below together with a complete description of what constitutes an object.

3.3 Objects and their management

If ARIUS were designed to manage a single CPU or a tightly coupled multi-processor machine, a single manager would be capable of handling all the space available in the machine. However for a truly distributed system an extra level of management is necessary to avoid the need for a central space manager (which would be both a bottleneck and a reliability problem).

The purpose of *space managers* therefore, is to share the namespace among locally requesting object managers. Instead of having object managers requesting small sections of the namespace many times from remote managers, the space managers allocate larger sections and provide it to local object managers as required. This circumvents two problems that would otherwise have affected the system. Firstly, scalability is increased by han-

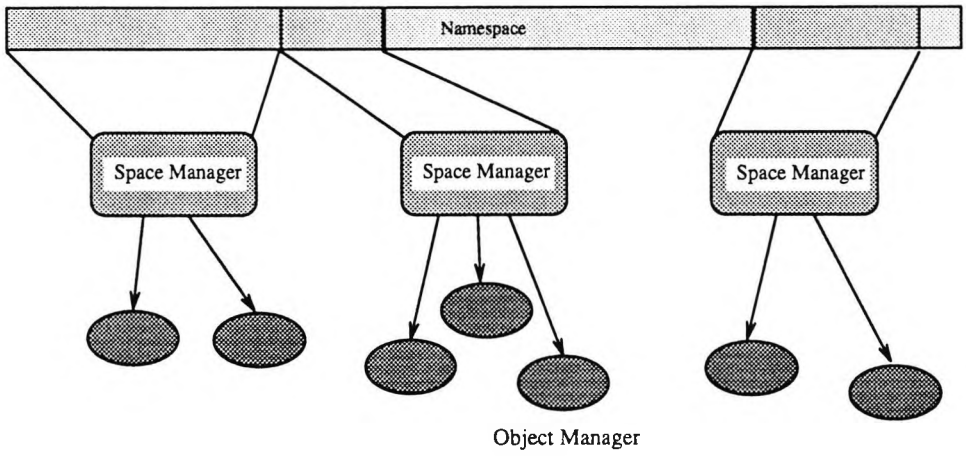


Figure 3.1: Namespace divided between a hierarchy of space and object managers

dling numerous small requests for space on the local machine. Secondly, failure of remote servers need not affect a local ARIUS machine, as long as the resident space managers had previously acquired sufficient namespace resources to fulfil expected local needs.

3.3.1 Object Managers

While the namespace is managed by Space Managers, objects are managed by *Object Managers*. A system is not restricted to a single object manager, in fact there are advantages in providing various different object managers. Firstly, different object managers can be designed for different purposes; one may provide general variable sized objects, another fixed sized objects designed for holding process state (which exists only for the lifetime of the process), and yet another to provide objects which can only be accessed by specific processors or machines[1]. Secondly, ARIUS is designed as a distributed system. Although many machines may cooperate in the provision of an ARIUS environment, they should not need to trust each other. The provision for multiple object managers allows a client to select a manager whom it trusts.

An Object Manager is responsible for providing storage space to requesting clients and granting access to them when presented with the correct capabilities. It is also the object managers' responsibility to remove objects which are no longer in use. Although explicit

[1]Memory mapped I/O devices for example can only be accessed from specific machine nodes.

object deletion is provided, clients cannot always be relied upon; either because of intentional misuse or error. It is impossible to provide simple reference counting, such as is used for UNIX file systems, to determine when objects can no longer be used; instead garbage collection must be employed. The actual implementation is left to the design of the manager. A manager holds information about all references to the objects it provides and so it is not an impossible task to implement efficient garbage collection. Of course, an object manager which provides objects which only live as long as the requesting process, greatly simplifies garbage collection.

3.3.2 An Object

Objects are provided by object managers to requesting clients. An object is a persistent, untyped, contiguous section of the namespace, in which data may be stored and later retrieved. All objects are created with a fixed base address and cannot be moved. This address is the only name for an object ARIUS understands; any other names are merely provided for the convenience of a user and will ultimately be presented to the system as 64 bit addresses.

Objects are created with a fixed size which cannot be changed. This is not as restrictive and problematic as it may first appear [vRTW89]. An object will usually only occupy physical space as that space is accessed, until then it places no demands on either volatile or persistent storage. Consequently, because of the large address space, generously sized objects can always be created "just in case".

Objects are accessed via capabilities which define both the object and the privileges inferred when it is resolved.

3.4 Object protection through capabilities

Object must support some kind of protection to prevent unauthorised access and modification. Generally it is only necessary to provide three forms of object protection:

1. **No access** - the object cannot be read or written,

2. **Read only access** - the object can only be read, and

3. **Read and write access** - the object can be read and written.

Other combinations, write only for example, make little sense. Although this provides general protection, extensions to this are possible to allow greater flexibility. For example, **execute only** might allow an object to be executed as a program although neither read or written. Other more specific additions are also possible. For example, UNIX provides a **setuid** bit in its file protection which effectively changes the user's identity when the file is executed. ARIUS provides a similar mechanism but does so in a more flexible way by using capabilities.

Capabilities [DVH66] are a well known and much used protection mechanism, particularly in earlier uni-processor operating systems [Lev90]. A capability is usually some form of typed data, the possession of which allows the process to access an object or service in a set of defined ways. In Amoeba [MvRT+90, TMR86] for example, a capability is a cryptically protected datum which may be passed freely between processes and, when used with the relevant server, gives access to a resource. These types of capability systems have various problems. Firstly, a capability can be used by anyone; possession is all that is needed for it to be valid. Secondly, once a capability has been given away, it cannot be revoked to prevent its use. Thirdly, encryption does not provide complete security and forces the cost of cryptically checking capabilities, in case of forgery, every time one is used.

The advantage of a capability based system is that is does not base protection on some system imposed principle, such as the notion of user and group identifiers in UNIX. ARIUS capabilities aim to provide flexibility and protection, but without the problems.

3.4.1 A Capability

ARIUS's capabilities allow access to objects in specific ways. Various types of access privilege are provided:

Read – Read access allows the contents of an object to be read by the resolving process.

Write – Write access allows the contents of an object to modified by the resolving process.

Execute – Execute indicates this object contains executable code.

Delete – Delete allows the object to be explicitly deleted.

Create – Create allows new capabilities to be created. These new capabilities must have a subset of the privileges of the parental capability.

Gate – Gate access allows an object to be present but neither readable nor writable. Access to it must be via subroutine call, upon which the object is entered at a controlled entry point in a readable state. Exact operation is detailed in §3.5.2.

The first two of these privileges are provided by the memory management of the hosting system, the others provide specific ARIUS services[2].

ARIUS does not use encrypted capabilities but, like Mach [SJR86], keeps them stored in typed and generally inaccessible storage; the object manager providing the object. When a capability is issued, the only information returned is a reference which identifies the capability to the manager when it is later re-presented. This information consists of the base address of the object and the address of the capability control information in the manager (128 bits in all).

But this is hardly secure, even if the search space is 128 bits in size. Simply presenting random numbers might yield access to an object at some point, especially if a rogue process can identify a pattern in the capability information returned. However, as we shall see, access to an object does not only require presentation of a valid capability.

3.4.2 Dependent capabilities

In a multi-user system, it is often necessary to temporarily change the effective user identity of the process being executed. Under UNIX this is achieved through the **setuid** file protection bit. However, such a system is very coarse, the process becoming the designated

[2]It should be noted that the absence of **execute** privilege does not necessarily prevent execution of code. Although desirable, many memory management systems do not support such a concept. Its presence modifies process behaviour when these objects are first used (see §3.6)

user, with access to all that users resources rather than access to the few actually required. ARIUS implements a finer scale mechanism using dependent capabilities.

The implementation of ARIUS' capabilities is original in that possession of a capability is only part of the requirement for obtaining the relevant object. To request an object from a manager, besides presenting a capability, a client must fulfil a set of requirements associated with that capability. These consist of a list of object and access privileges which must be possessed by the client for the capability to be valid. Therefore, although a process may know the capability to access the password database, unless it happens to be running the correct password program in the correct manner (via a gate perhaps) it cannot make use of it. Although a capability may be passed to another process, once received it could be completely useless.

This system is not only secure but extremely powerful. It provides object protection without the use of additional namespace, such as user-ids, and does not force specific protection policies on processes.

3.4.3 Operations on capabilities

By design, the operations which need to be performed on a capability are few. These are:

- **Create Object** (capability, size, privilege, dependent 1, dependent 2, ...)
- **Resolve Object** (capability, privilege)
- **Dissolve Object** (capability)
- **Delete Object** (capability)
- **New Capability** (capability, privilege, dependent 1, dependent 2, ...)

Create Object requests that the manager specified by the capability, creates a new object of the given size with the given privileges. The list of dependencies provides the condition under which the returned capability is valid.

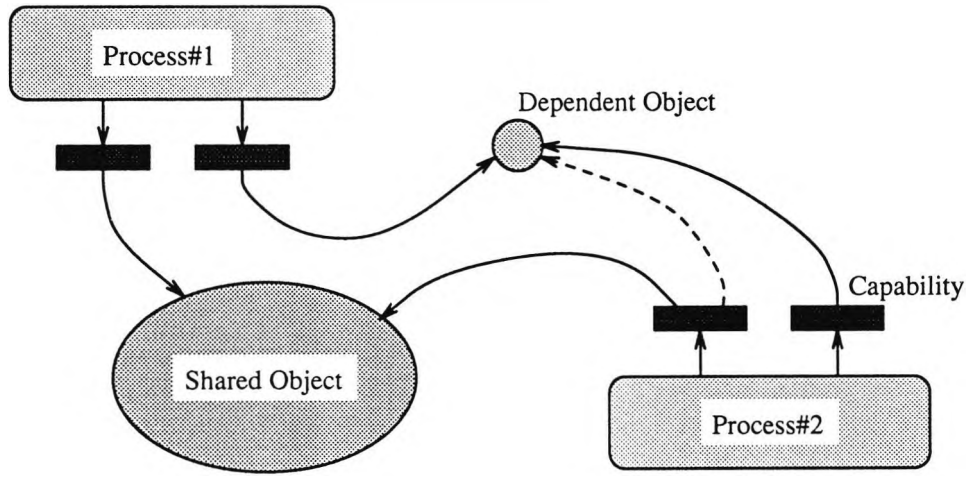


Figure 3.2: Two processes sharing an object using dependent capabilities

Resolve Object takes a capability and attempts to resolve it with the requested privileges. If resolution is successful, the object is made available to the process in the current protection domain.

Dissolve Object removes the object from the current protection domain. It does not delete the object from the object manager.

Delete Object instructs the manager to delete the object. Once this has happened it will no longer be available for resolution by any previously issued capabilities. If it is already resolved, the object will be removed from the relevant domains.

New Capability creates a new capability for the object referenced by the old one. This capability may have any subset of the privileges associated with the old and may have any dependents.

3.4.4 Revoking capabilities

Revoking capabilities is a traditional problem with capability systems; once a privilege is given away it is given for all time. The only way to revoke a privilege is to remove the service it refers to and recreate it. This not only invalidates the capabilities you want to invalidate but also invalidates all the others too.

By using *dependent capabilities* these problems are removed. Consider figure 3.2; suppose a source process (#1) wishes to share data with a destination process (#2) but must be able to remove this sharing at any time. It creates two objects, the first a dependent object of zero length[3], the second the one it wishes to share. It then gives away both objects by creating capabilities the destination process can use. The dependent object is given a simple capability whilst the data object is given a capability dependent on the other object. The destination, once it has resolved the dependent and then the data object, may access the shared information. The source may now withdraw access to the shared information by simply deleting the dependent object, so making it unresolvable by any capabilities, and removing it from any domains it is currently resolved in. Doing this makes it impossible for the destination process to resolve the data object. If both the dependent object and the shared object are active in a domain, removal of the dependent object will not prevent access to the shared object since dependents are only checked on capability resolution.

This situation can be extended to many sharing processes by the use of many dependent objects; the removal of each preventing one process from resolving the data object, or the removal of a single dependent object may prevent access by many processes.

3.4.5 Capability security

A high degree of security is important in an operating system which might be used to span tens, hundreds or even thousands of workstations, each of which may have several hundred MIPS or more of processing power. Complex capability encryption schemes are not practical as their overhead will severely punish legitimate users of the capabilities, while weak encryption schemes leave the system open to a malevolent hacker.

While capabilities can be weakly protected (as in Amoeba [TMR86] by redundant check fields) or strongly (as in Mach by passing them via the kernel [SJR86]), a rogue process registering itself as a service thread on a publicly known service port remains a problem.

[3] Although this object is of zero length, no two objects may reside at the same address, therefore a zero length object must have the size of 1 page or 1 address space increment (eg. a byte) depending upon the implementation.

The Amoeba group have presented a complex scheme to prevent this [TMR86, MvRT+90], which requires holding another set of codes, in addition to capabilities, in order to provide protection and verification.

This is unnecessary in ARIUS; the single address space enables the unique verification of the precise server required. The client makes a subroutine call to the entry address of the correct server binary. Thus two different binaries cannot be registered to provide the same service. Of course, such security ultimately depends on network security, in the form of encrypted data and signatures [KJ91, Kar91]. These techniques guarantee identity at the machine level leaving ARIUS to guarantee identity of services and processes through its capabilities.

3.5 Protection Domains

A protection domain comprises of a set of objects which may be accessed through normal load and store instructions by any process which references the domain. For example, in figure 3.3, Domain A contains three objects which may be accessed equally by any process referencing it, while Domain B contains two objects which may be accessed equally by any process referencing it. Notice that one object is common to both domains; this would typically be used to pass or share information between processes in the two protection domains.

The concept of protection domains is not new. However, in many previous operating systems they have not been separate entities. In UNIX for example, a protection domain is firmly attached to a process. In more novel systems such as Clouds [DLM88], domains are attached to objects. In ARIUS they are entities in their own right.

3.5.1 Domain construction

A mechanism is required to add and remove objects from a domain. Without this it would be impossible to execute new program, create new processes, or access stored data. A domain may have objects added or removed from it by any referencing process (there may

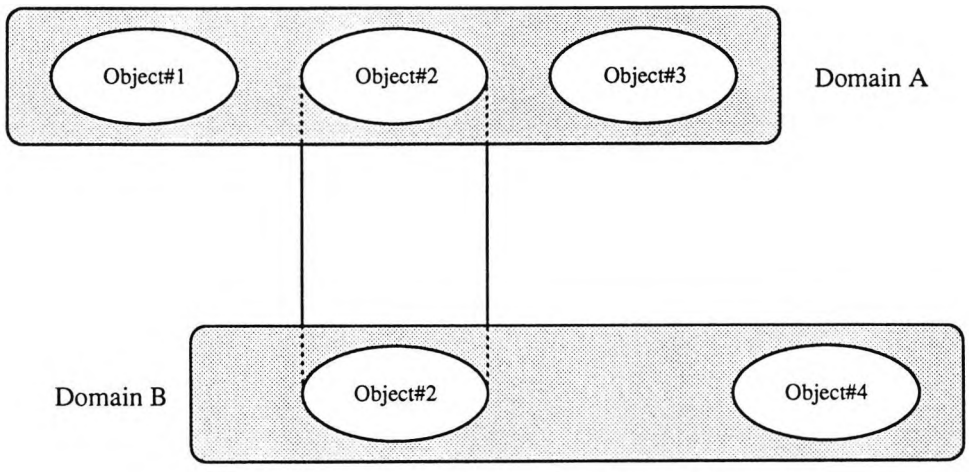


Figure 3.3: Two domains' views of the ARIUS namespace

be more than one). These objects may be accessed freely, privilege permitting, and can be thought of as "friends"[4]. Resolving an object places a reference to that object into the current domain's table object. This domain table is interpreted by the appropriate ARIUS memory server when page faults occur. It is important to note that ARIUS maintains domain information within the namespace; something it does with all "system" data. This greatly simplifies many mechanisms and also makes fault-tolerance much easier to implement.

3.5.2 Gateway objects

Resolving and dissolving objects enables domains to be altered but it is often useful to change a domain completely for short periods of time (to access a protected service for example). This facility is provided by designating an object as a *Gate*. A gate object, once resolved, can only be accessed as a subroutine. When this is done, the object is entered in a controlled manner to allow the request made to it to be validated. In doing this, the old domain is saved and a new one entered (figure 3.4).

A process may pass though a gate, change its domain, perform its task, then return to its previous domain. This operation can be considered a subroutine call with domain change.

[4]In the parlance of C++.

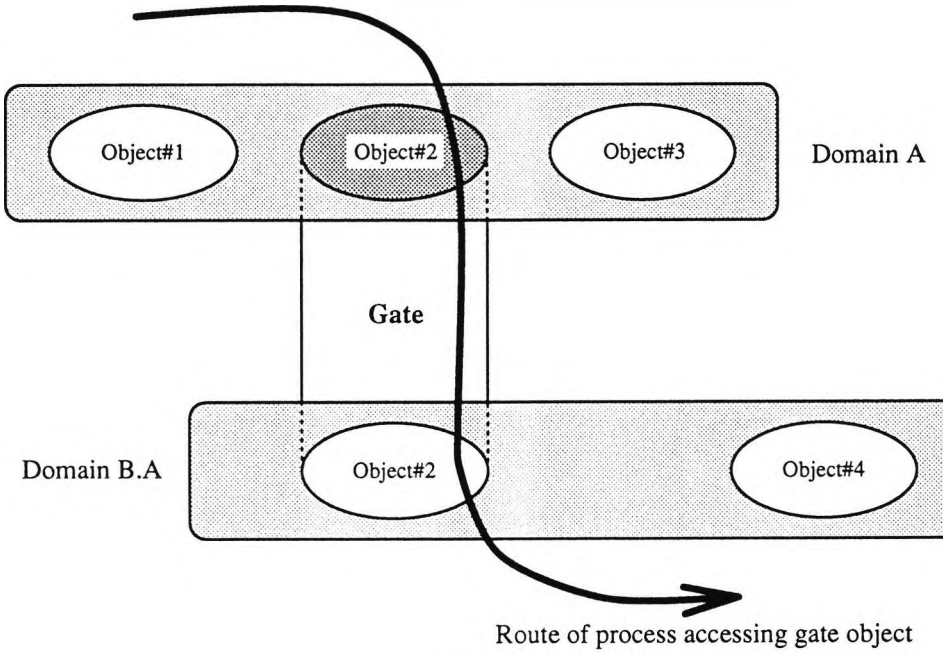


Figure 3.4: Two domains linked by a gate object

A processor kernel trap is a more crude form of this operation; a routine is called with the domain changed from user to supervisor.

3.5.3 Domain creation

The use of gate objects allows a domain to be changed but, how are they created? The first time a gate object is called, a new domain is created containing the gate object with new privileges and a stack for the entering process. Before validating the access requested by the entering process, the gate object is called upon to initialise itself. This initialisation is object specific but could consist of mapping in other objects, initialisation of a hardware device, or registering an inter-domain communication object. Once initialisation is complete, the domain is then re-entered with the original request presented.

A domain exists as long as its parental domain exists with the gate object in it. When this ceases to be the case, the gate domain is instructed to close down cleanly and is then terminated.

3.5.4 Domain optimisations

A typical domain operates as an open system; it does not trust its parent and expects its parent not to trust it. However, relaxation of these rules allows for speedier transfer of data and control between domains at the expense of reduced protection. If, for example, a caller trusts the callee, the callee may inherit all, or at least the relevant, objects from the parent. This is similar to the user/kernel domain change found in UNIX. Alternatively, if both callee and caller trust each other, then the inter-domain call reduces to a standard subroutine call.

3.6 Process Management

ARIUS attempts to provide a minimal process structure and flexible scheduling facilities. It does this by implementing a set of upcalls to inform processes of important events; such as scheduling requests, faults or interrupts.

3.6.1 Structure of a process

The ARIUS process structure is designed to hold minimal state. It consists of a process control block holding only the register set, a pointer to the upcall table, and a pointer to the current domain table. In this respect it is like a conventional "thread".

3.6.2 Upcalls

The use of upcalls [Cla85, MSLM91] has been demonstrated to be an efficient and clean mechanism for providing process exception handling. Previous mechanisms such as UNIX signals [ATT85] are too slow and clumsy to provide the fine control required by lightweight processes. ARIUS provides three types of upcall:

Object upcalls are handled within each object. They consist of mechanisms to allow objects to initialise when first used, validate gated subroutine calls, etc.

Process upcalls are handled by each process. They consist of mechanisms for catching segmentation faults, illegal instructions, etc.

Processor upcalls are handled by each processor. They consist of device interrupts, DSM faults, etc.

Upcalls need not be used by most processes (illegal instructions are usually of little interest to a process) in which case a default action is defined; for objects this will be nothing, for processes the process will terminate, and for processors the processor concerned will panic! The ability to modify upcalls is capability based; the ability to modify the relevant upcall table. Allowing general processes to modify the processors' upcalls is not recommended.

3.6.3 Migration

With the availability of a consistent and persistent continuous single address space comes the prospect of easy process migration. Moving a process from one processor to another (assuming they are homogeneous) is simply achieved by passing the thread of control from one to the other, all data structures, executable code are unchanged. In a distributed system, the DSM will invisibly take care of the movement of data from one processor system to another.

Such easily available process migration raises questions about load balancing and scheduling policy. Should a process be moved to a processor which has less work to do or, to a processor which has more of the process' working set cached or, to a processor which has more of the executable code cached? As the hardware complexity of systems increases, these issues are causing more problems, and the solutions are not as clear cut as they might first appear.

3.6.4 Scheduling

Like the other mechanisms, ARIUS leaves the scheduling decisions to the system or applications designers. Default schedulers will be available, but applications programmers will be able to fine tune the performance of their systems to suit their needs.

One might ask about very large distributed systems (over a University campus for example), and how an applications programmer can be given the freedom to write a unique scheduler and without disturbing the complete system. The answer is for each kernel's workspace and local scheduler queues to be made available as objects in the global address space, and accessible only to an applications programmer with the necessary capabilities.

3.7 Inter-process synchronisation

ARIUS provides only a shared memory model of computation. However, for processes to cooperate efficiently some form of synchronisation is necessary. In a message passing system, this synchronisation is implicitly provided with the data forming the message. In a shared memory system, synchronisation is explicitly provided by *locks*.

3.7.1 Spinlocks and Sleep/Wakeup

It has been observed [BALL89] that, if the number of physical processors is equal or greater than the number of processes in a parallel system, then the most efficient form of synchronisation is a *spinlock*. A spinlock is a two state lock which may only ever be held by one process at one time. If the lock is requested and is already held by another, the requester *spins* re-requesting the lock until it is granted to it.

Unfortunately, this mechanism is not efficient if the number of processes in a parallel system is larger than the number of physical processors, since processes may spend significant amounts of time spinning on locks when other processes could be executing. To avoid this situation, another form of locking, *sleep/wakeup* locks, may be used. Here, if a lock cannot be obtained, the process *sleeps* until released by a *wakeup* from another process releasing the lock.

In ARIUS locking is based around spinlocks. However, when a lock cannot be obtained after a defined number of spins and other processes are ready to run, the requesting process reverts to using a sleep/wakeup scheme instead. This composite scheme, as used in [BALL89], has been shown to be generally more efficient than either scheme in isolation.

To avoid the overheads associated with conventional sleep/wakeup schemes, ARIUS makes use of its DSM and upcall systems to assist in the provision of an efficient spinlock mechanism.

3.7.2 The Locking Mechanism

When a lock is requested and provided, the process continues immediately. If the lock fails, the process spins on the lock for either a set period of time, until the lock is provided, or forever if no other processes are ready to run. If the lock is not provided and the time period elapses, the process must give way and let another process execute. However, before doing this it notes its interest in the lock by marking the DSM page.

When the lock is later released, it is cleared by writing to the lock's DSM page. This change is propagated to all copies (rather than invalidating all copies). As each node applies the change to its copy of the page, if a lock is marked on the page, the relevant process is "upcalled" to inform it of the change in state. The process may then re-request the lock. Of course it may again fail to acquire it. It is important to note that if the upcall cannot be delivered, due to exhaustion of upcall space in the destination process, an overflow can simply be set. This indicates to the process that something has been lost. Re-running the process regenerates the locking request.

This scheme relies on the upcalls to inform processes of changes in *interesting* DSM pages rather than an explicit *sleep/wakeup* strategy. By doing this, spinlocks may change to sleep/wakeup locks in a transparent fashion.

3.8 Dynamic or Static Linking

Dynamic linking [HO91] has recently re-emerged in UNIX. Before this, every program contained a copy of all the library subroutines used, resulting in large program binaries. The use of dynamic linking enables programs to share a common, position independent version of these libraries. This has a number of advantages. Firstly, the code can be shared and so need only be present in memory once, so saving space; and secondly, the replacement

of an old library with a new one, perhaps with bug fixes and speed improvements, affects all programs immediately without the need to relink each by hand. The disadvantage is a decrease in speed. The library is loaded at an unknown address and, because this may change on different invocations, the linking of programs must be done *on the fly* as accesses are attempted.

In a single namespace system such as that provided by ARIUS, the sharing of libraries is greatly simplified; a library is seen at the same place by all observers. However, the policy of dynamic linking may not be sensible. Library code is hardly ever changed so the library object moves infrequently. Therefore, a form of static linking might be more suitable; the library code is still shared but the links are static and made at program compile time. Of course, some libraries do change but these can be handled by providing a level of indirection in the subroutine calls.

ARIUS does not force a particular dynamic linking policy on processes, instead it provides the necessary mechanisms for desired implementations to be constructed. However, a default strategy is provided called *pseudo-static linking.* This supplies a unique method of linking which offers the best of both dynamic and static linking systems. When an object is compiled, it is not linked but the libraries it requires are noted and the linking information concatenated with it (similar to a SunOS dynamic linking). When the object is first entered after resolution, it is instructed to initialise itself (as happens with gate objects, but now extended to include all executables). This initialisation performs an extra task. Any libraries the object may use are resolved and if newer than the object's last link time the object relinks itself using the stored information. The relinked binary is persistent so in future, as long as libraries are not changed, further relinking need not occur.

This mechanism has the efficiency of static linking without sacrificing the flexibility of dynamic linking. The only cost incurred, in the majority of invocations, is the resolution of any relevant libraries; a startup cost no greater than that incurred by standard dynamic linking schemes.

Other methods of relinking have also been considered. The design of ARIUS allows and encourages a mechanism to be chosen which best suits the application. However, the

first implementation of ARIUS is restricted to the basic pseudo-static linking mechanism described above.

3.9 Multi-processor and Distributed Machines

The use of a single namespace in ARIUS is central to it design. Consequently, remote machines must be addressed through this namespace rather than by the addition of another (such as an Internet address). To facilitate this and to keep the model simple, networks of machines are expected to share this single namespace, providing communications and data consistency through the use of distributed shared memory (DSM) techniques.

3.9.1 Distributed data sharing

There have been many implementations of DSM and its various properties have been extensively researched (see §2.4). The most interesting property is data sharing among distributed machines. Common data can be transparently cached and viewed locally without the need to resort to more specialised protocols and services. In ARIUS data exists at the same place in the namespace for all observers, remote or local. This view of a global namespace for all machines does have major consequences. Security cannot be ignored, as often happens with prototype systems, since here it is essential. Another concern is the limit of 64-bit addresses. For a locally distributed collection of machines this may be adequate but for a global system it may not be. To compromise the simple namespace to avoid these problems is not the correct solution; better to wait for 128 bit processors. Therefore the current design for ARIUS is restricted to locally distributed systems only.

3.9.2 Heterogeneous Architectures

ARIUS, because of its distributed aims, cannot afford to implement only one form of distributed memory system nor incorporate only one kind of machine architecture. The use of gate objects and intelligent object managers can solve the problems associated with

cross-architecture, inter-domain, calls by forcing invoked services to be run on specific types or even individually named machines.

Object and other service managers are free to implement any policies they see fit and are not limited by the standard operating system. For example, the DSM servers could even be used to provide object translation between architectures, such as big-endian/little-endian reversal.

3.10 The I/O system

ARIUS implements a persistent object space and so explicit I/O, in the form of file systems, is no longer necessary and should instead be seen as another level in the memory hierarchy. The use of I/O in ARIUS is therefore somewhat different from that of other systems.

3.10.1 Disk storage

Disk storage is the only persistent memory level of storage in ARIUS. Apart from this, it operates as any other memory level in the hierarchy. Generally because of its slow access time and great size, it forms the final level in this hierarchy; the final resting place for data no longer in active use.

3.10.2 Serial I/O

Serial I/O does not map into the single address space as disk storage does. Instead a serial channel, be it to a terminal or another machine, is treated as a stream of data accessed via services provided by objects.

3.11 The Services

The basic ARIUS microkernel provides few services, and those it does provide are very abstract. For example, there are no specified ways to write to a terminal, unlike UNIX

where `/dev/tty` might be used or DOS where `CON:` is available. ARIUS omits these and other services to avoid imposing them on the user, preferring to let users decide how best to tackle the problem.

Of course, users are not expected to write their own device drivers, a preposterous proposal for numerous reasons. Instead, ARIUS views these services and others in two halves; the lower half supports the raw service, be it a hardware device (such as a terminal, ethernet or disk) or software device (such as a nameserver or file system), whilst the upper half provides the necessary user interface. Consequently, a file system lower device would support the management of the files and storage space, whilst one of many available upper half devices would provide the relevant user abstraction (be it DOS, Unix, VMS or some customised other).

This division of service interfaces is for two reasons. Firstly, as demonstrated above, it allows the actual service function to be separated from the user interface; after all, most file systems provide much the same facilities. Secondly, it enables a set of ARIUS services to cooperate through their lower level interfaces without constraining the users to do likewise[5].

3.12 UNIX compatibility

One criticism of ARIUS is its abandonment of the UNIX process model. In order to provide some form of compatibility therefore, the design of a UNIX service has been examined. This has directly led to the development of a code generation system which supports UNIX without compromising the single address space. This work also indicates that the development of other operating system environments (such as MS-DOS, VMS or WINDOWS-NT) is possible. In this sense, despite its differences, ARIUS is a micro-kernel similar to MACH or Meshix.

[5] Although there is nothing stopping them!

3.12.1 The absence of fork()

The one facility absent from ARIUS which could cause problems was `fork()`. The creation of new processes using fork does not fit well with a single address space and a `spawn()` style of process creation has been adopted instead. However, many applications used fork semantics to duplicate data into each child process and although ARIUS supports the duplication of object using copy-on-write, each copy occupies a different virtual address range and so pointers in the duplicate are invalid.

Modifications to the C-compiler have already provided a means of placing the data segment at an arbitrary address. This was necessary to allow multiple invocations of the same application and to prevent corruption of the original image. These modification make use of a *data object base register* to indirect global memory accesses to the relevant data object. Although at first this may appear inefficient, experiments suggest the resulting code only suffers a 1–2% speed degradation. Based on this base register system, a method has been proposed to allow a fork semantic to be supported. By extending the current system so all memory access (rather than just global accesses) are indirected via a base register, it is possible to provide a fork operation by duplicating the data object for the child and changing the value of the base register. The cost of this in an application is a speed degradation of only 5–10% [Wil93].

3.13 Summary

This chapter has presented the design of the ARIUS operating system. The aim of ARIUS is to combine many traditionally separate mechanisms in order to provide a simpler but flexible parallel application environment. This is primarily achieved by basing the operating system around one single namespace, encompassing all data (volatile and persistent), and all service names (thread, processes, servers and lock). This namespace is supported by a single distributed shared memory system capable of operation on both parallel and distributed platforms. This removes the usual need to consider the position of data (local or remote) when writing applications.

These unifications does not limit the operation of the resulting system, since ARIUS is capable of supporting operating system extensions through the use of servers as are other micro-kernels (allowing a UNIX environment to be provided for example). However, unlike message passing micro-kernels which have inherent delays in their communication scheme, the shared memory system proposed here allows more rapid and more flexible data sharing.

The central tenet of ARIUS is the single namespace, termed the *Arius Massive Object Space* or AMOS. The unifications present in this single resource allows a single fault tolerance scheme to be used to encompass the entire system. The implementation of fault tolerance in such a namespace is the subject of the remainder of this thesis.

Chapter 4

Data coherency

This chapter considers the implementation of Distributed Shared Memory in the context of ARIUS's object store, AMOS. The first section examines the various constraints and properties required in a distributed shared memory system which is used to support an operating system. Five characteristics are identified and the relevant three are examined in detail. The second section details the DSM model implemented in AMOS. This model incorporates four different DSM policies in a single system to provide the required flexibility. The third section presents the DSM algorithms used. The fourth section builds upon these to provide some fault-tolerance for the system. This allows any single machine failure to be quickly handled and repaired if only duplicate data is lost. The final section summaries the chapter.

4.1 Arius' reliance on DSM

The ARIUS operating system is a non-intrusive system; it attempts to provide the required facilities for applications without forcing undesired policies upon them. To this end it provides a single shared address space implemented as AMOS, a distributed shared store. For simplicity, all control and data structures used by both ARIUS and applications are present in this distributed store and all communication takes place by the placement and retrieval of data from it.

This distributed shared store is implemented using distributed shared memory (DSM) techniques. Unlike many other systems, where distributed shared memory is implemented over a fundamentally different communications system, such as message passing, ARIUS relies only on shared memory for all application and system communications. The entire system's performance and reliability therefore depends on it.

This scheme has many strengths, the major ones being the simplification in the communications amongst machines and the assimilation of kernel and applications data structures into a single namespace available to all. The design of the AMOS DSM system attempts to make the most of these.

4.2 The adopted DSM model

AMOS' requirements for its DSM are more specific than those detailed in §2.4. Because its distributed memory system forms the core of all communications, storage and processor interactions; various constraints are placed on its characteristics. These are, in order of importance:

1. **Scalability** to allow efficient data sharing on a small and large scale,
2. **Strict, causal data coherency** but allowing a variety of policies to be used to provide it,
3. **Fault-tolerance** to prevent single node failure halting the entire system,
4. **Heterogeneity tolerance** to accommodate multiple networks of varying speeds, delays and topologies,
5. **Security** so that data cannot be obtained by devious means.

The last two of these constraints apply when AMOS is distributed over a wide area, making use of public, insecure networks for data exchange. For the purposes of this thesis however, we will consider only a local, single networked system and hence only constraints 1, 2 and 3 will be considered.

4.2.1 Scalability

Use of AMOS' for all data interactions means that any implementation must not only be efficient, but must also be scalable. Scalability is a very badly defined term; for example, it is said that hypercube networks are scalable because the available communications bandwidth scales with the number of nodes. Unfortunately, the hypercube node hardware is not scalable since at every step, another communications channel must be added to every node. Conversely, a mesh network may be considered scalable because the network hardware grows constantly with the network size. However, the available bandwidth does not scale.

For our purposes, it is more important that the DSM allows the number of processors sharing a piece of data to scale with the overall machine size rather than any other scaling property. This constraint instantly dismisses various schemes [TSF90, LLG+92, MR91] where fixed sized tables are used to hold DSM information. Although these tables could be increased until they are capable of holding information for all processors in a DSM machine, for large machines their size would be excessive, especially when multiplied by a large number of shared pages.

One article in the literature, SCI [KABJ89], also identifies this as the primary scaling property. The scheme described is based upon the notion of distributed doubly linked lists, where each sharing node holds a fixed DSM entry indicating its own state, together with pointers to the next and previous DSM entries, on remote nodes, in the list. This scheme allows the number of sharing nodes to expand indefinitely just by increasing the length of the list. The scheme in ARIUS uses similar principles.

4.2.2 Strict, causal data coherency

Most DSM systems are based upon single writer/multiple reader schemes, otherwise known as causal or coherent DSM. These systems operate by giving the exact semantics of physically shared memory even though the memory is distributed. This scheme is so popular because it emulates a situation which is readily understandable and easy to work with. Of course, such rigid coherency is not necessary in many cases. Some applications may only

require a coherent "snap-shot" to be made of data and are not interested in subsequent changes – graphical imaging of weather simulation for example.

With parallel programs data sharing falls into three categories; competitive, repetitive and cooperative. Competitive programs operate a number of processes which compete for machine time whilst interacting rarely. Repetitive programs operate a number of processes in a pipe line; each process processing data in a producer/consumer relationship with others. Cooperative programs operate a number of processes which interact extensively, each taking a small part of the problem's data set. A review of various parallel applications [Tot92] demonstrates that implementing a single coherency policy results in an inefficient, inflexible DSM system; any single policy is not suitable for all three parallel program categories. The review identifies four different policies for DSM coherency:

1. **Remote** access forces data being read or written by any processor to access a single static copy,
2. **Migrate** access moves a single copy of the data from processor to processor as each accesses it (read or write).
3. **Invalidate** is the most common strategy. A copy of the data is made locally for each reader, which is then invalidated when a master copy is modified. The master copy itself migrates on write accesses.
4. **Update** is similar to invalidate except that, instead of invalidating copies on write, the modification is propagated to all copies.

Different policies are used for different types of data accesses. For example, the *Remote* policy might best be used for handling the occasional update of processor load statistics whilst *Update* is useful for handling inter-process locks.

By nature of the review, the proposed implementation of these coherency policies was static. Each data structure was handled by one of these policies for its lifetime, the policy being chosen to suit the most common use of the data structure. In fact, is is unusual for a data structure to be so specifically used; most are treated in quite different ways during the course of a program's execution. A common example of this is a simple editor where

a data structure is accessed in three distinct ways; firstly data is loaded (migrate), then modified (update), and finally stored (migrate). In this example, it is usual for the size of the second set of accesses to far exceed the other two (although an editor is generally a uni-process so coherency policies are less important). This need not always be the case.

In AMOS, all four policies for coherency are employed. Additionally, the protocols are enhanced to allow all to operate concurrently on the same data structures. By doing this we allow dynamic choice of coherency policy, so increasing flexibility and efficiency. It is hoped that, at some future date, compilers will be able to take advantage of this flexibility by generating code to select how data is to be accessed rather than just where[1].

4.2.3 Fault tolerance

If DSM is to be used to provide all communications, the fault tolerance cannot be ignored. Providing this tolerance is considered in detail in §4.6 and in the following chapters. For the moment however, the operation of the DSM system will be established and the inclusion of fault tolerance incorporated afterwards.

4.3 Efficient implementation of AMOS' DSM

Although DSM mechanisms were first employed using software techniques [Li86], implementations are already being moved into hardware [HLH91, LLG+92, Del88, KABJ89]. This improves performance by increasing bandwidth and decreasing latency for data access. Consequently, much finer sharing of data becomes feasible; whilst software implementation must share virtual memory pages, hardware solutions may share cache lines. The performance advantages of finer levels of sharing are examined in [BS91].

[1] This might be particularly easy in object oriented languages where methods and data may be analysed in tandem.

4.3.1 Hardware DSM

Although ARIUS does not require hardware mechanisms for data sharing, a high performance AMOS implementation needs specialised support hardware. Such hardware support is no more complex than that employed in current bus based communication controllers [Bor88].

Hardware implementation of DSM imposes some limitations on the design of any protocols. Firstly, a device has limited state space. This means that simple table based protocols are not ideal since each cache line requires excessive table state which more often that not is unused. Secondly, only a limited amount of buffering for DSM requests is available. This makes it undesirable for an operation to block the hardware since doing so could require large buffers or excessive returning of unhandled messages for later retries. It also introduces the problem of deadlock. Thirdly, the implementation of complex protocols in silicon is difficult.

The protocols which will be proposed in the following take these limitations into account. They also recognise the need to provide fault tolerance, so enabling the system to handle the sudden loss of a machine without halting the DSM system for all others.

4.4 DSM coherence policies and AMOS protocol

By analysing the coherence policies detailed in §4.2.2 it has been determined that five DSM protocol elements can be designed to cover all cases. These are:

- **Request copy** to obtain a coherently managed page copy,
- **Request read** to read a single data item from any available page copy,
- **Request owner** to obtain control over a page,
- **Request invalidate** to remove all copies of a page except the one held with the owner, and
- **Request update** to write a data item through to all copies of a page.

Using these protocol elements, the policies may be implemented as follows:

- Remote coherency (read)
  1. Request remote read of page (request read).
- Remote coherency (write)
  1. Modify all page copies (request update).
     If only remote coherency is used then there is only be a single copy to modify.
- Migrate coherency (read)
  1. Request a copy of the page if one is not present (request copy).
  2. Request ownership if now owned by requesting node (request owner).
  3. Request invalidate of all other copies (request invalidate).
     These last two steps guarantee that only a single copy of the data exists and is located locally.
  4. Read from page
- Migrate coherency (write)
  1. Request a copy of the page if one is not present (request copy).
  2. Request ownership if now owned by requesting node (request owner)
  3. Request invalidate of all other copies (request invalidate).
  4. Write to the page.
- Invalidate coherency (read)
  1. Request a copy of the page if one is not present (request copy)
  2. Read from page
- Invalidate coherency (write)
  1. Request a copy of the page if one is not present (request copy).
  2. Request ownership if now owned by requesting node (request owner)

3. Request invalidate of all other copies (request invalidate)
4. Write to the page.

- Update coherency (read)
  1. Request a copy of the page if one is not present (request copy).
  2. Read from page
- Update coherency (write)
  1. Request a copy of the page if one is not present (request copy).
  2. Request update of all other copies (request update).

Because all the DSM protocol elements requests are compatible, these coherence policies may be intermixed. For example, reads from a data item might be done using *invalidate coherency* whilst writes to the similar data item could use *remote coherency.*

One property these coherency policies demonstrate is that separating *request owner* from *request copy* is not useful. A combined owner and copy request would be quicker since it requires fewer messages. Throughout this thesis, such requests are treated individually since they are semantically separate.

4.4.1 Implementation of coherency in AMOS

Rather than provide these policies explicitly, by tagging data structures [BCZ90], AMOS decomposes the actions into three types of read and three types of write. These can be used together to produce all of the policies already examined plus a mixture if required. The read coherencies are:

- **Read remote** reads the specific item of data from any copy found in the system. This copy can be local, if a copy has been obtained by another policy, or remote.
- **Read copy** retrieves a copy of the page in which the data resides, if one is not present, and provides the data from it.

- **Read migrate** retrieves a copy of the page in which the data resides, obtains ownership, and then invalidates all others. The data is provided from the copy.

The write coherencies are:

- **Write remote** writes to all copies of the data. It does not retrieve a copy of the data, although if one exists it too is updated.
- **Write invalidate** retrieves a copy of the page in which the data resides, if one is not already present, obtains ownership, and then invalidates all other copies before performing the modification.
- **Write update** retrieves a copy of the page in which the data resides, if one is not already present, and then writes the modification to all copies.

The decision on which policy to use can be implemented in several ways. Firstly, the data structures can be tagged indicating which policy to use. Secondly, the instruction set of the desired processor can be augmented to support the three types of read and the three type of write. Thirdly, three images of AMOS can be made in the processor's virtual address space; writing and reading from different images can perform the different policies[2]. Whatever the interface between processor and DSM system, the DSM system itself is not affected.

4.5 The AMOS DSM protocol

The AMOS DSM protocol elements are designed for an arbitrary network of nodes where broadcast, at a reasonable cost, is not possible. Such a network can be constructed using a scheme similar to that proposed in the Fibre Channel [FC991] standard where a number of nodes connect to a *Fabric* which provides the routing of point to point data transfers (figure 4.1). It is assumed that this system guarantees ordered, reliable delivery between any two nodes attached to the Fabric.

[2]Additionally, some form of hybrid scheme can be adopted using two AMOS images for general policies, only using read and write remote when the data is marked *uncachable* in the VM state.

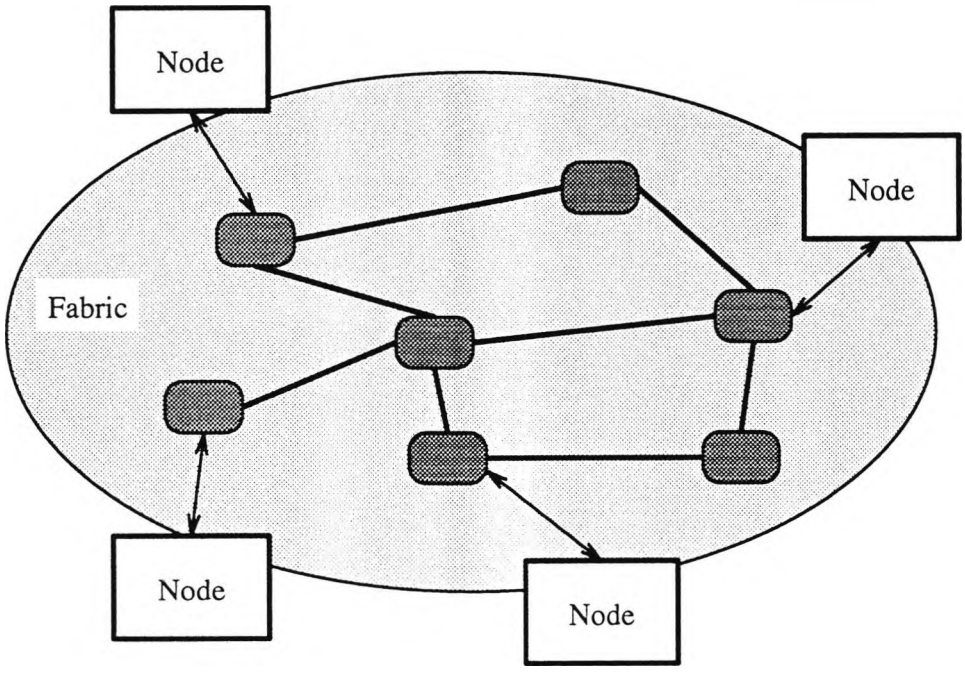


Figure 4.1: Nodes on a routing "fabric"

The DSM system is implemented over this network. Each shared page is held, by those using it, as a double linked virtual ring[3] (figure 4.2), each node holding a page copy also holds a pointer to both the upstream and downstream nodes also holding copies. Each page is coordinated by its own virtual ring of nodes. This avoids any non-participatory nodes from incurring any costs by forming part of a ring they are not interested in.

Using a ring structure to hold the copyset has a number of advantages. Most importantly, it is easy to demonstrate that a doubly-linked ring is capable of tolerating a single node failure. Additionally, a ring structure can handle a large copyset efficiently. Other systems for maintaining the copyset have been proposed. One system, used in the Stanford Dash [LLG+92], maintains each copyset centrally. Although it behaves similarly for most operations with a ring, invalidation poses the greatest strain on the copyset. For it to invalidate a data item a message is first sent to the central copyset site. This then sends a message to each processor in the copyset, receives a reply in turn, and then finally acknowledges the requester. The time to perform this operation cannot be simply defined since it is possible for some message transmissions to overlap each other (although only

[3]The original protocols were designed to use a singly linked ring but did not provide fault tolerance.

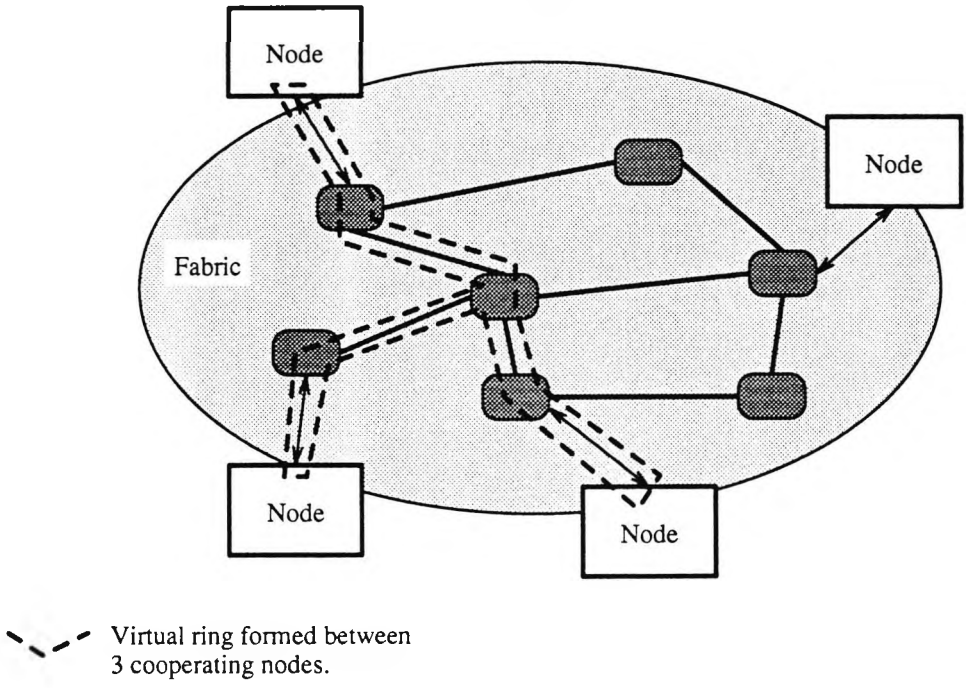


Figure 4.2: Nodes cooperating in the AMOS DSM system

one may be sent or received at once). However, if the time the message is in the network is negligible [Dal90] then the operation time is proportional to the number of messages sent; two messages per node in the copyset. However, in a ring based system with similar assumptions, the number of messages is only one per node. Therefore, a ring copyset is more efficient. Finally, a ring copyset only has a single message in flight during an invalidation resulting in less network congestion.

The protocols have been designed to be non-blocking and incur only a linear network load. These two properties are more important in a hardware implementation than a software one but it is expected that such systems will ultimately be moved to hardware (as examined in §4.3.1). The non-blocking protocol means a DSM request is received, processed by the system to modify local state, and a reply sent. At no point is it necessary for a protocol to block until it receives acknowledgements from other DSM servers. This avoids problems of buffering partially completed operations and deadlock detection. The system also incurs linear load; any protocol request which is received may produce at most a single request or reply in response. This is in the nature of the ring structure and avoids exponential traffic building up on the network.

4.5.1 Protocol implementation

The complete AMOS DSM protocol is now described. It has been split into its components. First, the client/server interface is described. Second the server loop is presented. It receives requests and dispatches them to the appropriate protocol elements. The five protocol elements which were introduced in §4.4 are presented next. Note that in all cases, the requesting node is assumed to have been part of the relevant DSM page chain in the past and that no node has "removed" its DSM page chain entry. Consequently, these protocols do not deal with the first page request, when a page is unknown to a node, nor do they deal with the removal of a node's participation in a DSM page chain. These less frequent operations are examined separately.

DSM client/server interface

Requests are made of the DSM system by the local processor in order to obtain the necessary data in the required form (for reading or for writing). A client will issue a request to the DSM system, and then wait until a reply is provided[4]. This reply will indicate either success or error. Success allows the process to continue. An error forces the client processor to back the request off and retry at a later time. An exponential scheme similar to Ethernet is used to prevent deadlock or starvation.

Algorithm 4.1

```
Begin client request
Repeat {
    Issue request to DSM server
    Wait for reply
    If reply is marked "error" {
        Back requester off for a time
    }
}
Until reply is not an error
```

---

[4] Alternately, the process may be descheduled in favour of another.

DSM server

Requests are made of the DSM server by either local processes, through hardware faults on the virtual memory or cache systems, or by external messages received across the network. In all cases, the requests are treated identically.

Algorithm 4.2

```
DSM server begin
Repeat forever {
    Receive message
    Find page message refers to
    If message is a "request copy", call algorithm 4.3
    Else if message is a "request read", call algorithm 4.4
    Else If message is a "request ownership", call algorithm 4.5
    Else if message is a "request invalidate", call algorithm 4.6
    Else if message is a "request update", call algorithm 4.7
    Else indicate an error!
}
```

DSM request copy

The algorithm described below obtains a copy of a page from a node which already has one. The request is made to the local DSM server and is passed from node to node in the direction of the page's last owner until a copy is found. The page's last owner will nearly always have a page copy and so forwarding requests toward it is most likely to succeed. In a few cases, it is possible that the page's owner does not have a page copy; but it will know where to find one in this situation. Once a page is located, a copy is sent to the requester and the requester informed of the success. For data coherency purposes, it will be necessary to locate the copy at a future date. To do this, the requester is linked into a double linked chain of copy holders. This forms the distributed copyset, a chain of nodes each holding an identical copy of the page. Note that the chain is only used in a singly linked fashion for standard DSM operations. The double linking is only provided for fault recovery.

Algorithm 4.3

Begin request copy
If message is marked "error" {
    Unbusy the page
    Release requester for retry
}
Else if message is marked "link" {
    Set page's "downchain-nextcopy" to message's source
}
Else if message is marked "copy" {
    Set page's "lastowner" to "lastowner" in message
    Set page's "upchain-nextcopy" to "nextcopy" in message
    Set page's "downchain-nextcopy" to the source of the message
    Mark message as a "link"
    Forward message to page's "upchain-nextcopy"
    Add copy to page's DSM state
    Busy the page
    If page is marked dirty, dirty the page's VM state
    Request TLB flushes
    Release requester
}
Else if this node does not hold a copy of the page {
    Forward message to page's "lastowner"
}
Else if this node is the originator and already has a copy of the page {
    Release requester
}
Else if page is marked "busy" {
    Mark message as "error"
    Send it back to originator
}
Else {

```
        Mark message as a "copy"
        Mark message "lastowner" to be page's "lastowner"
        Mark message "nextcopy" to be page's "upchain-nextcopy"
        If page is dirty, mark page as dirty in message
        Set page's "upchain-nextcopy" to be originator
        Add copy to page's DSM state
        Request TLB flushes
        Send message to originator
    }
    End request copy
```

DSM request read

Rather than obtain a complete page on a read request, it may be desirable to obtain only the specific item. This has the advantage of requiring no cache coherency state information to be maintained for the request, since the item of data is considered copied from memory to register. Such a mechanism is useful for reading remote uncached data, such as device registers, or data the destination node will not provide coherent copies of when it does not wish to pay the penalty of invalidation at a later time. In operation, this algorithm is identical to *request copy* except that, once a copy is found, the data is returned and the node is not linked into the DSM page chain.

Algorithm 4.4

```
Begin request remote read
If message contains copy {
        Set page's "lastowner" to "lastowner" in message
        Insert data into halted requester
        Release requester
}
Else if this node does not hold a copy of the page {
        Forward message to page's "lastowner"
}
Else if this node is the originator and already has a copy of the page {
        Release requester
}
Else {
```

```
        Mark message to contain a copy
        Place requested data in the message
        Send message to originator
    }
    End request remote read
```

DSM request owner

The following algorithm obtains ownership of a requested page. Ownership is only ever assigned to one copy of a page and designates the node responsible for controlling accesses to it. For example, ownership must be obtained before an invalidation of a page may be performed. This prevents two nodes invalidating the same page at the same time, so resulting in its loss from the system. The last owner fields are used to locate the current owner in the least possible time.

Algorithm 4.5

```
Begin request owner
If message is marked "error" {
        Unbusy the page
        Release requester for retry
}
Else if page is marked "busy" and message is from a remote node {
        Mark message as "error"
        Send it back to originator
}
Else if message contains ownership {
        Add ownership to page's DSM state
        Set page's "lastowner" to be this node
        Request TLB flushes
        Release requester
}
Else if this node is the originator {
        If page is already owned {
            Release requester
        }
        Else if this node does not have a copy of the page {
```

```
            Unbusy the page
            Release requester to allow page to be obtained
        }
        Else {
            Busy the page to prevent removal of copy
            Forward message to the page's "lastowner"
        }
    }
    Else if this node does not own the page {
        Forward message to the page's "lastowner"
    }
    Else {
        Mark message to contain ownership
        Invalidate page's ownership on this node
        Set page's "lastowner" to be requesting node
        Request TLB flushes
        Send message to originator
    }
    End request owner
```

DSM request invalidate

Once ownership of a page has been obtained, a node may invalidate all other copies so it may modify it in a coherent manner. This form of coherency is useful if the owner is to make many modifications to the page before copies are requested from other nodes. An invalidate request is propagated from node to node around the DSM page chain until it returns to the requester. If at any point the invalidate fails (because the page is busy), a reply is sent directly to the requester indicating the error. The DSM page chain is then adjusted by modification of *upchain-nextcopy* to point to the failed node. This removes the successfully invalidated entries from the chain but leaves in the remaining ones. A later *request invalidate* can then be issued to finish the task.

Algorithm 4.6

```
    Begin request invalidate
    If message is marked "error" {
```

```
    Set page's "upchain-nextcopy" to be the source of the message
    Unbusy the page
    Release requester for retry
}
Else if page is marked "busy" and message is from a remote node {
    Mark message as "error"
    Send it back to originator
}
If message contains "owner_one" {
    Set page's "upchain-nextcopy" to be this node
    Set page's "downchain-nextcopy" to be this node
    Set page's DSM state to indicate a single writable copy is present
    Busy the page
    Request TLB flushes
    Release requester
}
Else if this node is the originator {
    If this node holds a single writable copy {
        Release requester
    }
    Else if this node owns the page {
        Busy the page to prevent movement of ownership
        Forward the message to "upchain-nextcopy"
    }
    Else {
        May not invalidate
        Release requester for retry
    }
}
Else {
    If node has this page {
        Remove copy from DSM state
        Request TLB flushes
    }
    If page's "upchain-nextcopy" is originator, mark the message "owner_one"
    Forward the message to the page's "upchain-nextcopy"
}
End request invalidate
```

DSM request update

Update coherency operates by updating all copies of a page with any changes rather than invalidating them. If only a few changes are made to the page before copies are re-requested, then this scheme is more efficient than invalidation. Updates are first forwarded to the page's owner. This is necessary to guarantee that modifications will always be made in the same order in all copies. If this were not the case and two nodes modified the same datum, some copies could take one value whilst others could take the other.

Algorithm 4.7

```
Begin request update
If message contains "owner" {
    Release requester
}
Else if message contains "owner_one" {
    Unbusy the page
    If this node initiated the update {
        Release requester
    }
    Else {
        Mark the message "owner"
        Forward message to initiator
    }
}
Else if message contains "owner_many" {
    Set page's "lastowner" to be message's "lastowner"
    Apply the update if a copy is held
    If page's "upchain-nextcopy" is originator, mark the message "owner_one"
    Forward the message to the page's "upchain-nextcopy"
}
If this node owns the page {
    Busy the page to prevent movement of ownership
    Apply the update
    Mark the message "owner_many"
    Mark the message's "lastowner" to be this node
    Forward the message to "upchain-nextcopy"
}
Else {
```

Forward the message to the page's "lastowner"

}

End request update

Use of the busy indicator

When any of these algorithms succeed in changing the local page's DSM state (such as obtaining a read copy of a page or successfully obtaining ownership) the page entry is marked *busy*. Whilst this flag is maintained, the page state cannot be changed; so ownership cannot be removed or copies taken of unique data. This marker is maintained until the access causing the DSM fault has been fulfilled. Without this facility, a page need never persist on any given node long enough for accesses to be completed. Circumstances can be imagined where this would produce livelock.

This facility is also useful if *delta timers* are added. Such a timer would be started when the busy flag is set and cleared some time later. Delta timers have been shown to improve DSM efficiency [CF89, FP89]. The busy flag can also be used to implement page locking [AMMR90], forcing data to remain resident until explicitly released.

4.5.2 First time page requests

So far the problem of acquiring the first link into the DSM system for a page has not been examined. This mechanism has been excluded from the general DSM system for three reasons; it increases the complexity of any hardware DSM implementation, it is an operation performed only once per page, and it is expensive to provide in limited hardware when the number of machines involved may be thousands. First acquisition is therefore considered a software task regardless of the DSM implementation. However, there is no reason why the DSM system should not assist in the location of pages.

Figure 4.3 shows how the DSM system can be used to provide the necessary page location information. When a page first becomes active, it is added to the *DSM location table* which provides an initial location for all pages in an ARIUS system. This tables operates as a multi-level index, a root block pointing to sub-blocks, which in turn point to others,

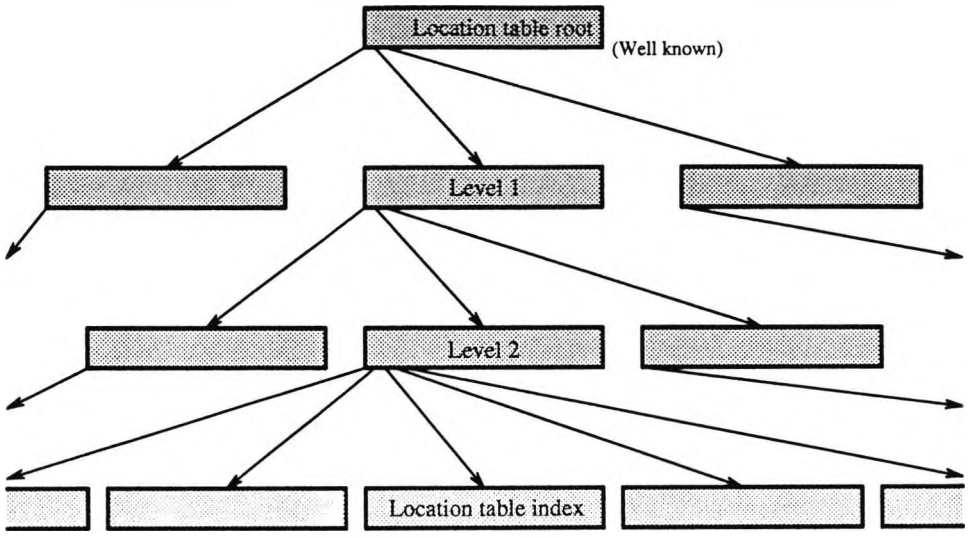


Figure 4.3: DSM tree for initially locating pages

which eventually point to the actual location information. By careful organisation of this table so every page of the next level is logged in the previous one, it is possible to locate any page whilst initially only providing access to the tree's root.

4.5.3 Page entry removal

When a page becomes inactive, the space for its DSM entry might be recovered for reuse and so the page entry must be removed. Doing so presents problems since the old entry may be the entry listed in the DSM location table or may still be part of a chain leading from the location table to the current owner. Therefore, to avoid any problems a page entry removal must first reinstall the current owner in the DSM location table.

This event is also handled in software and requires the addition of a *raise exception* mechanism in the proposed DSM scheme. This tag is added to standard DSM request but on reaching the action point, *request owner* reaching the current owner or *request copy* reaching a valid copy, an exception is raised on the node rather than the default action being taken. This may be caught and handled in software.

Removal of a page entry does not prevent other nodes from forwarding request for that page to it, especially if other nodes consider it to be the *last owner*. Since it is impossible

to search a system to find all references to the page entry, references must be invalidated as they are used. Therefore, when a DSM server receives a request, it first determines whether it holds the corresponding DSM entry. If it does not, then the request is returned to the originator marked with a *page unknown error*. The originator must then re-install its DSM entry by performing a *first time request* for the page.

4.6 Fault tolerance of DSM

Most DSM systems in the literature are incapable of handling even the most minor faults. As systems grow in size and distribution, such short sighted schemes are unacceptable. In AMOS this problem is compounded by the total reliance of ARIUS on its DSM system. Relatively little work on reliable DSM has been reported in the literature, although [WF89, WF90, TH90b, Fle90, SZ90] do present some ideas (see §2.5).

AMOS classifies DSM faults into three types:

One of many page loss

If a node fails whilst holding a DSM page copy, then no information is actually lost, but the page entries chain is broken. This must be repaired before operations on the page can continue.

One of many page and owner loss

If a node fails whilst holding a DSM page copy and the ownership, no information is lost but ownership must be reassigned. The DSM entries chain must therefore be repaired and ownership must be recovered and assigned to another page copy. Note that unmodified pages also fall into this category, since a copy will exist on a failure independent media (eg. disk).

Only page and owner loss

If only a single copy of a DSM page exists, then its loss removes data from the system. This data must be recovered before operations can continue.

The first two of these faults constitute *minor DSM faults*. These are faults which are recoverable without affecting any other pages in the DSM system. The final fault constitutes

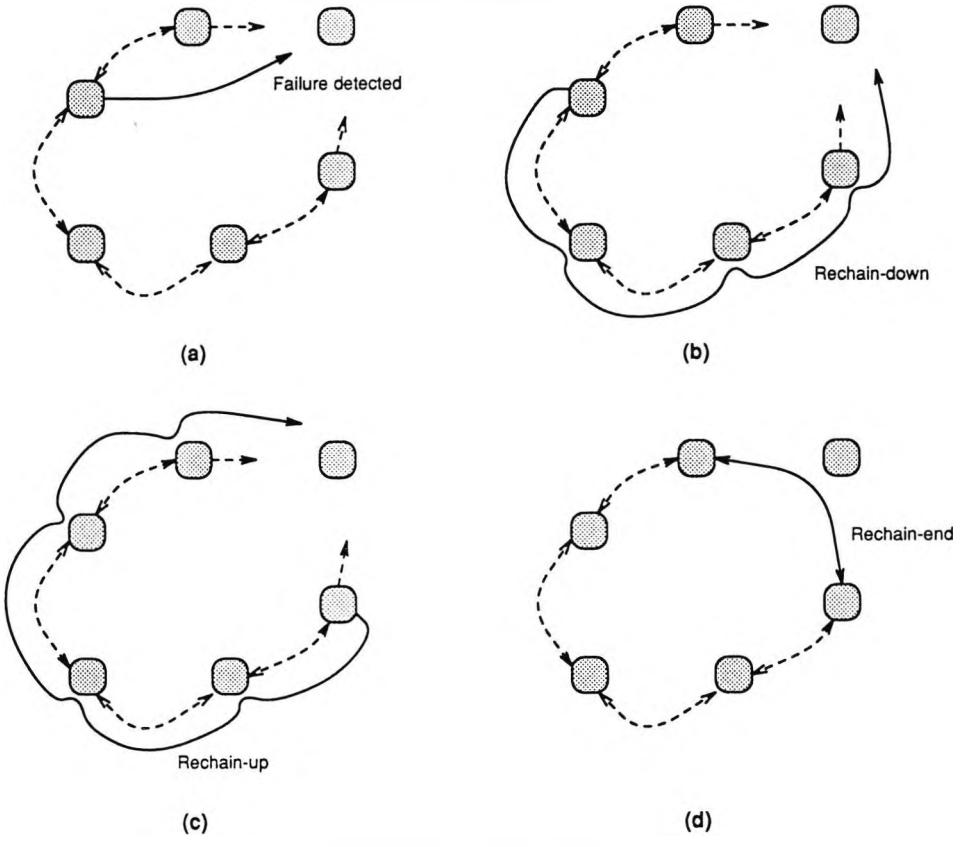


Figure 4.4: Rebuilding a DSM page chain in the presence of a single fault

a *major DSM fault* because its recovery implies alteration to other pages in the system. Such faults are more complex and dealt with in chapter 6.

4.6.1 Minor DSM faults

A node failure is noticed during DSM-to-DSM message exchanges. Once detected, the entries chain must be repaired, and then ownership reassigned, if it no longer exists.

Algorithm 4.8 is capable of repairing a DSM page chain in the presence of a single node failure (figure 4.4). The detecting node (a) injects a rechaining message into its local server which marks it *rechain-down*. This is forwarded down the DSM chain until it reaches the node in the chain which logically adjoins the failed one (b). It then relabels itself *rechain-up* and notes the current node in the message. The message then propagates back up the chain, repairing *lastowner* and *downchain-nextcopy* entries as it goes, until it reaches the

other logically adjoining node (c). It then reconnects both ends of the DSM chain (d), and if the owner was not encountered during the repair, the old downchain end of the repaired chain is designated the new owner. To handle more than one node attempting a page repair at the same time, when the downchain end of the list is reached the node is marked *failed.* Any subsequent rechaining requests which encounter the node are returned to their originator marked *rechain busy.*

The complete algorithm is presented below:

Algorithm 4.8

```
Begin DSM rechaining
If message is marked "rechain busy" or "rechain done" {
    Inform requester
}
Else if message is marked "rechain-up" {
    If this node owns the page {
        Mark the message "owner"
    }
    Else if message is marked "owner" {
        Set the page's "lastowner" to be message's source
    }
    Else {
        Set the page's "lastowner" to be page's "upchain-nextcopy"
    }
    If "upchain-nextcopy" either equals the failed node or the message's "nextcopy" {
        Reached up end of entries chain
        Set the page's "upchain-nextcopy" to be that held in the message
        Mark message "rechain-end"
        Note this node's address in the message's "nextcopy"
        Forward message to "upchain-nextcopy" node
    }
    Else {
        Forward message to "upchain-nextcopy" node
    }
}
Else if message is marked "rechain-down" {
    If page's "upchain-nextcopy" does not equal message's source or "downchain-nextcopy"
       does equal the failed node {
        Reached down end of entries chain
        If page is marked "failed" {
```

```
                Mark the message "rechain busy"
                Forward message to it originator
            }
            Else {
                Mark page "failed"
                Set the page's "upchain-nextcopy" to be the source of the message
                Mark message "rechain-up" with this node's address in message "nextcopy"
                Forward message to "upchain-nextcopy" node
            }
        }
        Else if message is back at its start node {
            Someone else must have fixed the chain!
        }
        Else {
            Forward message to "downchain-nextcopy" node
        }
    }
    Else if message is marked "rechain-end" {
        Set the page's "downchain-nextcopy" to be that held in the message "nextcopy"
        If the message does not contains ownership {
            Make this node the owner of the page
            Set page's "lastowner" to be this node
        }
        Else {
            Set the page's "lastowner" to be message's source
        }
        Mark page "active"
        Mark message "rechain done"
        Forward message to its originator
    }
    Else {
        If "downchain-nextcopy" equals the failed node {
            Forward message to "upchain-nextcopy" node
        }
        Else {
            Mark message "rechain-down"
            Note the start node in the message
            Forward message to "downchain-nextcopy" node
        }
    }
    End DSM rechaining
```

4.7 Summary

This chapter has presented the AMOS DSM system on which ARIUS is based. By first examining the requirements for a DSM system which must support various types of parallelism, a protocol have been designed capable of supporting multiple coherency policies simultaneously. This is necessary since any single solution is insufficient for the purposes of all the systems components and all potential applications. However, all policies maintain strict data coherency. This allows any policy to be used in place of any other, with the only effect being on efficiency.

All DSM policies are supported on a "ring" copyset. This allows an unlimited number of node to participate in a DSM exchange[5]. Rather than forcing the use of physical ring, which would restrict the networks on which the policy could operate, a virtual ring is used. This removes any topological restrictions and does not effect any nodes not involved with the DSM ring. Additionally, a ring can efficiently support all common DSM operations (get copy, invalidate, update, etc.) whilst limiting the congestion in the interconnection network. Page location is managed by using a chain of "last owners" which tracks a pages movement between nodes. If this fails or the page is not known, a hierarchical "DSM location table" is used which allows all pages to be traced from a well known root.

Implementation of the DSM protocol is suited to both software and hardware. This is made possible by the decomposition of the individual DSM actions into simple operation and the use of a non-blocking scheme removing the need for buffering (which would be necessary to prevent deadlock).

The DSM system is capable of handling "minor faults". The copyset ring is capable of tolerating a single failure without any DSM copies being lost (except the failure). This allows many simple faults to be recovered by simply rebuilding the ring and reassigning ownership as necessary. "Major faults", where the only copy of data is lost, is not handled by the DSM system. Instead, a more general fault recovery system must be used. This is described in the following chapters.

[5] Limited only by addressing.

Chapter 5

Providing fault tolerance

This chapter describes what fault tolerance is and what it attempts to achieve in the context of operating systems. How such fault tolerant systems are designed and constructed is then explained together with the various trade-offs among them. From this basis, the design decisions behind the fault tolerant system described in this thesis are examined.

5.1 The need for reliability

As systems grow from single, isolated processors to parallel machines with hundreds of processing elements and networks with hundreds of machines, the problems of reliability, often ignored due to the rarity of faults, must be addressed. The probability of system failures increases with machine size due to the interdependencies present in distributed operating systems and applications. Examples of these problems were highlighted in Chapter 2.

When an operating system attempts to unify many machines in a physically distributed environment, the ability to tolerate the sudden absence of some machines from the system is essential. It is also desirable that the "faults" are handled gracefully. Ideally, any failure would be handled transparently but in many cases this is not possible. For example, if the data being accessed is held only on the machine which has failed, then it cannot be accessed. In the majority of cases however, data is either available locally or has already

been cached elsewhere. Failure in such cases need not affect running applications if there is some degree of fault tolerance.

5.1.1 Types of faults

Typically two types of faults may be present in a system:

- Fail-stop failures, and
- Byzantine failures.

Fail-stop failures [Sch84] are the easiest to consider. When a component of a system fails, the failure is detected by other components and the system halted. A good example of this is parity checked memory; when a parity error is detected, it is reported and the system stopped. Such failures can be considered "clean", that is they are detected and the system stopped before further damage can result.

Byzantine failures [LSP82] are more complex. A failure may be malicious and involve collusion with other components to make it invisible to simple detection mechanisms. Consequently such failures are more difficult to isolate.

This thesis is concerned with only the first type of failure. It assumes byzantine failures are transformed into fail-stop failures through lower level detection mechanisms (eg. memory and processor error checking and correction techniques, network packet checksums, etc.).

5.2 Methods for providing fault tolerance

Any method for providing fault tolerance ultimately involves a cost in the form of some extra resource [Nel90]. One reason why fault tolerance has been little used in general systems is that this cost has been prohibitive when compared with the cost of the machine. Any fault tolerance mechanism has therefore been an expensive optional extra (if available at all).

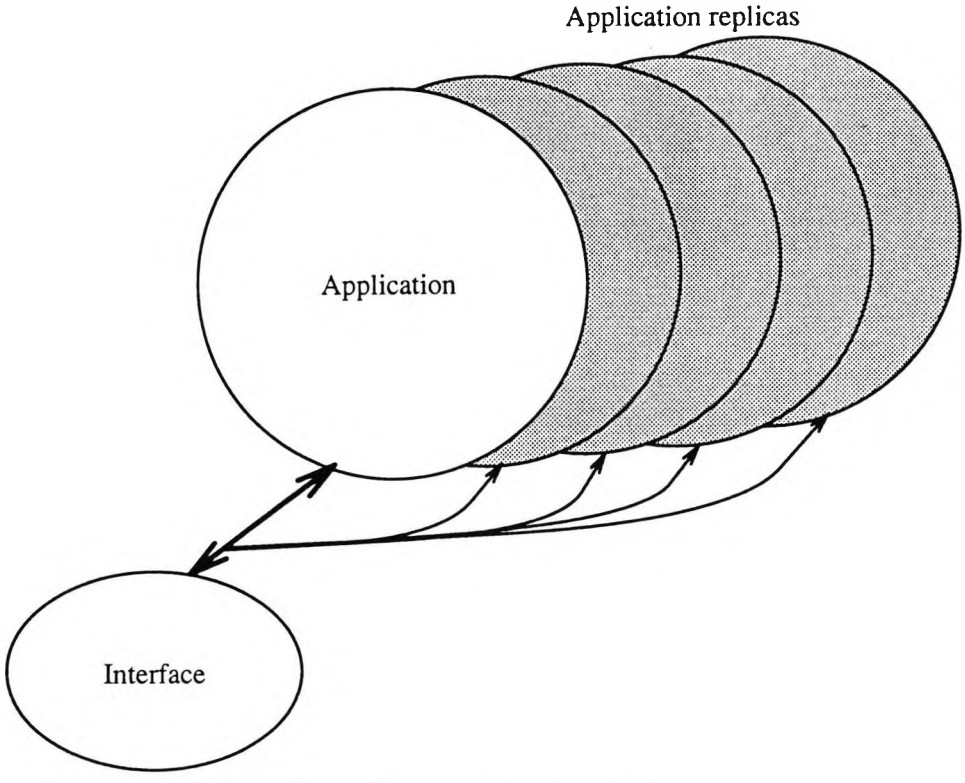


Figure 5.1: Replicated application

There are two general mechanisms for providing fault tolerance:

- Replication, and
- Checkpointing.

5.2.1 Replication

If a single instance of an application is executing and suffers a failure, then the application fails. However, if two instances of the application are executed in parallel, either can fail leaving the other one to complete the task. Similarly, by executing N instances of the application, N-1 instances can fail and a result will still be returned.

This rather simplistic method of providing fault tolerance is attractive for many reasons. Firstly, if the replication is handled by the operating system, no modifications are required to the application. Secondly, by determining how many replicas are executed, the number

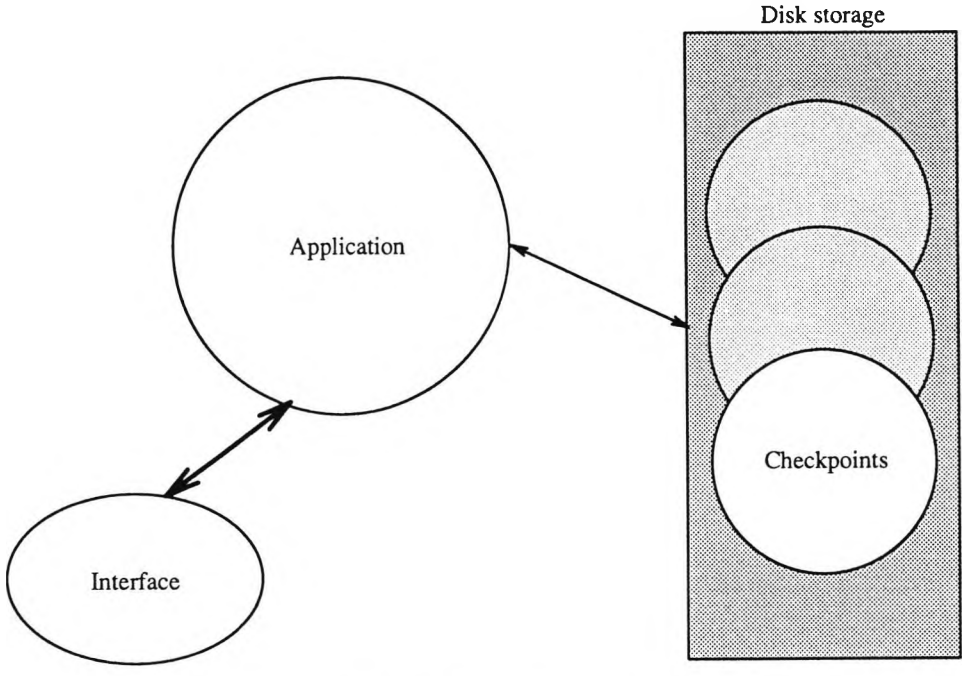


Figure 5.2: Checkpointing application

of faults to be tolerated can be adjusted. Thirdly, all applications execute at full speed and any failure does not effect the execution speed of the others. This is very important in real-time applications.

The cost of replication is obvious; processing resource. If two instances are executed concurrently then twice the number of processors must be available. If the application were executed on a parallel machine this would effectively reduce its size by a factor of two—a large penalty to pay for infrequent faults.

5.2.2 Checkpointing

Instead of executing multiple instances of an application, checkpointing periodically takes a "snapshot" of the running program and saves it to disk (or some other failure independent media). This snapshot contains a complete copy of all active program data, files, etc. If the application then fails, the checkpoint can be recovered and execution restarted using it as the starting point.

Checkpointing[1] is attractive for various reasons. Firstly, the checkpointing can be handled by the operating system and then entails no modification to the application. Secondly, the period between checkpoints can be varied to provide a compromise between checkpoint overheads, which are proportional to fault frequencies, and lost time (if the time between checkpoints is "T", then a fault could at worst cause the application to take an extra time T to complete). Thirdly, multiple faults can be tolerated by increasing the number of snapshot copies made, each additional snapshot allowing the tolerance of another failure.

With checkpointing the cost is not as apparent as in replication. Here the predominating costs are extra space for the checkpoint copies plus the processor time to make the copies. The total cost of both these probably does not exceed that required for replication (replication also requires extra space for the individual instances) but because the processor is responsible for generating the checkpoint, the application will take longer to execute even if there are no failures.

5.3 Fault tolerance in parallel machines

Neither replication nor checkpointing in its simple form is suitable for distributed machine architectures. Consider an application consisting of many parallel parts.

Replication — If a parallel application is replicated and a single part fails, then what should happen to the remainder? One possibility is to terminate the whole of the replica to which the part belonged. This is unnecessarily harsh, since another replica of the part could provide the missing answer for the remaining parts thereby enabling the replica to continue. This would provide good tolerance of multiple failures. However, if this approach were adopted then it is necessary to maintain careful synchronisation so that different replicas cannot provide temporarily incorrect data to other replicas (due to one part executing faster than another).

Checkpointing — If a single part of a parallel application fails then it cannot simply recover the most recent checkpoint for that part since it may be temporally incorrect

[1]The recovery process is known as "rollback".

with respect to the rest of the application (ie. contain data which depends on data which is no longer available). In such a situation it is necessary to "rollback" other parts of the application until a consistent point is found. This is essentially the same problem as described above for replication, except that here it is checkpoints that must be correctly synchronised.

No matter which fault tolerance strategy is adopted, for a parallel or distributed machine, the problem of coordinating the parallel tasks is essentially the same when recovery is necessary.

5.4 Fault tolerance for Arius

ARIUS is designed to make best use of parallel or distributed groups of machines. Fault tolerance in such situations is considered mandatory rather than optional. It is recognised that any solution must be both efficient, so as to limit the effect on the application performance, and involve little or no modification to the applications, so requiring no effort by the programmer to make use of it. Additionally, ARIUS is intended as a general purpose operating system and so does not support real-time constraints, nor is it realistic to sacrifice half the available machine to handle failures.

For all these reasons, ARIUS adopts a checkpoint based solution to fault tolerance. These checkpoints are supported within the operating system. An alternative solution would be to provide primitives to applications to allow checkpoints to be made and trust they are used correctly. This was not considered a sufficiently comprehensive solution, especially when parallel programs increase the complexity of such operations. By adopting an operating system bias, system checkpointing can be provided efficiently and can be strictly enforced so allowing the operating system to depend upon applications being fault tolerant. This greatly simplifies the design of system servers which need no longer worry about client failures.

5.4.1 Structure of Arius reliability system

The design of the reliability system is based around a few basic concepts:

1. It must be application transparent,
2. It must be provide efficiently, to have minimal impact on application performance when no faults are detected,
3. It is unreasonable to duplicate hardware to provide fault tolerance,
4. It is unreasonable to use any part of the machine to support fault tolerance exclusively,
5. Checkpointing an entire machine in a single "snapshot" is difficult and inefficient, and
6. Both checkpointing and rollback algorithms must be scalable.

The goal is to produce a solution which is scalable to large machines (1000 processors), does not require specialised hardware, and does not require application intervention. This can all be achieved by considering the provision of fault tolerance in two linked systems, one providing *volatile reliability*, and the other *persistent reliability.*

5.4.2 Volatile reliability

Figure 5.3 illustrates the general structure of a parallel machine which would support ARIUS. The machine consists of many *nodes*, each node being a processor (or a group of processors), memory and disk. These nodes are linked by a network operating a DSM communication system. Such a system inherently contains redundancy since it consists of many identical units, each capable of operating in the place of another[2].

Volatile reliability makes use of this duplication to provide fast checkpointing and rollback. Each node is essentially independent, and so failure independent. When a node makes a

[2]Generally, a *unit* must contain at least one processor, a memory, and at least one disk. A diskless node is not independent since it depends on another disk.

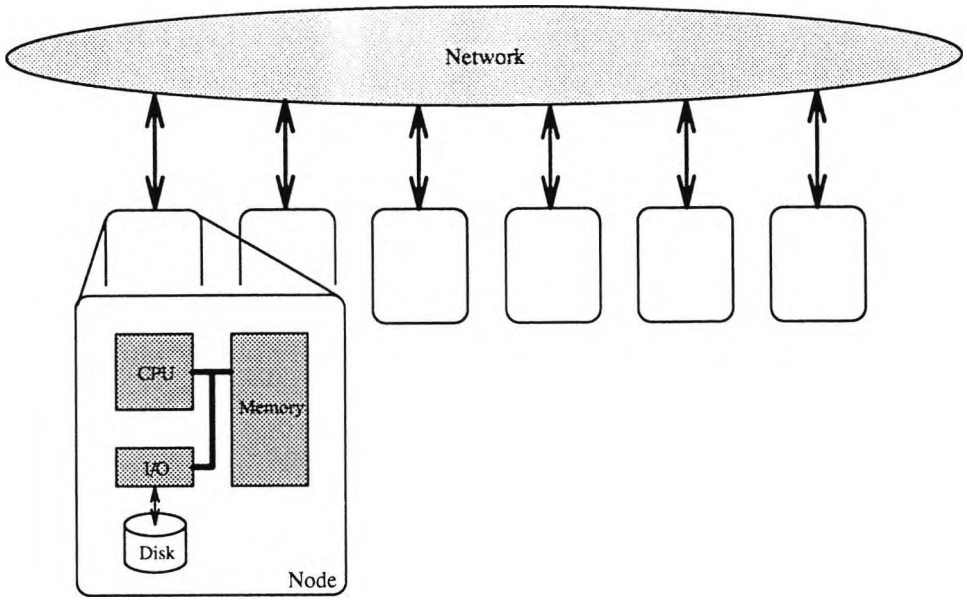


Figure 5.3: General machine structure

volatile checkpoint, rather than storing it on disk (so it may be recovered after failure), it is stored to another node. This allows checkpoints to be completed in a much reduced time and also provide faster rollback. Most of the work in volatile reliability is performed by the DSM. The DSM system is used in three ways. Firstly, it coordinates the grouping of interdependent applications (those which have communicated) so allowing fine grain checkpoints to be made; secondly, it provides a mechanism to export the checkpointed data to other nodes; and thirdly, after a rollback it provides a means for migrating checkpointed data on demand.

The volatile reliability mechanism supports the following operations:

Checkpoint — Checkpoints may be initiated by applications or, more usually, will be initiated on an application's behalf by the system. Once a checkpoint is initiated, the application is allowed to continue before the checkpoint has completed.

Commit — Applications may explicitly wait for a checkpoint to be committed. Once this has happened the previous checkpoint is guaranteed to be recoverable.

Rollback — The system initiates rollback on behalf of applications when faults are detected. Rollback always returns to the previous committed checkpoint.

Generally, these operations are used by the operating system only, applications being unaware of them. However, explicit control is provided to perform checkpoints or wait for commitment.

Storing checkpoints in other machines' memories does not ensure that data persists. Therefore, a system to support checkpointing to disk is also provided.

5.4.3 Persistent reliability

Volatile reliability provide a means of efficiently making checkpoints, storing them, waiting for commitment to take place, and recovering after a fault. Persistent reliability does not attempt to duplicate these services. It provides only a mechanism to make a checkpoint to disk, and does not provide a mechanism for rollback or commitment.

This is a policy decision rather than the inability of the system to support commitment and rollback. Volatile reliability can provide the required level of fault tolerance more efficiently that persistent reliability. Consequently, excessive use of persistent checkpoints would degrade the performance of the machine. The ability to make persistent checkpoints is primarily provided to allow systems to be shutdown without data loss, and to allow periodic system initiated checkpoints to prevent large scale data loss in the event of a catastrophic failure.

5.5 Summary

This chapter has introduced the need for fault tolerance in distributed operating system design and highlighted the relevant points in the two strategies available to provide it. The problems associated with implementing either strategy in a parallel machine were then outlined. The solution adopted in ARIUS is checkpointing. The main reasons for this decision and the level at which it is implemented have been explained. The next two chapters describe in detail how such a fault tolerance framework can be constructed under the ARIUS operating system and how it operates. Volatile reliability will be considered in chapter 6. It enables failures in a running network of machines to be handled quickly

and in an application transparent fashion. Persistent reliability is considered in chapter 7. It provides a mechanism to store checkpoints to disk to protect data against catastrophic failure.

Chapter 6

Volatile reliability

The algorithms detailed in the following chapter provide reliability through the use of data duplication, checkpoints and rollback. This allows failures, resulting in data loss, to be tolerated by recovery of the duplicated data. For networks of failure independent machines, a single machine failure may be tolerated. After such a failure is detected and compensated for, the system may then tolerate another single machine failure.

Any faults resulting in the failure of more than one machine at any one time cannot be handled and are considered catastrophic. However, such faults will *fail safe*, that is they will neither corrupt persistent data nor breach security barriers.

This *single machine fault tolerance* may be expanded to handle multiple simultaneous failures by use of *failure domains*, wherein a set of machines are considered failure dependent.

6.1 Introduction

Volatile reliability provides a reliable run-time system by providing a hidden, fault-tolerant virtual memory system. This system is constructed by additions to both the virtual memory management and the DSM structures of AMOS.

We will consider a mechanism to provide fault tolerance through the use of checkpointing and rollback [TKT89] and show how these systems can be implemented using additions

to first the virtual memory management system and then the distributed memory system. Results are presented in chapter 8 indicating the overhead this mechanism imposes.

The remainder of this chapter is divided into two parts. We first develop algorithms for fine grain, single node checkpointing and rollback. These are then expanded to handle the problems inherent in distributed systems.

6.1.1 Checkpointing and rollback

Checkpointing and rollback is a mechanism for providing fault tolerance. Periodically, the entire image of a running process is copied elsewhere (usually to disk). This operation is termed a *checkpoint.* If at a later date the process fails, due to the processor crashing for example, the process may be restarted elsewhere by using the previously recorded checkpoint. This operation is termed a *rollback.*

This operation may appear to be quite simple but in this basic form is almost useless. Checkpointing in this manner is extremely expensive in processing time. Not only must all processes be halted during the checkpointing, but the quantity of data which requires copying is excessive. There are also implementation difficulties. The algorithm assumes that the checkpoint can be made atomically; that is, once a checkpoint is initiated it proceeds correctly to completion. Unfortunately, it is quite possible for an error to occur during the checkpoint operation so losing both the checkpoint and the processes themselves. Consequently, a better error handling scheme is required.

6.1.2 Challis' Algorithm

Challis algorithm [Cha78] provides a way of producing an apparently atomic checkpoint whilst coping with the possibility of faults during the actual checkpointing operation.

Assuming that a checkpoint is to be written to disk, the algorithm allocates two disk blocks at known locations. These are known as root blocks and contain the necessary information to allow the system to map memory blocks to disk blocks. When the system checkpoints, the current memory image is written to newly allocated disk blocks and an

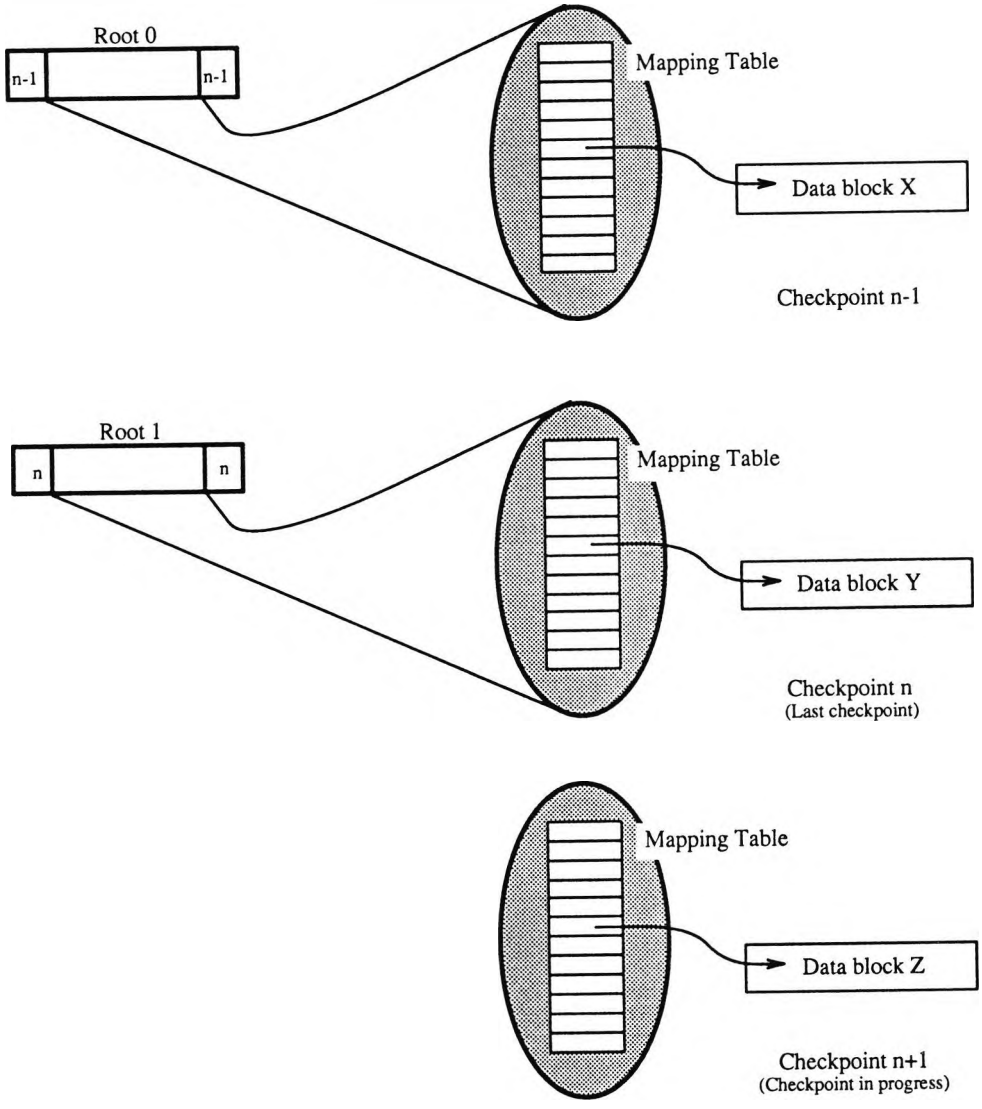


Figure 6.1: Organisation of data in Challis' Algorithm

appropriate root created, indicating where these blocks are, and also written to disk. The root blocks are written alternately, each with a version number at the beginning and end[1] (see figure 6.1).

When a system crashes and recovers, both root blocks are examined. The root block with the newest version number, which is the same at the beginning and end, is selected and used to reload the system.

The algorithm provides the illusion of atomic update by producing a checkpoint which happens in two stages. The first stage is the writing of the current image, the second is the writing of the root block. Only once this root block is written correctly is the checkpoint *committed*; that is, written consistently so it may be rolled back to in the event of system error. Such algorithms are said to use *two stage commit protocols.*

The Challis Algorithm reduces the problems of atomic checkpoints since it copes with errors during commitment without the system failing completely. In this form however, it is still necessary to write a complete image to disk at every checkpoint.

6.1.3 Improving Challis' Algorithm

The obvious improvement to Challis' Algorithm is to avoid copying the entire image and only copy the changes. One such scheme [RHB+90] does this using *shadow paging.* This scheme, used in the MONADS operating system, uses the page modification information present in the virtual memory management subsystem, to determine which pages of an image have been modified since the last checkpoint and only incorporates those pages into the new checkpoint. This dramatically reduces the quantity of data which must be stored at each checkpoint. The cost of this improvement is the increased complexity of determining which data requires checkpointing and which disk pages can be reused when no longer associated with a checkpoint or current image.

[1] By placing similar version numbers at the beginning and end of the block, it becomes possible to determine if the block was written completely during the recovery procedure.

6.1.4 Distributed machine checkpointing

Challis' Algorithm was designed to checkpoint single machines. So too is the improved algorithm used in MONADS although some attempt has been made to incorporate the notion of distributed memory checkpointing.

There are a number of issues to be addressed in checkpointing a distributed system.

Where do we checkpoint to?

In a uniprocessor system, there is only the processor's memory and disk available as independent storage entities. Consequently, checkpoints are made from memory to disk. In a distributed system, other machines are as good a repository for checkpoints as disks, if they are failure independent. Using other machines could also decrease checkpointing time and speedup recovery.

Should the whole system be checkpointed at the same time?

If Challis' Algorithm were to be used, it would be necessary to halt the entire system, checkpoint the entire system, and then resume the entire system. This might be an acceptable solution for small multiprocessor machines. For hundreds of machines distributed across many sites, it would be unusable since it could easily halt the entire system for minutes.

Can each machine in the system be checkpointed individually?

If all machines cannot be checkpointed together, then the other extreme would be to checkpoint them individually. This is possible as long as there are no interactions among them. Unfortunately, when interactions occur they can cause the loss or duplication of data after rollback. Consider the following example where data is duplicated:

Example 6.1

Both "machine A" and "machine B" checkpoint,

"Machine A" copies a piece of data to "machine B",
"Machine B" checkpoints,
Both machines crash,
Both machines rollback to their last completed checkpoints,
"Machine A" copies a piece of data to "machine B" as before,
"Machine B" gets the data again and was not expecting it!

Another example where data is lost:

Example 6.2

Both "machine A" and "machine B" checkpoint,
"Machine A" copies a piece of data to "machine B",
"Machine A" checkpoints,
Both machines crash,
Both machines rollback to their last completed checkpoints,
"Machine A" continues; it does NOT resend the data,
"Machine B" expects data from "machine A" but never receives it!

Both these examples highlight why individual checkpointing cannot provide correct rollback after machine failure. Somehow, the interactions between machines must also be captured.

Can we checkpoint only those entities with data dependencies?

An algorithm is required which allows a system to be checkpointed in small sections but maintains the necessary data dependencies so data is neither duplicated nor lost. One such solution would be to determine which machines have communicated with which between checkpoints and force these to be checkpointed simultaneously. A better solution can be found by considering the problem on a finer scale.

Modified data is produced by processes running given programs. A process reads data, processes it, and writes results accordingly. Therefore, since a process is the smallest unit of data transformation, it seems the sensible unit for checkpointing. In fact, this turns out to be true only in a system like UNIX where one process inhabits one protection domain. In ARIUS, many processes may inhabit a single domain, each transforming data, therefore, it is domains that must be checkpointed. Every process interacts with others in the system

through joint data structures maintained via the DSM mechanisms. These interactions must be captured if complete checkpoints are to be made and the problems highlighted in §6.1.4 avoided. The DSM and virtual memory allow interactions between nodes to be captured at the page level, and so this level of data dependency can be used.

6.2 Local data dependent checkpointing

A scheme will now be examined to provide parallel volatile checkpointing by using virtual memory and DSM together with failure independent machines for checkpoint deposition. The algorithm to provide data dependencies on a single machine will first be examined, followed by its enhancement to incorporate remote machines through distributed shared memory. Finally, the exporting of checkpointing data to other machines will be explained.

6.2.1 Data dependences in a single domain

A single domain interacts with memory in a simple way. Data is read from memory pages, operated upon by the processes in the domain according to their programs, and the results written back to memory pages. Therefore, the dataflow can be considered to be a flow from pages, through the domain, and out to other pages (see figure 6.2). A simple rule can define this relationship:

Rule 6.1

All modified pages are dependent on all accessed pages.

The pages which have been accessed and the pages which have been modified can be determined by examining state information kept in the virtual memory subsystem.

Rule 6.2

A locally referenced page may have one of four states; "unaccessed", "read only", "modified only" or "read and modified"[2].

[2]Conventional memory management support does not provide facilities to separate the two states "modified only" and "read and modified"

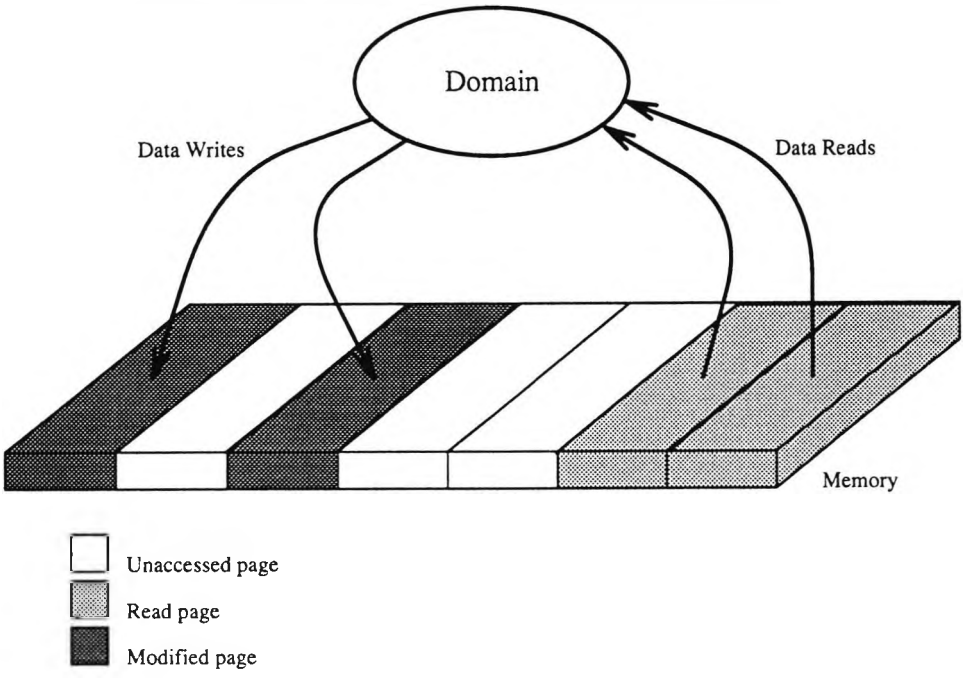


Figure 6.2: The flow of data in a single domain

This state information can be gathered in one of two ways: either by examining state information kept by the virtual memory subsystem at the moment of checkpointing or, by noting the memory accesses as they occur so that it is immediately available at the moment of checkpointing. This last method is the most attractive. When a page is first accessed, a page fault will be taken to allow its virtual-to-physical mapping to be determined and cached. Likewise, when a modification is first made to a page, a page fault will be taken to *dirty* the page. In both these cases, a note of the access can also be made.

It is therefore possible to determine, for any measuring period, which pages are dependent upon which others. Although it would be more exact if the order of modification and accesses were maintained, this would place an increased overhead on the virtual memory subsystem, requiring many more page faults to note this information. It is debatable whether the information gathered would be any more useful. Only a limited amount of information is therefore kept.

For a single domain on a single machine, the operation of checkpointing is relatively trivial and can be separated into three sections. The first section actually determines which pages require checkpointing. The processes are halted, the pages marked, and the processes

restarted. The use of a *copy-on-write*[3]*-checkpoint* mechanism allows the processes to be restarted *before* the data has been written in the checkpoint. Any subsequent attempts to modify these pages will force a copy to be made and modified instead. The second section performs the actual checkpointing in a lazy fashion. The remaining section commits the modifications by writing the index block. Once all this has been completed correctly, the checkpoint is complete. The algorithm is detailed below:

Algorithm 6.1

Begin the checkpointing operation,
Halt the processes in the domain,
Iterate through all the pages the domain has modified since the last checkpoint and mark each one "copy-on-write-checkpoint",
Restart the processes,
Iterate through the "copy-on-write-checkpoint" pages and write each one to a failure independent store (eg. a disk),
Write the domain's "index block" so committing the checkpoint,
End the checkpointing operation.

Parallels may be drawn between this system and the improved Challis' Algorithm; the use of copy-on-write for modified pages before commit here is similar to the use of shadow pages. Notice that, in this simple system, only information about modified pages is required to make the checkpoint. In fact, only modified pages will ever need writing during a checkpoint but information regarding which pages have been accessed is important as will be shown later.

6.2.2 Data dependences between two domains

Now consider a more complex system; that of two domains which interact on a single machine (see figure 6.3).

In this system, two domains access pages and modify both unique and shared pages as a result. What data must be saved if Domain#2 performs a checkpoint? Domain#2 will

[3]Copy-on-write is a technique which allows lazy copying of data. A page of data may be virtually copied but only physically copied when either the original or virtual copy is modified.

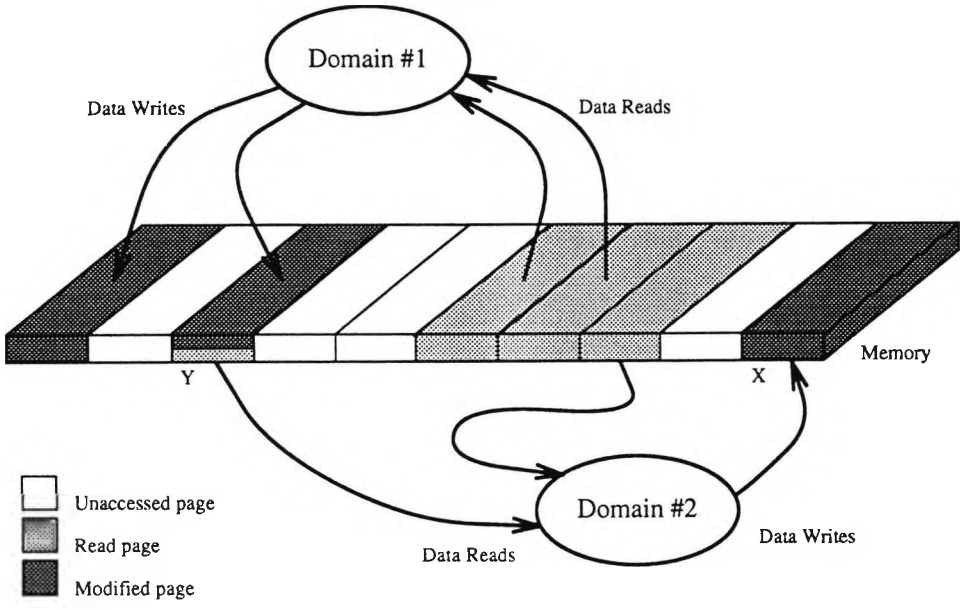


Figure 6.3: The flow of data for two domains (1)

commit the page it has directly modified (page X). The domain also depends on two other pages (see Rule 6.1) one of which has been modified by Domain#1. Failure to take account of this domain interdependency could result in incorrect rollback.

Figure 6.3 can be considered as two scenarios. Page Y is shared between the two domains; Domain#1 modifies it whilst Domain#2 reads it. Firstly, Domain#1 may write data to Page Y which is then read by Domain#2. Secondly, Domain#2 may read data from Page Y which is then overwritten by Domain#1.

In the first scenario, information is transferred from Domain#1 to Domain#2 via Page Y. This establishes a data dependency from Domain#2 to Domain#1; if Domain#2 is checkpointed then so must Domain#1. The converse is not true since no data passes in the opposite direction. In the second scenario, no physical information is transferred between domains. However, data that Domain#2 depends upon is overwritten by Domain#1. This can be viewed as "invalidation" of the old data by Domain#1 which then depends upon this invalidation. Consequently, a dependency from Domain#1 to Domain#2 is produced. No converse dependency is generated.

Unfortunately, in the fixed periods of time over which dependency information is gener-

ated, it is impossible to determine ordering of accesses and modifications by the domains. Therefore, it must be assumed that either scenarios could have taken place. The only solution therefore is for both domains to depend upon each other. This may be encapsulated in the following rule (refined from Rule 6.1):

Rule 6.3

At the time of checkpointing, all modified pages depend on all accessed pages which have themselves been modified.

In this example system, with two domains operating on a single machine, the checkpointing operation is quite simple and detailed in the following:

Example 6.3

Begin the checkpointing operation,
Halt the processes in domains #1 and #2,
Iterate through all the pages these domains have modified since the last checkpoint and mark each one "copy-on-write-checkpoint",
Restart the processes,
Iterate through the "copy-on-write-checkpoint" pages and write each one to a failure independent store (eg. a disk),
Write the domains' "index blocks" so committing the checkpoint,
End the checkpointing operation.

There is a problem here. In the penultimate step, the checkpoint is committed by the writing of *two* index blocks. Problems would arise if only one index block were written before a rollback became necessary; the checkpoint should not be restored since it was not completed. Unfortunately, the written index block would be restored to its domain—this is incorrect.

The example shown, and checkpointing steps given, do not constitute a complete solution (even disregarding the problem of index blocks for the moment). In order to generalise, consider figure 6.4. Here both domains modify private data pages whilst reading other shared pages. No data dependencies are generated between them because there is no flow of information between them; they only depend on unaltered data. Consequently, each domain may be checkpointed independently.

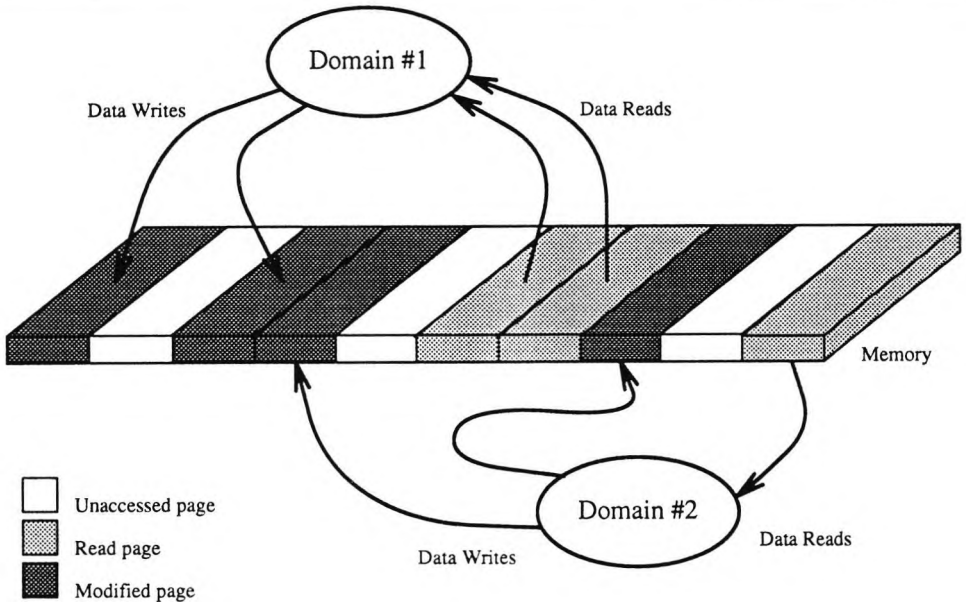


Figure 6.4: The flow of data for two domains (2)

6.3 General algorithm for local data dependent checkpointing

Figure 6.5 shows a matrix of domains and pages. Each domain's page reference is marked showing how it was used in the measured period of time. Three page states are possible (see Rule 6.2) and are indicated in the figure. By applying Algorithm 6.2 to this system, two independent checkpoints can be found; one containing domains #1, #2, #3, #5 and #8, the other containing domains #4, #6 and #7.

Inspection of this example and application of Rule 6.3 allows the development of a general domain checkpointing algorithm:

Algorithm 6.2

Begin the domain checkpoint operation
If the domain is not marked "active" then return immediately,
Mark the domain as "checkpointing"
Halt all processes within this domain
For each page accessed by this domain {
    If this page has not been modified, go onto the next page
    If this page is marked "active" {

```
            Mark this page "checkpointing"
            Flood the checkpoint to all local domains which have accessed this page
        }
    }
    Mark the domain as "checkpointed"
    Synchronise with other dependent domains
    Release any processes halted in this domain
    For each page which has been marked "checkpointing" {
        Write the page to failure independent media
        Mark the page "checkpointed"
    }
    Write the index block so committing the checkpoint
    Mark the domain "active"
    End the domain checkpointing operation.
```

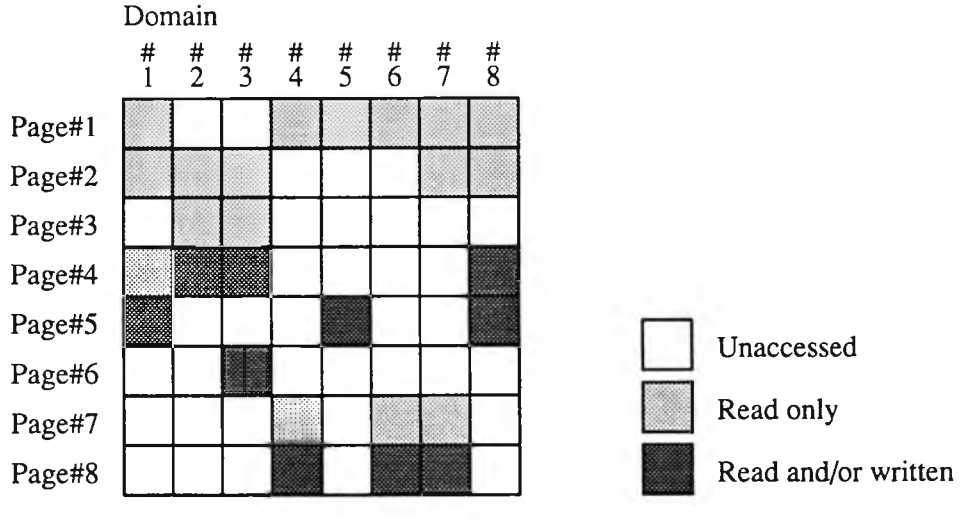


Figure 6.5: Matrix of domains and pages

The algorithm uses a "flood-fill" technique to find and checkpoint modified pages. Pages which have been modified (indicated by a row containing a dark block in figure 6.5), allow other accessing domains to be "flooded", so including them in the initiator's checkpoint. Pages which have no modifications are not "flooded" since there has been no change in the accessed data. A typical implementation of this algorithm would construct a list of accesses to each page, accumulated by the page fault handler of the VM subsystem, to enable the page searching to be done quickly.

6.3.1 Implementation problems

Unfortunately, the algorithm has a three implementation problems. Firstly, the step to *synchronise with other dependent domains* presents problems with multiple simultaneous checkpoints, a situation where the flood-fills initiated by multiple checkpoints collide resulting in a number of partial checkpoints. Together these would provide a correct checkpoint but separately they are incomplete. This situation is very likely to arise in a shared memory multiprocessor and a solution is non-trivial.

Secondly, it is possible that processes are modifying pages at the same time they are being checkpointed. The copy-on-write mechanism allows snapshots to be taken of each page but this must be carefully coordinated if a correct checkpoint is to be produced. Consider the following example. Two pages, A and B, exist in two separate domains and both are modified. When a checkpoint is initiated, these pages must be "copy-on-write" protected. First, A is protected. It is then modified, so causing a copy to be taken, by a process which then modifies page B. Page B is then "copy-on-write" protected as it is added to the checkpoint. This checkpoint is causally incorrect; data in page B exists without the causally preceding data in page A. It is not difficult to see how this can happen, especially in multiprocessor shared memory machines.

Finally, the checkpoint commits multiple *index blocks*, one for each included domain. If some index blocks were not committed before a machine crashed, it would be possible to recover incorrectly after failure. A scheme is required to guarantee that a single action commits the whole of the checkpoint, like the single *root block* in Challis' Algorithm.

6.3.2 Solutions: Multiple simultaneous dependent checkpoints

The simplest solution would be to allow only one checkpoint to occur at any one time. This is clearly undesirable because it introduces a bottleneck into the system and is a solution which cannot be easily extended for distributed memory machines.

A better scheme therefore is to provide some form of domain synchronisation whereby each domain proceeds to a synchronisation point and then waits until all other domains have reached this point before continuing. This may resemble simple barrier synchronisation but

it is more complex since the dependencies among domains are not known until each domain has completed its first stage of checkpointing. Therefore, unlike barrier synchronisation, neither the dependencies nor the number of them are known in advance.

The scheme is similar to the flood-fill technique used to spread the checkpoint from one domain to another and relies on simple domain prioritorisation and domain specific locks.

Algorithm 6.3

```
Begin synchronisation of dependent domains (assuming priority and checkpointing locks are
    set)
Release domain's checkpointing lock
If domain's dependent set contains any domains with a higher priority than my own {
        Select the highest from the set
        Wait for the release of its priority lock
}
Else {
        Recursively examine my dependent set {
            If examined domain has a greater priority than my own {
                Abort examination
                Wait for the release of its priority lock
            }
            Else if examined domain has its checkpointing lock set {
                Wait for the release of the checkpointing lock
            }
        }
        I am the highest priority domain, so mark myself "master"
}
Release my priority lock
End synchronisation of dependent domains
```

Firstly, the domain sets its *priority lock* and *checkpointing lock*. Then it performs the first stage of checkpointing. When this is done, it releases its *checkpointing lock*. Next each domain examines its set of dependents, the set contains those domains to which a flood was sent. If this set contains any domain with a higher priority than its own, the current domain waits on the highest domain's *priority lock*. If the set contains only lower priorities, then the domain begins to recursively examine its dependents, avoiding any cycles in the graph by logging those it has already visited. If at any stage a dependent is discovered with its *checkpointing lock* set, then the domain waits until it is released. If a domain is

encountered with a higher priority than the current domain, then the domain aborts it search and waits on the high priority domain's *priority lock*. Eventually, all domains will be waiting on another except for one, the highest priority domain or *master*. When this one has completed its search it has determined two things; that it is the highest priority domain and that all other domains have completed their checkpointing. The domain then releases its *priority lock* and proceeds with the rest of the checkpointing process. When this lock is released, other domains will restart. They immediately release their own *priority locks* and continue with the rest of their checkpoints.

An example of this algorithm in operation is shown in figure 6.6. Here six domains are linked by various dependencies (a). Of the six, only 1 & 2 need to initiate a recursive search of the graph to determine the highest priority, the others simply synchronise on their set's highest (b). 2 soon discovers that domain 1 is of a higher priority (c) whilst 1 examines the whole graph to discover it is the highest in the dependency group (d). It then releases its *priority lock* (e) so releasing the others (f).

This algorithm may appear complex but it has the advantage of incurring no additional cost in the page fault handler (the page accesses were logged for checkpointing purposes anyhow) nor does it facilitate false dependencies or bottlenecks. As will be seen later, it also extends to distributed checkpoints.

6.3.3 Solutions: Causally correct checkpoints

A causally incorrect checkpoint can occur in two ways; if a process modifies a page of data which has already been marked *copy-on-write checkpoint* and then modifies a page which has not, or if a process modifies a page of data which has already been marked *copy-on-write checkpoint* and then modifies a *clean* page which is shared by a processes not yet checkpointed.

All problems are caused by a process generating a "copy-on-write" fault and then modifying further, as yet uncheckpointed, pages. By halting a process, when it generates a copy-on-write fault, the further modifications which cause the problem are prevented. The halted process may be restarted when all appropriate pages have been marked.

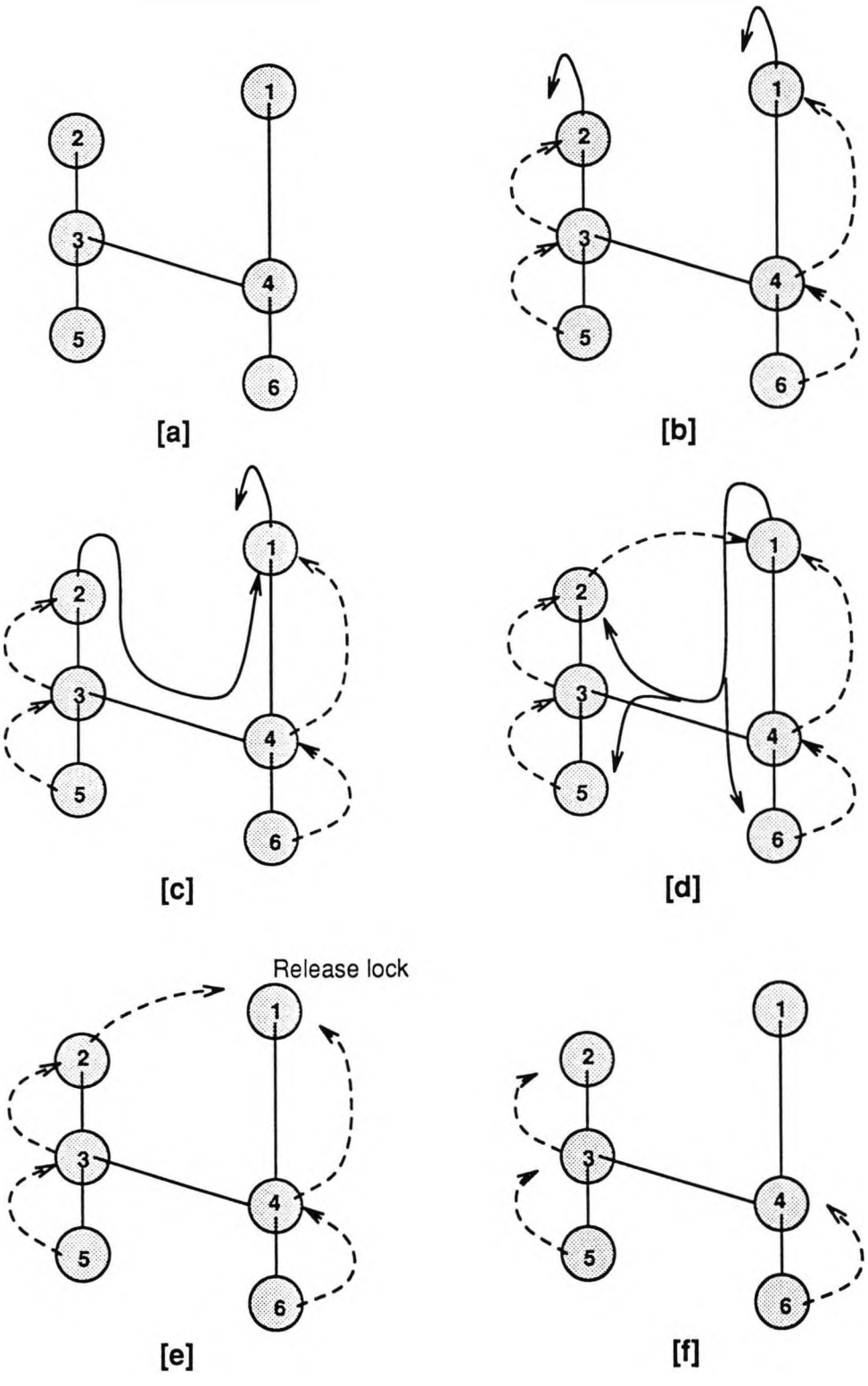


Figure 6.6: Synchronisation among six domains

Algorithm 6.4

Begin the "copy-on-write" fault handing
If the faulting page is marked "checkpointing" and the current domain is not marked "checkpointed" {
    Halt the process until the checkpoint instantiation responsible wakes it up
}
Perform the copy-on-write operation, marking the new page as "active"
End the "copy-on-write" fault handing

Here a process is halted when a copy-on-write fault is generated, the faulted page is marked "checkpointing" and the current domain marked "checkpointed". This situation will arise when a page forms part of a checkpoint in another domain but, as yet the checkpoint flood has not progressed to the current domain to halt any processes it may contain. Once the checkpoint flood procedure has completed, the process is woken and allowed to proceed. At this point any checkpointed pages will either be marked "checkpointing" and the domain marked "checkpointed", or marked "checkpointed" and the domain marked "active". Both these cases do not halt processes. This simple algorithm guarantees that any further page modification and copy-on-write faults will form part of the next checkpoint and not the current one.

6.3.4 Solutions: Single action commitment

By providing each domain with its own *index block*, the granularity of checkpoints is decreased. Unfortunately, when a checkpoint is made, the commitment of these index blocks must be correctly coordinated if the result is to be consistent. Failure to do so could result in rollback recovering only part of a checkpoint.

The algorithm described in §6.3.2 provides a way to organise the index blocks so that correct recovery can be guaranteed. This algorithm elects one domain as *master*, the rest being *slaves*. The identity of the master can be propagated to the slaves as part of the priority lock releasing mechanism.

Algorithm 6.5

Begin index block commitment (assuming commitment lock has been set at the beginning of the checkpoint operation)
If domain is a "slave" {
    Commit the checkpoint marking it "slave to master"
    Release the domain's commitment lock
    Wait for release of master's commitment lock
}
Else {
    Recursively examine the dependency set {
        If the examined domain's commitment lock is set, wait until it is cleared
    }
    Commit the checkpoint marking it "master"
    Release the domain's commitment lock
}
End index block commitment

The checkpoint operates as originally described except that an extra *committed lock* is set before it begins. The operation proceeds until the commitment of the index block is reached. If the domain is a slave, the commitment is made but the block is marked *slave to master "n"* where "n" is a unique identifier determined during synchronisation. Once this is done, the *committed lock* is released and the domain pauses until the master releases its own *committed lock*. If the domain is the master, its dependency set is recursively examined as in the previous synchronisation procedure. Here, if any domain is encountered with its *committed lock* set, the domain pauses until it is cleared. When the examination is complete, all slave domains must have committed their index blocks. Therefore it only remains for the master to commit its own, marked *master "n"*, and release its own *committed lock* to complete the checkpoint.

Recovery of any index block marked *master "n"* will always succeed but recovery of an index block marked *slave to master "n"* will only succeed if the associated master has also been recovered.

6.4 Local Rollback

On a single machine, failure of the machine is equivalent to the failure of the system. Checkpointing allows the machine to be restarted from its last consistent position making use of data stored on a failure independent store, most probably a disk. The rollback procedure must analyse the stored data, determine which are the most up-to-date and complete checkpoints, and use this data to restart.

Algorithm 6.6

Begin rollback
Retrieve index blocks from failure independent store
Collect together a set of master index blocks
Collect together a set of slave index blocks
For each block in the slave set {
    If the slave's master is not in the master set, discard the index block
}
Re-initiate the virtual memory system from the recovered block indexes
End rollback and restart the system

This algorithm operates in three phases. Initially, the index blocks are recovered and ordered into two sets, one containing all the master index blocks, the other containing all the slaves. The slave blocks are then examined in turn and, if no associated master is present, they are discarded. Finally, the resulting image is assimilated into a new virtual memory stage from which the the machine may restart. It is important to note that, if the failure independent media cannot provide the index blocks due to a media error, recovery cannot take place. Such an error is best handled by using conventional archival backups.

6.5 Remote checkpointing

A complete solution to the distributed checkpoint problem must also handle remote domains sharing data through the DSM mechanism of chapter 4. Any dependencies which exist between these domains and local ones should cause the checkpoint operation to "flood" between machines. It should also be possible to use these remote machines as

repositories for checkpointed data (if they are failure independent). This provides the system with the required volatile checkpoints. In the following sections, the inclusion of remote data dependencies will be analysed and the use of DSM to support volatile checkpoints will be discussed.

6.5.1 Remote data dependencies

A complete checkpoint in a distributed machine must include not only locally dependent data but also remotely dependent data. Analysis of the DSM information kept for distributed shared pages should indicate which data must be saved and which domains are dependent.

Figure 6.7 shows a number of interacting domains split between two physical machines (the VM state marked in the top matrix is the same as in figure 6.5). The bottom matrix indicates the DSM state of each page. A comparison of these two matrices shows there is insufficient state held to determine remote dependencies correctly. Page#8 provides a good example of this; the page is owned by domain#4 and shared with domain#6. However, domain#7 has also made use of the page during the checkpoint period, as indicated in the VM state, but the DSM state does not provide enough information to determine this. Some additional DSM augmentation is therefore necessary to enable these dependencies to be determined.

6.5.2 DSM modifications

The DSM system currently holds a chain of DSM entries describing the machine designated owner as well as those holding copies for each page. At the time of a checkpoint, this chain does not necessarily hold all machines which have accessed and modified a page during the checkpoint period. A further list is required to hold this additional information (figure 6.8).

The DSM supports this information in a DSM checkpoint chain (DCPC). This chain is similar to that used to hold the "DSM copies" information but, unlike it, it is not truncated by page invalidations.

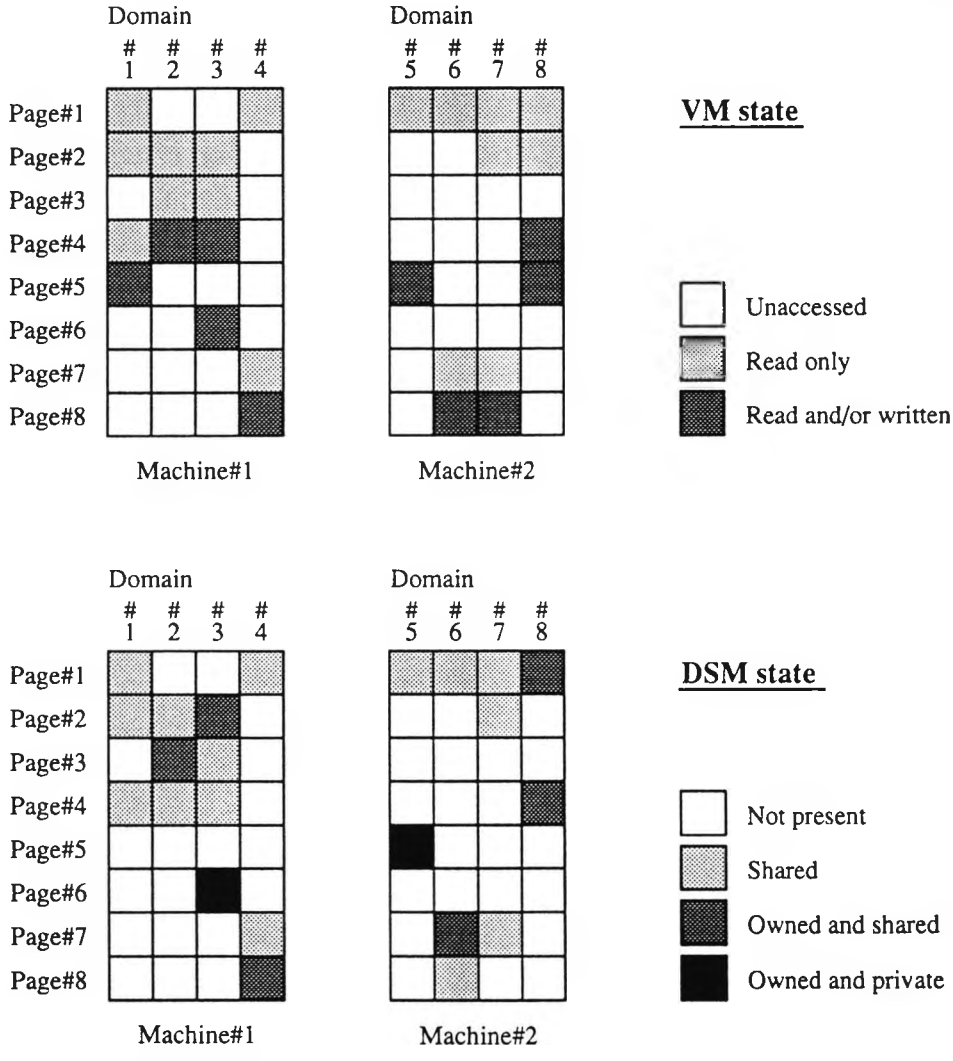


Figure 6.7: Matrix of processes and pages for two processors

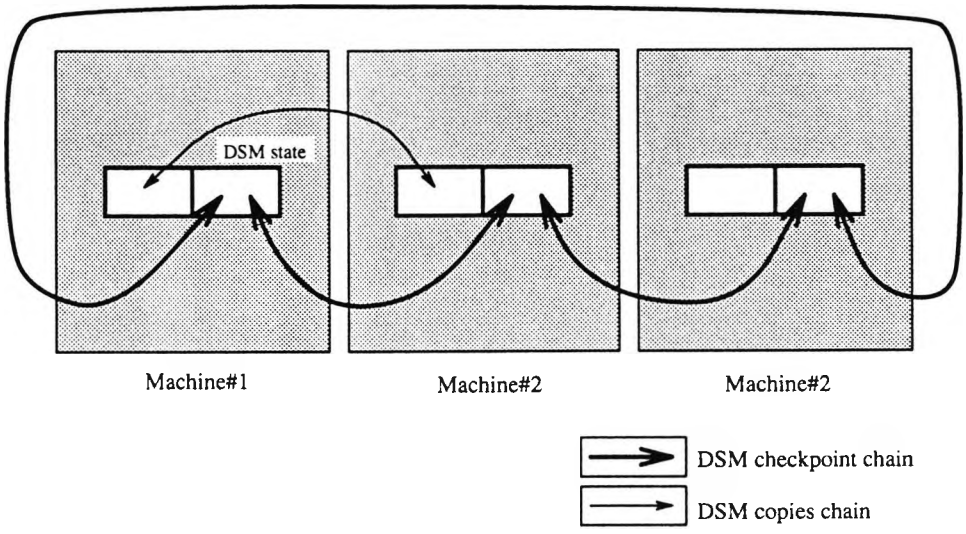


Figure 6.8: DSM copies and data dependencies

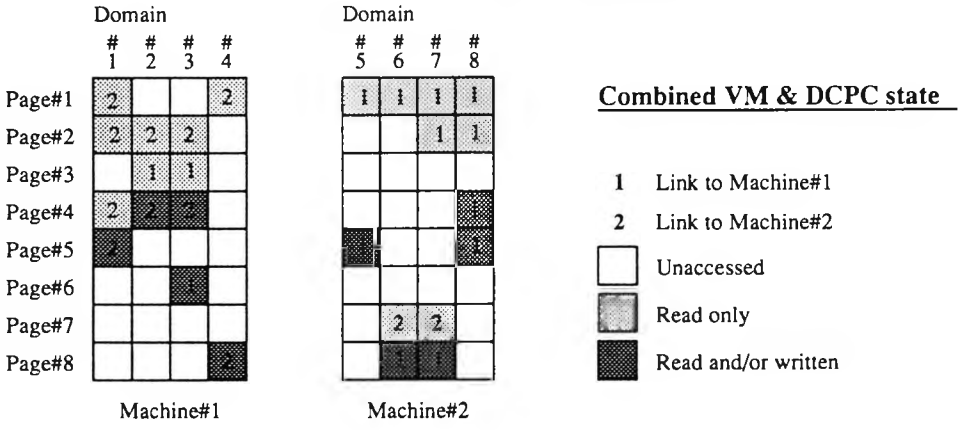


Figure 6.9: DSM checkpoint information

Rule 6.4

A machine is added to a DSM checkpoint chain if it accesses a modified page or holds an unmodified page which is invalidated.

Rule 6.4 describes the conditions under which a machine is added to the DCPC. Note that DCPCs are not constructed for clean data pages since they cannot affect the current checkpoint state.

Figure 6.9 shows both the VM state, used for checkpointing local systems, together with the DCPCs which show how page copies have migrated to machines during the checkpoint

period. The circular nature of the chains allows any machine to locate all other machines which are dependent.

6.5.3 Using DCPCs for distributed checkpoints

DCPCs provide a mechanism to locate all machines which depend on a page. By combining this with Algorithm 6.2, a new algorithm may be designed capable of handling distributed checkpoints.

Algorithm 6.7

Begin the domain checkpoint operation
If the domain is not marked "active" then return immediately,
Set all relevant locks
Mark the domain as "checkpointing"
Halt all processes within this domain
For each page accessed by this domain {
    If no domain has modified this page, go onto the next page
    If this page is marked "active" {
        Mark this page "checkpointing"
        Flood the checkpoint to all local domains which have accessed this page, adding any domains to the dependency set
        If the DCPC points to any machines which are not this one, flood the checkpoint to the remote machines' domains
    }
}
Mark the domain as "checkpointed"
Synchronise with other dependent domains
Release any processes halted in this domain
For each page which has been marked "checkpointing" {
    Write the page to failure independent media
    Mark the page "checkpointed"
}
Perform index block synchronisation
Mark the domain "active"
End the domain checkpointing operation.

6.5.4 Implementation problems

This algorithm has three problems. Firstly, Algorithm 6.3 relies upon the creation of a dependency set to allow correct synchronisation to take place later. No attempt is made to handle the addition of remote domains to the dependency set. Secondly, as with Algorithm 6.2, it is possible to obtain causally incorrect checkpoints. Although the local case is still handled by Algorithm 6.4, the remote case could still allow data to propagate incorrectly. Thirdly, although commitment of index blocks on each machine is correctly coordinated by Algorithm 6.5, the commitment between machines is not controlled. This problem is associated with the need to add distributed synchronisation. However, this problem may be better addressed by removing the need for a root index block altogether and providing a more distributed commitment solution.

6.5.5 Solutions: Multiple simultaneous dependent checkpoints

The DSM checkpoint chain for any given page is a doubly linked circular list of nodes which have modified or referenced the modified page. Consequently, any node following the links will eventually return to itself. Also, ARIUS relies on DSM to provide access to all data in the system. Therefore, accessing a remote domain's structure is no more difficult than accessing a local one. By combining this essential property with the properties of DCPCs, the production of a distributed dependency set can be described.

In Algorithm 6.7, whenever a flood is made to a remote domain, the domain is added to the dependency set. This is sufficient to allow correct synchronisation to take place without further modification. It is not necessary to add the flooding domain to the flooded domain's dependency set. The circular nature of the list ensures that the necessary dependency will exist.

6.5.6 Solutions: Causally correct checkpoints

Remote causally incorrect checkpoints can occur when a process obtains a checkpointing page from a remote node and then modifies a local page. This results in the obtained

page occurring twice in the checkpoint in two different forms. To overcome this problem, it is necessary to prevent the DSM providing copies of the page whilst it is checkpointing. Therefore, if a page is marked *checkpointing*, any DSM request made to it should be returned as an error. The behaviour is similar to the page being marked *busy*.

6.5.7 Solutions: Single action commitment

Challis' algorithm uses a root block to commit a checkpoint. This data resides at the root of the tree of changes and, once committed atomically, makes all other data in the checkpoint available. Similarly, the multiple *index blocks* of Algorithm 6.2 are organised as a master/slaves tree, commitment of the master committing the whole checkpoint. Although such a mechanism is easy to implement in a distributed system, such as ARIUS, volatile checkpoints provide an opportunity for a more flexible solution (§6.7.3).

6.6 DSM storage for volatile checkpoints

A DSM system provides an ideal way of implementing volatile checkpoints; that is the maintenance of checkpoints in distributed memory rather than on disk or other persistent stores. DSM data is often duplicated on many machines due to sharing in parallel programs, and such data could be checkpointed at almost no cost since the replica needed to provide fault-tolerance is already present. By enhancing this inherent property, volatile checkpoints are easily implemented.

6.6.1 DSM checkpoint copies

When a checkpoint is performed on a machine, the modified data must be written to an independent failure storstore. For volatile checkpoints this is another machine's memory. For every DSM state, which is marked *copy* or *owner_many*, at least two copies of the page must exist so that there is no need to produce duplicates. For states which are marked *owner_one*, a copy must be exported. In nearly all cases, copies of the page are being moved between dependent domains since the page in question has been modified and

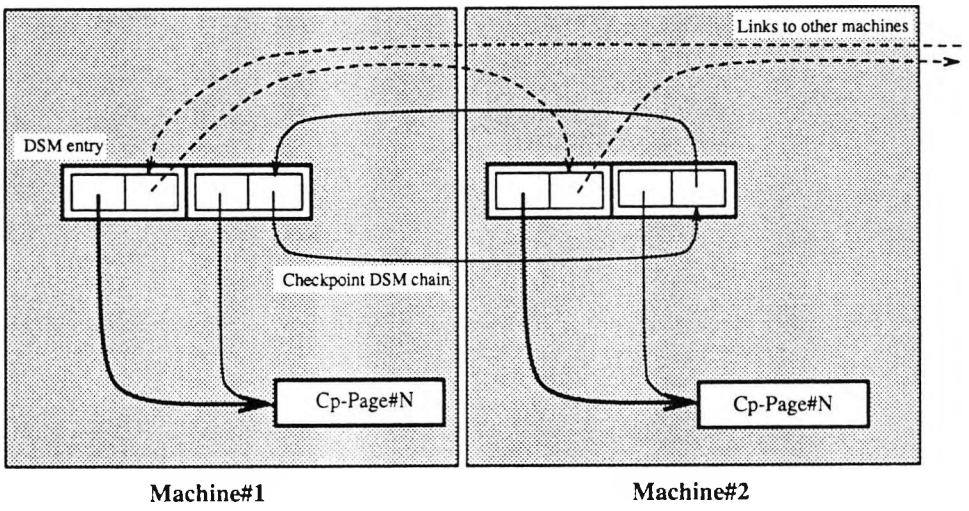


Figure 6.10: DSM checkpoint copies (1)

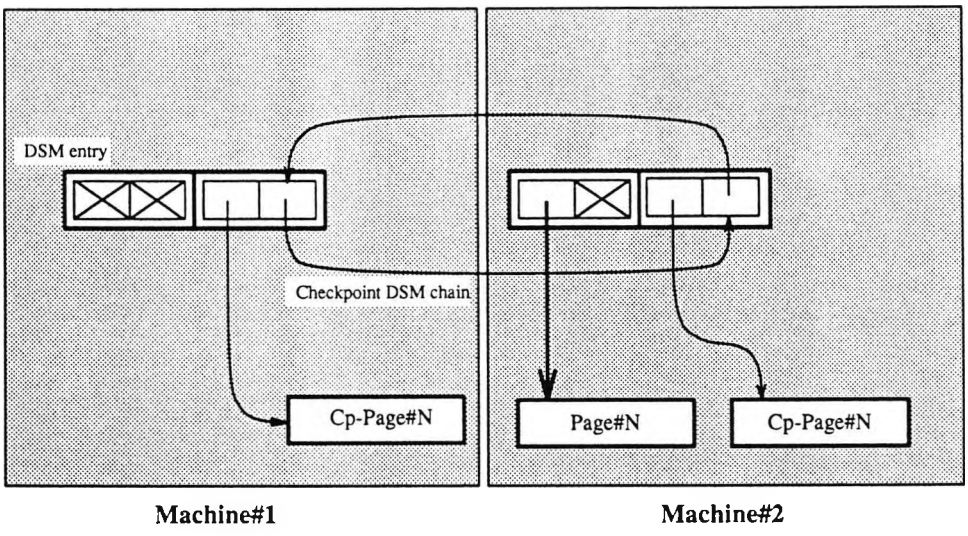


Figure 6.11: DSM checkpoint copies (2)

accessed by the two domains involved. The page will therefore be incorporated into both domains' respective index blocks. In one case, the page has been accessed and modified by only one domain. In this instance, the page must be exported to a dependent of the domain so it will be included in any rollback which may subsequently take place. If this cannot be done, then a *fake domain* is created remotely. This domain's sole purpose is to hold these checkpoint pages and, on failure, perform the required rollback operation to recover the domain.

Once copies exist, it is necessary to mark them as *checkpoint copies*, that is pages which are part of a checkpoint and must not be modified or deleted. Modification of DSM copies is not permissible anyhow, but disposal of copies is normally legitimate and so must be prevented.

Figure 6.10 shows the state of two machines after a single page has been checkpointed. When the checkpoint was initiated, copies of this page were present on numerous machines, indicated by the dotted dependency links, including machines #1 and #2. The checkpoint is initiated from machine#1 and, since Page#N has been modified in the checkpointing period and, is *owned and shared* by #1, it is declared a *checkpoint copy*. Machine#1 then instructs its downstream neighbour to do likewise. This results in the circular checkpoint DSM chain (indicated by the narrow lines). The current DSM version of the page may share the checkpointed version until a modification is attempted, at which point a copy must be made to prevent alteration of the checkpoint.

In figure 6.11 such a modification by machine#2 has forced a copy of the page to be made and all others to be invalidated. The checkpointed pages are still maintained on machines #1 and #2 via the checkpoint DSM chain.

6.7 Distributed rollback

When a failure is detected, because some communication with another machine has errored, the detecting machine initiates a rollback. Only those domains which have direct or indirect DSM links to the failed machine are affected by the failure and these need only be returned to their state at the previous checkpoint. The checkpoint dependency

information constructed at the time of the failure describes how the failure affects other domains and, consequently, describes which other domains need be rolled back.

The rollback algorithm is identical to a checkpoint algorithm except that, where a checkpoint would mark a page "checkpointing", rollback discards the page and reinstalls the one written into the previous checkpoint.

Algorithm 6.8

Begin rollback operation — DSM error
If the domain is marked "in rollback" then return immediately,
Set all relevant locks
Mark the domain as "in rollback"
Halt all processes within this domain
For each page accessed by this domain {

If page is clean, go onto the next page
Flush this page so returning to the last checkpoint version
Flood the rollback to all local domains which have accessed this page, adding any domains to the dependency set
If the DCPC points to any machines which are not this one, flood the rollback to the next DCPC entry, adding it to the dependency set

}
Synchronise with other dependent domains
Release any processes halted in this domain
Mark the domain "active"
End the domain rollback operation.

6.7.1 Implementation problems

This algorithm fails to take account of three problems. Firstly, a failed machine may lose other machines' checkpoint data. Although copies are recovered during rollback, only one copy of checkpointed pages may then exist. Unless these are duplicated, further failure could result in loss of these remaining copies and so an unrecoverable failure. Secondly, failure during checkpointing is not considered. Thirdly, whilst the rollback is flooding to dependents, it is possible for domains which are not rolled back to access those which are.

6.7.2 Solutions: Rollback data duplication

The problem of rollback data duplication can be divided into two; the first is the problem of locating and re-duplicating the necessary pages, the second is informing all relevant machines that a fault has taken place.

Page re-duplication

After a machine failure and rollback, it may be necessary to re-duplicate pages which are no longer fault tolerant because only a single copy remains in the system; the duplicate having being lost on the failed machine. The use of circular DSM chains allows any page copy to determine the other machine holding a copy, so it is simple for a page to be checked after rollback to establish whether the failed machine held a copy or not. If a copy were lost, then the page can be re-checkpointed.

It is more difficult to establish the lost pages if they are not part of the current rollback. Although a complex list strategy could be used to maintain a record of which pages have duplicates on which nodes, the infrequency of crashes makes the cost of maintaining this information prohibitive. A simple linear page search is therefore used when a machine failure is noted.

Machine dependents' notification

Informing all relevant dependents of lost data is unlike checkpointing because it does not cause rollback. It is quite conceivable, though perhaps unlikely, for a machine failure not to cause rollbacks in AMOS yet require various nodes to re-duplicate data. In effect, machine failures are causing partial re-checkpointing because data which is not immediately relevant has been lost.

The problem of re-duplication has already been considered, but the process of informing the relevant machines has not. Initially, only a single machine will notice the failed machine, and this may have no need to re-duplicate pages itself nor know of others that might. How is the failure to be reported to those machines to which it matters?

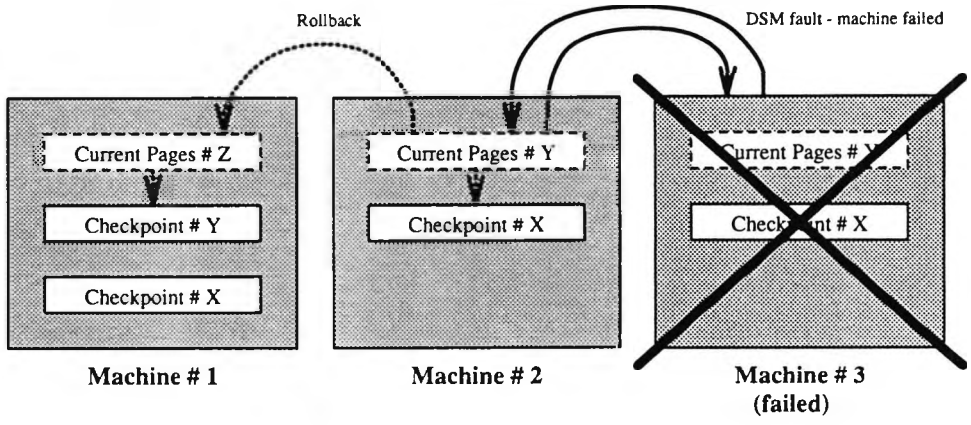


Figure 6.12: Example of failed rollback

If a broadcast mechanism is considered to be prohibitively expensive, then the simplest solution is for each machine to be responsible for determining the failure of any dependent machines; when a failure is noted any necessary pages are reduplicated to another failure independent machine. Such a scheme has the advantage of being distributed, no one is responsible for reporting failures, but has the disadvantage of forcing each machine to check the state of its dependents periodically in case they fail. Since this checking is not linked directly to the failure, being time oriented instead, the period between failures is increased from the period of recovery to the period between these *state checks*. If these checks are made too frequently, then any network will be swamped; if they are made too infrequently, the period between failures will be excessively large.

A slightly more elegant solution is to group machines together into *state groups*. These groups operate as already proposed but, when a failure is detected, the detector informs others in the group. By staggering the state check times of different machines in the group, the inter-failure time becomes the period between machine state checks divided by the number of machines in the state group.

6.7.3 Solutions: Rollback during checkpointing

As a checkpoint proceeds, it may discover that an involved domain is on a machine which is no longer functioning. This machine failure can only be handled by a rollback operation to the preceding checkpoint. Doing this is problematic. Machines involved in the

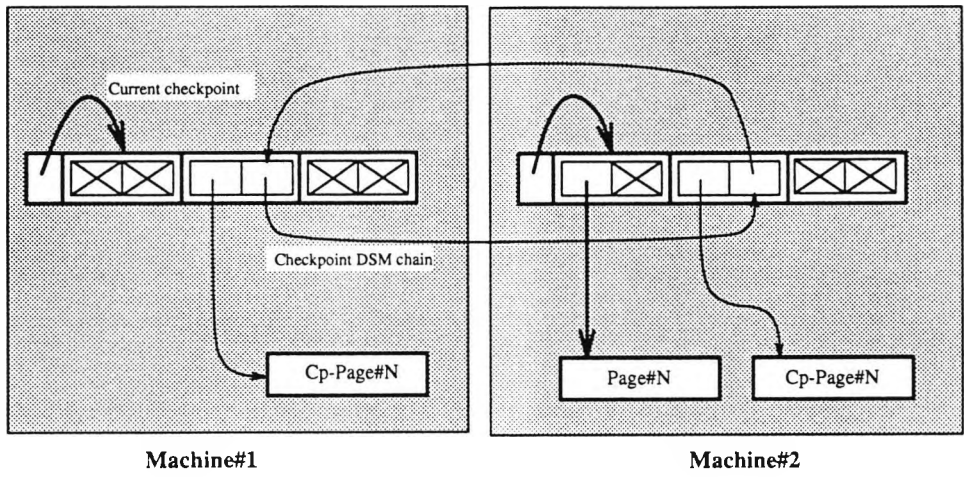


Figure 6.13: New checkpoint dependency structures

checkpointing operation will be at different stages; some having only just started, others near completion whilst others have completed. These different states make it impossible to rollback successfully in the way examined so far.

In figure 6.12, Machine#2 detects the failure of Machine#3 during a checkpoint operation. It therefore rolls back to its last checkpoint – it has not yet begun the current one before detecting the failure. It then floods the rollback to Machine#1.

Machine#1 has already completed its section of the new checkpoint when the rollback instruction is received. When it unrolls the dependent domains, it returns them to the incorrect checkpoint; the one it has just completed instead of the one preceding it. To avoid this situation and correctly rollback, a more intelligent checkpoint structure is required.

A new checkpoint dependency structure

Figure 6.13 illustrates a different checkpointing structure. Instead of using a current page and a checkpoint page, three page stages are used together with a pointer indicating which is the current one. When a page is checkpointed, rather than moving the current page to the checkpoint page and allocating a new current page, the current pointer is moved to the next page stage in a circular manner.

It is important to note that three stages are used rather than two as in the previous

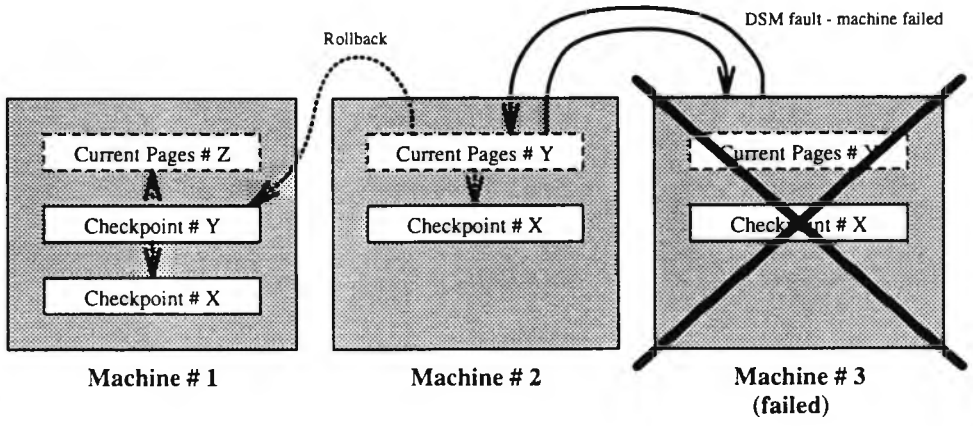


Figure 6.14: Successful rollback

system. This is necessary to handle failures during checkpoint operations. If only two were used, then at any given time it would only be possible to hold a current state and previous checkpoint state. As demonstrated in figure 6.12, it may be necessary to rollback two checkpoints at once and two stages is insufficient to handle this.

The inter-machine DSM links are also modified. Rather than just pointing to another machine's DSM entry, a link also points to a particular page stage. By making inter-machine DSM links between specific page stages, local checkpointing cannot affect the remote interpretation of these pointers. This new structure allows rollback to operate correctly, no matter when the fault is detected.

Figure 6.14 demonstrates the correct rollback of the previous example. The rollback request made from Machine#2 to Machine#1 now specifies which version of the checkpoint is to be rolled back, in this case Checkpoint#Y. Machine#1 therefore discards both this and its current page.

Reclaiming volatile checkpoint space

Further analysis of Algorithm 6.7, in coordination with the new checkpoint structure, allows old checkpoint pages to be discarded quickly. After a checkpoint has been committed, three potential page copies exist per real page; the current page, the last checkpoint, and the last but one checkpoint. It is now possible to immediately discard the last but one

checkpoint since it is impossible for it to be used. This allows space to be reclaimed quickly and can be carried out on individual machines without any communication.

It may appear that a machine accessing a page, having it checkpointed, and then never referring to it again, will never be able to discard the checkpointed version even if other machines subsequently modify it. However, Rule 6.4 specifies that a machine will be added to the current DCPC if it holds a page copy which is invalidated. This will happen in the above example, so making the machine dependent on a subsequent checkpoint operation. This operation will find nothing new to checkpoint but will be able to discard the old checkpoint page which is no longer relevant.

It is also possible to discard the access list associated with the last checkpoint. Although this must be preserved until commitment has been achieved, in case a fault occurs, after the checkpoint is stable it is no longer required.

6.7.4 Solutions: Access by unrolled back domains

It is easy to envisage a situation where a number of dependent domains require rolling back. The rollback is initiated from any in the dependent set and floods out to the others. Whilst the flood is in operation, domains which have yet to be rolled back may access others' unrolled data. This is a similar problem to checkpointing domains that are being accessed by uncheckpointed domains (see §6.3.3).

To overcome this problem, an addition to the DSM system is required, so that as well as returning an error when a page is marked *busy* or *checkpointing*, it also returns an error if the page is marked *in rollback*. This prevents the data from migrating until the rollback operation is complete.

6.8 Corrected distributed rollback

Algorithm 6.9 details a revised rollback scheme which handles the problems of rollback during checkpointing.

Algorithm 6.9

Begin rollback operation — DSM error
If the domain is marked "in rollback" then return immediately,
Set all relevant locks
Mark the domain as "in rollback"
Halt all processes within this domain
For each page currently accessed by this domain (on the current access list) {
    If page is clean, go onto the next page
    Mark the page "in rollback"
    Flush this page so returning to the last checkpoint version
    Flood the rollback to all local domains which have accessed this page, adding any domains to the dependency set
    If the DCPC points to any machines which are not this one, flood the rollback to the next DCPC entry at the specified stage, adding it to the dependency set
}
Use to supplied page/stage information to locate the relevant access list
If this access list is different from the current access list {
    For each page on this access list {
        Mark the page "in rollback"
        Flush this page stage so returning to the previous checkpoint version
        Flood the rollback to all local domains which have accessed this page stage, adding any domains to the dependency set
        If the relevant stage DCPC points to any machines which are not this one, flood the rollback to the next DCPC entry at the specified stage, adding it to the dependency set
    }
}
Synchronise with other dependent domains
Release any processes halted in this domain
For all pages that were marked "in rollback", mark them "active"
Mark the domain "active"
End the domain rollback operation.

When rollback is initiated from a domain, the domain's current *domain access list* is traversed as described in Algorithm 6.9 and any modified pages discarded. Where an inter-machine dependency is found, the rollback is flooded to the necessary machine along the specific DSM link (including the stage information). When another machine receives the rollback request, it does likewise. However, it is now possible to receive a rollback

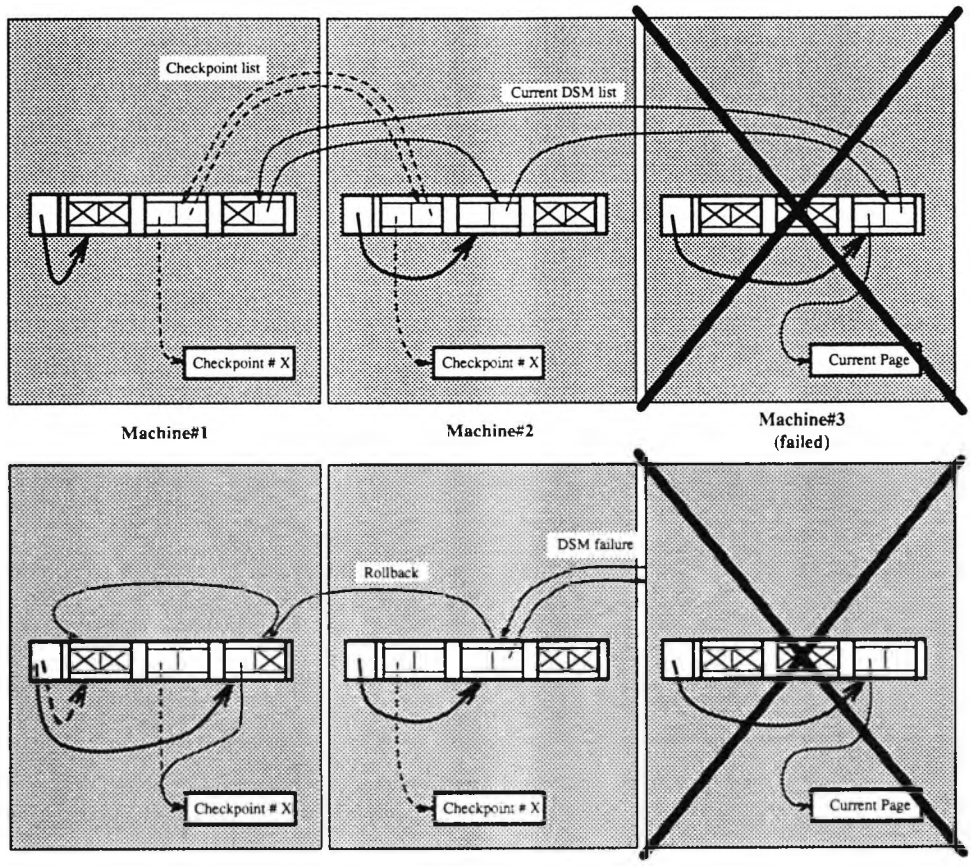


Figure 6.15: Operation of distributed rollback

from a stage other than the current one—the last checkpoint. In such a case the domain is rolled back to this checkpoint and both current state and last checkpoint are discarded, with dependents flooded as necessary.

Figure 6.15 illustrates how such a rollback traverses a system, invalidating checkpointed as well as current pages in the process. The top of the figure illustrates the page dependencies when Machine#3 fails; the bottom shows how the rollback floods the remaining machines. It would seem possible that such a system could result in a rollback invalidating all data; a rollback flooding backwards from domain to domain until all data is discarded. To avoid this a rule is imposed on the algorithm's use.

Rule 6.5

A domain may not take part in a new checkpoint until its participation in an old checkpoint is complete.

A domain becomes part of a checkpoint in one of two ways. Either a process initiates the checkpoint within it, or a checkpoint from a remote domain is flooded into it. The domain is considered part of this checkpoint until the checkpoint is committed. Therefore, until this commitment takes place, the domain is prevented from initiating another checkpoint of accepting any participation in another flooded checkpoint (simultaneous checkpoints are handled correctly as demonstrated in §6.3.2). This constraint makes it impossible to rollback more than two stages since the worst case can only result in the removal of the current domain image and the currently checkpointing (but not yet committed) image.

The inclusion of this means an additional synchronisation in Algorithm 6.7 is required to inform all domains that the checkpoint has completed. This additional synchronisation, together with the three stage checkpoint structure and Rule 6.5, provides distributed commitment. By the time the final synchronisation point is reached, all checkpoint data is stable. The three stage checkpoint entries and specific DSM links guarantee that failure at any point will correctly rollback the machines. Consequently, there is no need to provide a specific *index block* style commitment.

6.9 Recovery of lost domains

Finally, it is necessary to recover and restart the domains lost on the crashed machine. This is trivial. ARIUS is structured so all kernel data is stored in the object store AMOS. When this is recovered after failure, it contains complete domain information. The only problem to be overcome is the movement of the domain to a different machine. This can be done by the first machine to notice the domain is attached to a faulty machine, simply attaching it to itself instead. The domain and any contained processes would then become runnable on the new machine.

6.10 Summary

Volatile reliability is a method of providing fault-tolerance in a distributed machine by checkpointing to other machines' memories rather than to local or remote disk. This

enables faster checkpointing, faster recovery after failure, and a more unified checkpointing mechanism.

In order to described how this is achieved in a distributed machine supporting DSM, the algorithms for parallel checkpointing and rollback have been derived from simpler single machine principles, there problems exposed and solutions described. The resulting system is based on the ability of the DSM system to track data use and sharing in the machine as well as provide a mechanism to distribute the checkpointed data.

Like the DSM system, the checkpointing system attempts to be as parallel as possible and avoid sequential algorithms where at all possible. This is particularly true in the methods used to generate the checkpoints, and the checkpoint commitment system which does not need the restriction of an "index" to commit the checkpoint, instead relying on a distributed and inherently parallel index. The rollback algorithm is identical to that used to generate the checkpoints. However, rather than checkpointing pages, it discards them so restoring those made at the previous checkpoint. Consequently, a rollback operation is as efficient and parallel as the checkpoint operation. This is essential if faults are to be quickly handled.

Volatile reliability through DSM enables efficient and transparent fault-tolerance to be provided to any program. However, any network of machine cannot be expected operate indefinitely—at some point they must be powered down. Volatile reliability does not provide a means for its checkpointed data to be stored on a persistent media. The next chapter examines how this service can be provided.

Chapter 7

Persistent reliability

The previous chapter examined volatile reliability. This is neither adequate nor complete. Although single failures may be tolerated and recovered, a catastrophic fault consisting of two or more simultaneous faults will still result in system failure. At this point it may be necessary to reboot part of the system, so losing volatile checkpointed data. A double fault is considerably less likely if the machines are failure independent and so such fault tolerance can be provided by a slower, more persistent mechanism; effectively a system restart. Providing a solution in such a way also allows its use for restarting a system after a controlled shutdown.

Logically the inclusion of persistent reliability should be an extension of the volatile reliability already detailed; so allowing both mechanisms to be easily integrated. The following sections consider how persistent reliability should be incorporated with volatile reliability and what kind of service it should provide.

7.1 The need for persistent reliability

If the possibility exists that a machine will be unpowered, then a persistent store is necessary else all data will be lost. This is usually realised as a *file system store*. Additionally, although a system may possess sufficient volatile memory for a few executing processes,

it is unlikely there will be sufficient memory to hold the entire working set of complex parallel applications. The impression of a large store is achieved by using *virtual memory* and a *swap space store*. In both these cases the store takes the form of magnetic disks, providing cheap persistent space at the cost of increased access time.

ARIUS does not make a distinction between these two form of persistent storage. Instead, if considers all memory in the address space, AMOS, to be persistent. In this way it more closely resembles a swapping system since there is no explicit file system. Unlike a swapping system and like a file system however, data persists beyond the lifetime of the creating process.

Persistent memory in AMOS is used in two ways. Firstly, it allows currently unused data to be moved from expensive volatile memory to cheaper but slower persistent memory. Secondly, it provides a store for checkpointed data to enable the system to recover after a catastrophic failure (eg. when a power failure has lost all volatile checkpoints). However, to place every checkpoint onto disk would reduce the efficiency of the system and remove the advantages of using volatile checkpoints. Instead, movement of data to disk should be done lazily and persistent checkpoints only completed rarely or when explicitly forced by system shutdown.

7.1.1 A reliable machine unit

Before persistent reliability can be considered further, we must first define what constitutes a persistent reliable unit. If a disk in an AMOS system were to fail, then all the data on it would be lost. Consequently, all machines using that disk would fail. If many machines directly use a disk therefore, they become failure dependent.

Therefore, a failure independent machine is one which contains a number of processing elements and volatile and persistent storage. The only interactions between it and other machines are via the distributed shared memory system.

7.1.2 Persistent reliability—a new checkpoint system?

Does persistent reliability require a new set of algorithms to implement it? Such a solution does not seems sensible. Instead, the reliability of volatile checkpoints should be used to implement persistent checkpoints.

Consider the following. At any point a machine will hold data in three states:

1. **Active** data is part of a current domain and in use by processes within it.

2. **Last checkpoint** data is part of the last checkpoint of a domain. This data may not be stable if a checkpoint is still in operation.

3. **Last but one checkpoint** data is part of a previous checkpoint but not the current one. This exists while a new checkpoint is being performed. When the new checkpoint is complete, it will be deleted.

The last of these, *last but one checkpoint data*, is completely stable but represents a state to which, if the machine were returned, adverse side-effects would result.

A stable persistent checkpoint can be made at anytime by combining all *last checkpoint* data and all *last but one checkpoint* data. By committing this to persistent memory, the machine's checkpoint may be recorded. By coordinating many of these checkpoints, a system wide persistent checkpoint can be made.

7.2 Construction of a local persistent checkpoint

Determining what data should be contained in a persistent checkpoint at any given time is fairly simple. However, this is not a solution in itself. Three other aspects must be considered. Firstly, how is the data to be written efficiently; secondly, what data need actually be written to disk; and thirdly, when should a checkpoint be made.

7.2.1 Log based checkpoint stores

There has been great interest in log based file systems in recent years [RO90] arising out of the need to increase the efficiency of disk utilisation. The principles applied to file systems are also relevant to log based checkpoint stores. Large memory systems act as massive disk caches, so reducing the number of reads made from disks in comparison with writes. A disk may therefore be utilised better by optimising the performance for writes rather than reads. This is best done by aggregating large numbers of small writes into much larger, single writes. By providing such a system, disk throughput can be dramatically increased. However, to achieve this, the way in which data on disk is organised must also be dramatically altered. This is because large continuous section of the disk must be kept available in which to place the large data writes.

A full account of log based disk stores will not be given here but the principles will be applied to the design of the persistent reliability mechanisms.

7.2.2 Determining the data set

Instead of attempting to provide many independent checkpoints, as was done for volatile reliability, persistent checkpoints contain all modified data on the local machine. Experience of log based techniques has shown that large write operations give better disk bandwidth utilisation. The ability to provide large writes would be reduced by producing may independent checkpoints. Additionally, the infrequency of persistent checkpointing makes the overhead of coordinating many small checkpoints prohibitive.

§7.1.2 explained how the checkpoint data set is determined. However, this set will contain both modified and unmodified data pages as well as committed and uncommitted ones. To write all this data to disk on every persistent checkpoint would be pointless; a more efficient solution is required.

The following algorithm constructs the data set for a new persistent checkpoint.

Algorithm 7.1

```
Begin persistent data collation
Prevent any domain becoming "active"
Build a "persistent checkpoint list" of all modified pages in the machine
For each page in the "persistent checkpoint list" {
    If the page is in a domain which is not active {
        Delete the page from the "checkpoint domain list"
    }
    Else if the page has a newer version in the list {
        Delete the older one from the "checkpoint domain list"
    }
    Else {
        Allocate the page a persistent page address (eg. a disk block number)
        Mark the page "unmodified"
    }
}
Allow domains to become "active"
End persistent data collation
```

This algorithm is very simple. It creates a list of all modified pages on a machine (therefore eligible to be transferred to disk). It then discards the pages which form part of an unstable volatile checkpoint or have newer duplicates. Any remaining pages are allocated disk space and marked *unmodified*. The resulting list forms the persistent checkpoint. It is important to note the disabling of domains becoming *active* whilst the persistent checkpoint list is being assembled. This is necessary to prevent half a volatile checkpoint being included unintentionally.

The allocation of persistent pages, or disk block numbers, uses the log based principles outlined in §7.2.1. At the end of Algorithm 7.1, a number of large blocks of data are ready to be written to disk. These may be written lazily by the system and committed using a *root block* scheme similar to that described in §6.1.2.

Algorithm 7.2

```
Begin persistent checkpoint commitment
Whilst there are unwritten pages in the "persistent checkpoint list" {
```

Construct and perform a large disk write operation

}

Commit operations by writing next disk root block

End persistent checkpoint commitment

7.2.3 When to produce persistent checkpoints

There are three situations where a persistent checkpoint is necessary; when explicitly requested, when volatile space becomes scarce, or after a given period of time. The first occurs when an application considers it vital that the information is physically present on disk and the guarantee of volatile reliability is not good enough. A typical example of this is when a system is powered-down and volatile data will be lost.

The second occurs when the data set within a machine grows larger than the available volatile memory. Often, it may be possible to discard some old data pages no longer in use. Alternatively, some data pages may be shipped to another machine for temporary storage via the DSM mechanism. Ultimately however, it will be necessary to write these pages somewhere more permanent, especially if they have not been accessed for some time. By performing a persistent checkpoint at this point, the volatile memory occupied by the volatile checkpoint is released by the transfer of the data to disk.

The third occurs when it is necessary to make sure data does get copied to disk after a period of time, even if space is not required or an explicit request made. If this is not done, a situation could be imagined where modified data left on a machine weeks before could suddenly be lost by a catastrophic failure.

7.3 Construction of a distributed persistent checkpoint

Perhaps the most problematic aspect of implementing a distributed persistent checkpoint is whether it is necessary at all. In the three situations discussed in §7.2.3, in two instances checkpoints are performed in order to free memory and make sure long unused data is not lost. The only situation where a distributed checkpoint is strictly necessary is when it is

requested by an application. In such circumstances it is likely that a distributed volatile checkpoint would be sufficient. For completeness however, a mechanism is included which provides a distributed persistent checkpoint service.

We have already considered how to produce a localised persistent checkpoint. To extend this mechanism to a distributed system, it is only necessary to add some form of distribution. Final synchronisation has not been added and so it is impossible to say when exactly a distributed persistent checkpoint will have completed, only that it will have within a fixed period of time after local commitment. The necessary synchronisation has not been provided because of the additional complexity involved and the belief that volatile checkpoints make it unnecessary. However, such a system could be provided using the same principle described in §6.3.2.

These checkpoints are sufficient to guarantee data is present after a double machine failure but, unlike volatile checkpoints, do not provide rollback recovery. Any active processes are lost.

7.3.1 Distributing the checkpoint

To distribute a checkpoint it is necessary to flood the request to all the dependents. This kind of information is readily available for volatile checkpoints. However, whilst a volatile checkpoint only floods the checkpoint request to the active dependent domains, a persistent checkpoint must be flooded to all machines holding domains which were part of any volatile checkpoint since the last distributed persistent checkpoint was issued.

Constructing this information when a persistent checkpoint is requested is impossible. If such a checkpoint is initiated, by either a time based of space based event, then the checkpoint is not flooded to other machines and so the dependencies are not removed. However, the pages which form part of the dependency will be written locally and possibly reused. Consequently any dependency information associated with them will be lost. Therefore, to provide this information to persistent checkpoints, it is necessary to construct a separate dependency table.

Constructing the distributed persistent checkpoint table

When a volatile checkpoint becomes stable, the pages' DSM checkpoint chains (DPCPs) are examined to determine which machines the volatile checkpoint is dependent upon. This information is used to add to a reference count table of dependent machines.

Algorithm 7.3

Begin constructing distributed checkpoint table
For each page which has formed part of the current volatile checkpoint {
    Increase the reference count in the "persistent checkpoint table" for the machine indicated by the DPCP
}
End constructing distributed checkpoint table

When a persistent checkpoint is made the *persistent checkpoint table* is used to flood the checkpoint to all dependents. Once this is done, the table is cleared.

The use of a single table gives some problems. If no application ever performs a distributed persistent checkpoint, then this table will grow forever. If a checkpoint is eventually requested, then it will be flooded to every machine the local machine has ever interacted with. This is clearly not sensible, since most of the information that forms part of the dependency will have been committed to disk hours before.

This problem may be solved by considering the inclusion of the time related persistent commitment. If each machine commits its volatile data to disk every T *seconds*, then any dependency need only persist for that period of time. Rather than attempt to time dependencies, two *persistent checkpoint tables* can be used. After every time T, the oldest table is discarded and a new one created. Dependencies are always added to the newest table. When a distributed persistent checkpoint is made, the dependencies from both tables are used to flood it.

This scheme means a dependency will persist for at least T *seconds* and at most $2T$ *seconds*, depending on when it was inserted into the newest persistent checkpoint table. Therefore, every dependency will persist for at least the time between checkpoints but eventually be discarded when it can no longer be relevant.

7.3.2 The distributed persistent checkpoint algorithm

The algorithm to perform a distributed persistent checkpoint is given below:

Algorithm 7.4

Begin persistent checkpoint
If the node is marked "checkpointing", return immediately
Mark the node "checkpointing"
Collate the persistent checkpoint (Algorithm 7.1)
Whilst there are unwritten pages in the "persistent checkpoint list" {
    Construct and perform a large disk write operation
}
For each non-zero entry in either of the "persistent checkpoint tables" {
    Flood the persistent checkpoint to the relevant machine
}
Discard the "persistent checkpoint tables" and allocate a new one
Commit operations by writing next disk root block
Mark the node "active"
End persistent checkpoint

This algorithm performs the distributed checkpoint by constructing a local one and flooding the operation to all others. It does not bother to wait for replies to its operation, so there is no synchronisation between machines as to when the distributed checkpoint is stable. However, it should be possible to determine a time limit beyond which all data must have been committed.

7.3.3 Distributed persistent checkpoint - an example

Figure 7.1 illustrates the operation of a persistent checkpoint between four machines. Each machine holds its two persistent checkpoint tables indicating which machines it is dependent upon (a). The checkpoint is initiated from Machine#1 and spreads via its dependents (b) to all other machines (c). As each machine completes its persistent checkpoint, it deletes both persistent checkpoint tables (d) and is ready to begin accumulating a new persistent checkpoint.

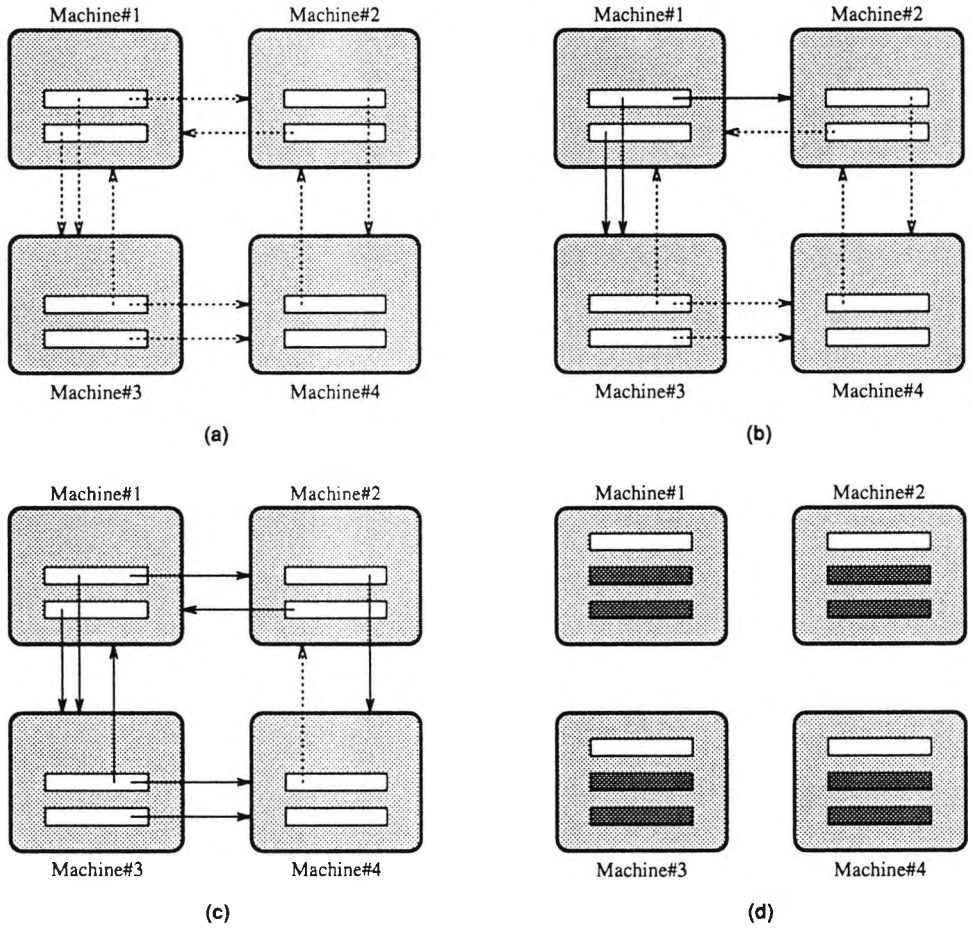


Figure 7.1: Persistent checkpoint between four machines

7.4 Summary

This chapter has presented a means of implementing persistent reliability by constructing persistent checkpoints from volatile ones. Persistent checkpoints provide a means of committing modified data to disk, which is necessary if machines are to be turned off or catastrophic failure tolerated without total system data loss.

Persistent checkpoints may be initiated in order to make space in volatile memory, to make sure old data is stored to disk, or at an application's insistence. The coordination of these checkpoints uses a time based approach to reduce the necessary machine interaction.

Persistent checkpoints are not intended to be used instead of volatile ones. Therefore, no rollback is provided (any active processes will be lost when a fault occurs) and commitment of data cannot be waited on. The exclusion of these features is not a restriction in the design (which could support them) but a philosophical decision. Volatile checkpoints provide the necessary level of robustness and are more efficient than persistent ones.

The next chapter details a software model using the checkpoint algorithms so far described. It examines their effect and efficiency, and also attempts to determine the impact on system performance they have.

Chapter 8

Modelling AMOS

A model of AMOS has been designed and written in C++ in order that the DSM and fault-tolerance systems can be studied. The model provides information in two important areas; firstly, correct working of the algorithms proposed for DSM and reliability; and secondly, what costs the operation of reliability places on the system as a whole. Extensive experiments have been carried out using the model and the results are presented here.

8.1 Design of the AMOS model

The purpose of the AMOS model is to allow a number of performance related experiments to be made to determine how efficient, in both time and resources, the reliability system is. A model, rather than a real implementation, was chosen for three reasons. Firstly, it enables various parameters to be altered which would be impossible in a real system (page size and cache size for example). Secondly, it provides a more controlled environment. Thirdly, since a real implementation does not exist, writing one would be an unreasonably large piece of work when a model would be more reliable, easier to debug, and more flexible.

The following details various aspects of the model.

8.1.1 Process creation

Process creation is handled simply. When a new process is created, it is spawned onto the next available processor (they are selected in order) from which it never migrates. This process executes its given function and then exits.

8.1.2 Object creation

Objects are created by the system by a single manager. The first object has a base address of zero and a length specified by the call. The next object has a base address of the next page after the start of the previous object; and so on. Objects may be deleted but the space is never reclaimed[1].

8.1.3 DSM system

The program models the DSM system completely, although it only uses the *write-invalidate policy* for data accesses and the *write-update policy* for lock accesses. Each message is allocated a transit time so requests for data do not happen instantaneously and processes are correctly scheduled whilst they are being processed, forwarded and fulfilled.

One simplification made to the DSM system in the model was in fulfilling *first time page requests*. Rather than implement the system described in §4.5.2, a single globally accessible table was used to record and obtain page information when the DSM entry was first created. This simplification does not affect the system in any relevant way.

8.1.4 Volatile reliability

Volatile reliability was modelled exactly as the algorithms describe. In doing this, the model had to implement a *virtual processor*. This allowed page faults to be generated at

[1]The programs being executed on the model were from the UNIX based parallel applications suite SPLASH. This meant it was more convenient to allocate one large object to include the complete data segment of the UNIX program. The use of a single object does not affect the efficiency of the reliability system under test.

the correct points by forcing *memory access events* (see §8.3.2) to be checked with the model's virtual-to-physical translation cache. This also allowed the detection of modified data to be made, an essential component of the reliability algorithms, as well as providing the flexibility to modify cache and page sizes for experimentation.

8.1.5 Persistent reliability

Explicit persistent reliability is ignored in the model, since its inclusion provides nothing which cannot be determined from experiments with volatile reliability—both persistent and volatile reliability systems are based upon the same principles and algorithms so demonstration of the functionality of one demonstrates the other. It is likely that persistent reliability will add a small execution time to the application in general. However, the scheme proposed in chapter 7 uses disk rarely and since disk transfers can take place in parallel with program execution, any effects would be negligible. This assumes that the resident set size for applications does not exceed the size of physical memory—which is true for all simulations performed.

8.1.6 Rollback

The model does not simulate the occurrence of rollback. This is because it was not possible to extract all the necessary state information from the threads used in the model (see the following section). However, this did not prevent internal consistency checks being used in order to show that rollback could be successfully achieved.

In order for rollback to be correctly achieved, the rollback algorithm must correctly delete all dependent data pages which have been modified since the previous checkpoint. With these removed, the active processes are returned to the state saved at the last checkpoint, a state known to be a correct point for process resumption.

In order to verify the rollback algorithm, the algorithm was initiated in various programs by deliberatly inserting the event in them at various points. The algorithm then proceeded to remove all the relevant data using the similar flood fill technique demonstrated for checkpoints. Once this was completed, it was possible for the simulation to halt all activity

and examine all the processors' memory systems. The simulator then checked that no modified data pages still existed within them and hence that the processes had been correctly returned to the previous checkpoint. This being the case, rollback was considered to have been successful.

In a true ARIUS system, the processes could then have been restarted. However, due to the noted lack of internal thread state this resumption was impossible. Despite the absence of this final step, it was felt that the above consistency checks were sufficient to determine that rollback was achieved correctly—the model was primarily designed to determine the impact of checkpointing in working systems (the norm).

The latency for completing rollback after a fault is important since a recovery time of several hours is unacceptable. However, since the rollback algorithm is essentially the same as the checkpointing algorithm (see §6.7), rollback takes a similar amount of time.

8.2 Implementation of the model

The AMOS model was written in C++ on a SUN Sparcstation using SUN's lightweight process library to provide parallel threads of execution. However, since the thread library was written in C, it was first necessary to encapsulate it in a C++ library [Wil92].

The model runs on a uniprocessor system. The threads provide the illusion of a multiprocessor but it was also necessary to provide some kind of scheduling amongst them in order to obtain *correct* multiprocessor execution.

Figure 8.1 illustrates the difference between two processes executing simultaneously on a true parallel system and on a uniprocessor system. It was considered necessary to provide the behaviour exhibited by the left example even in a uniprocessor model if any correct conclusions were to be drawn from the model. To provide this, the threads within the model were scheduled using *age event scheduling.*

Age event scheduling always schedules the youngest thread to run. The age of a thread is determined by its age at creation plus the accumulation of time executing and waiting for locks; the lower this is, the younger the thread is. A thread is allowed to execute

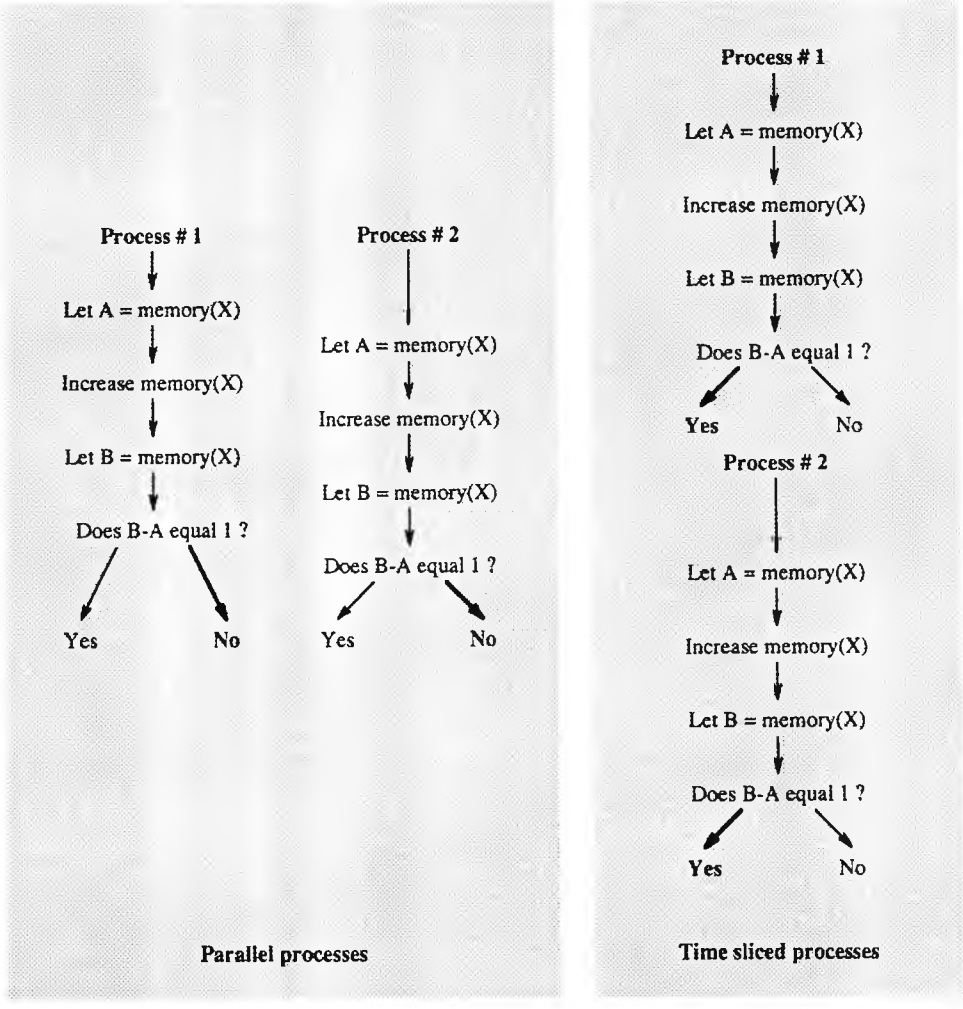


Figure 8.1: Parallel execution on a parallel system and on a uniprocessor system

until it issues an event, such as a memory access to a shared object. At this point its age is recomputed and if it is still the youngest, then it is allowed to continue. If not, the new youngest thread is scheduled instead. Using this system, correct parallel operation is achieved[2].

8.3 Running parallel application on the model

There are two general methods available for evaluating the cost of program execution; trace based and execution based. The original intention was to use trace based simulation but it was soon discovered to be inadequate for the AMOS simulation (see below). Instead, an execution based model was adopted. Whilst this causes massive performance degradation, it did enable accurate measurements and adjustments to be made with relative ease.

8.3.1 Abstract Execution (AE)

Abstract Execution [Lar90] (AE) is a technique developed for producing address trace data of executing programs with relatively little impact on the performance of the program itself. The original intention was to use this method to produce trace data to feed to the AMOS model in order to evaluate the overhead of checkpointing.

However, it was soon discovered that the AE system is not suitable for use in parallel programming environments where the program behaviour can be effected by time delays in the DSM system. For example, if a process is blocked waiting for a DSM page to arrive, then another process may run on the node. In a true shared memory system this is not the case since the blocked process halts the node until the page is supplied, the delay being many times shorter. This assumption means that parallel trace data produced by AE on a shared memory machine, the only one available to generate the trace data, cannot accurately reflect the behaviour of the model.

For this reason, AE was abandoned and execution based simulation was used instead.

[2]In actual fact, the system employed is slightly more complex since two "processes" on the same physical processor do not execute in parallel but one after another in round-robin fashion.

8.3.2 Modified GNU C-compiler 2.1 (GCC)

Execution based simulation may take two forms; instruction level simulation or event level simulation. In the first, the model interprets the program, instruction by instruction, in order to obtain an accurate simulation of the program's execution. This approach is very time consuming and involves the simulation of the entire processor rather than just the operating system functionality.

The second approach, which was adopted, is event simulation where individual entities are executed until they perform some event which interacts with another (this is achieved using age event scheduling as described in §8.2). However, in order to provide the necessary event information, it was necessary to modify a C-compiler, GCC 2.1.

Compiler modifications

The compiler was modified to issue subroutine calls to special event handlers within the model under two circumstances, *load instructions from memory* and *store instructions to memory*. These are the only ways in which different threads within a parallel program can interact, more complex operations, such as locks and barriers, are implemented using some form of these.

Example SPARC code from modified C-compiler

An example piece of code produced by the compiler is shown below:

```
_printtree:
        !#PROLOGUE# 0
        save %sp,-144,%sp
        !#PROLOGUE# 1
        add %fp,68,%g3
        add %g4,3,%g4
        call __sim_store,0
        nop
        st %i0,[%g3]                ! st %i0,[%fp+68]
        add %fp,68,%g3
        add %g4,4,%g4
```

```
call __sim_load,0
nop
ld [%g3],%l0                    ! ld [%fp+68],%l0
```

Before a load is issued in the code (last line), the address from which it is to be made is generated and placed in register *%g3* and the routine `_sim_load` is called. After this call the actual load instruction is issued. Similarly, before a store is issued (9th line), the address to which it is made is placed in register *%g3* and the routine `_sim_store` is called. After this call the actual store is issued. It is important to note the use of register *%g4*. This register is used to count instructions executed, excluding the extra ones inserted to generate the events. This provides the *event's age* to enable correct scheduling to be performed[3].

Each routine call does not actually perform the operation but, instead, makes the model perform the necessary operations to enable the instruction to take place. In fact, all data is physically shared in the model and no data movement need take place. However, the model imposes the necessary costs on each event so that, for example, when a read is made to a DSM page which is not present, the process is blocked and another is scheduled until the page is delivered sometime later.

Using the modified C-compiler

To use the modified C-compiler, a C program is compiled to object form using the modified compiler. It is necessary to include a special header file, `amos-model.h`, in each code file. This redefines various conventional C functions, `main()` for example, and provides the necessary prototypes for creating processes as well as locking and barrier primitives.

The resulting object code is then linked with similarly compiled libraries. In addition `libmodel.a` is appended to the program. This is a dynamically linked library containing the AMOS model itself. Then the program may then be executed in the usual way. In doing so, the AMOS model is first called to allow it to initialise itself. It then calls back into the compiled C program.

[3]This assumes that each instruction takes only one cycle to execute, something which is not always true. However, in a RISC processor such as the SPARC, it is accurate enough for the simulation's purposes.

The provision of the AMOS model as a dynamic library enables it to be changed quickly without recompiling any of the test programs. This reduces the time required to experiment with modifications and removes the possibility of errors, by making it impossible to execute a test program with an old version of the model.

8.4 Experimentation suite

All experiments with the model were made using the suite of applications collectively known as SPLASH [SWG92]. These programs were developed at Stanford University USA, to provide parallel programmers and machine designers with a *standard* set of applications with which to experiment and compare results. They proved well suited to experiments with the AMOS model since they not only demonstrated different patterns of data sharing but also provided the ability to vary the number of parallel threads of execution within them.

8.4.1 Incompatibilities between Arius and Splash

The SPLASH programs were all designed to support a particular set of parallel operations [LO87]. Fortunately this model closely resembles the one provided within the ARIUS environment. Creation of processes using a *spawn* semantic rather than *fork* is particularly important. Spawn creates a new process, sharing the same code and shared data spaces but with a new stack and duplicated private data, which starts executing from a supplied function and terminates when it returns from this function. This is the model used in ARIUS. Fork requires the complete duplication of data and stack for the new process and is impossible to emulate in ARIUS.

There was only one incompatibility between ARIUS's model and SPLASH's. In SPLASH it is assumed that the private data space will be duplicated in the new process; in ARIUS it is assumed to be shared also. Fortunately, the applications make little use of this facility. Usually, a loop is used to *spawn* a number of processes to perform the work. Each process uses the value of the loop counter as its ID. Because this is shared in ARIUS it was necessary

to pass this ID as an argument to the new process. This, once understood, was easy to accomplish.

8.4.2 Purposes of the experiments

The experiments' first task was to demonstrate that the DSM system and the volatile reliability protocols worked correctly. Once this was shown, various modifications were made to the model to determine how the efficiency was affected by different page sizes, VM address translation cache sizes, frequency of checkpoint operations, etc. The results of these experiments are presented in the following sections.

8.4.3 Experiments, configuration and results

The majority of the initial experiments were conducted using the program `Mp3d` from the Splash suite. This program provides flexible control of the amount of parallelism as well as the size of the problem. One difficulty with the Splash suite is the length of time the applications take to run. On a real system the times would be considered reasonable but on a simulator, with a much increased execution time, they made it difficult to experiment.

Final results were obtained using three of the Splash applications; `Mp3d`, `Water` and `Barnes-Hut`.

Mp3d solves problems in rarified fluid flow simulation. Rarified flow problems are of interest to aerospace researchers who study the forces exerted on space vehicles as they pass through the upper atmosphere at hypersonic speeds. Such problems also arise in integrated circuit manufacturing simulation and other situations involving flow at extremely low density.

Water is an N-body molecular dynamics application to evaluate forces and potentials in a system of water molecules in the liquid state.

Barnes-Hut simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which every body is modelled as a point mass and exerts forces on all other bodies in the system.

(extracts from "SPLASH: Stanford Parallel Applications for Shared-Memory" [SWG92])

Configuration

A typical configuration script is shown below:

```
10      ; Number of clusters
1       ; Number of processors per cluster
1024    ; Page size (bytes)
32      ; PTE data cache size
80      ; Access table size
50      ; Miss fault time (cycles)
50      ; Write fault time (cycles)
30      ; Copy-on-write fault time (cycles) excluding copy time
100     ; Dsm message delivery time
40      ; Dsm buffer size
1       ; Do checkpoints (1 = yes, 0 = no)
0       ; Checkpoint period (0 = none)
```

The values shown were chosen to represent reasonable operating conditions for a real machine; times are in clock cycles and sizes are in bytes.

Results

Each simulation terminates by producing a table of results, for example:

```
Simulator run started Wed May 27 09:35:13 1992

Cycles                    :      2488839
Loads                     :       452026
Stores                    :        95990
Tlb hits                  :       446613
Miss faults               :         6132
Write faults              :         3553
Copy-on-write faults      :          465
Accessed page additions   :         1335
Dsm messages              :        40187
Dsm owner request         :         3440
```

```
Dsm copy request        :        3741
Dsm inval request       :        3200
Dsm copies sent         :        4554
Max coremap size        :        1094
Checkpointed pages      :        1177
Checkpoints             :          31

Simulator run finished Wed May 27 09:46:51 1992
```

These results indicate various measured aspects of the simulation. For example, *Cycles* is the total time for the program to execute, *Loads* is the sum of loads from memory executed by all processes, and *Max coremap size* is the maximum number of pages active at any one time.

Each simulation generates fourteen individual results and each simulation is run with one to ten processes in a number of configurations. This produces a multitude of information, of varying utility. Full sets of results are given in Appendix B. Here are presented the most relevant. These are:

- **Cycles**

  The total execution time of the program in processor clock cycles,

- **Max coremap size**

  The total core used by all processes and processors,

- **Checkpointed pages**

  The number of pages checkpointed by the application,

- **Copy-on-write faults**

  The total number of copy-on-write faults which occur due to checkpointing,

- **Access page additions**

  Total number of new page accesses noted by the VM subsystem,

- **Dsm messages**

  Total number of DSM messages sent between nodes.

8.5 Experiments — Overview

Before experiments could proceed, it was first necessary to validate the model. Validations was performed in two ways:

1. Liberal use of assertions within the model to make sure DSM and checkpoint integrity is always maintained, and

2. Comparison of results from single process and multiprocess simulations.

The first of these was simple to provide and has the advantage of checking the model at all stages of operation rather than in just a few engineered examples. The second set of validations was done by executing various Splash applications with the same configuration but with varying degrees of parallelism. Unfortunately, when this was done the answers in some of the applications were almost, but not quite identical. Examination of the relevant applications exposed that random numbers were used in various calculations. When the random number generator was used by a single process, one answer would result. However, when used by two or more processes, the random numbers would be extracted in a different order and so give a different answer. No solution to this problem was found. If a random number generator were used for each process the answer would still be different from the single process case.

8.6 Experiments — Mp3d

`Mp3d` was the initial test program and so was used for the first experiments. It allowed small problems to be analysed in a reasonable time (an average of twenty minutes per run, a set of runs consisting of about fifty tests) whilst providing flexible control of its parallelism.

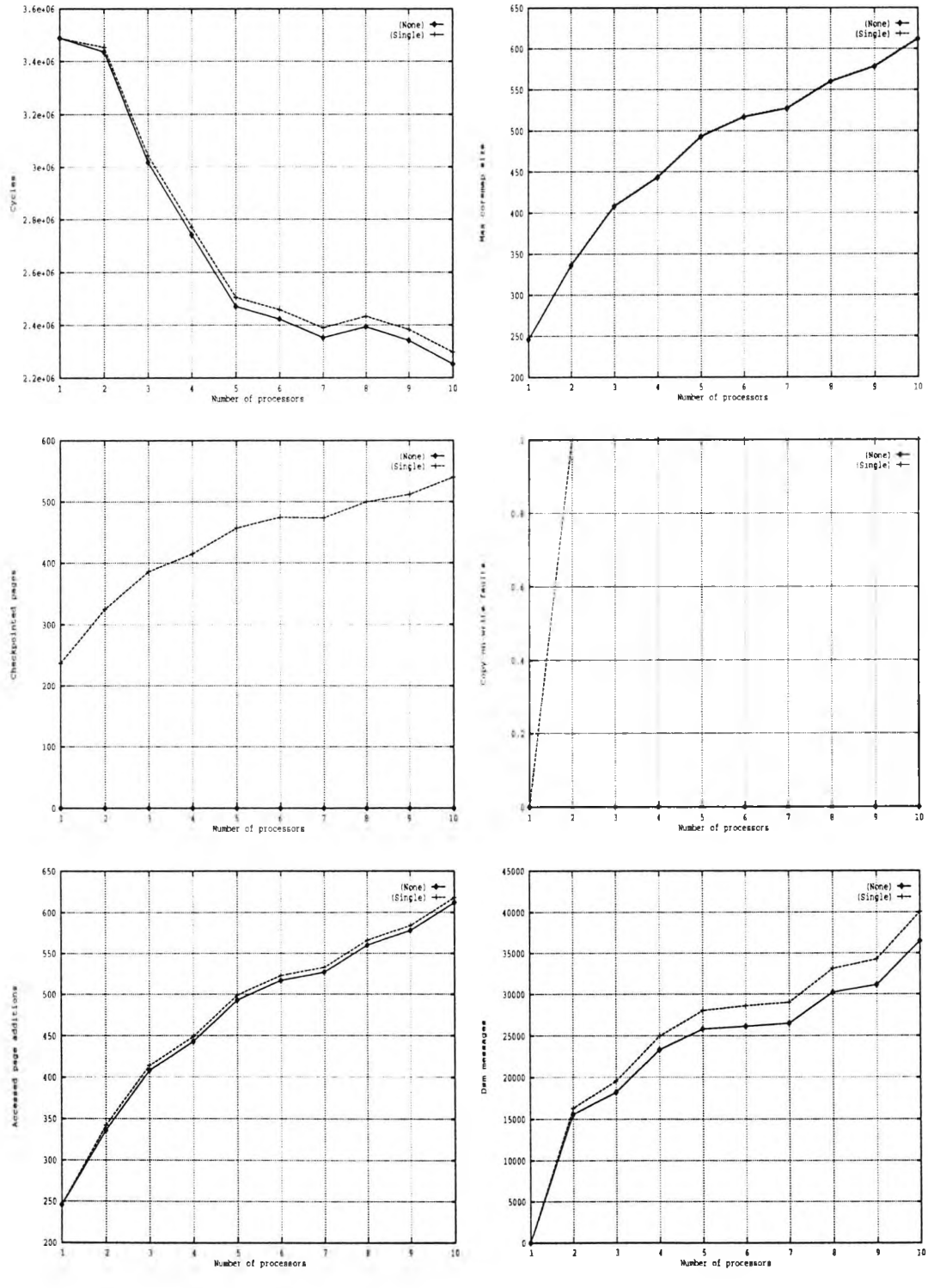


Figure 8.2: Control experiments

8.6.1 Control

Initially, two experiments were carried out to provide a control for all future tests. The results of these are shown in figure 8.2. One experiment was conducted without any checkpointing enabled, so providing a "best" performance measurement. The other was performed with checkpointing enabled but only activated when a process exits, the point at which only persistent or shared data need be checkpointed. This provides the "best" checkpointing enabled performance.

8.6.2 Size of access table

The control experiments examine two non-useful aspects of reliability, one where there is none and the other where reliability is only established on termination. A more useful reliability system would operate checkpoints at periodic intervals so that rollback will only lose either a set amount of data or a set time of computations. From the users point of view, the time aspect is more important whilst from a systems point of view the amount of modified data lost, and so the amount of space required, is more important. AMOS supports a time and space related checkpointing system. In `mp3d` however a time based checkpoint system is not useful since sufficient data is modified to prevent the time period from elapsing.

The results presented in figures 8.3 and 8.4 show how the size of the access table, that is the number of pages accessed by a process in a given checkpoint, affects the resources used by the application.

As expected, the smaller the access table, the longer the program takes to execute, due to the increased number of checkpoints being made. Note that with ten processes the different access table sizes make little difference to performance. This occurs as the working set size of each process approaches the access table size.

The behaviour of the coremap size is also interesting. As the parallelism in the program increases, the total core used rises, as expected, with the increased number of DSM copies. However, after a certain limit, the core size begins to fall and then begins to rise again in parallel with the control results (this can also be expected from the checkpoint access

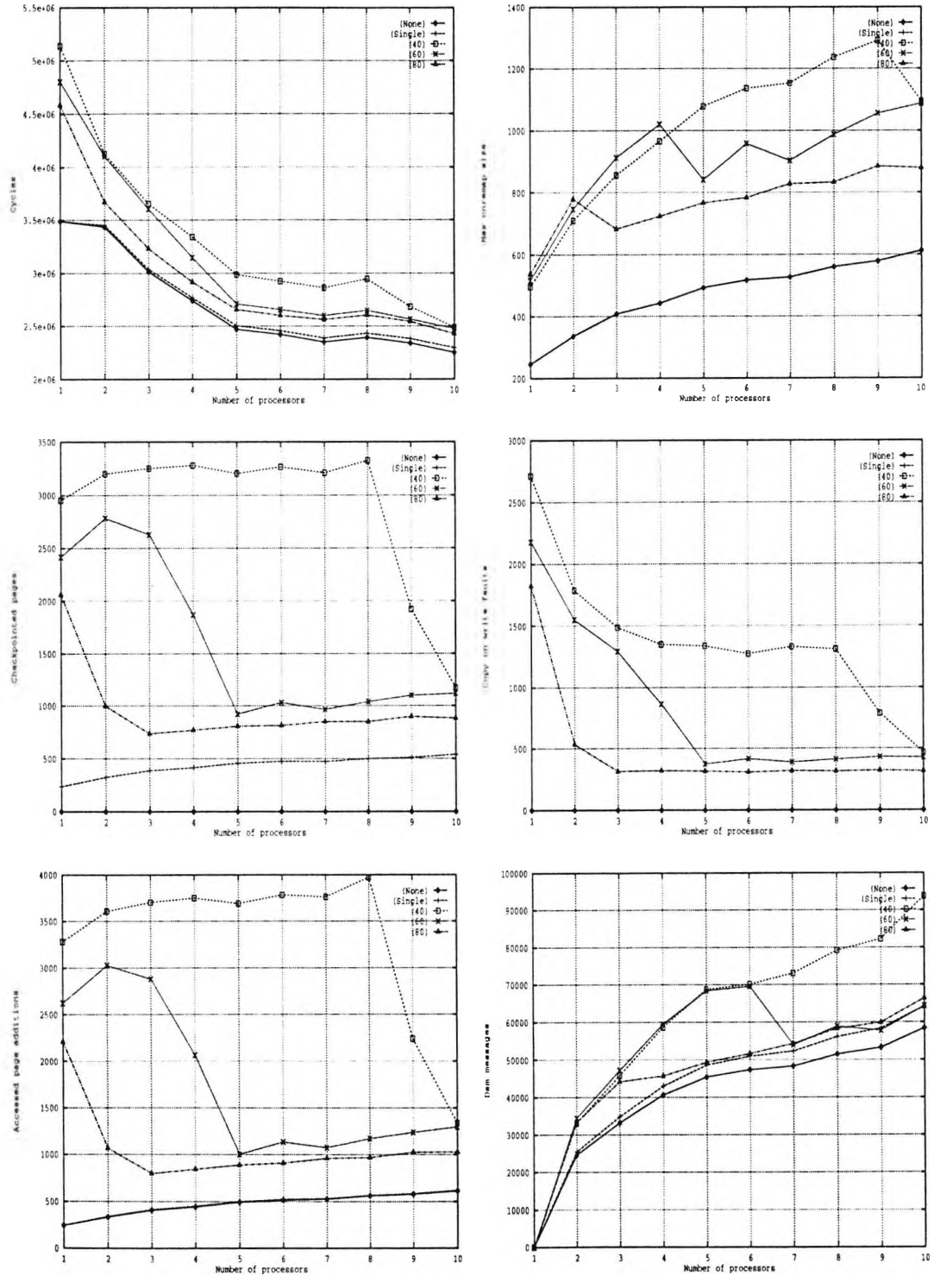


Figure 8.3: Access table experiments – 160 molecules

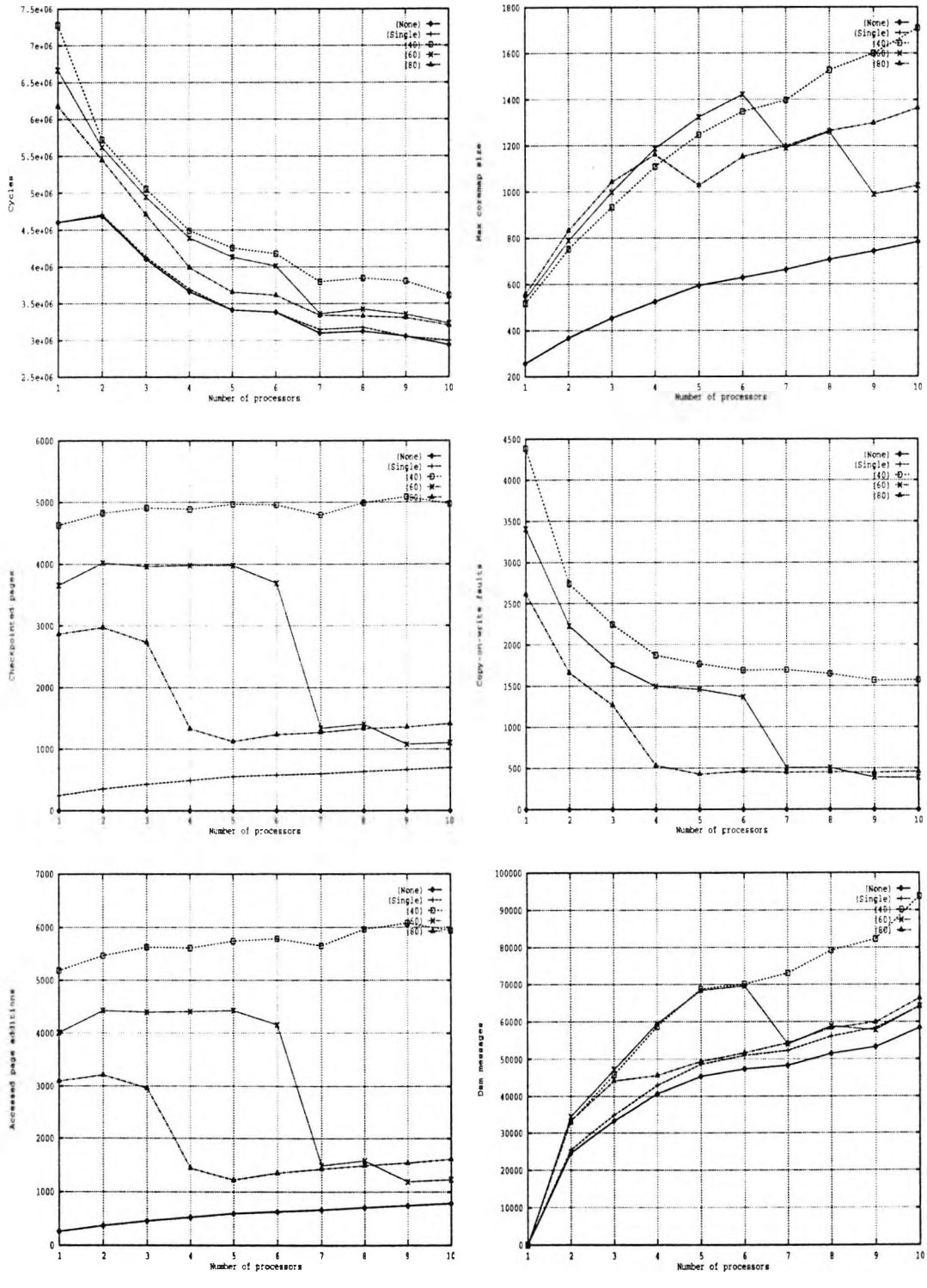


Figure 8.4: Access table experiments – 320 molecules

table with a size of forty, although it is not shown on the graph). Why is this?

A clue lies with comparing this result with that for checkpoint pages. Here the number of pages checkpointed rises then suddenly falls and then begins to rise in parallel with control. What is happening is this. With a fixed sized problem, as the number of processes increases the amount of the problem associated with each decreases. At different points, depending on the size of the access table, this subproblem size approaches the access table size so resulting in fewer checkpoints. Fewer checkpoints mean less core used to store them and fewer pages processed. This corresponds to the reduction in core used and fewer checkpointed pages. This interpretation is further backed up by a similar behaviour in the accessed page additions.

The results for copy-on-write faults were unexpected. It had been assumed that these would increase steadily in line with the number of pages checkpointed. In fact although they do indeed follow the sudden decreases in checkpointed pages, they tend to decrease regardless of this influence. This can be explained by examining the behaviour of the system, and particularly the size of the access table, more closely.

The problem is configured so there is little false sharing[4]. In general therefore a page is either accessed by only one processor or intentionally shared by all. Once the working set of a processor's data fits within the access table, the "accessed page additions" should fall dramatically since it no longer continually overflows the access table (which result in checkpoints operations), so reducing the number of copy-on-write faults possible. As the parallelism increases, the chance of the requested page being present on the relevant node decreases (assuming the page is shared equally). Therefore, although a node may hold a checkpoint copy it is less likely to be up-to-date. When a modification to a page is requested, it is more usually the case that the page is requested from another machine, using the DSM, than a copy-on-write fault being generated to produce a copy of the local checkpointed version. These two trends account for the behaviour seen with copy-on-write faults.

The graphs of "DSM messages" indicate a general increase in network traffic as the par-

[4]Modification to the source were necessary to make this happen.

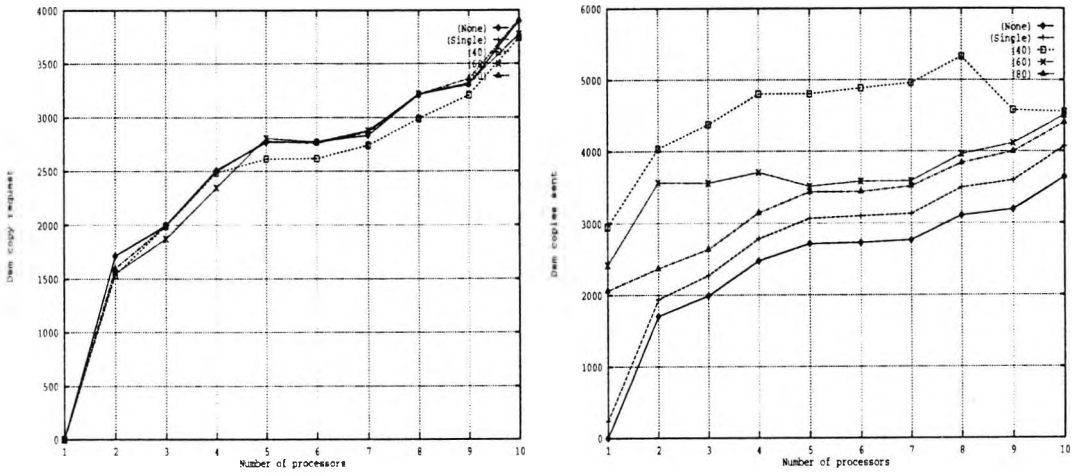


Figure 8.5: DSM copies traffic – 160 molecules

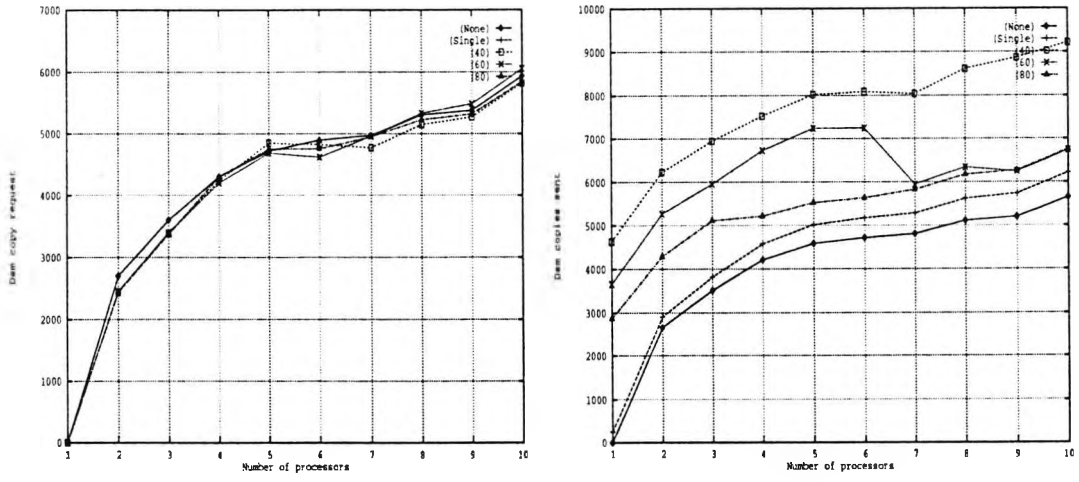


Figure 8.6: DSM copies traffic – 320 molecules

allelism increases. However, these increases are similarly affected by the changes in access table size as shown in the data for checkpointed pages. By analysing two further graphs (figures 8.5 and 8.6), of DSM copies sent and DSM copies requested, we can determine that these reductions are due to the checkpoints distributing pages to nodes which will later use them. This avoids the necessary DSM traffic to request the copies which would otherwise take place.

Conclusions

The access table size in relation to the subproblem size has been demonstrated to be very important to the efficiency of the application's execution. If the access table is too small, then excessive numbers of checkpoints are performed. Making the table infinitely large and checkpointing only on process termination is by far the most efficient solution but could result in the loss of too much data and subsequently affect too many other dependent processes. The most appropriate solution would be based on a time related system combined with an access table scheme. This would force a checkpoint to occur when either the time limit is exceeded or too much modified data is accessed or generated.

8.6.3 Page size and constant Access Table capacity

Variations in the page size affect various aspects of the model. The DSM shares data using a page size granularity. Increasing this size increases the risk of false sharing, where two independent data structures lie in the same page and therefore cannot be written to simultaneously by two distinct processors. Larger pages also produce larger grain checkpoints. This increase the likelihood of false checkpointing, where data which has not been modified lies in the same page as data which has been changed. However, by decreasing the page size, the number of page faults to manage the page translation entry (PTE) cache is increased. Conversely, the quantity of data rendered inaccessible by the invalidation of a PTE entry (by the DSM system) is reduced.

Changing the page size does not obviously improve or degrade the system. Experiments were therefore carried out in order to determine the "correct" page size for a checkpointing

DSM system. These used an access table size of 80 Kbytes and a PTE cache size of 32 Kbytes. The configuration of the model operates in page size units so the relevant parameters were adjusted in different simulations to maintain these data sizes.

The results presented in figures 8.7 and 8.8 show how the page size affects the resources used by the application.

With the smallest page size, the application executes in the fewest cycles. This may appear counter-intuitive; it would be expected that smaller pages would increase the number of page-faults and the number of DSM requests made so increasing the execution time. This is not the case. Firstly, by examining the graphs of DSM messages, it can be seen that a smaller page size reduces the number of messages. Secondly, by examining two further graphs of PTE miss faults and PTE write faults (figures 8.9 and 8.10) it is observed that the number of PTE faults is also reduced with small pages.

This behaviour is the result of false sharing. When independent data resides in the same sharable unit (in this case a page), it is possible that two processors will attempt to write to the page at once. This results in additional faults and DSM transfers, as the page is moved back and forth between the two processors, yet at no time does either access of modify the others data. This behaviour is a property of the DSM system and not of the application algorithm. There are two common ways to avoid this problem; either provide a more loose form of DSM coherency or arrange data so this problem does not occur. It is still interesting to note that, despite false sharing, the general trend is for execution speed to increase as the number of processors increased.

Copy-on-write faults tend to decrease as parallelism increases for reasons already given in §8.6.2. However, the effect of page size on the numbers of faults as parallelism increases is quite interesting. With a single processor, the number of copy-on-write faults simply indicates the distribution of modified data. As the number of processors increases to four, the copy-on-write faults fall into two groups, small pages (512 and 1024 bytes) and large pages (2048 and 4096 bytes), with small pages producing a third as many faults. As the number of processors increases towards ten, the large pages begin to approach the small pages in number of faults (4096 byte pages can be expected to converge also), even surpassing them in some cases. How can this distinct two phase behaviour be explained?

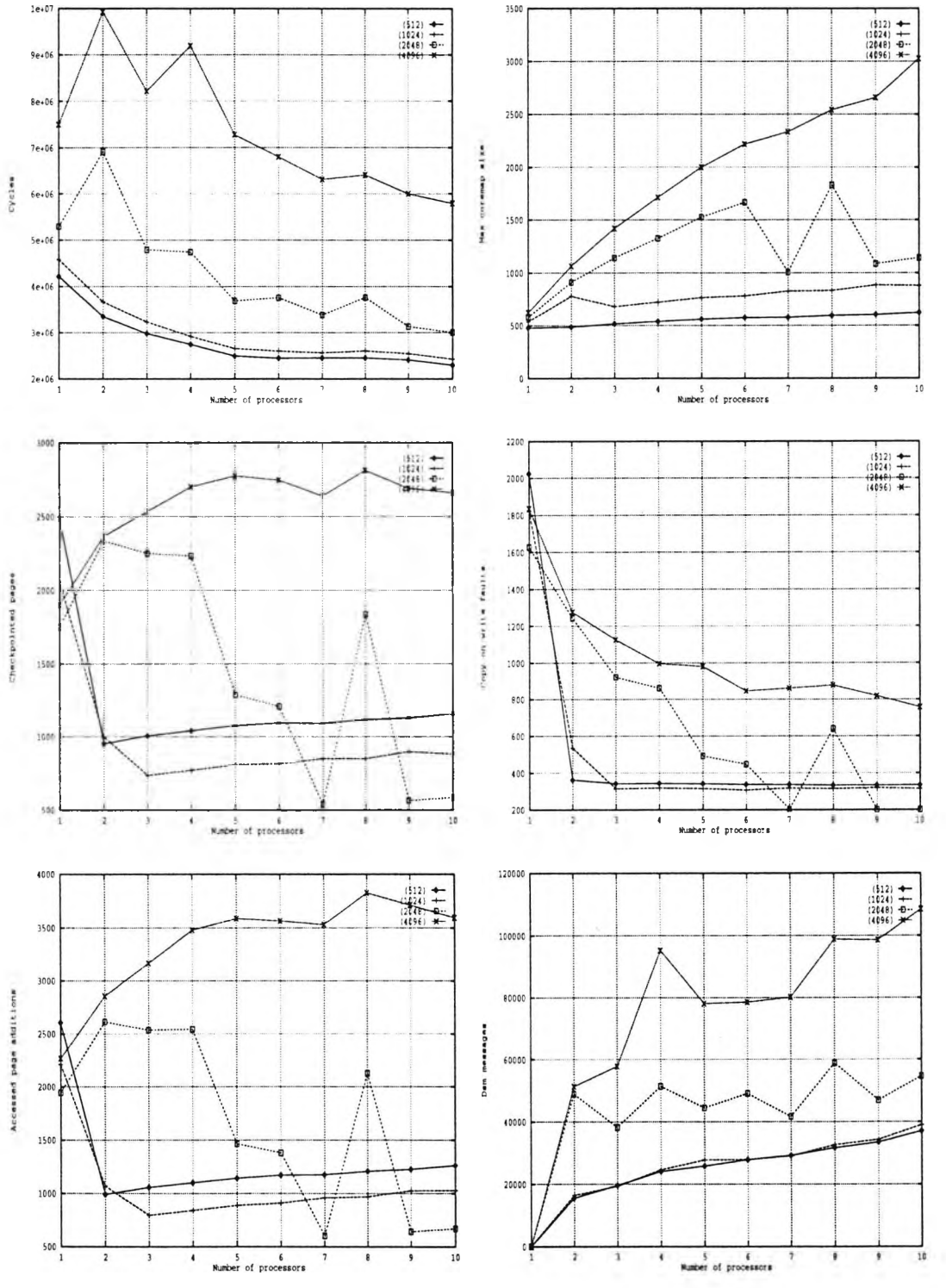


Figure 8.7: Page size experiments – 160 molecules

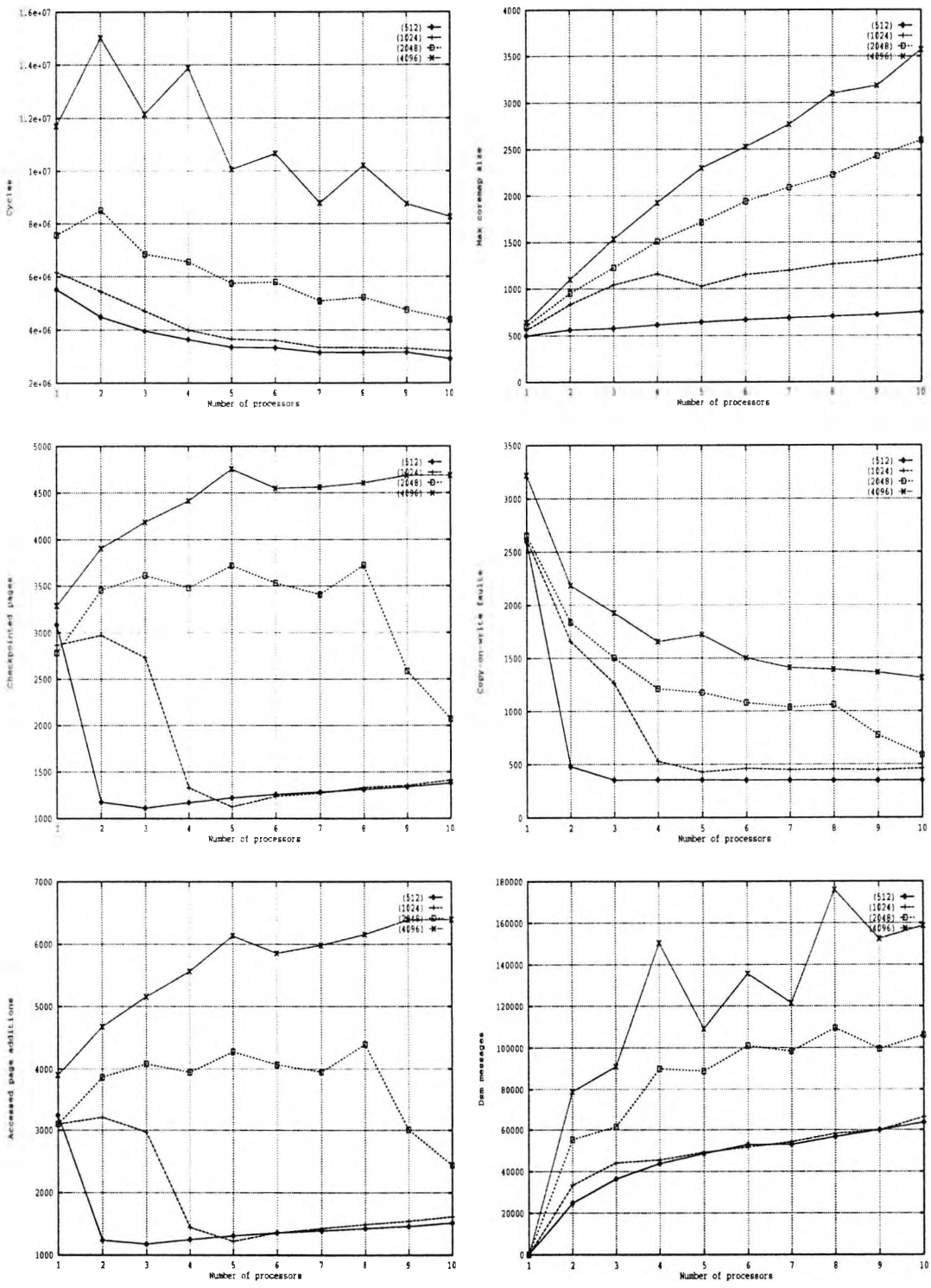


Figure 8.8: Page size experiments – 320 molecules

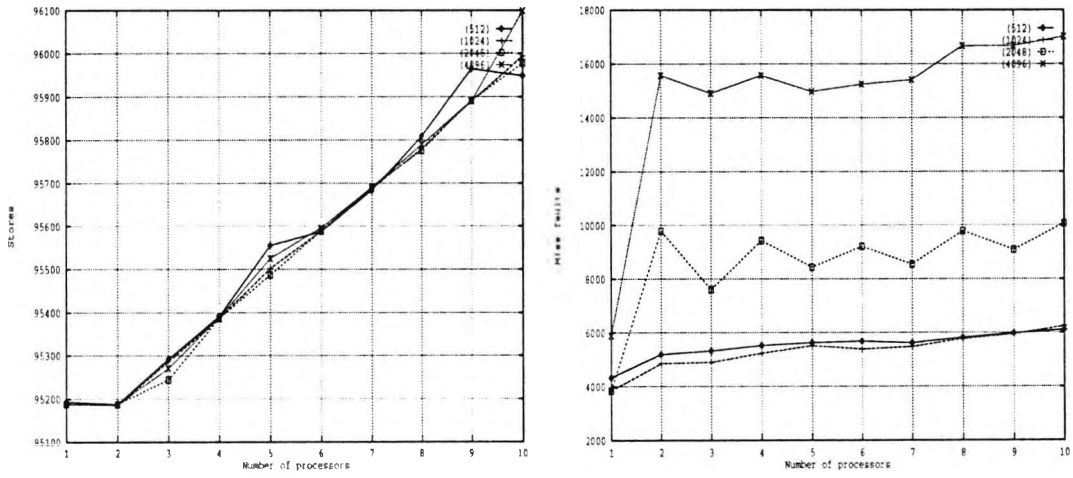


Figure 8.9: PTE faults – 160 molecules

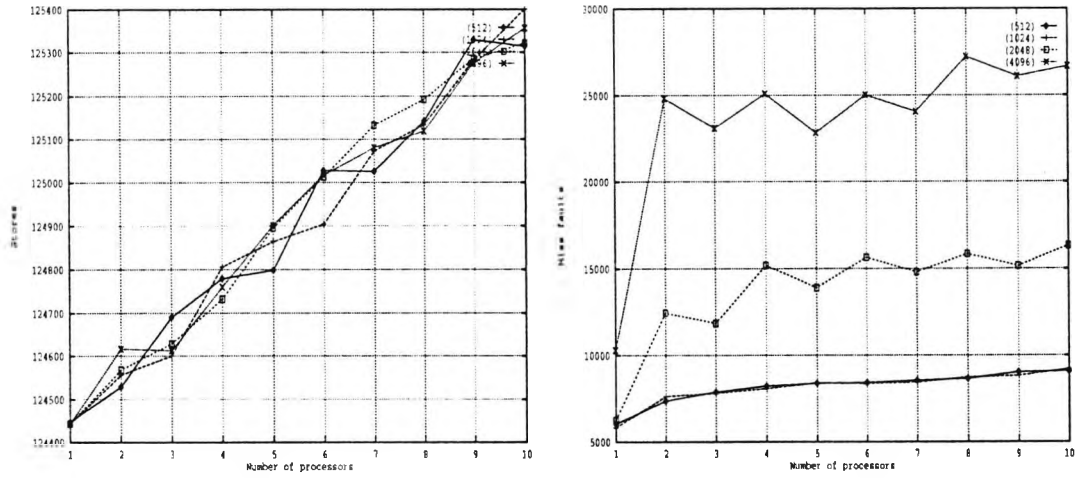


Figure 8.10: PTE faults – 320 molecules

Examination of "accessed page additions" and "Dsm messages" helps to clarify what is happening. As the number of processors increases, the quantity of data each processor accesses decreases. For `Mp3d` the working set of data for each of these processors is small but inter-page locality is not good (it may access page N but is unlikely to access page N+1). A large access table size is therefore required to accommodate the data set. However, since the access table is configured to allow the same amount of data to be referenced in all experiments, best results should be obtained by using a large access table and small page size.

In §8.6.2, we have seen that an access table able to accommodate the processor's working data set results in fewer checkpoints. With small page sizes there is little false sharing and a larger access table. Therefore, the working set is more easily accommodated so resulting in fewer checkpoints. With large pages there is more false sharing and a larger effective working set size of data. Many more processors are required before the data is divided sufficiently for the working set to fit within the access table. However, once this does happen, the page additions reduce below those for small pages since the access table is smaller and any overflow produces fewer copy-on-writable pages.

The number of copy-on-writes is therefore directly related to the "accessed page additions" and number of processors in the experiment. A decrease in accessed pages indicates a decrease in the number of checkpoints occurring and so a decrease in the potential number of copy-on-write faults. An increase in the number of processors makes it more likely data is found on another node rather than locally and so also decreases the number of copy-on-write faults.

Small page sizes

Experiments so far seem to indicate that small pages are best. However, results presented in figures 8.11 and 8.12 suggest there is a limit. Physical limitations dictate that the PTE cache is of finite size. Consequently, a limit of thirty-two entries is used in the experiments. Therefore, by reducing the page size beyond a certain size results in many more page faults and is not compensated by either improved false sharing, smaller checkpoints or fewer copy-on-write faults. Overall performance is degraded.

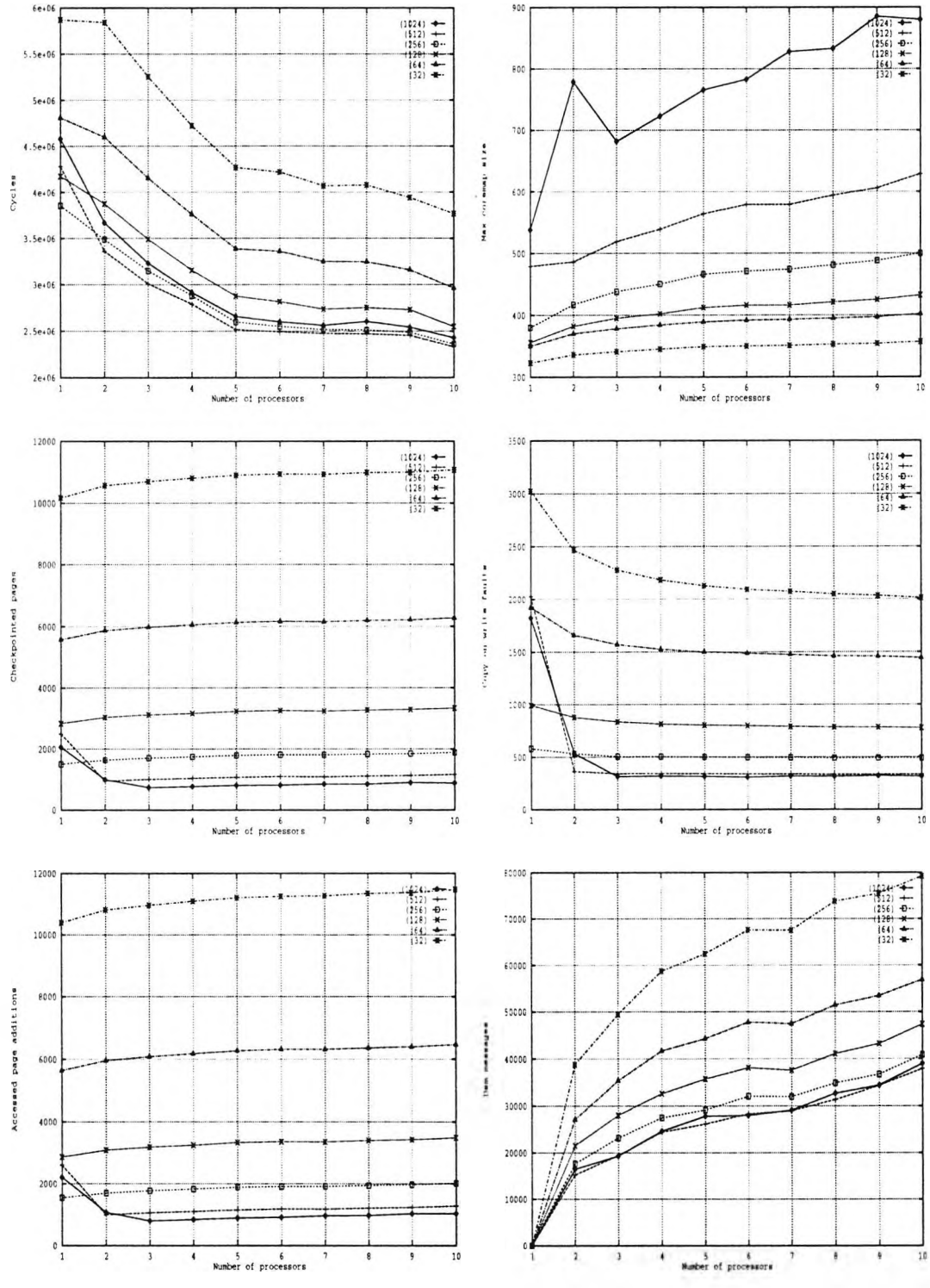


Figure 8.11: Small page size experiments – 160 molecules

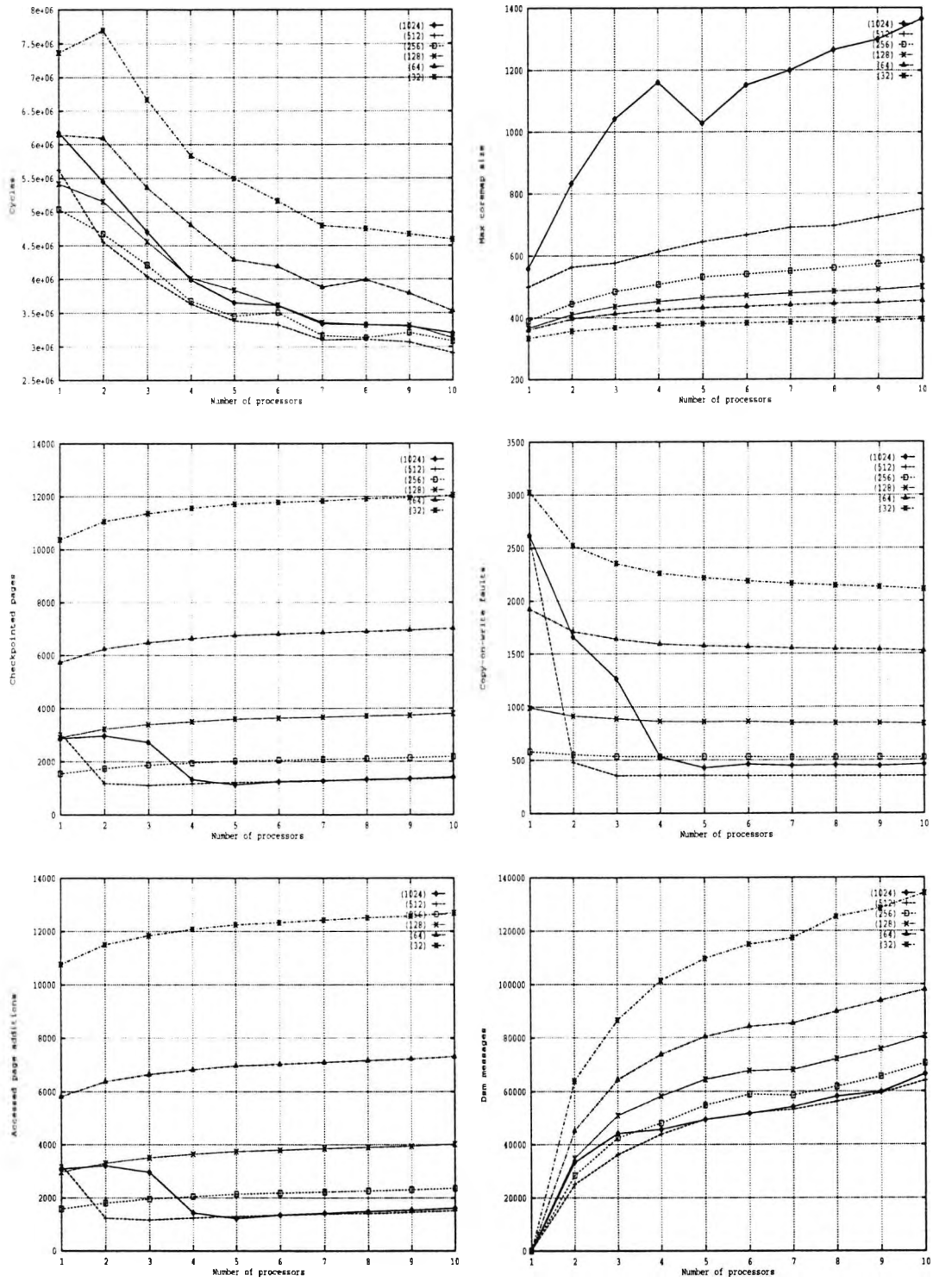


Figure 8.12: Small page size experiments – 320 molecules

8.6.4 Conclusions

Larger page sizes increase the execution time of the applications by increasing the size of checkpoints, increasing the number of copy-on-write faults, increasing the false sharing, and increasing the number of DSM messages. Also, large DSM page transfers and large copy-on-write faults take longer to execute. All this suggests that small page sizes are best.

However, reducing the page size beyond a certain point is not advantageous. Once there is good inter-page locality (if page N is accessed then so is page N+1), smaller pages merely produce more DSM requests and page faults whilst improving neither the size of checkpoints, memory usage or false sharing. The experiments carried out here suggest a page size of 512 bytes to be optimal.

8.7 Experiments — Water

`Water` was the second test program to be used with the simulator. Like `mp3d`, it allows both the problem size and parallelism to be changed easily. However, unlike `mp3d`, large problem sizes proved extremely difficult for experimentation since the work increased $O(n^2)$. A single simulation with 128 molecules took a minimum of 24 hours to execute; a full simulation set of fifty runs would therefore take two months to complete! Much smaller simulations were therefore undertaken with 16 and 32 molecules respectively.

8.7.1 Size of access table

Experiments were carried out using no checkpoints, terminating checkpoints and an access table limited checkpoints. In all cases (as shown in figure 8.13 and 8.14) there was very little difference in performance. How can this be so?

Examination of the graphs and Splash documentation indicate that the subproblem size is very small and will easily fit within the access table sizes used in the experiments. Additionally, unlike `mp3d`, no work is undertaken once the parallel processes have termi-

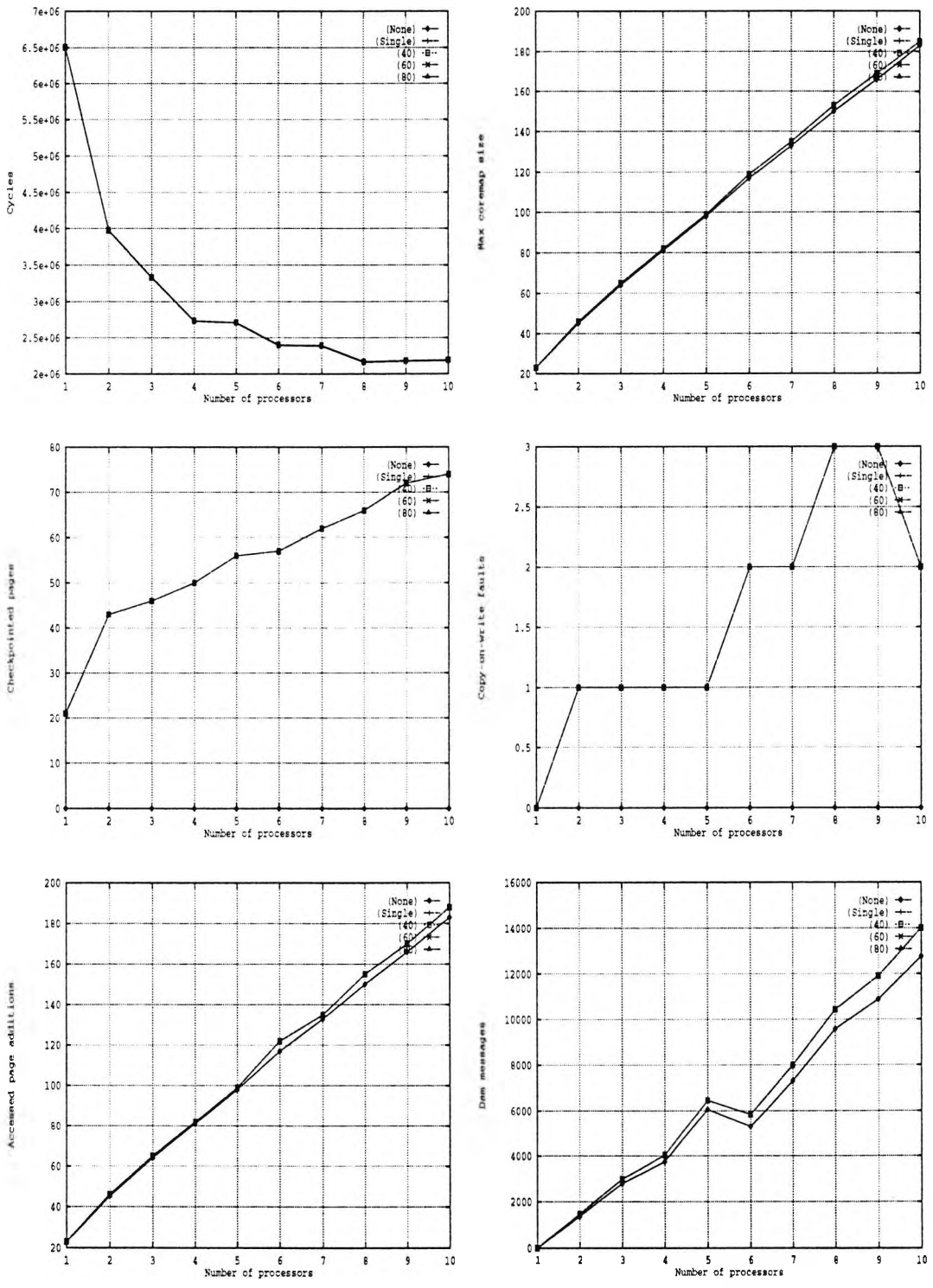


Figure 8.13: Access table experiments – 16 molecules

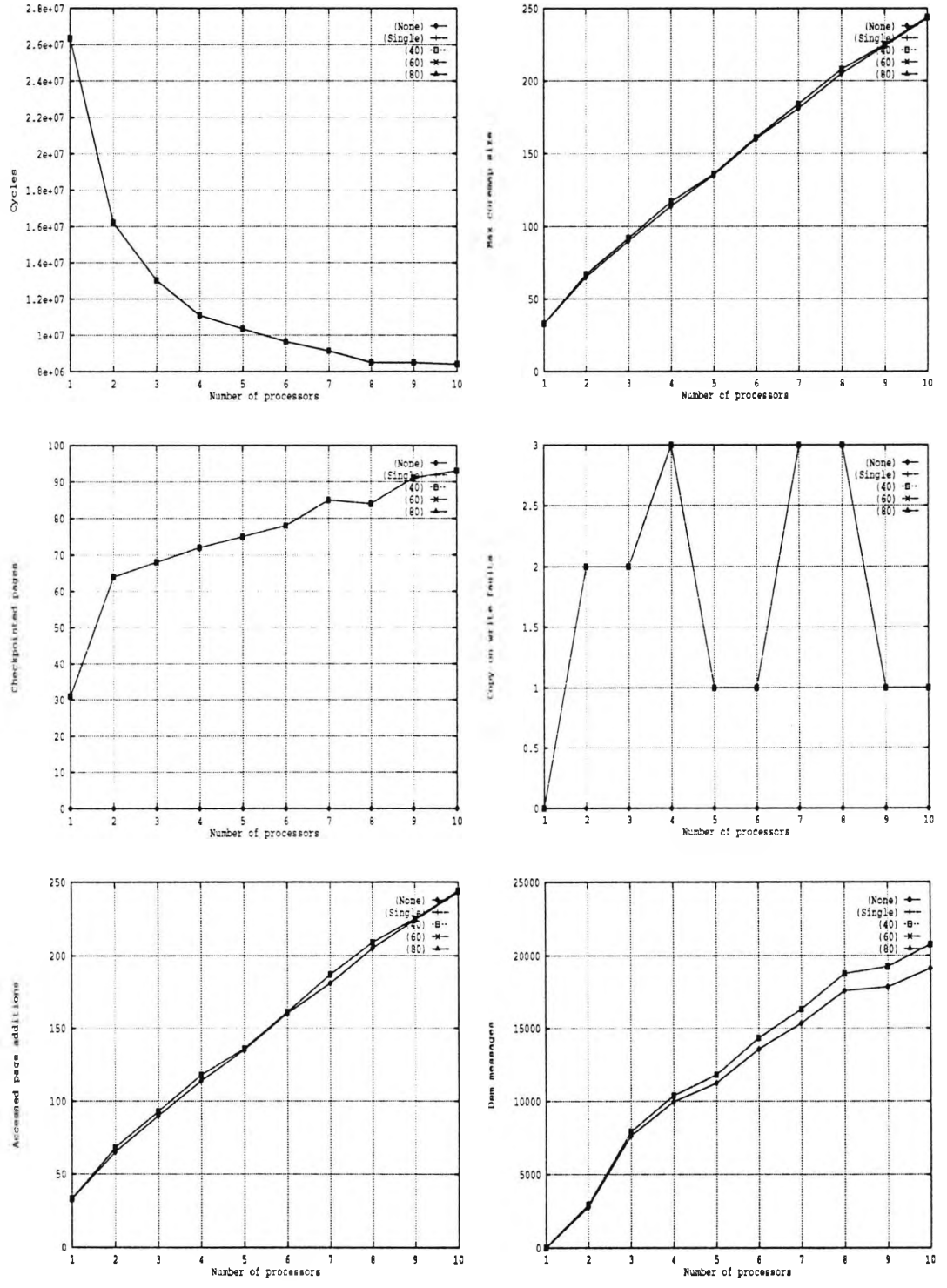


Figure 8.14: Access table experiments – 32 molecules

nated, so few copy-on-write faults are generated. The combination of these two elements results in an application which easily fits within the access table size and so generates little checkpointing overhead.

Conclusions

Although these results may seem on first examination to prove very little, they do in fact graphically demonstrate that the checkpoint system imposes little overhead as long as the access table size is large enough to accommodate the subproblem size. Unfortunately, when this is the case a problem could execute for a large period of time without any checkpoints ever being made. In order to overcome this problem, it is necessary to impose a time based checkpoint trigger as well as an access table based one (as mentioned in §8.6.2).

8.7.2 Time based

Experiments were carried out using no checkpoints, terminating checkpoints, and three different periods between checkpoints (50000, 100000, and 200000 cycles). The results are given in figures 8.15 and 8.16.

As expected, the more frequent the checkpoints the longer the execution of the program. In fact, halving the checkpoint period adds an approximately fixed period to the execution time regardless of the parallelism employed. This constant factor makes the percentage performance loss increase as the parallelism increases. The loss is accounted for by additional page faults and copy-on-write faults generated by the checkpoints. There is also an increase in DSM traffic due to the exporting of checkpoints. This has little effect on either ownership or page copy requests, indicating that the exported pages are generally of little use to the receiving nodes.

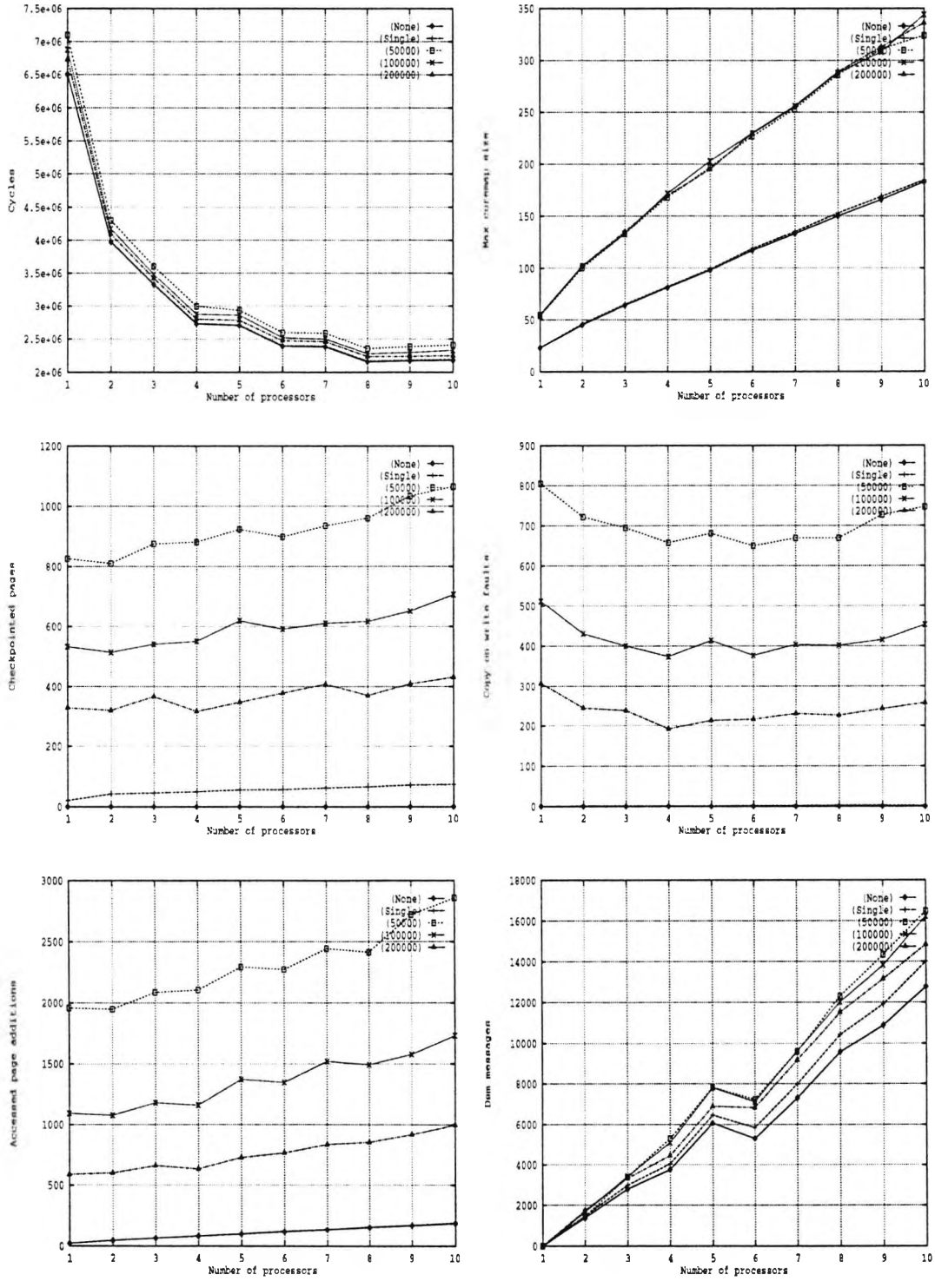


Figure 8.15: Time based experiments – 16 molecules

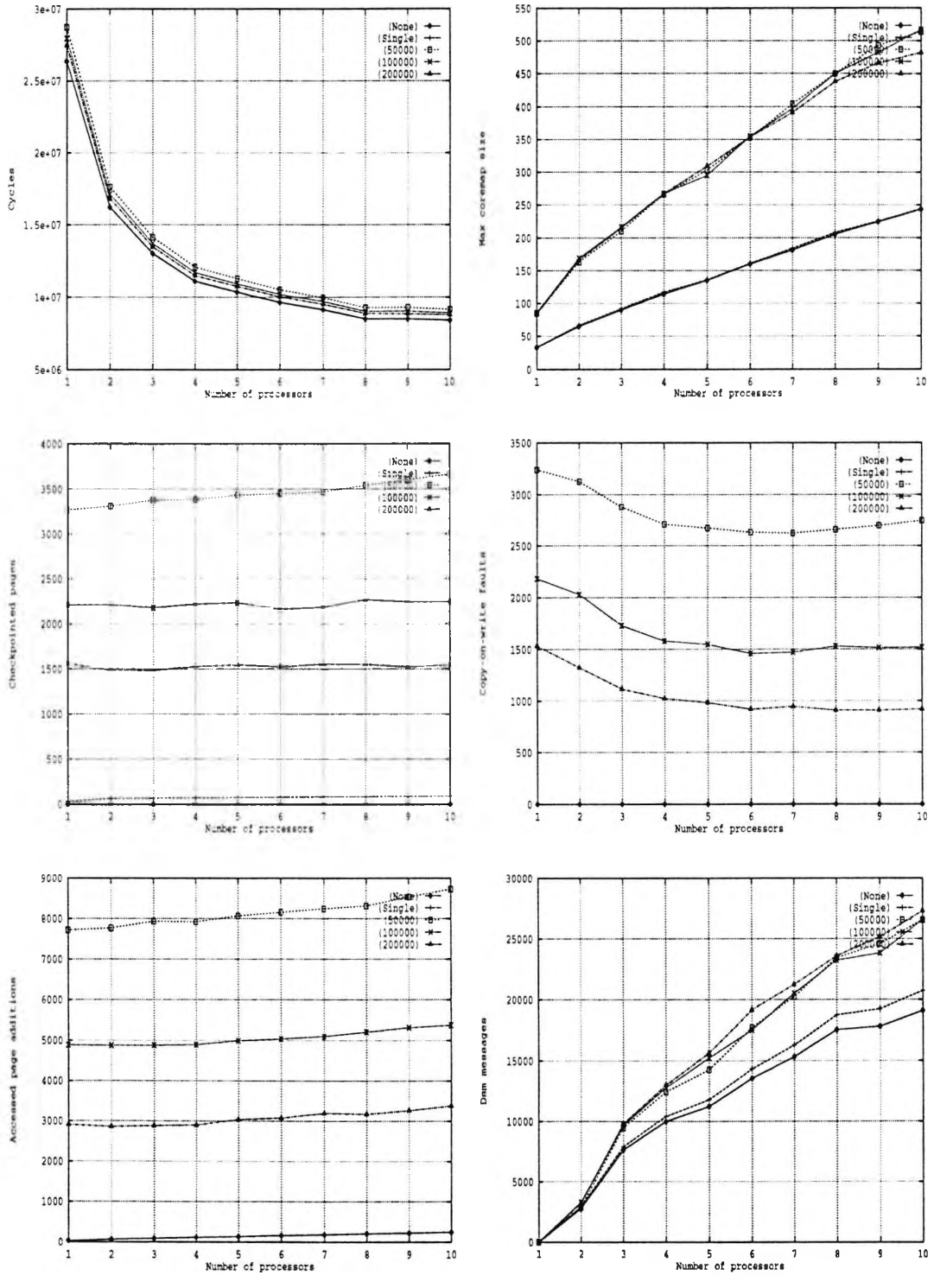


Figure 8.16: Time based experiments – 32 molecules

Conclusions

The result of adding a time constraint to checkpoints is unremarkable; more checkpoints incur additional costs resulting in a constant slow down of the application. However, such a mechanism is necessary if the subproblem size in the application is very small.

It should be noted that the time based experiments are based around very short checkpoint time periods, with checkpoints occurring between 100 and 500 times a second. In a real system checkpoints would occur less frequently and the constant cost would be much smaller than seen here.

8.7.3 Page size and constant Access Table capacity

Experiments of similar page size to those with the `mp3d` program were carried out using the `water` program. The results of these are shown in figures 8.17 and 8.18.

The general trend of smaller pages to decrease execution time and DSM message traffic is again observed in these results, although the changes are not as dramatic. However, in contrast to `mp3d`, smaller pages result in an increase in the number of pages checkpointed and accessed. This does not cause an increase in checkpointed data, since the number of checkpointed pages does not quite double when the page size is halved. From this we can conclude that, whilst `mp3d` accesses a very loosely clustered data set, `water` accesses a much tighter set of pages resulting in fewer page faults with large page sizes.

Conclusions

Like `mp3d` the predominating effect of reducing the page size is to reduce false sharing, so decreasing the DSM traffic and therefore speeding the execution of the application. The effect this has on checkpointing, and its contribution to execution time, appears to be negligible. Although a greater number of pages are checkpointed, the actual quantity of data checkpointed does not vary—the negative effect of one is compensated by the positive effect of the other.

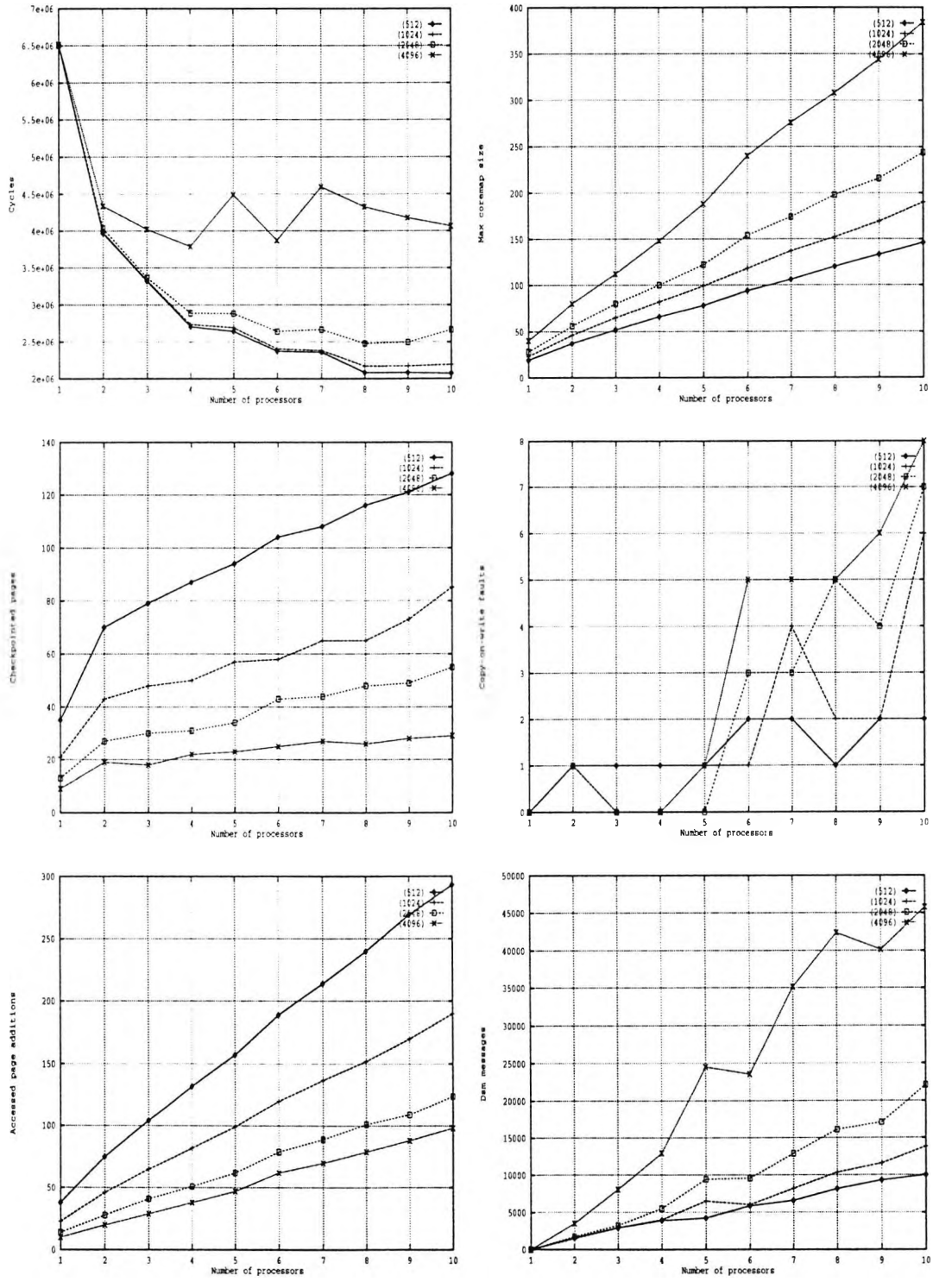


Figure 8.17: Page size experiments – 16 molecules

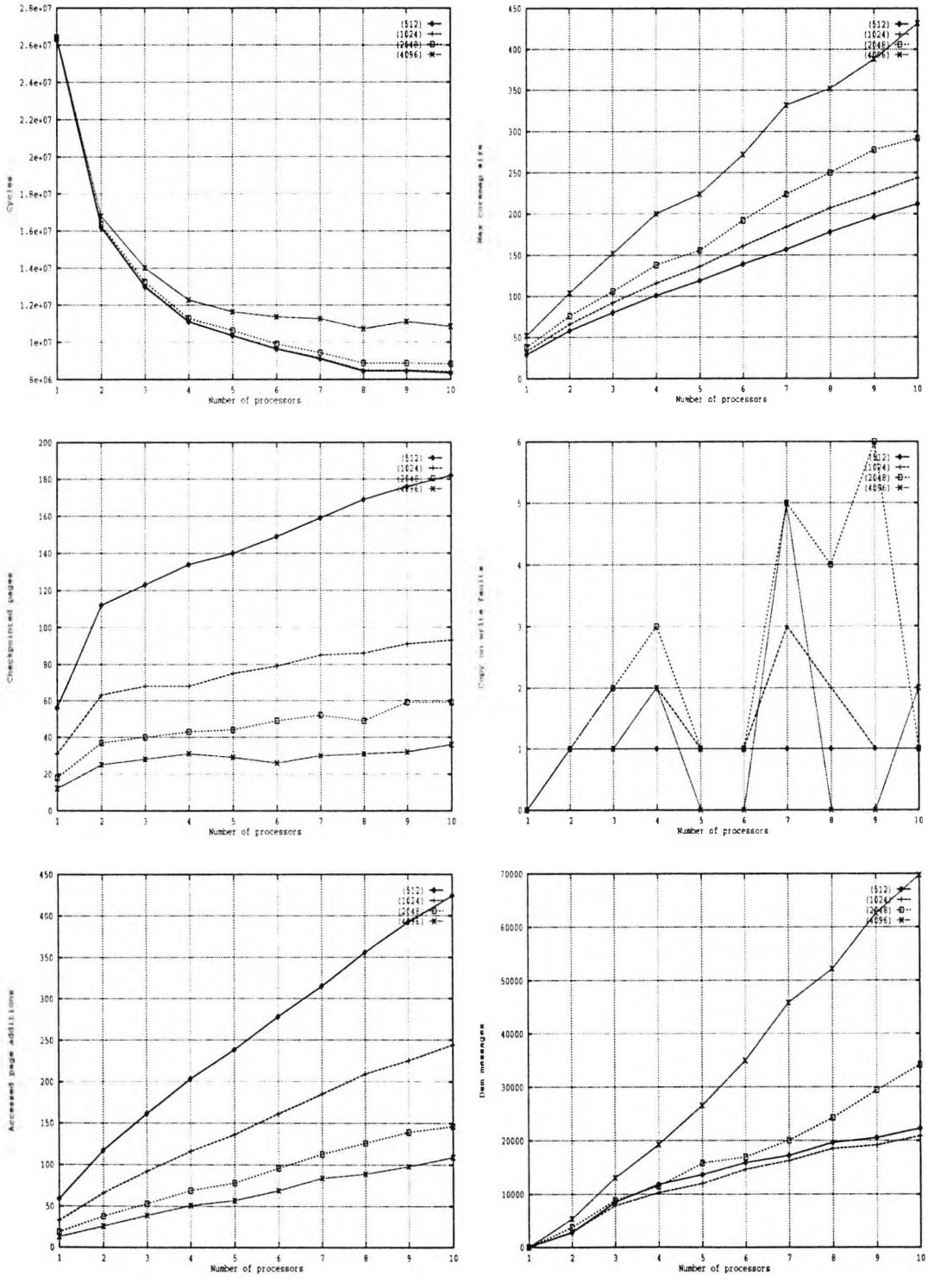


Figure 8.18: Page size experiments – 32 molecules

8.7.4 Large scale simulation

One set of simulations using 128 molecules was completed although there has been insufficient time to complete further large scale experiments. The results are given in figure 8.19.

This set of simulations is configured in a similar way to those in §8.7.1. Despite the dramatic increase in problem size, the modifications in access table size make little difference to the execution time of the application. However, with a table size of only 40 entries, the subproblem only just fits within it. This is demonstrated by a much large number of checkpointed pages and access page additions for this case; although still insufficient to make an impression on the execution time.

8.7.5 Conclusions

`Water` exhibits similar speedup behaviour to `mp3d` under the same conditions—this is encouraging. It also demonstrates that if the subproblem size fits well within the access table, then checkpointing adds no overhead (since the do not happen). To overcome this problem, a time related checkpoint element was introduced whereby a checkpoint would always occur within a set time from the last one. This appears to add a constant latency to execution regardless of the parallelism of the application but proportional to the problem size.

8.8 Experiments — Barnes-Hut

`Barnes-Hut` was the final program used with the simulator. As with the previous two examples, both the problem size and the parallelism could be changed easily. Also, in common with `Water`, large scale simulations were impossible to perform due to the excessive execution times required. Simulations were therefore carried out using 64 and 128 masses.

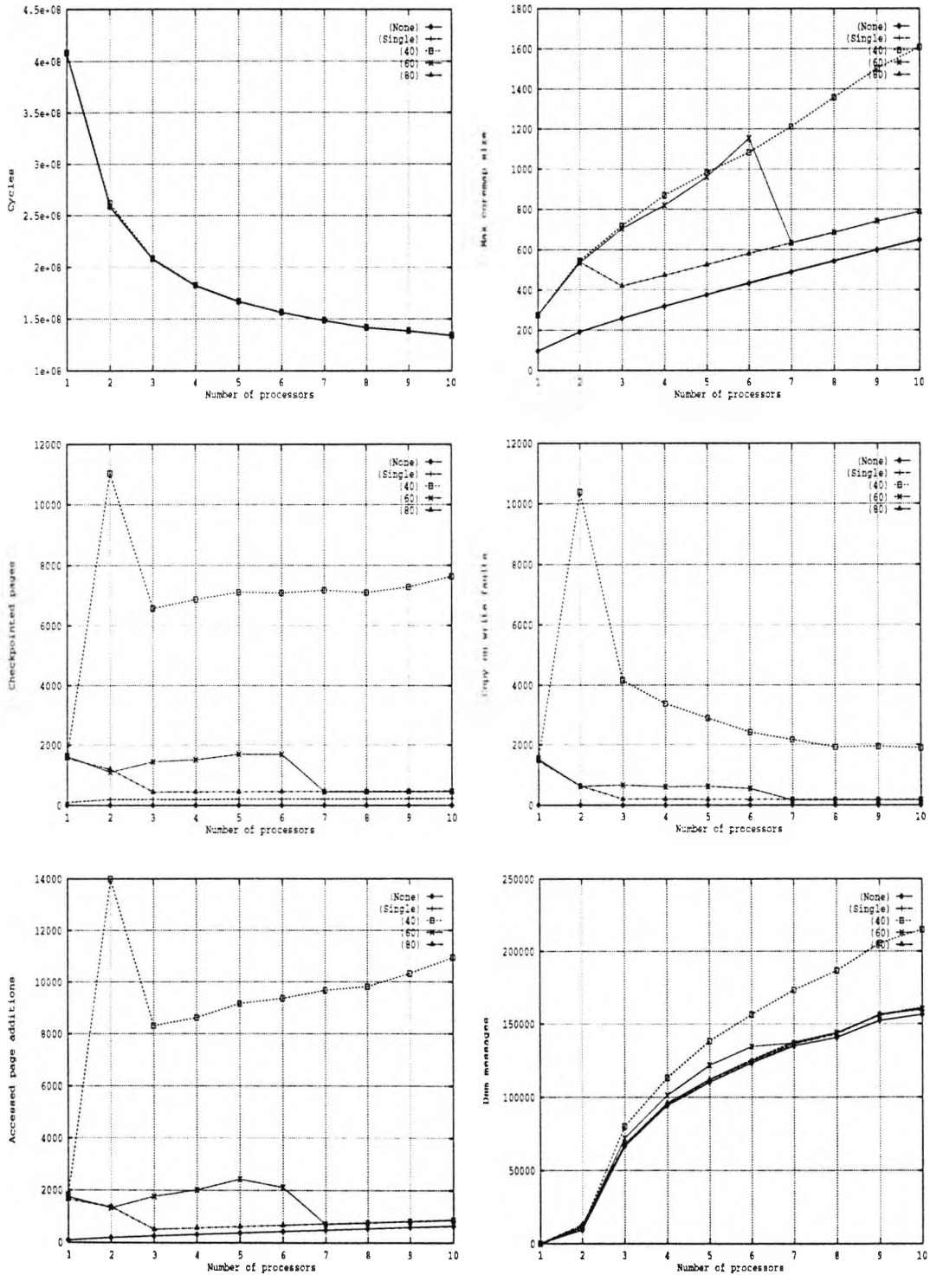


Figure 8.19: Access table experiments – 128 molecules

8.8.1 Size of access table

Experiments were carried out using no checkpoints, terminating checkpoints and access table limited checkpoints. The results of these are shown in figures 8.20 and 8.21.

The first thing to be observed is that the graph characteristics closely match those seen in `Water`. In fact, `Barnes-Hut` exhibits similar treatment of data, producing results which are not significantly effected by checkpoints. The reasons why such behaviour is expected will not be repeated (see §8.7).

8.8.2 Conclusions

Further experiments with `Barnes-Hut` are not included here due to the observed, and expected, similarity of the results with those of `Water`. Nothing new was discovered; the conclusions drawn from observing the `Water` algorithm are equally applicable here.

8.9 Overall conclusions

Overall conclusions can be drawn from the experiments performed. Firstly, if the access table size is too small to accommodate the subproblem size of a parallel application, then excessive checkpoints are performed. Secondly, if the subproblem fits easily into the access table, checkpoints are never performed unless a time based checkpointing element is also present. The mechanism described provide for both these cases.

Ideally, the access table size and timed checkpoint elements should not be fixed statically but determined dynamically, so as to provide the best compromise between data in each checkpoint, time between checkpoints (and so necessary delay incurred on rollback), and execution time of the application. An algorithm to adjust these parameters on a per application basis is the subject of further research.

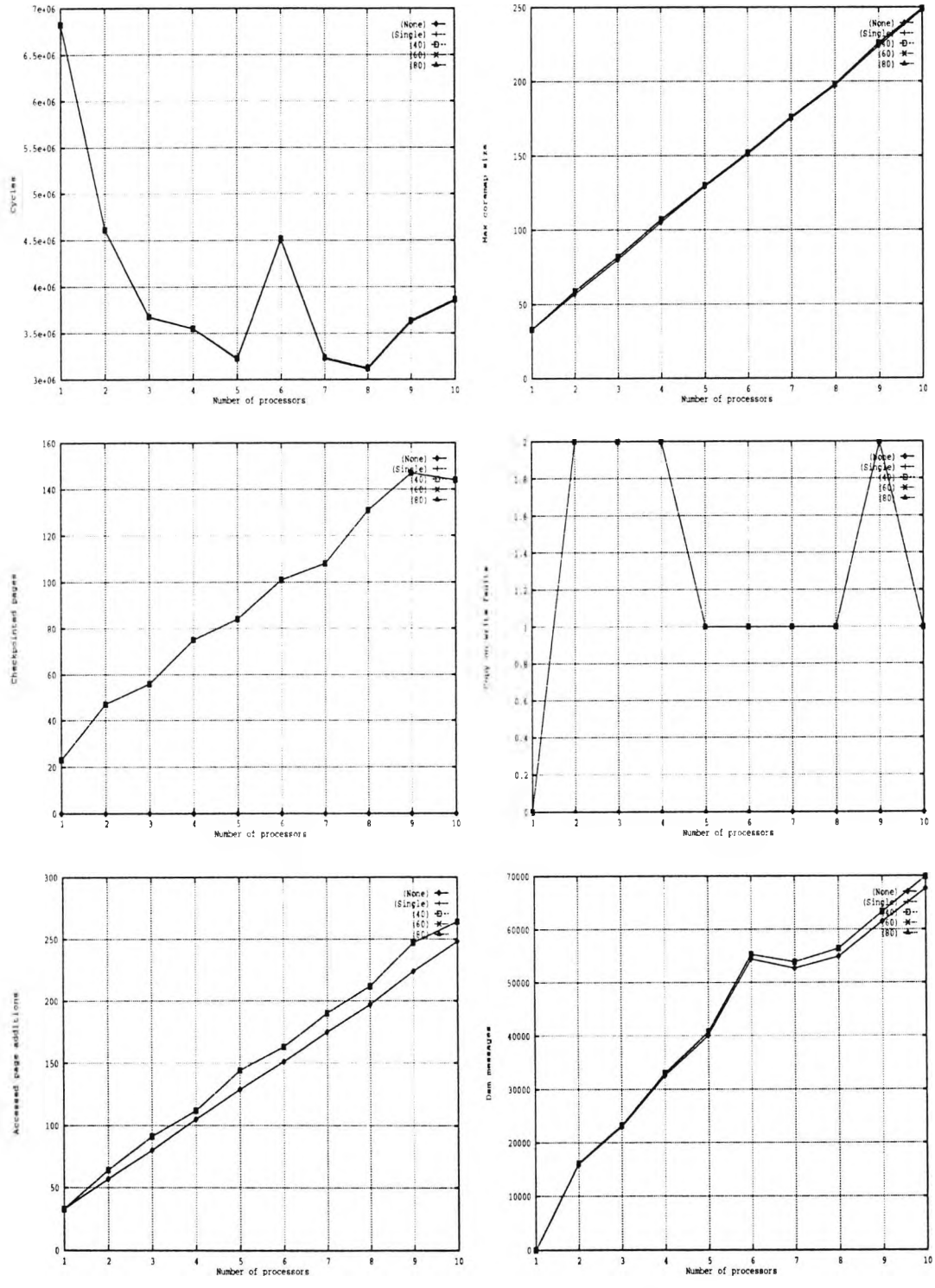


Figure 8.20: Access table experiments – 64 masses

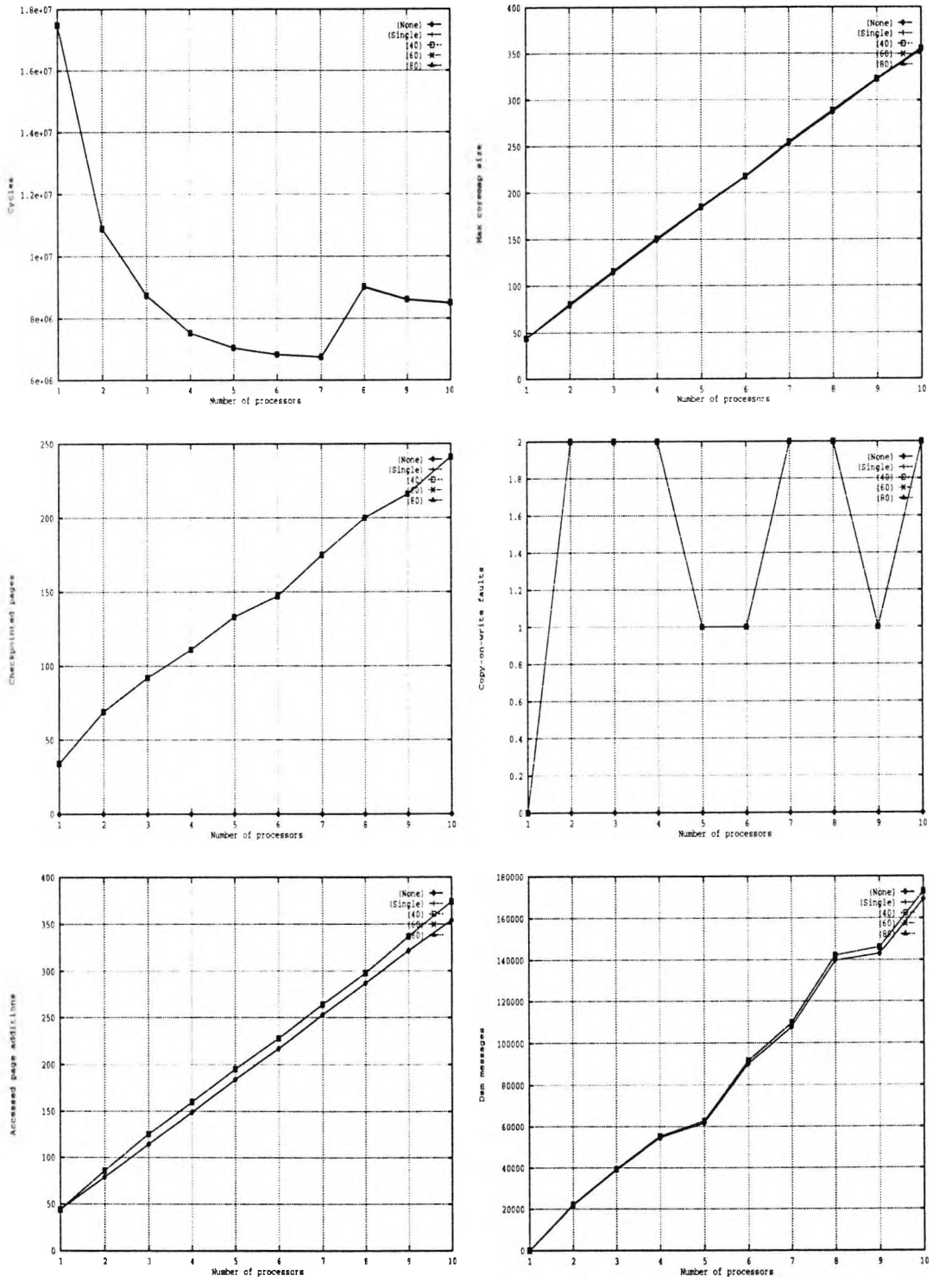


Figure 8.21: Access table experiments – 128 masses

8.10 Summary

This chapter has described the design of the ARIUS and AMOS model which has been used to analyse the performance characteristics of the reliability mechanisms. The model was developed as a simulation library which handles events generated by code from a modified C-compiler. This was found to be a flexible environment for experiments.

Various applications were used to examine the checkpointing system, and the relevant results presented here (complete results can be found in Appendix B). The primary goal of these experiments was to minimise the effect on execution time of the checkpointing system. The other resource considerations, such as extra memory used, were of secondary importance.

The results demonstrate a need for a checkpointing system comprising of a time related component (to prevent large period of computation being lost when a fault occurs) and a space related component (to prevent large quantities of memory being consumed when a checkpoint occurred). By optimising these parameters, it is possible to reduce the checkpoint latency to between 2% and 10% of the original execution time. This is much better than originally expected, and better than any system reviewed.

Chapter 9

Conclusions

In the introduction to this work, the question was posed "How can fault tolerance be provided efficiently in a distributed parallel operating system?" This thesis has answer the question by: examining a new style of operating system designed to exploit parallel machine architectures and the potential of 64-bit processors, demonstrating a scalable DSM system capable of tolerating some failures, and finally demonstrated an efficient fault tolerance system which exploits the inherent resource duplication in a parallel machine.

9.1 The Arius operating system

Large address space processors offering 64+ bits of address make it possible to unify networks of machines and disks into one, single namespace. This greatly simplifies the development of parallel and distributed applications. Importantly, ARIUS can support a UNIX service if required. This is essential if it is to gain any general acceptability. Unlike other system this can be done without compromising ARIUS and still allows UNIX applications to access the unified services if they require. Consequently, there is a simple migration pathway from one system to the other.

9.2 Distributed shared memory

ARIUS adopts distributed shared memory as its only means of inter-processor communication and as the kernel's communication mechanism. The design demonstrated here is therefore constrained to provide a service acceptable to both operating system services as well as general applications.

This resulted in a design capable of supporting various coherency policies in a unified fashion. This proved essential since a single policy cannot support general data sharing (using a write-invalidate policy) and barrier synchronisation (using a write-update policy) efficiently. The DSM system adopted a scalable approach to maintaining the copyset. In small systems a distributed bit-vector copyset may suffice. With 1000 processors this becomes impractical. The adoption of a circular linked list provided a scalable solution, which on further analysis proved as or more efficient than a bit-vector solution, and with doubly-linked entries provided the necessary fault tolerance.

9.3 Reliability

Originally, the reliability mechanisms designed for ARIUS were only intended to provide data integrity so preventing large scale data corruption when a machine failed. However, this was soon expanded to a fully comprehensive reliability strategy. It was hoped that the use of a DSM scheme for the majority of the work would result in a performance loss of at best 20%. Experiments which demonstrated a performance loss of only 2%–10% were an unexpected pleasure. The decision to divide reliability into *volatile* and *persistent* was essential to this result by allowing easy incorporation of fault tolerance into the DSM and the subsequent scalability that resulted from this combination.

Inefficient, processor hungry mechanisms to provide fault tolerance have never had wide acceptance except for critical applications. The mechanism developed here demonstrates that such inefficient fault tolerance systems are no longer required. An efficient system can now be implemented on any parallel architectures, and hopefully such systems will become generally accepted.

9.4 Future work

As with most pieces of work, as many new questions and problems arise as are solved. In this thesis, an attempt has been made to describe a new 64 bit operating system architecture based on distributed shared memory and design such a system to provide the reliability necessary in any large scale parallel machine. This has resulted in various pieces of potential research.

9.4.1 Work on multi-policy DSM protocols

The design of the DSM protocols given here are relatively primitive in the way different policies operate together. Remote writes, for example, are not efficiently implemented. Much could be done to improve this situation. This includes work to reduce the searching of owner chains when obtaining ownership and allowing ownership to pass outside the copyset so simplifying remote access operations.

Hierarchy of chains

A hierarchy of chains may prove a more efficient method of handling invalidations and updates. This would allow updates to be carried out in parallel in large copysets so reducing the write latency. However, it is necessary to investigate how such a structure can be supported in a resilient fashion and what the gain would be for the added complexity.

9.4.2 Algorithm to adjust reliability parameters

Experiments in chapter 8 demonstrate how the parameters, checkpoint time and access table size vary the composition and frequency of checkpoints. However, no work has been done in determining the best values for these parameters for a given application. An algorithm is therefore required to monitor an application and adjust these parameters in order to obtain the best performance whilst maintaining a required level of reliability.

9.4.3 Implementation of Arius and AMOS

The complete ARIUS operating system would allow many other claims and aspects of this work to be analysed more thoroughly. However, such a task is significant since it requires not only the basic system but the necessary support compilers and programs to allow the result to useful.

Additionally, Arius leads onto many interesting aspects of work. By not explicitly supporting the UNIX model of processing, a freedom is available to experiment with such diverse areas as graphical user interfaces and system call and service interfaces.

9.5 Summary

This chapter has presented general conclusions and thoughts on future work. In doing so, it has summarised the answer to the question posed in the introduction, how to provide efficient fault tolerance. This thesis demonstrates that in parallel machines, efficient fault tolerance is possible.

Bibliography

[AB86] J. Archibald and J. Baer. "An evaluation of cache coherent solutions in shared-bus multiprocessors,". *ACM Transactions on Computer Systems*, February 1986.

[ABG+86] N. Accetta, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "MACH: A new kernel foundation for UNIX development,". In *USENIX Summer Conference*, July 1986.

[ADL90] M. Ahamad, P. Dasgupta, and R.J. LeBlanc. "Fault-tolerant Atomic Computations in an Object-based Distributed System,". *Distributed Computing*, 4(2), May 1990.

[AMMR90] R. Ananthanarayanan, S. Menon, A. Mohindra, and U. Ramachandran. "Experiences in Intergrating Distributed Shared Memory with Virtual Memory Management,". Technical Report GIT-CC-90/40, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, 1990.

[ATT85] *System V Interface Definition.* AT&T Customer Information Centre, 1985.

[Bac86] M.J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Inc, 1986.

[BAHK+88] J.M. Bernabéu-Aubán, P.W. Hutto, M.Y.A. Khalidi, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, P. Dasgupta, and U. Ramachandran. "The Architecture of Ra: A Kernel for Clouds,". Technical Report GIT-ICS-88/25, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, 1988.

[BALL89] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. "Lightweight remote procedure call,". *ACM Operating Systems Review*, 23(5):102–113, December 1989.

[BCZ90] J. Bennett, J. Carter, and W. Zwaenepoel. "Munin: Distributed shared memory based on type-specific memory coherence,". In *1990 Conference on the Principles and Practice of Parallel Programming*, March 1990.

[BG91] M. Bayer and K. Gresser. "Functions of an operating system in a fault-tolerant real-time system,". In *Prozessrechensysteme '91. (Process Computer Systems '91)*, pages 169–178, 1991.

[BO90] M. Baker and J. Ousterhout. "Availability in the Sprite distributed file system,". In *Fourth ACM SIGOPS European Workshop*, September 1990.

[Bor88] P. Borrill. "VMEbus – The Next 5 Years,". VMEbus in Research, October 1988.

[BS91] W. Bolosky and M. Scott. "A Trace-based Comparision of Shared-Memory Multiprocessor using Optimal Off-Line Analysis,". Technical Report, Computer Science Department, University of Rochester, NY, 1991.

[CD89] R.C. Chen and P. Dasgupta. "Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation,". In *Ninth International Conference on Distributed Computing Systems (IC-DCS)*, pages 121–128, June 1989.

[CDG92] A. Clematis, G. Dodero, and V. Gianuzzi. "Process checkpointing primitives for fault tolerance: definitions and examples,". *Microprocess. Microsyst. (UK)*, 16(1):15–23, 1992.

[CF89] A. Cox and R Fowler. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum,". *ACM Operating Ssystems Review*, 23(5):32–44, March 1989.

[Cha78] M.F. Challis. "Database Consistency and Integrity in a Multi-user Environment,". In B. Schneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 245–270. Academic Press, 1978.

[Cla85] D. Clark. "The structuring of systems using upcalls,". In *Tenth ACM Symposium on Operating Systems Principles*, December 1985.

[Dal90] W. Dally. "Express Cubes: Improving the Performance of k-ary n-cube Interconnection Networks,". Technical Report, Artificial Laboratory, M.I.T., USA, April 1990.

[DCM+91] P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabéu-Aubán, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilkenloh. "The Design and Implementation of the Clouds Distributed Operating System,". Technical Report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, 1991.

[Del88] G. Delp. *The Architecture and Implementation of MemNet: A High Speed Shared Memory Computer Communication Network.* PhD thesis, University of Delaware, Computer Science Department, 1988.

[DLAR91] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. "The Clouds Distributed Operating System,". Technical Report, College of Computing, Georgia Tech., Atlanta, 1991.

[DLM88] P. Dasgupta, R. LeBlanc, and B. Marsh. "The Clouds distributed operating system: functional description, implementation details and related work,". In *International Conference on Distributed Computing Systems*, 1988.

[Dob92] Dobberpuhl and others. "A 200Mhz 64-bit Dual Issue CMOS Microprocessor,". In *International Solid-State Circuits Conference*, February 1992.

[DVH66] J.B. Dennis and E.C. Van Horn. "Programming Semantics for Multiprogrammed Computations,". *Communications of the ACM*, 9:143–154, March 1966.

[FC991] "Fibre Channel Rev 0.93,". Working Draft ANSI, December 1991.

[Fle90] B. Fleisch. "Reliable distributed shared memory,". In *IEEE Workshop on Experimental Distributed Systems*, pages 102–105, 1990.

[Fow86] R. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 1986.

[FP89] B. Fleisch and G. Popek. "Mirage: A Coherent distributed shared memory design,". *Operating systems reviews*, 23(5):211–223, May 1989.

[HL82] M. Herlihy and B. Liskov. "A value transmission method for abstract data types,". *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[HLH91] E. Hagersten, A. Landin, and S. Haridi. "DDM – A Cache-only Memory Architecture,". Technical Report Research Report R91:19, Swedish Institute of Computer Science (SICS), November 1991.

[HO91] W. Ho and R. Olsson. "An approach to genuine dynamic linking,". *Software – Practice and Experience*, 21(4), April 1991.

[KABJ89] E. Kristiansen, K. Alnæs, B. Bakka, and M. Jenssen. "Scalable Coherent Interface,". In *Eurobus Conference, Munich*, March 1989.

[Kar91] A.T. Karila. *Open Systems Security – an Architectural Framework.* PhD thesis, Helsinki University of Technology, 1991.

[KJ91] B. Kaliski Jr. "An Overview of the PKCS Standards,". Technical Report, RSA Data Security, Inc., June 1991.

[KLA+90] M.L. Kazar, B.W. Leverett, O.T. Anderson, V. Apostolides, B.A. Bottos, S. Chutani, C.F. Everhart, W.A. Mason, S.T. Tu, and E.R. Zayas. "DEcorum File System Architectural Overview,". In *USENIX Summer Conference*, pages 151–163, June 1990.

[Kle86] S.R. Kleiman. "Vnodes: An Architecture for Mulitiple File System Types in Sun UNIX,". In *Usenix Conference*, pages 238–247, 1986.

[Lar90] J. Larus. "Abstract Execution: A Technique for Efficiently Tracing Programs,". Technical Report, Computer Science Department, University of Wisconsin, January 1990.

[Lev90] H. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Mass., 1990.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, Department of Computer Science, 1986.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. "The Stanford DASH multiprocessor,". *IEEE Computer*, 25(3), March 1992.

[LMKQ89] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[LO87] E. Lusk and R. Overbeek. "Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors,". Technical Report ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

[LSP82] L. Lamport, R. Shostak, and M. Pease. "The Byzantine generals problem,". *ACM TOPLAS*, 4(3):382–401, 1982.

[MBC+80] R. Metcalfe, D. Boggs, C. Crane, E. Taft, J. Shoch, and J. Hupp. "The Ethernet Local Network: Three Reports,". Technical Report, Palo Alto Research Centres, February 1980.

[MIP91] MIPS Computer Systems Ltd. *RISC Microprocessors, V_r4000 - User's Manual.* NEC, 1991.

[MR91] A. Mohindra and U. Ramachandran. "A survey of distributed shared memory in loosely-coupled systems,". Technical Report, College of Computing, Georgia Institute of Technology, January 1991.

[MSLM91] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos. "First-Class User-Level Threads,". Technical Report, Computer Science Department, University of Rochester, NY, 1991.

[MvRT+90] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. "Amoeba: A distributed operating system for the 1990's,". *IEEE Computer*, pages 44–53, June 1990.

[Nel90] V. Nelson. "Fault-Tolerant Computing: Fundamental Concepts,". *IEEE Computer*, July 1990.

[NWO88] M.N. Nelson, B. Welch, and J. Ousterhout. "Caching in the Sprite Network File System,". *Transactions on Computer Systems*, 6:134–154, February 1988.

[Ous92] J. Ousterhout. "Sprite permissions,". Personal correspondance, February 1992.

[PPTT89] R. Pike, D. Presotto, K. Thompson, and H. Trickey. "Plan 9 from Bell Labs,". In *Summer UKUUG Conference, London*, pages 1–9, July 1989.

[PPTT91] R. Pike, D. Presotto, K. Thompson, and H. Trickey. "Plan 9, A Distributed System,". In *Spring EurOpen Conference, Troms.*, pages 43–40, May 1991.

[PW91] J. Pendry and N. Williams. "Amd: The 4.4BSD Automounter - Reference Manual,". Technical Report, Imperial College, London, U.K., March 1991.

[Ram92] U. Ramachandran. "Clouds object permissions,". Personal correspondance, January 1992.

[RHB+90] J. Rosenburg, F. Henskens, F. Brown, R. Morrison, and D. Munro. "Stability in a Persistent Store Based on a Large Virtual Memory,". Technical Report CS/90/6, Department of Mathematical and Computational Sciences, University of St. Andrews, 1990.

[RO90] M. Rosenblum and J.K. Ousterhout. "The LFS Storage Manager,". In *Usenix Summer Technical Conference*, June 1990.

[RO91] M. Rosenblum and J.K. Ousterhout. "The Design and Implementation of a Log-Structured File System,". In *13th ACM Symposium on Operating Systems Principles*, 1991.

[Sch84] F.B. Schneider. "Byzantine generals in action: Implementing fail-stop processors,". *ACM TOCS*, 2(2), 1984.

[SCI91] "SCI: Scalable Coherent Interface,". IEEE Standards document P1596/D1.7, August 1991.

[SJR86] R. Sansom, D. Julin, and R. Rashid. "Extending a Capability Based System into a Network Environment,". Technical Report CMU-CS-86-115, Carnegie-Mellon University, April 1986.

[SLM+88] M.L. Scott, T.J. LeBlanc, B.D. Marsh, T.G. Becker, C. Dubnicki, E.P. Markatos, and N.G. Smithline. "Implementation Issues for the Psyche Operating System,". Technical Report, University of Rochester, Department of Computer Science, 1988.

[SLM89a] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocesors,". Technical Report, University of Rochester, Department of Computer Science, March 1989.

[SLM89b] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "Memory Management for Large-Scale NUMA Multiprocessors,". Technical Report, University of Rochester, Department of Computer Science, March 1989.

[SLM89c] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "A Multi-User, Multi-Language, Open Operating System,". Technical Report, University of Rochester, Department of Computer Science, April 1989.

[SNS88] J.G. Steiner, C. Neuman, and J.I. Schiller. "Kerberos: An Authentication Service for Open Network Systems,". Technical Report, Massachusetts Institute of Technology, March 1988.

[Sun89] Sun Microsystems. "Network file system protocol specification,". Request for comments RFC-1094, SRI-NIC, March 1989.

[SW92] A. Saulsbury and T. Wilkinson. "The Design of a Unifying 64-bit Distributed Operating System,". Unpublished, Swedish Institute of Computer Science, 1992.

[SWG92] J. Singh, W-D. Weber, and A. Gupta. "SPLASH: Stanford Parallel Applications for Shared-Memory,". Technical Report, Computer Systems Laboratory, Stanford University, 1992.

[SZ90] M. Stumm and S. Zhou. "Fault tolerant distributed shared memory algorithms,". In *Second IEEE symposium on Parallel and Distributed Processing*, pages 719–724, 1990.

[Tan88] A.S. Tanenbaum. *Computer networks.* Prentice-Hall, 2nd edition, 1988.

[TH90a] V. Tam and M. Hau. "Fast recovery in distributed shared virtual memory systems,". In *10th International conference on distributed computing*, 1990.

[TH90b] V. Tam and M. Hau. "Token transactions: Managing fine-grained migration of data,". In *9th ACM Symposium on principle of database systems*, April 1990.

[TKT89] Z. Tong, R.Y. Kain, and W.T. Tsai. "A low overhead checkpointing and rollback recovery scheme for distributed systems,". In *Eighth Symposium on Reliable Distributed Systems*, pages 12–20, 1989.

[TMR86] A.S. Tanenbaum, S.J. Mullender, and R. Renesse. "Using Sparse Capabilities in a Distributed Operating System,". In *Sixth International Conference on Distributed Computer Systems, IEEE*, pages 558–563, 1986.

[Tot92] B.K. Totty. "Experimental Analysis of Data Management for Distributed Data Structures,". Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[TSF90] M. Tam, J. Smith, and D. Farber. "A taxonomy-based comparison of several distributed shared memory systems,". Technical Report, Distributed Systems Laboratory, Dept. CIS, University of Pennsylvania, May 1990.

[vRTW89] R. van Renesse, A. Tanenbaum, and A. Wilschut. "The design of a High-performance File Server,". In *Ninth International Conference on Distributed Computer Systems, IEEE*, pages 22–27, 1989.

[WBD+89] B. Welch, M. Baker, F. Douglis, J. Hartmann, M. Rosenblum, and J. Ousterhout. "Sprite Position Statement: Use Distributed State for Failure Recovery,". In *Second Workshop on Workstation Operating Systems (WWOS-II)*, pages 130–133, September 1989.

[WF89] K-L. Wu and W.K. Fuchs. "Recoverable distributed shared virtual memory: memory coherence and storage structures,". *Nineteenth International Symposium on Fault-Tolerant Computing*, pages 520–527, 1989.

[WF90] K-L. Wu and W.K. Fuchs. "Recoverable distributed shared virtual memory,". *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[Wil90] P. Wilson. "Pointer swizzling at page fault time: Efficiently supporting hugh address spaces on standard hardware,". *ACM SIGARCH Computer Architecture News*, 19(4), December 1990.

[Wil92] T. Wilkinson. "A lightweight process library for C++,". Unpublished, 1992.

[Wil93] T.J. Wilkinson. "Compiling for a 64-Bit Single Address Space Architecture,". Technical Report TCU/SARC/1993/1, SARC, City University Computer Science Department, March 1993.

[Win92] P. Winterbottom. "Plan 9 permissions,". Personal correspondance, January 1992.

[WSO+92a] T. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly. "Angel: A Proposed Multiprocessor Operating System Kernel (Extended Abstract),". In *European Workshop on Parallel Computing*, March 1992.

[WSO+92b] T. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly. "Angel: A Proposed Multiprocessor Operating System Kernel,". Technical Report, Systems Architecture Research Centre, City University, March 1992.

Appendix A

Complete DSM algorithms

A.1 DSM server loop

Algorithm A.1

DSM server begin
Repeat forever {

Receive message
Find page message refers to
If message is a "request copy", call algorithm 4.3
Else if message is a "request read", call algorithm 4.4
Else If message is a "request ownership", call algorithm 4.5
Else if message is a "request invalidate", call algorithm 4.6
Else if message is a "request update", call algorithm 4.7
Else indicate an error!

}

A.2 DSM page copy request

Algorithm A.2

```
Begin request copy
If message is marked "error" {
    Unbusy the page
    Release requester for retry
}
Else if message is marked "link" {
    Set page's "downchain-nextcopy" to message's source
}
Else if message is marked "copy" {
    Set page's "lastowner" to "lastowner" in message
    Set page's "upchain-nextcopy" to "nextcopy" in message
    Set page's "downchain-nextcopy" to the source of the message
    Mark message as a "link"
    Forward message to page's "upchain-nextcopy"
    Add copy to page's DSM state
    Busy the page
    If message is marked dirty {
        Dirty the page's VM state
        If checkpoint chain point to myself, install chain entry from message
    }
    Request TLB flushes
    Release requester
}
Else if this node does not hold a copy of the page {
    If this node is the originator, copy checkpoint chain entry into message
    Forward message to page's "lastowner"
}
Else if this node is the originator and already has a copy of the page {
    Release requester
}
Else if page is marked "busy" {
    Mark message as "error"
    Send it back to originator
}
Else {
```

```
        Mark message as a "copy"
        Mark message "lastowner" to be page's "lastowner"
        Mark message "nextcopy" to be page's "upchain-nextcopy"
        If page is dirty {
            Mark page as dirty in message
            If message's checkpoint chain entry points to the originator, exchange the mes-
                sage's version with the page's version
        }
        Set page's "upchain-nextcopy" to be originator
        Add copy to page's DSM state
        Request TLB flushes
        Send message on to its originator
}
End request copy
```

A.3 DSM remote read request

Algorithm A.3

```
Begin request remote read
If message contains copy {
    Set page's "lastowner" to "lastowner" in message
    Insert data into halted requester
    If message is marked dirty {
        Dirty the page's VM state
        If checkpoint chain point to myself, install chain entry from message
    }
    Release requester
}
Else if this node does not hold a copy of the page {
    If this node is the originator, copy checkpoint chain entry into message
    Forward message to page's "lastowner"
}
Else if this node is the originator and already has a copy of the page {
    Release requester
}
Else {
    Mark message to contain a copy
    Place requested data in the message
    If data's page is dirty {
        Mark data as dirty in message
        If message's checkpoint chain entry points to the originator, exchange the mes-
            sage's version with the page's version
    }
    Send message on to originator
}
End request remote read
```

A.4 DSM request ownership

Algorithm A.4

```
Begin request owner
If message is marked "error" {
    Unbusy the page
    Release requester for retry
}
Else if page is marked "busy" and message is from a remote node {
    Mark message as "error"
    Send it back to originator
}
Else if message contains ownership {
    Add ownership to page's DSM state
    Set page's "lastowner" to be this node
    Request TLB flushes
    Release requester
}
Else if this node is the originator {
    If page is already owned {
        Release requester
    }
    Else if this node does not have a copy of the page {
        Unbusy the page
        Release requester to allow page to be obtained
    }
    Else {
        Busy the page to prevent removal of copy
        Forward message to the page's "lastowner"
    }
}
Else if this node does not own the page {
    Forward message to the page's "lastowner"
}
Else {
    Mark message to contain ownership
    Invalidate page's ownership on this node
    Set page's "lastowner" to be requesting node
    Request TLB flushes
    Send message on to originator
}
End request owner
```

A.5 DSM invalidate request

Algorithm A.5

```
Begin request invalidate
If message is marked "error" {
    Set page's "upchain-nextcopy" to be the source of the message
    Install checkpoint chain entry from message in current page's
    Unbusy the page
    Release requester for retry
}
Else if page is marked "busy" and message is from a remote node {
    Mark message as "error"
    Send it back to originator
}
If message contains "owner_one" {
    Set page's "upchain-nextcopy" to be this node
    Set page's "downchain-nextcopy" to be this node
    Set page's DSM state to indicate a single writable copy is present
    Install checkpoint chain entry from message in current page's
    Busy the page
    Request TLB flushes
    Release requester
}
Else if this node is the originator {
    If this node holds a single writable copy {
        Release requester
    }
    Else if this node owns the page {
        Busy the page to prevent movement of ownership
        Copy checkpoint chain entry into message
        Forward the message to "upchain-nextcopy"
    }
    Else {
        May not invalidate
        Release requester for retry
    }
}
Else {
    If node has this page {
```

If page's checkpoint chain entry points to the current node, exchange the message's version with the page's version
Remove copy from DSM state
Request TLB flushes
}
If page's "upchain-nextcopy" is originator, mark the message "owner_one"
Forward the message to the page's "upchain-nextcopy"
}
End request invalidate

A.6 DSM update request

Algorithm A.6

```
Begin request update
If message contains "owner" {
    Release requester
}
Else if message contains "owner_one" {
    Install checkpoint chain entry from message in current page's
    Unbusy the page
    If this node initiated the update {
        Release requester
    }
    Else {
        Mark the message "owner"
        Forward message to initiator
    }
}
Else if message contains "owner_many" {
    Set page's "lastowner" to be message's "lastowner"
    If page is present {
        Apply the update
        If page's checkpoint chain entry points to the current node, exchange the message's
            version with the page's version
    }
    If page's "upchain-nextcopy" is originator, mark the message "owner_one"
    Forward the message to the page's "upchain-nextcopy"
}
If this node owns the page {
    Busy the page to prevent movement of ownership
    Apply the update
    Copy checkpoint chain entry into message
    Mark the message "owner_many"
    Mark the message's "lastowner" to be this node
    Forward the message to "upchain-nextcopy"
}
Else {
    Forward the message to the page's "lastowner"
}
End request update
```

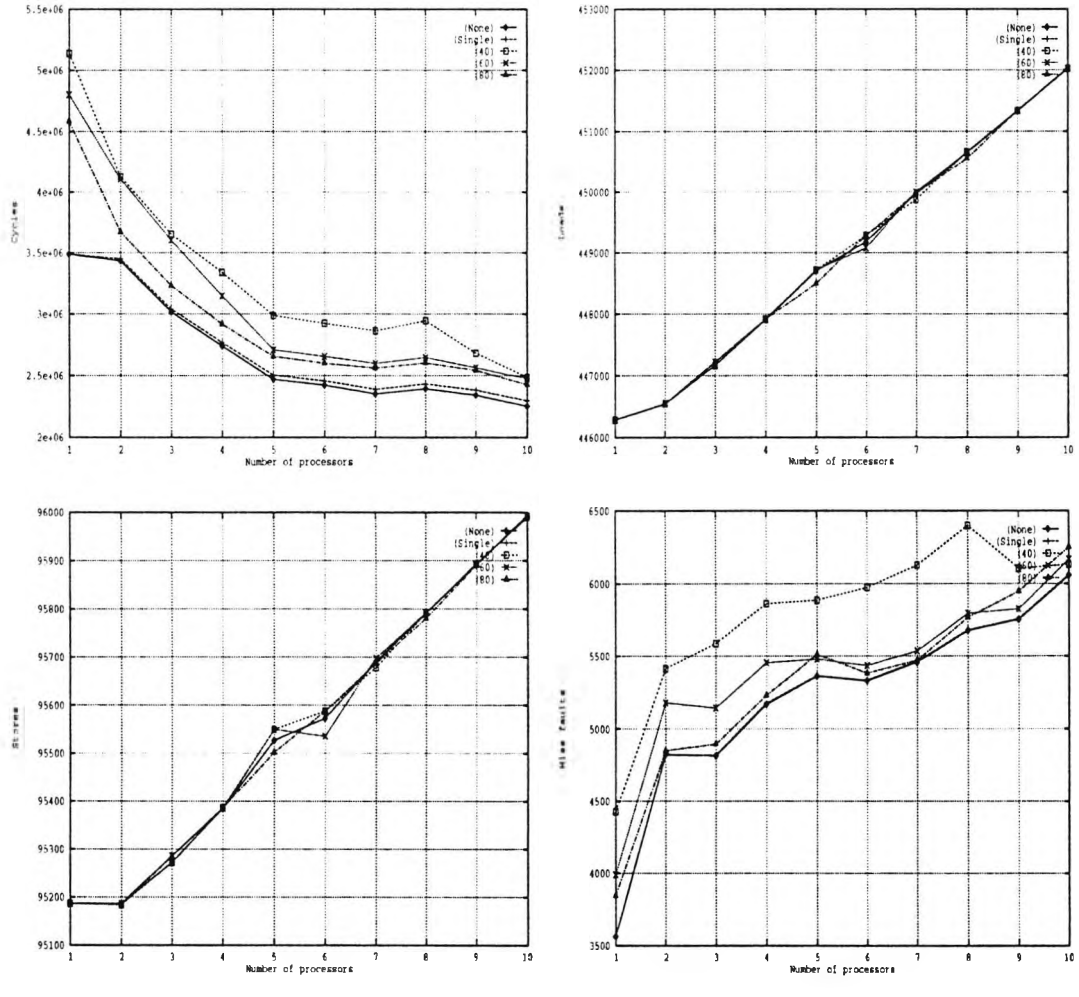
Appendix B

Complete results

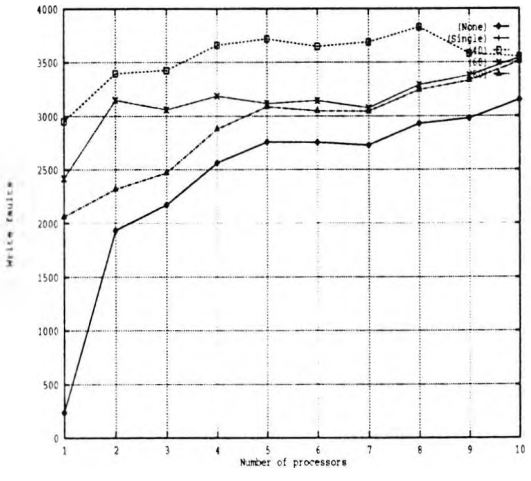
B.1 MP3D

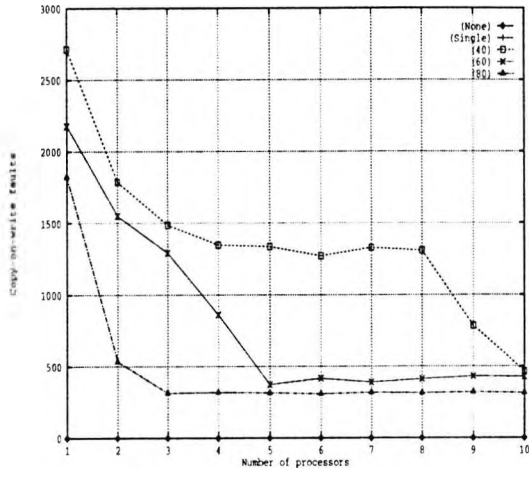
B.1.1 160 molecules – Access table size

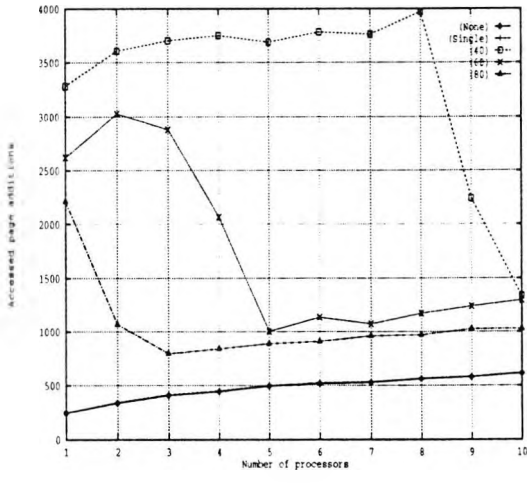
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

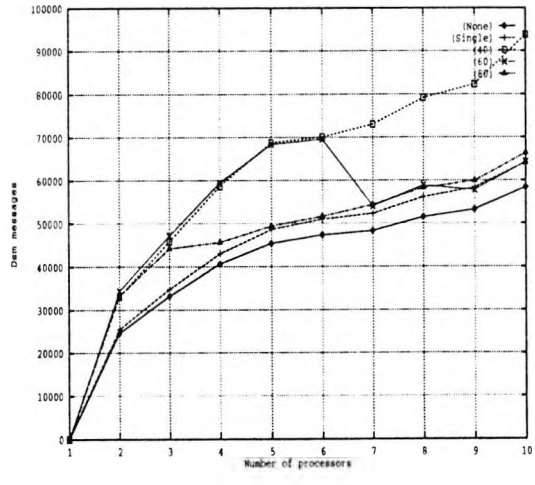


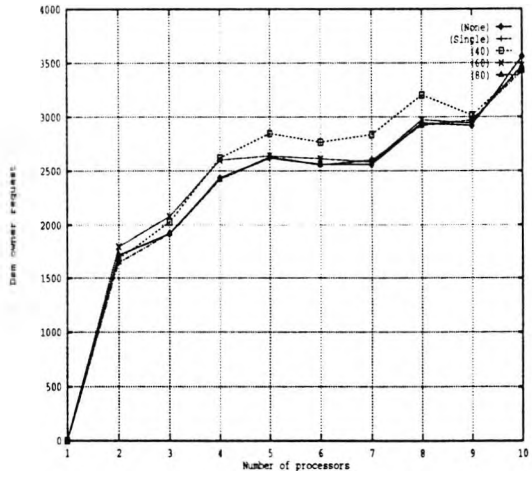
B.1. MP3D

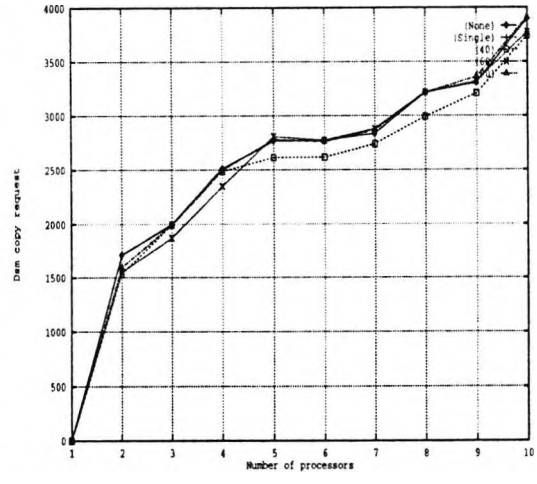

Write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

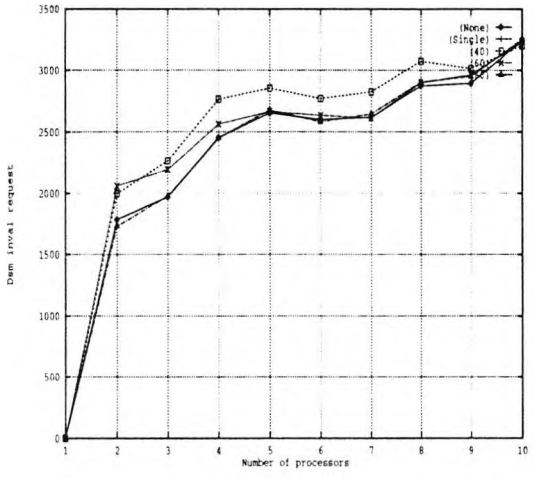

Copy-on-write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

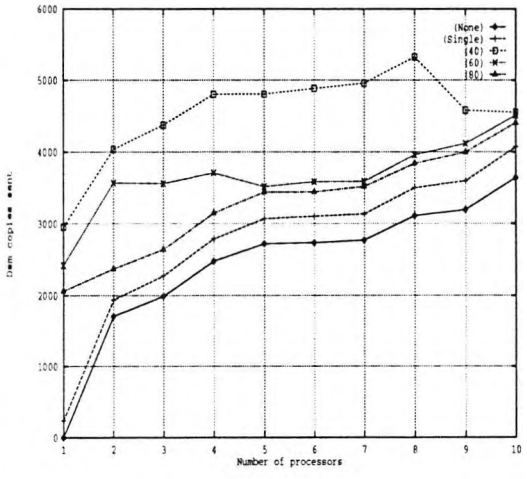

Accessed page additions
Number of processors
(None)
(Single)
(40)
(60)
(80)

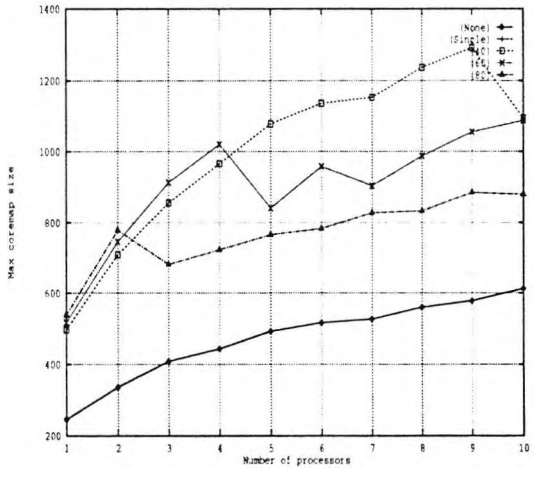

Dam messages
Number of processors
(None)
(Single)
(40)
(60)
(80)

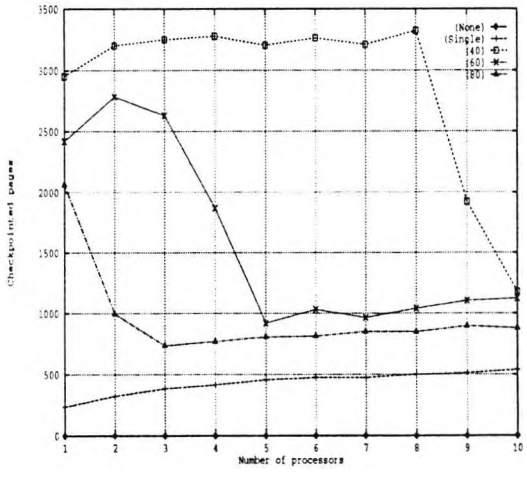

Dam owner request
Number of processors
(None)
(Single)
(40)
(60)
(80)

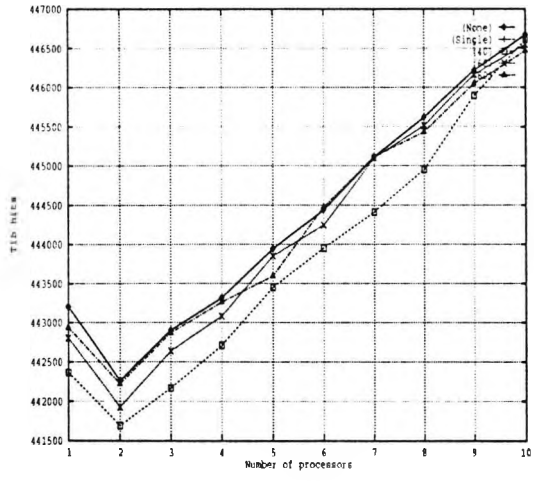

Dam copy request
Number of processors
(None)
(Single)
(40)
(60)
(80)

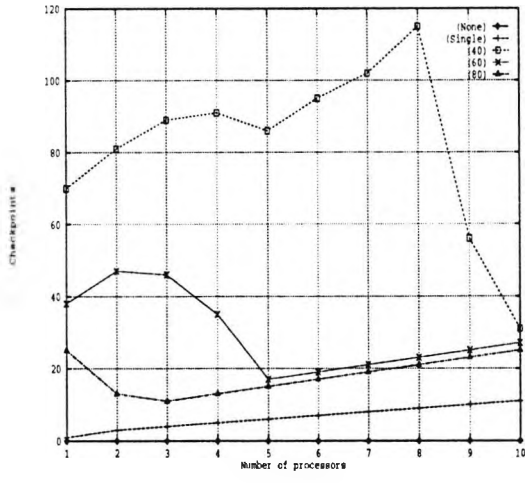

Dsm inval request
Number of processors
(None)
(Single)
(40)
(60)
(80)


Dsm copies sent
Number of processors
(None)
(Single)
(40)
(60)
(80)


Max coremap size
Number of processors
(None)
(Single)
(40)
(60)
(80)

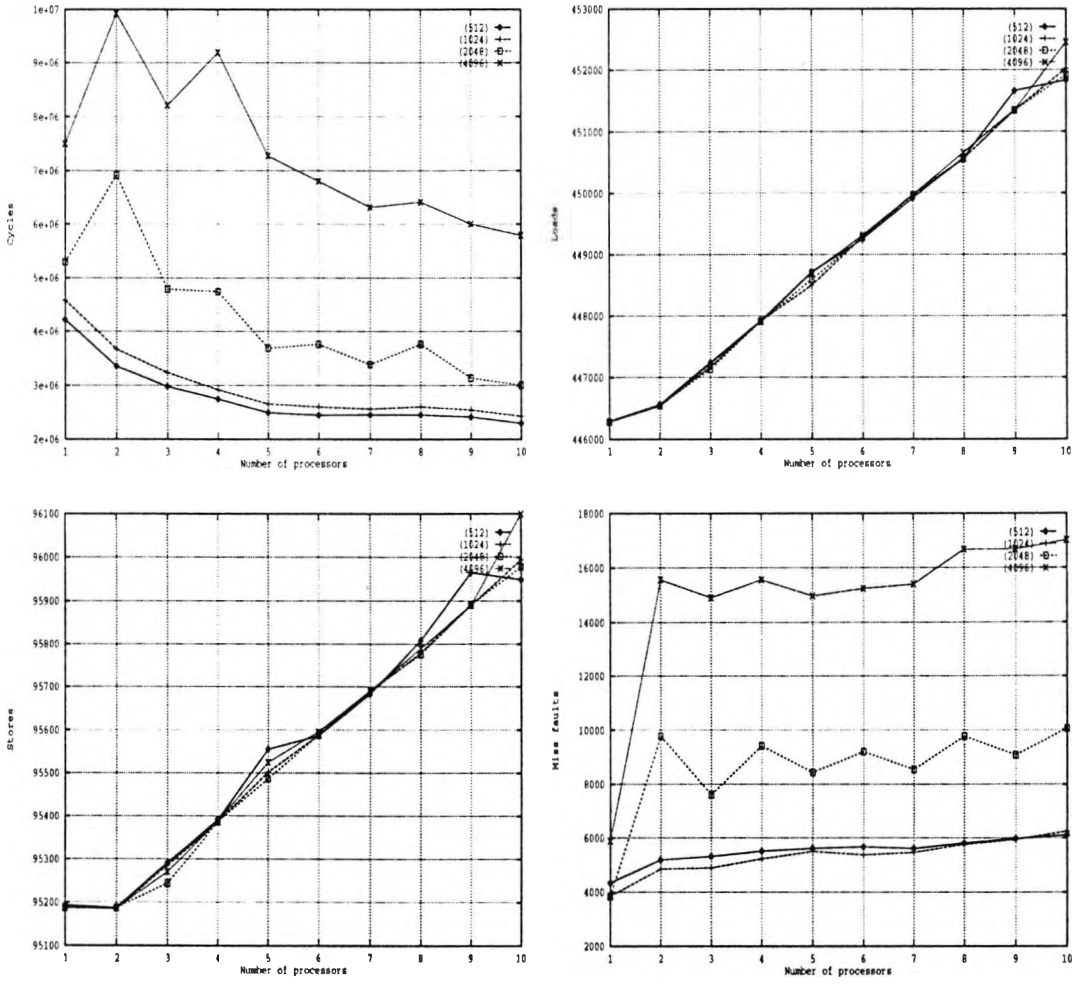

Checkpointed pages
Number of processors
(None)
(Single)
(40)
(60)
(80)

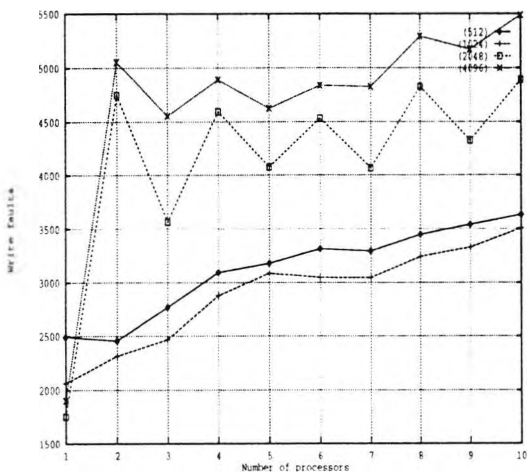

TLB hits
Number of processors
(None)
(Single)
(40)

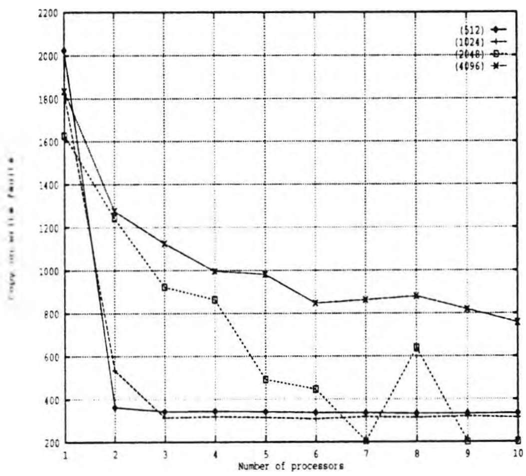

Checkpoints
Number of processors
(None)
(Single)
(40)
(60)
(80)

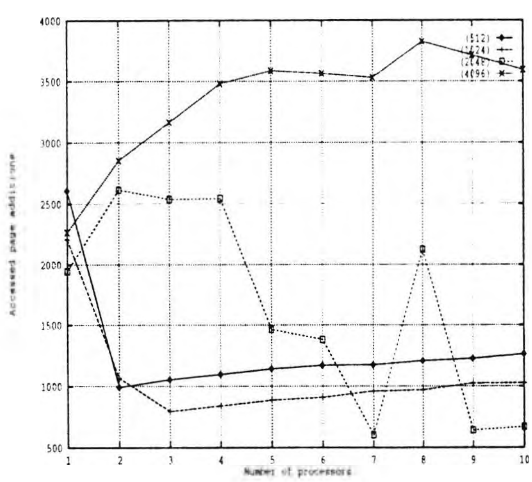
B.1.2 160 molecules – Page size

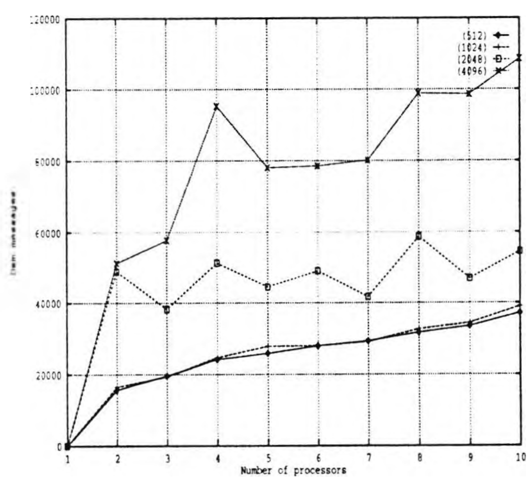
One to ten clusters,
One processor per cluster,
80 entry access table,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

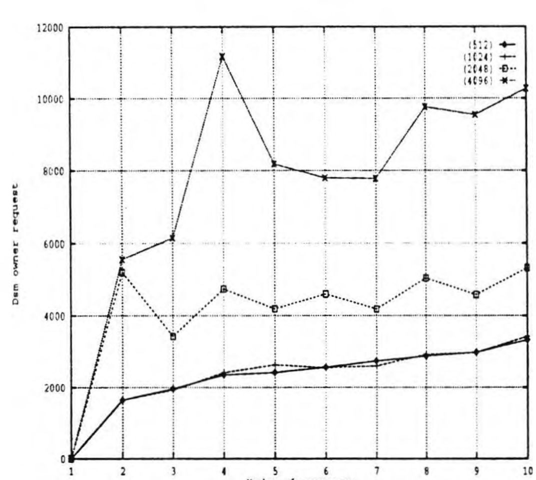


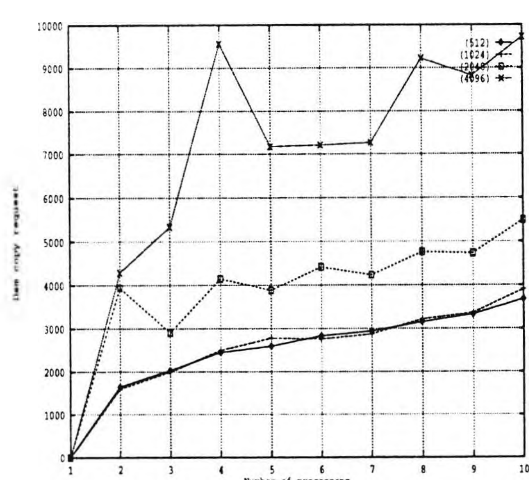

(512)
(1024)
(2048)
(4096)
Write faults
5500
5000
4500
4000
3500
3000
2500
2000
1500
1 2 3 4 5 6 7 8 9 10
Number of processors

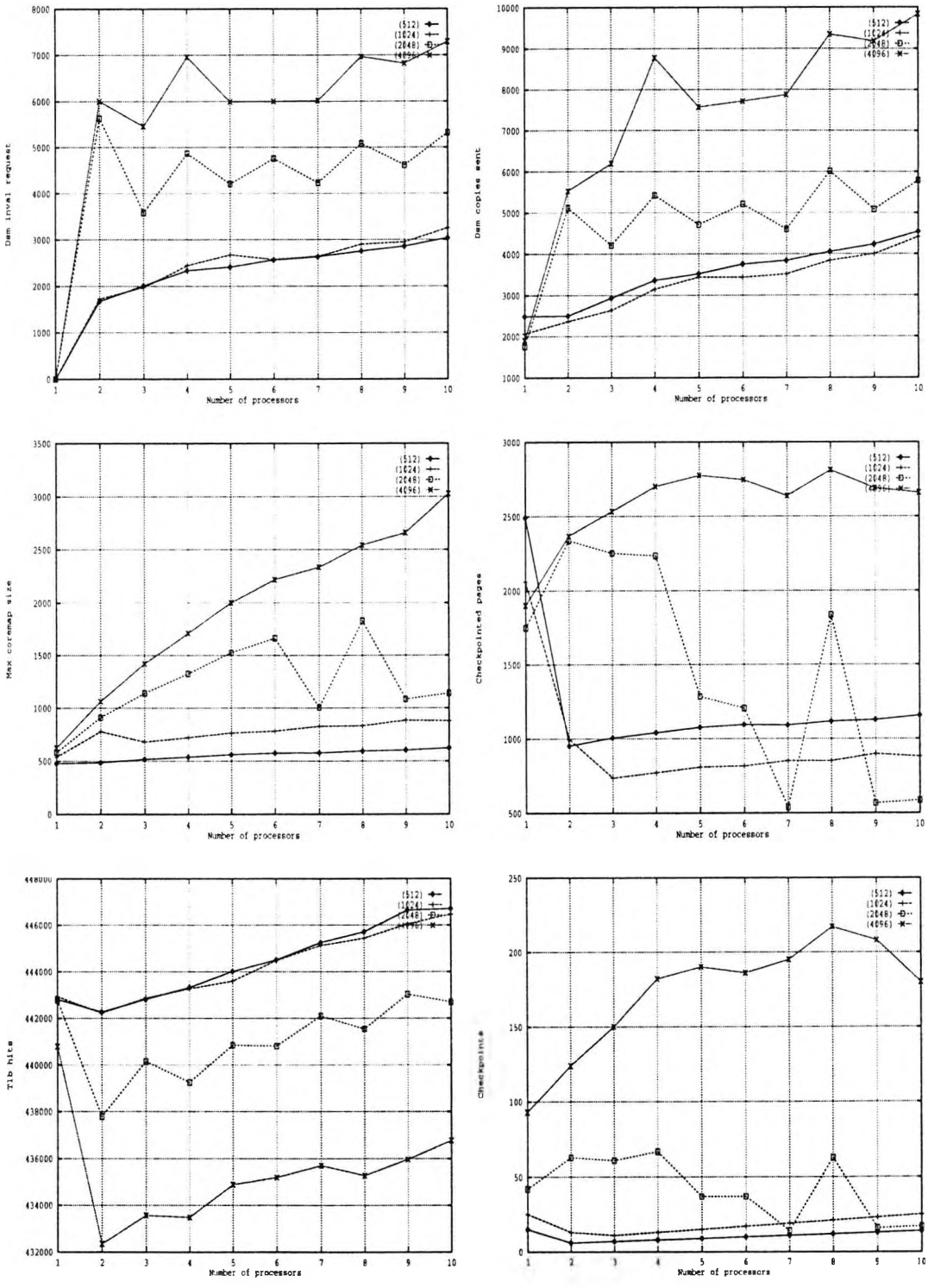

(512)
(1024)
(2048)
(4096)
2200
2000
1800
1600
1400
1200
1000
800
600
400
200
1 2 3 4 5 6 7 8 9 10
Number of processors


(512)
(1024)
(2048)
(4096)
Accessed page additions
4000
3500
3000
2500
2000
1500
1000
500
1 2 3 4 5 6 7 8 9 10


(512)
(1024)
(2048)
(4096)
Dsm messages
40000
20000
0
1 2 3 4 5 6 7 8 9 10
Number of processors

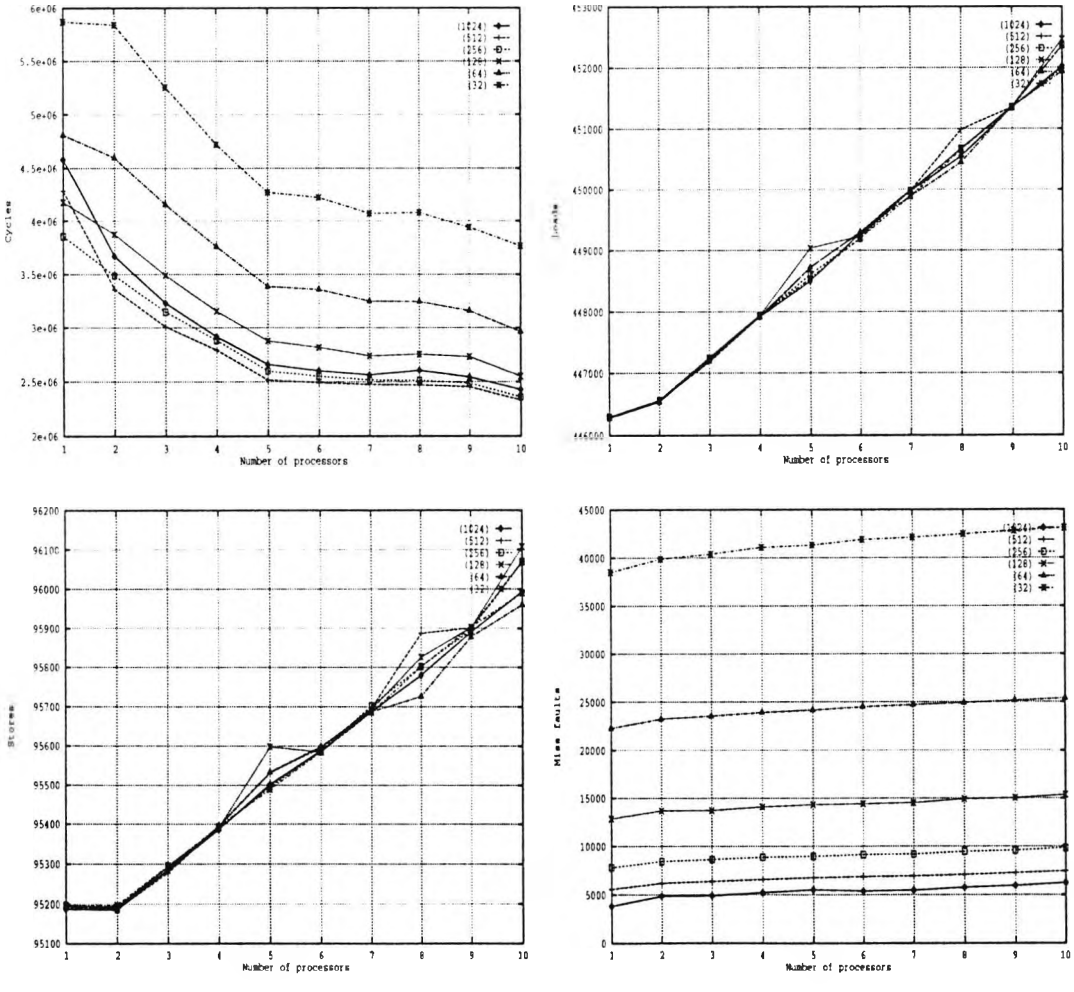

(512)
(1024)
(2048)
(4096)
Dsm owner request
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10
Number of processors


(512)
(1024)
(4096)
Dsm copy request
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10
Number of processors

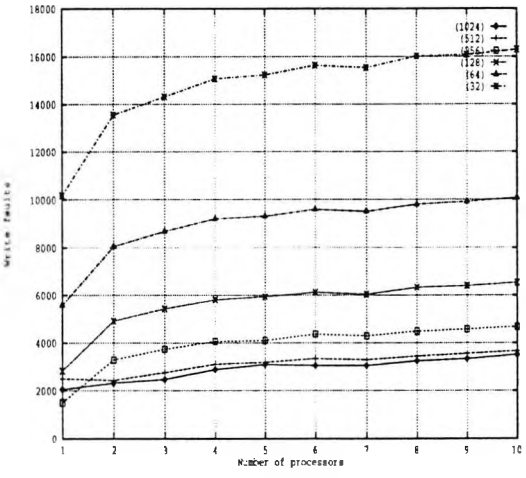

Dam inval request
Number of processors
(512)
(1024)
(2048)
(4096)
Dam copies sent
Number of processors
Max coremap size
Number of processors
Checkpointed pages
Number of processors
Tlb hits
Number of processors
Checkpoints
Number of processors

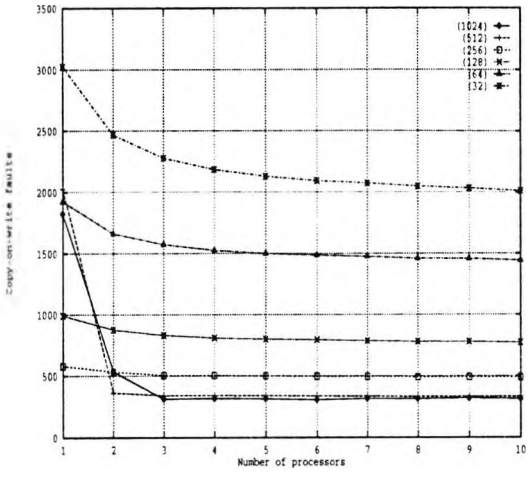
B.1.3 160 molecules – Small page size

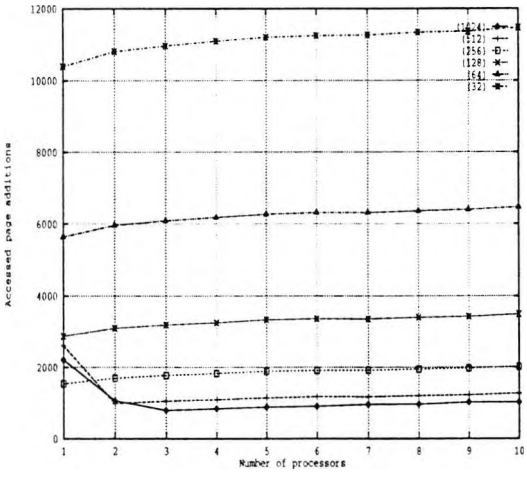
One to ten clusters,
One processor per cluster,
80 entry access table,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

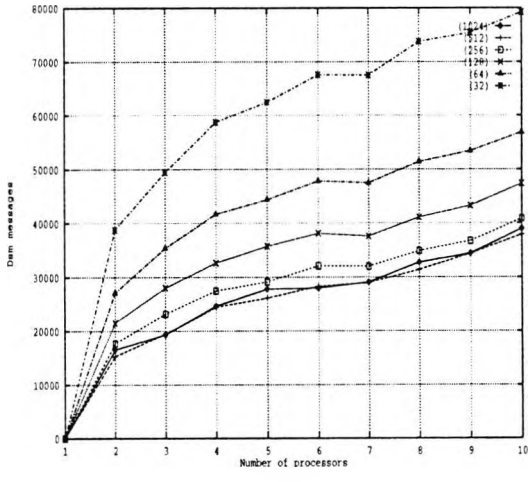


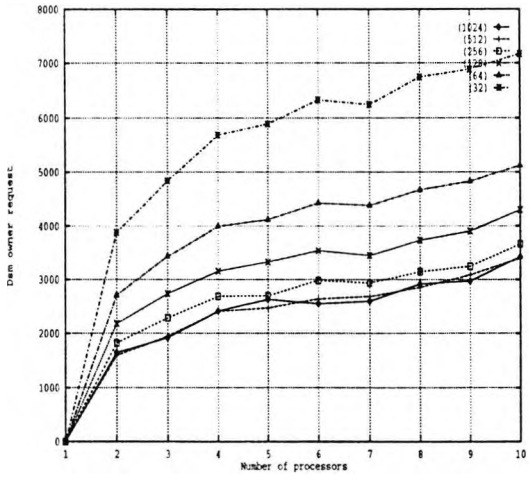
B.1. MP3D

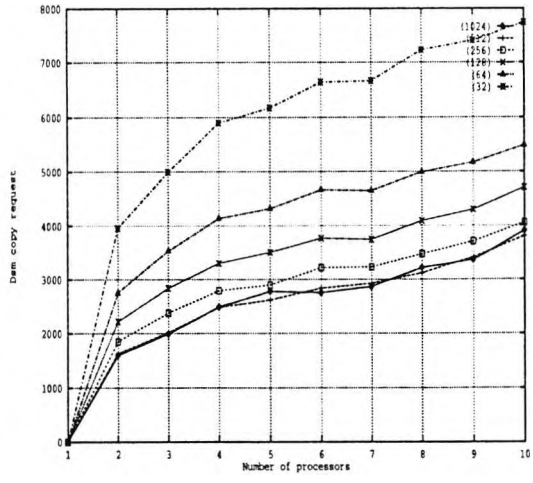

Write faults
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

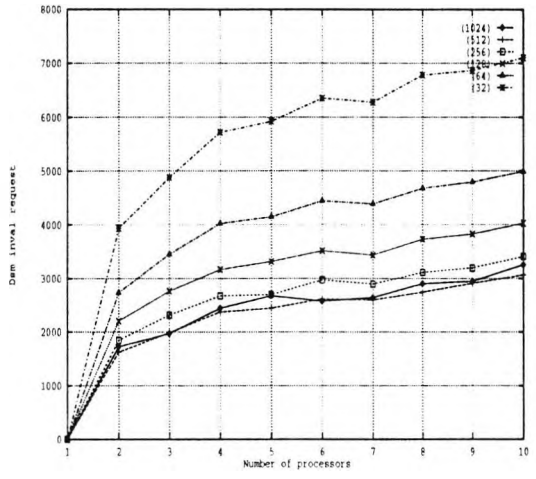

Copy-on-write faults
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

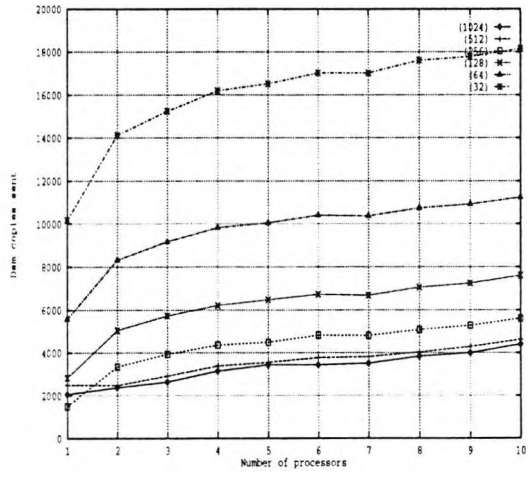

Accessed page additions
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

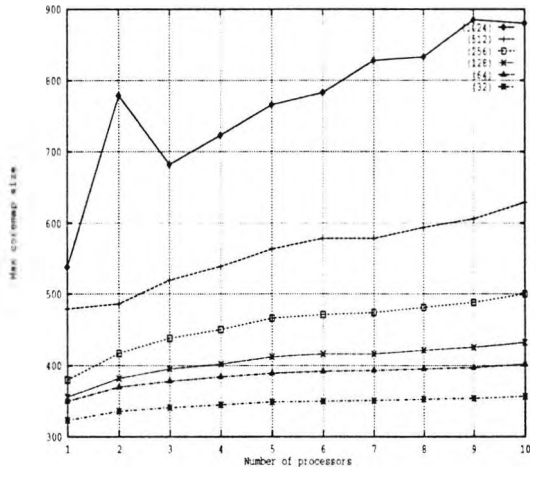

Dsm messages
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

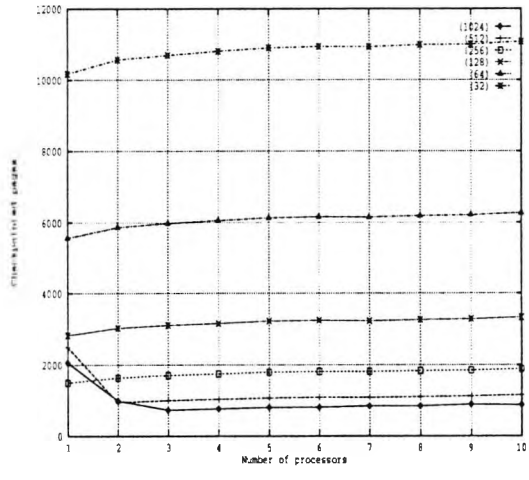

Dsm owner request
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

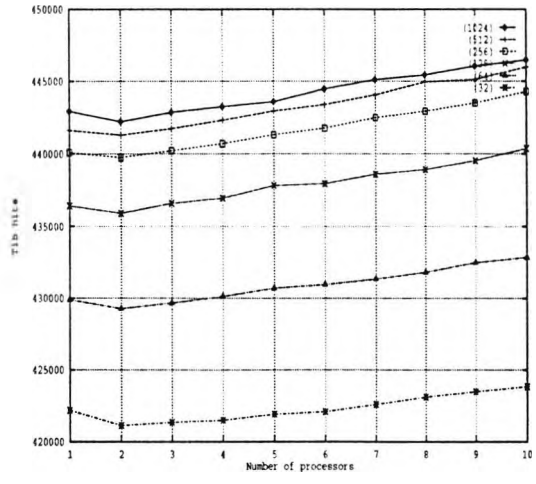

Dsm copy request
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

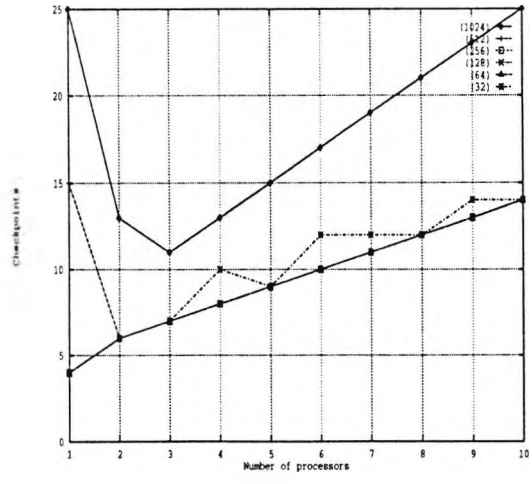

Dem inval request
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)


Dem replies sent
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)


Max coremap size
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

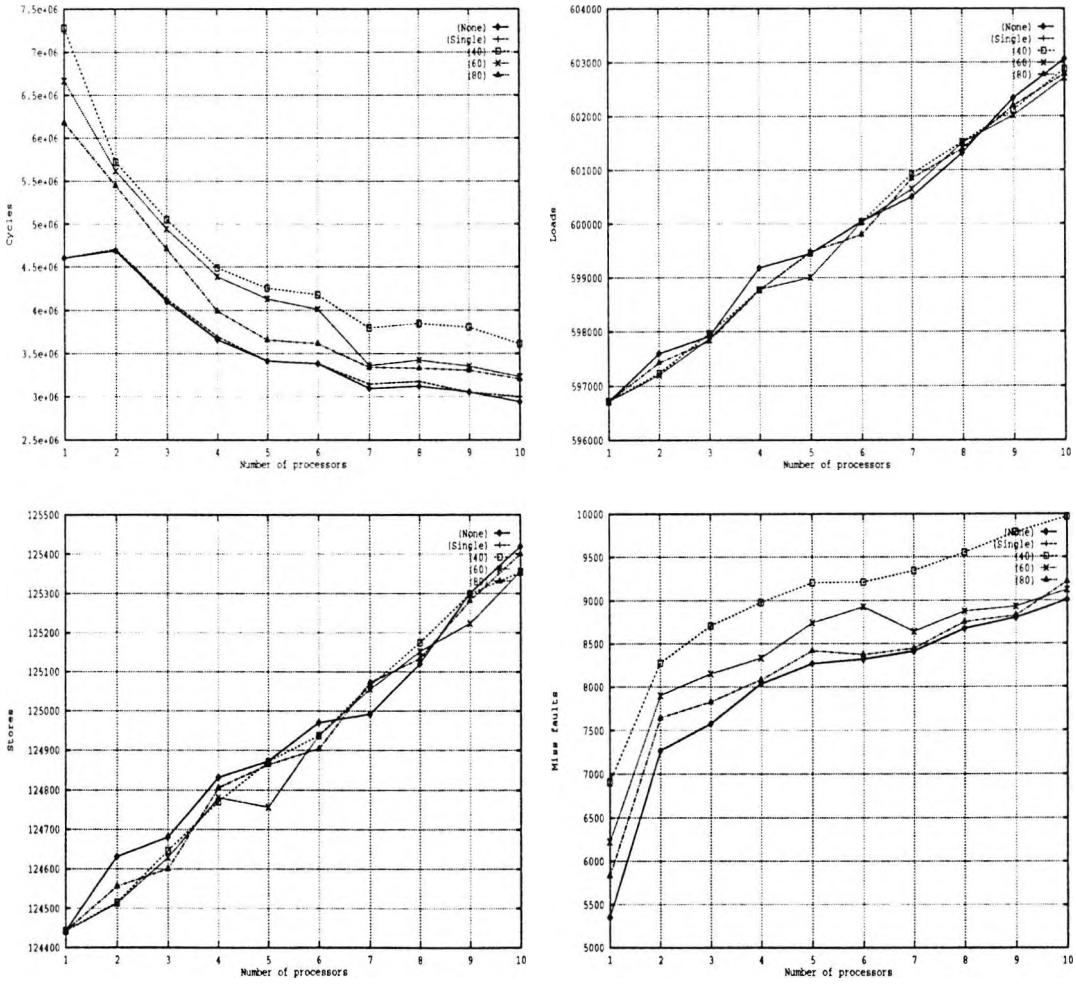

Checkpointed pages
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

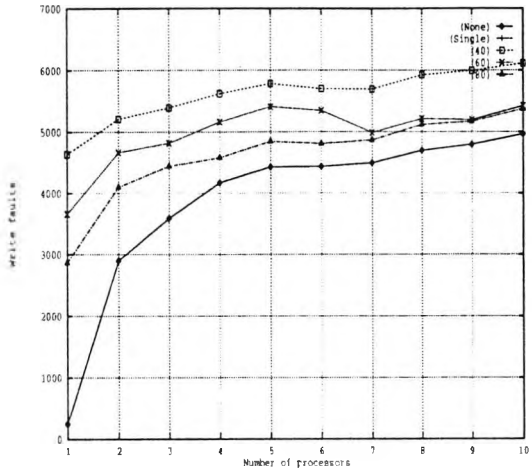

Tlb hits
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

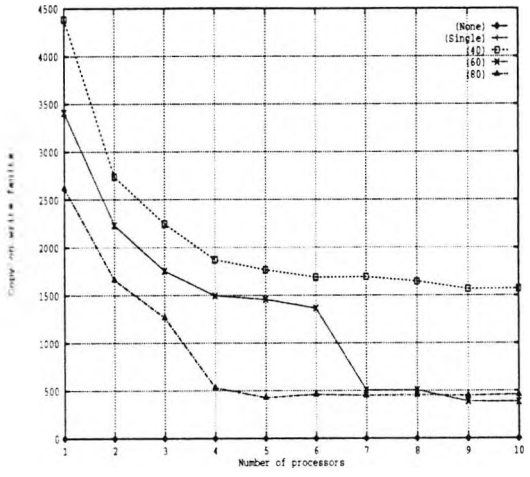

Checkpoints
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

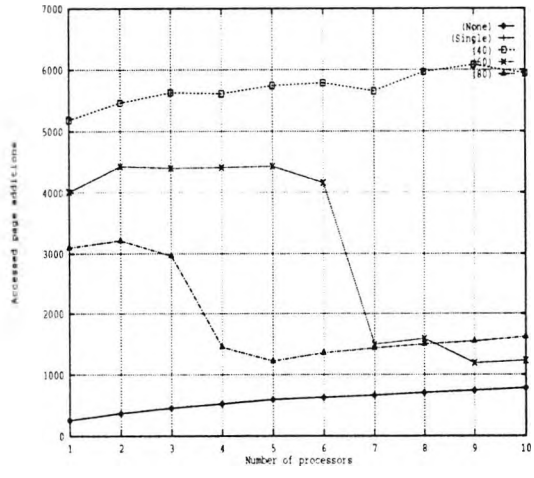
B.1.4 320 molecules – Access table size

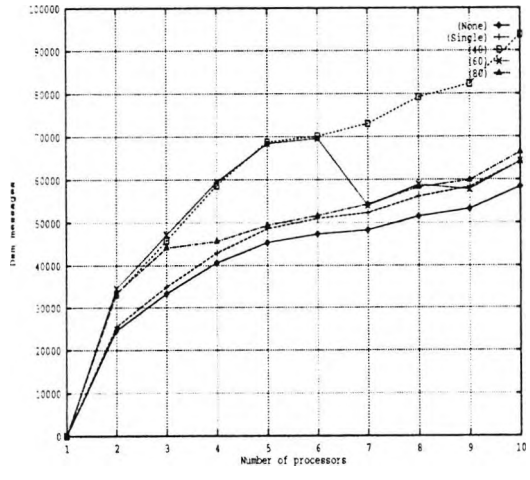
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

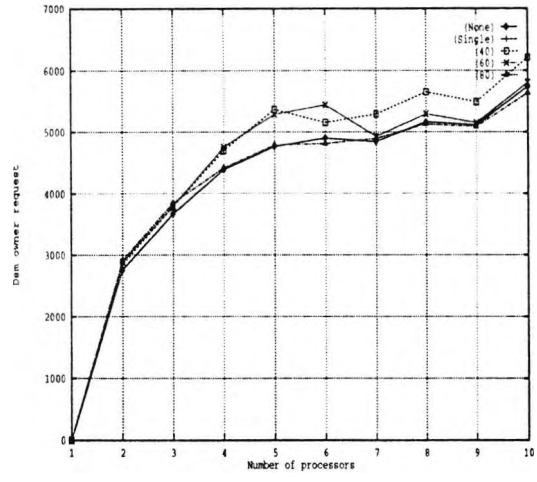


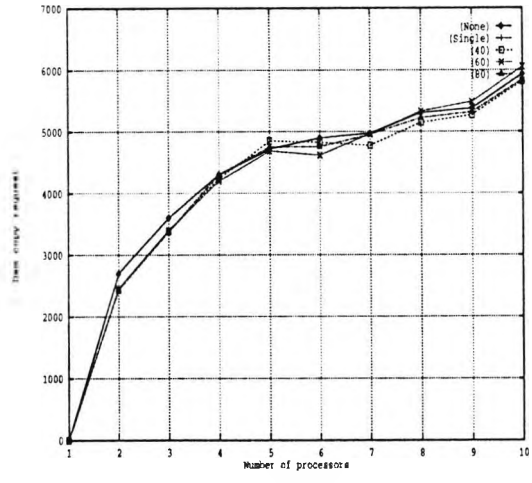

(None)
(Single)
(40)
(60)
(80)
write faults
Number of processors

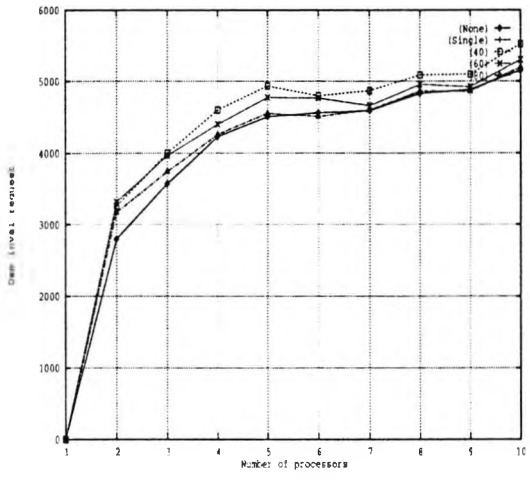

(None)
(Single)
(40)
(60)
(80)
Number of processors

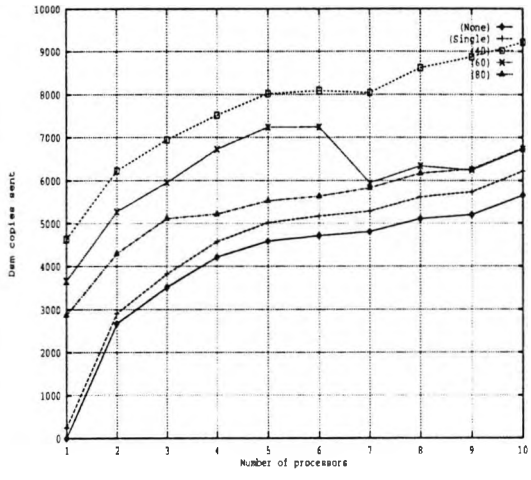

(None)
(Single)
(40)
(60)
(80)
Accessed page additions
Number of processors

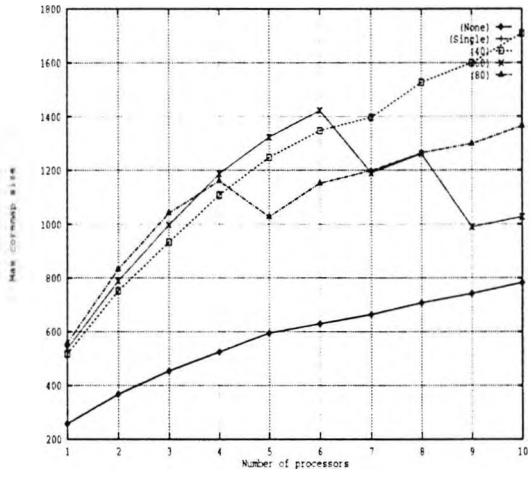

(None)
(Single)
(40)
(60)
(80)
Dam messages
Number of processors

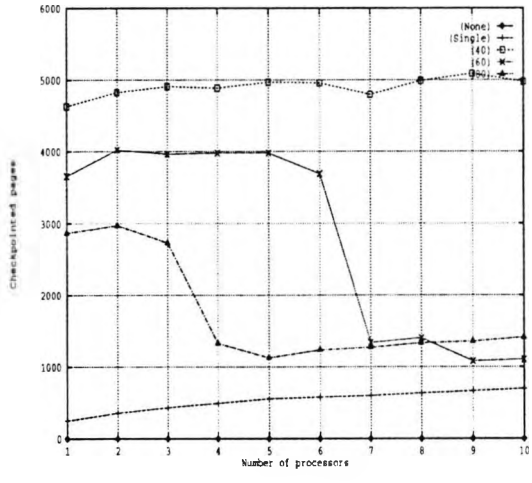

(None)
(Single)
(40)
(60)
(80)
Dam owner request
Number of processors

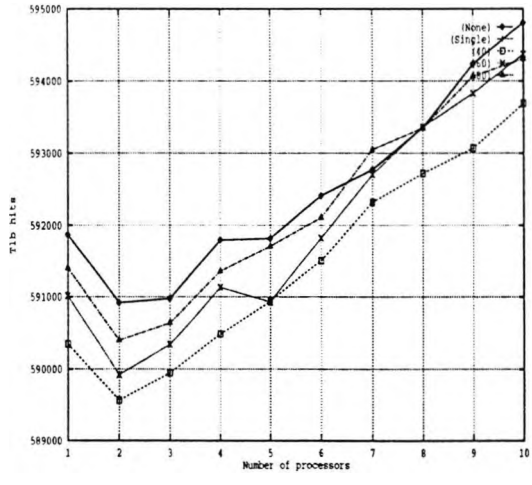

(None)
(Single)
(40)
(60)
(80)
Dam copy request
Number of processors

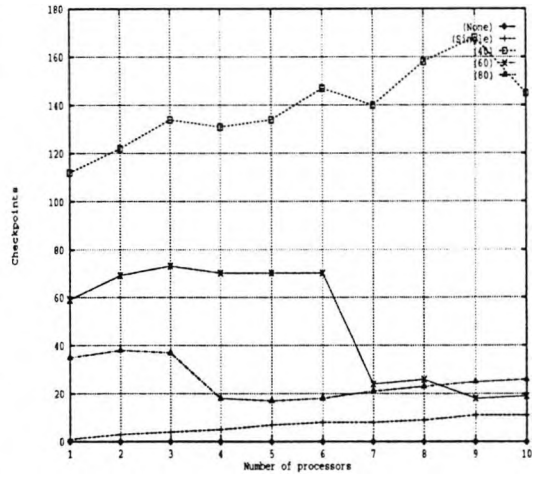

Dem level removes
Number of processors
(None)
(Single)
(40)
(60)
(80)


Dem copies sent
Number of processors
(None)
(Single)
(40)
(60)
(80)


Max Corresp size
Number of processors
(None)
(Single)
(40)
(60)
(80)

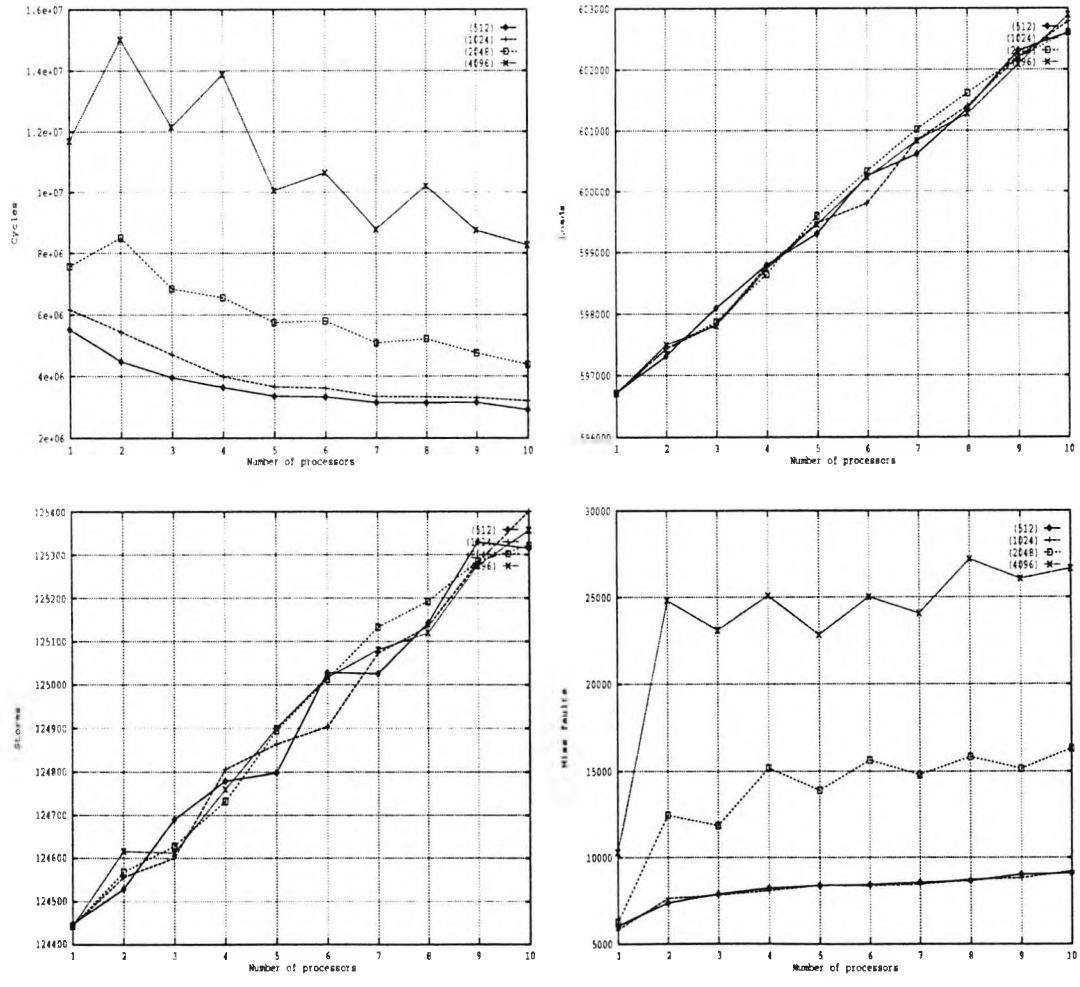

Checkpointed pages
Number of processors
(None)
(Single)
(40)
(60)
(80)


Tlb hits
Number of processors
(None)
(Single)
(40)
(60)
(80)

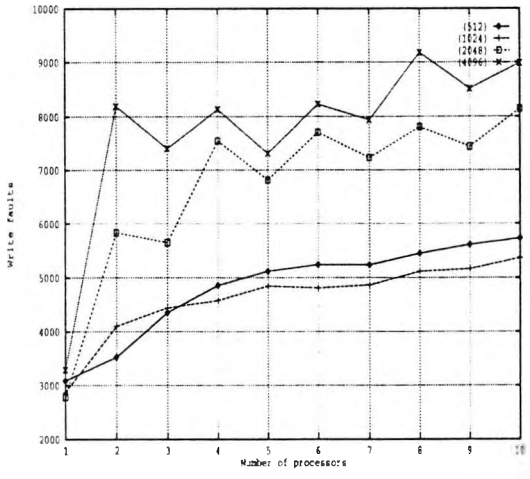

Checkpoints
Number of processors
(None)
(Single)
(40)
(60)
(80)

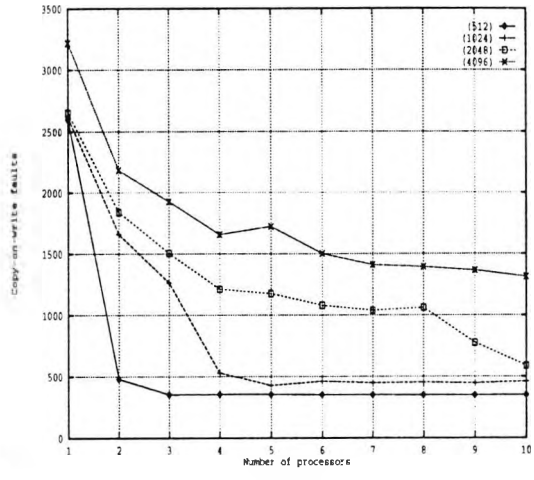
B.1.5 320 molecules – Page size

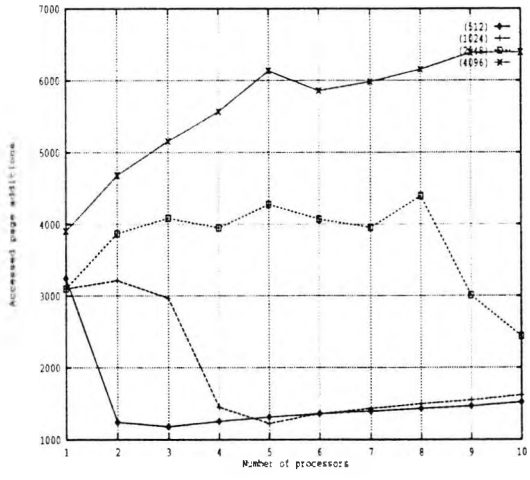
One to ten clusters,
One processor per cluster,
80 entry access table,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

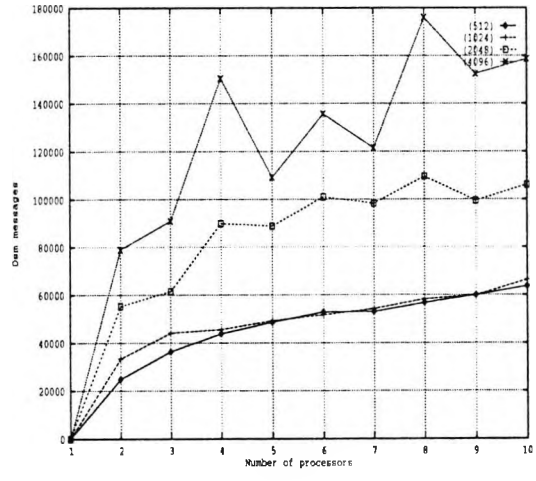


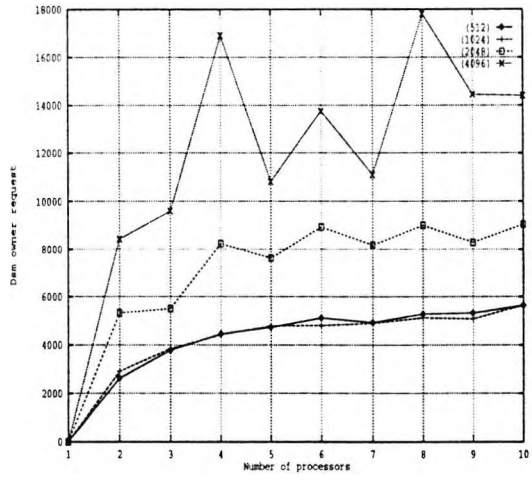
B.1. MP3D

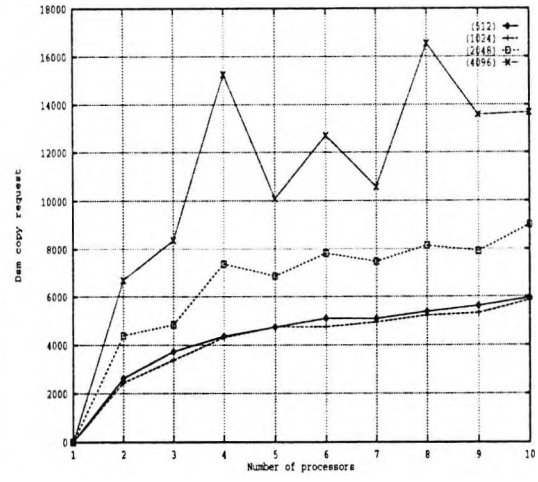

Write faults
Number of processors
(512)
(1024)
(2048)
(4096)

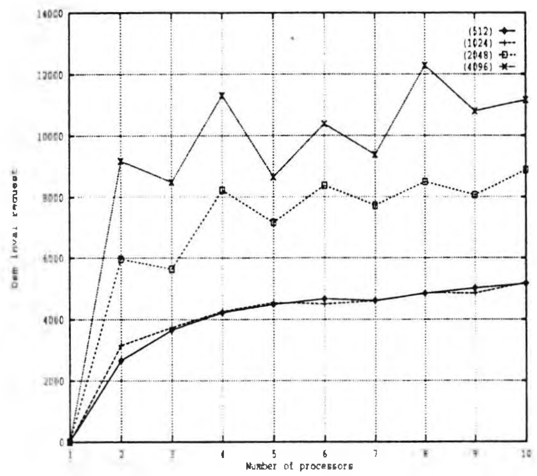

Copy-on-write faults
Number of processors
(512)
(1024)
(2048)
(4096)

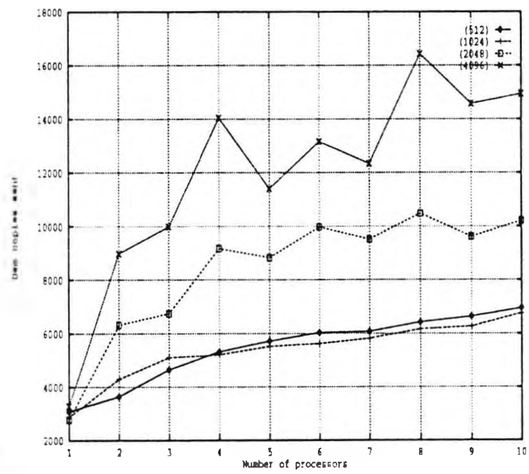

Accessed page additions
Number of processors
(512)
(1024)
(2048)
(4096)

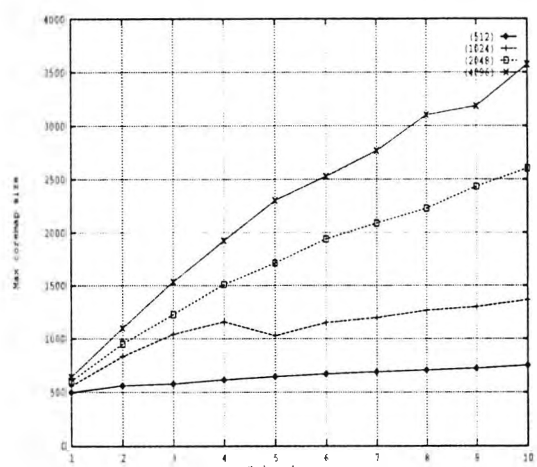

Dsm messages
Number of processors
(512)
(1024)
(2048)
(4096)

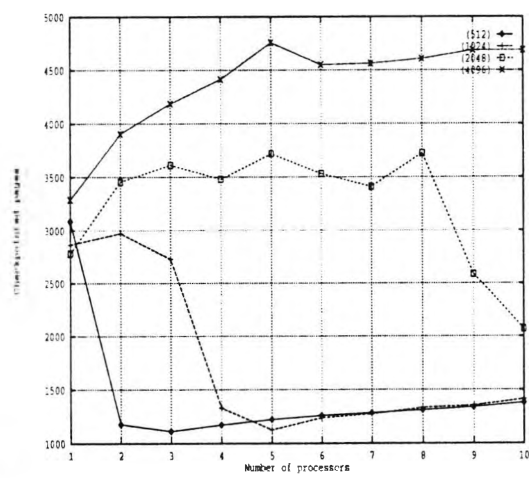

Dsm owner request
Number of processors
(512)
(1024)
(2048)
(4096)

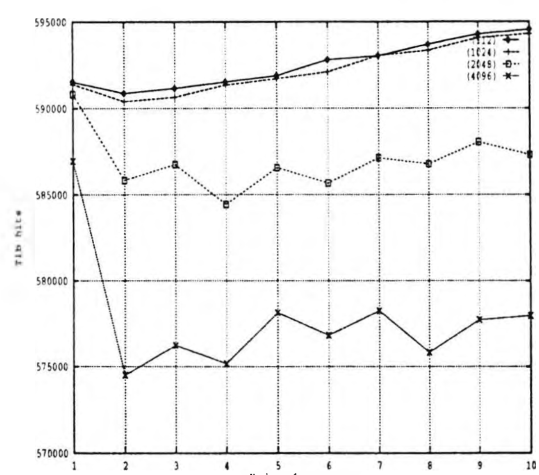

Dsm copy request
Number of processors
(512)
(1024)
(2048)
(4096)

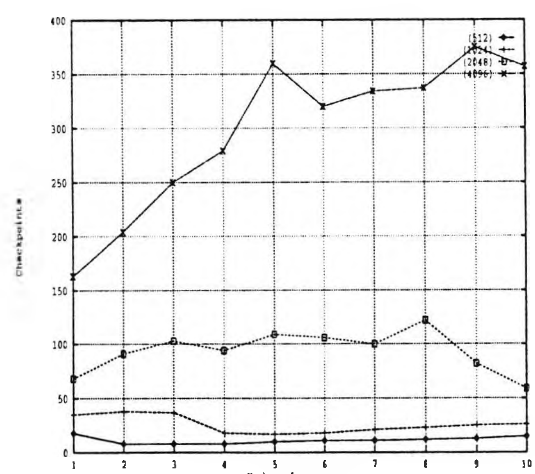

(512)
(1024)
(2048)
(4096)
Number of processors


(512)
(1024)
(2048)
(4096)
Number of processors


(512)
(1024)
(2048)
(4096)
Max coremap size
Number of processors

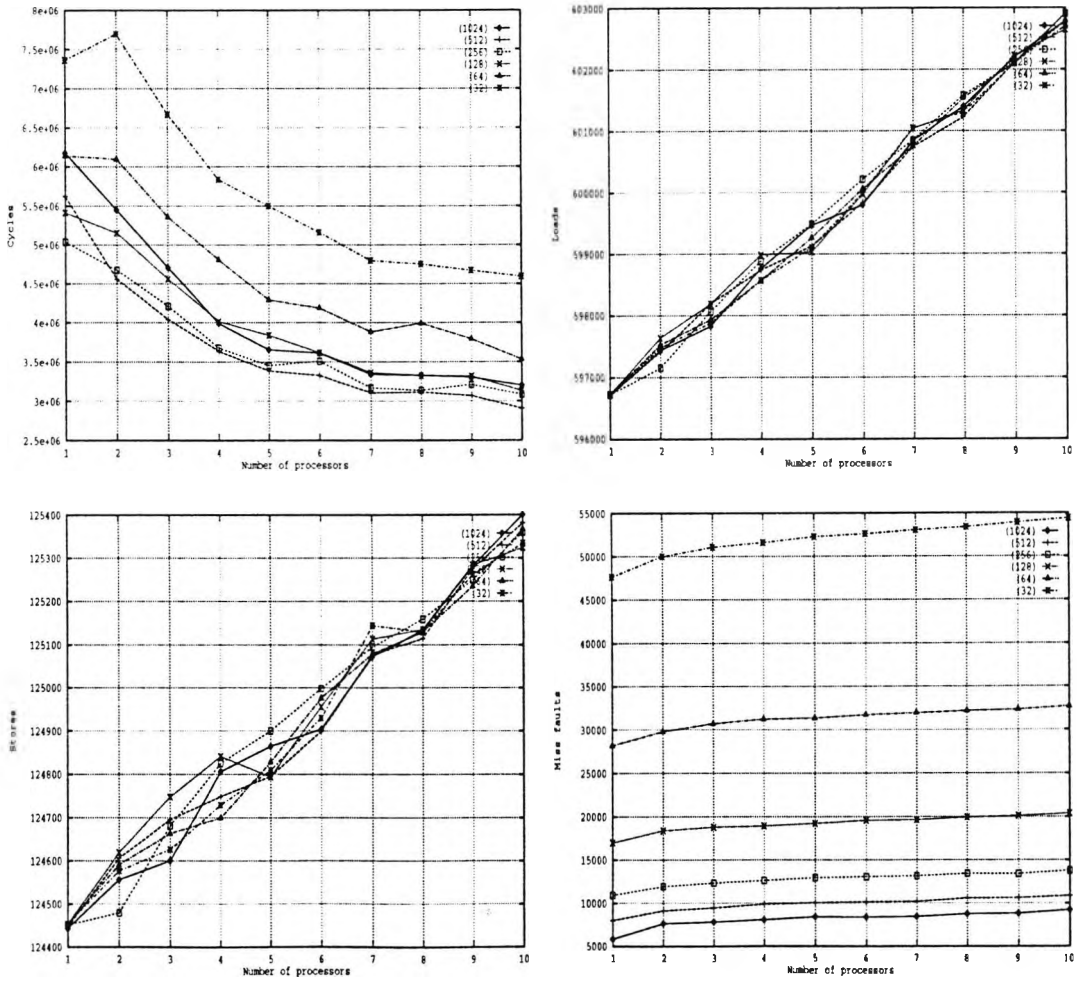

(512)
(1024)
(2048)
(4096)
Number of processors

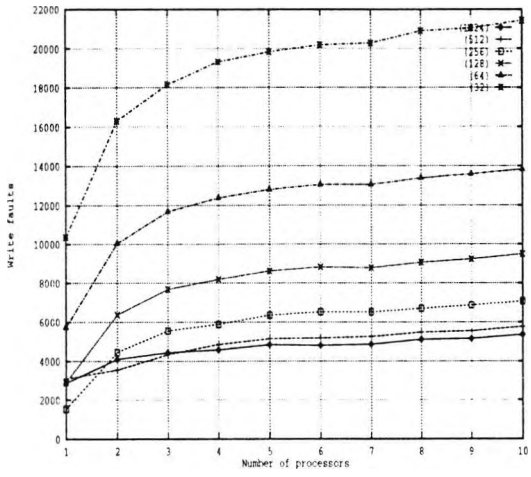

(512)
(1024)
(2048)
(4096)
Tlb hits
Number of processors

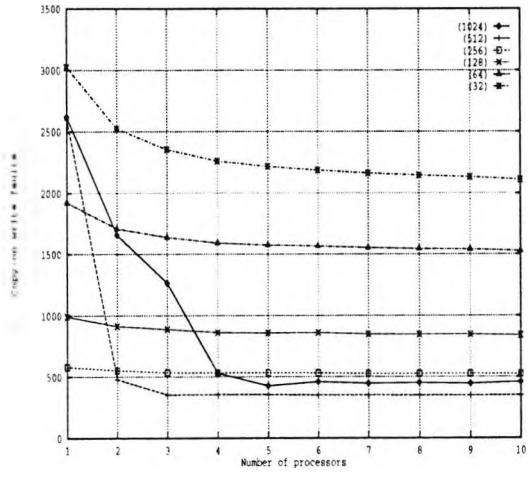

(512)
(1024)
(2048)
(4096)
Checkpoints
Number of processors

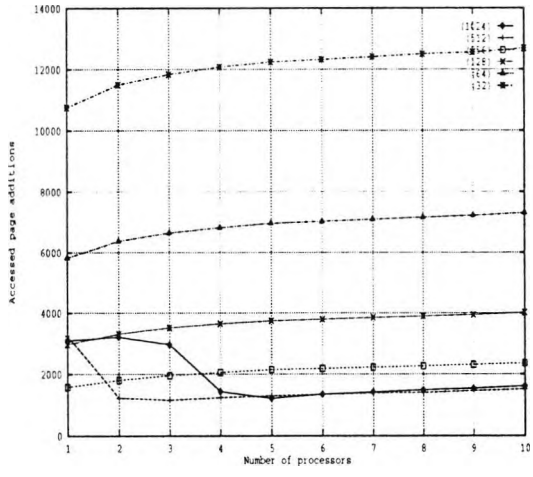
B.1.6 320 molecules – Small page size

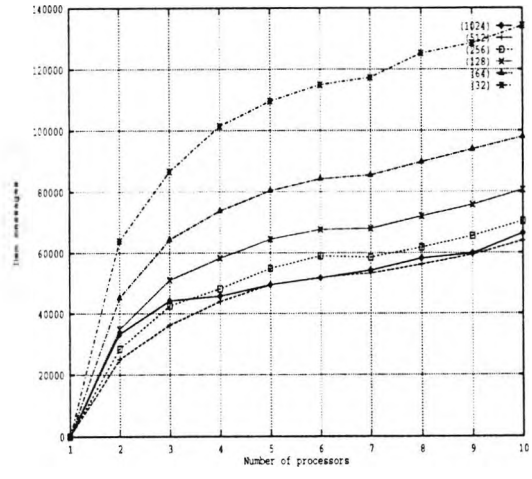
One to ten clusters,
One processor per cluster,
80 entry access table,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

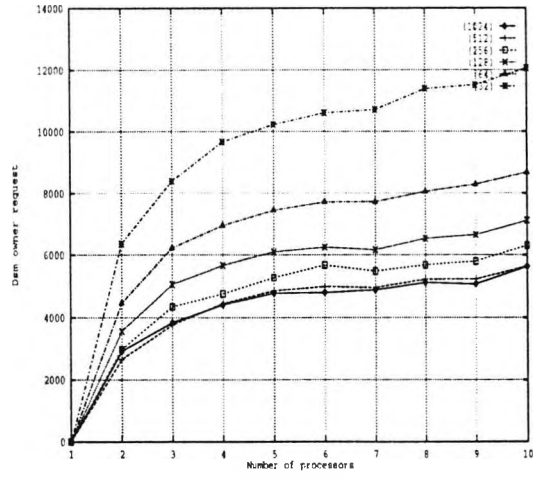


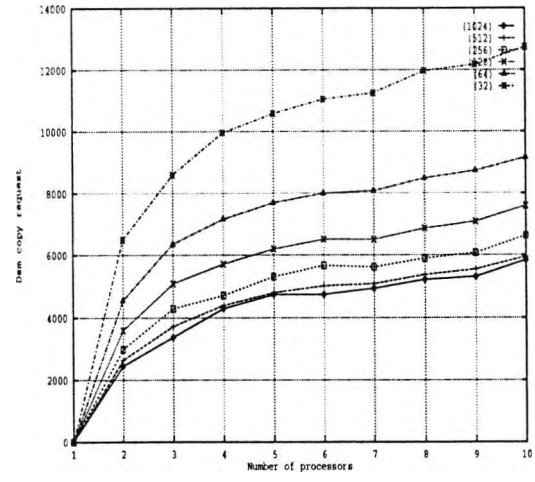

Write faults
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)


Copy on write faults
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

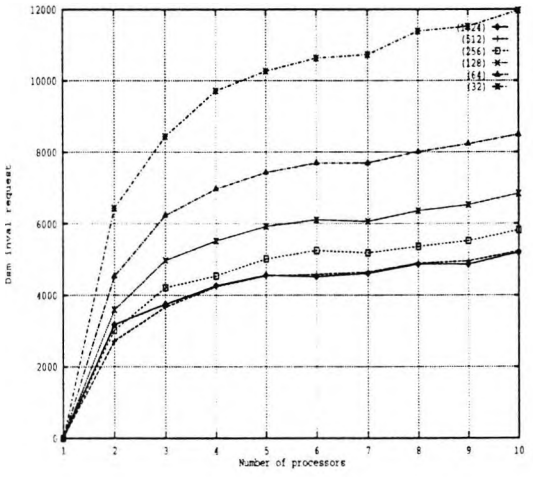

Accessed page additions
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

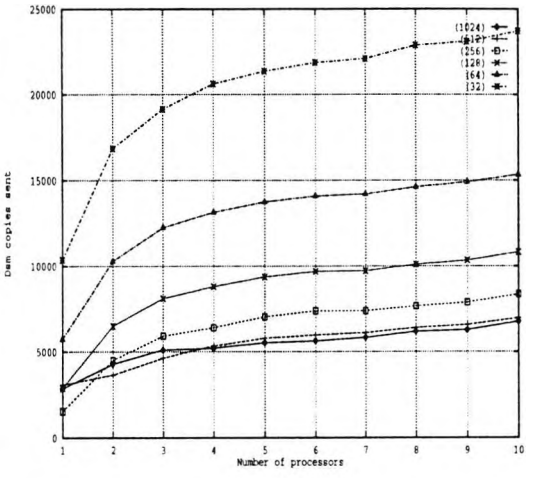

Dsm messages
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

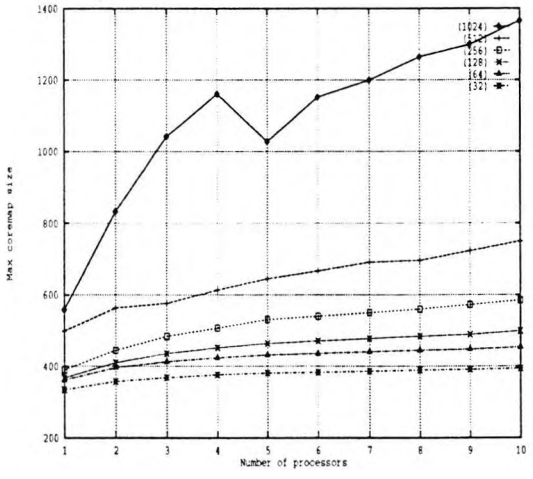

Dsm owner request
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

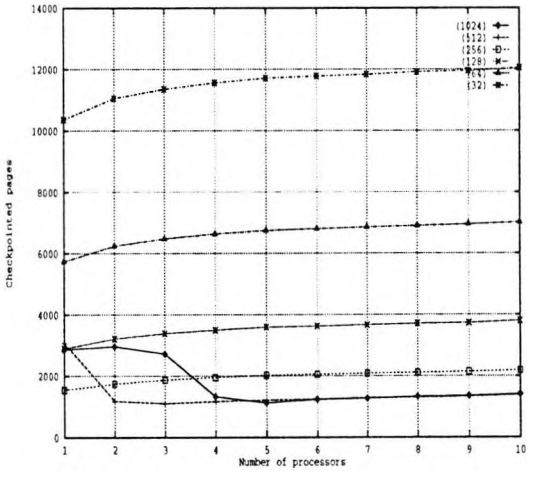

Dsm copy request
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

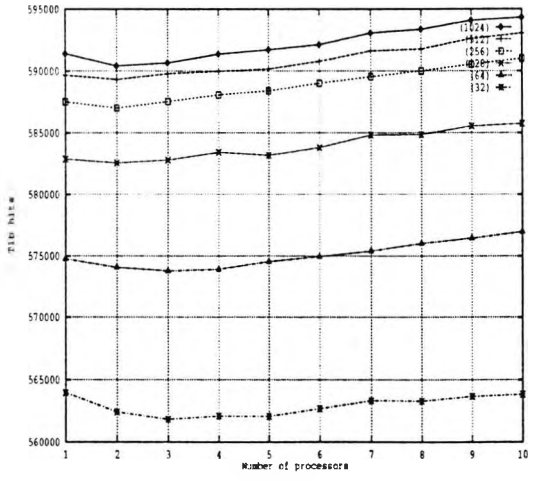
B.1. MP3D

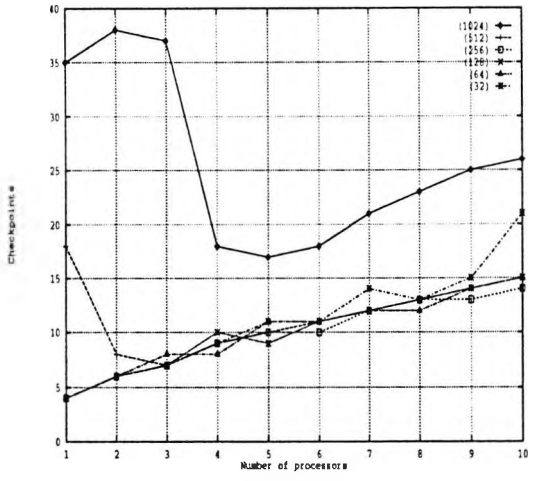

Dam inval request
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)


Dam copies sent
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)


Max coremap size
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)


Checkpointed pages
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

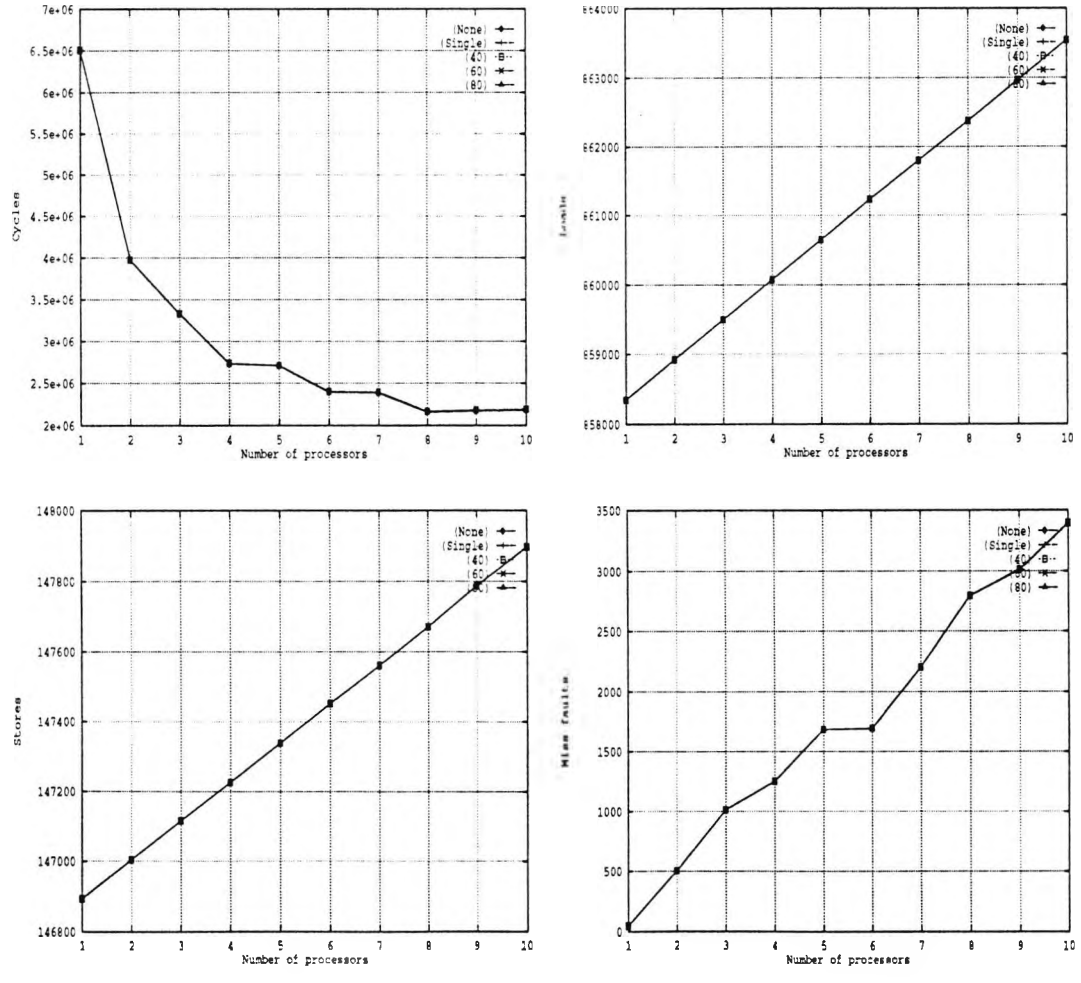

TLB hits
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

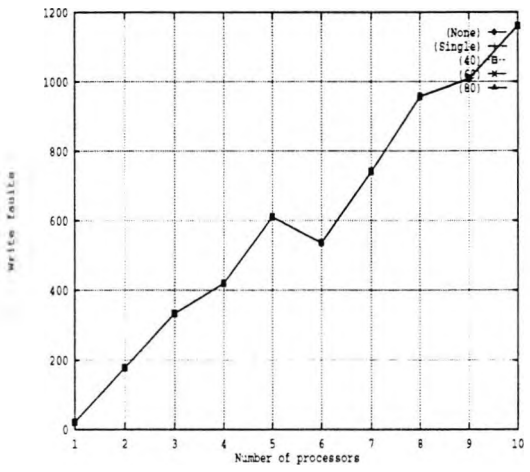

Checkpoints
Number of processors
(1024)
(512)
(256)
(128)
(64)
(32)

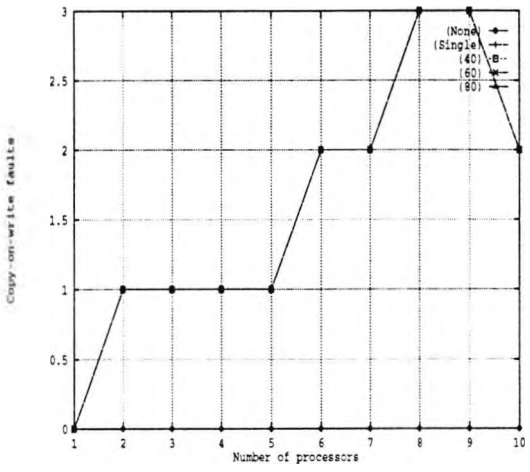
B.2 Water

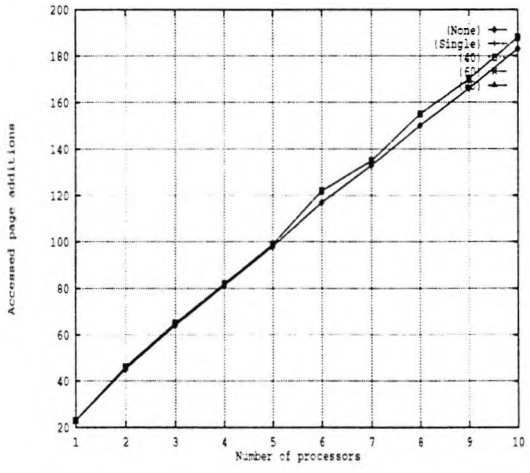
B.2.1 16 molecules – Access table size

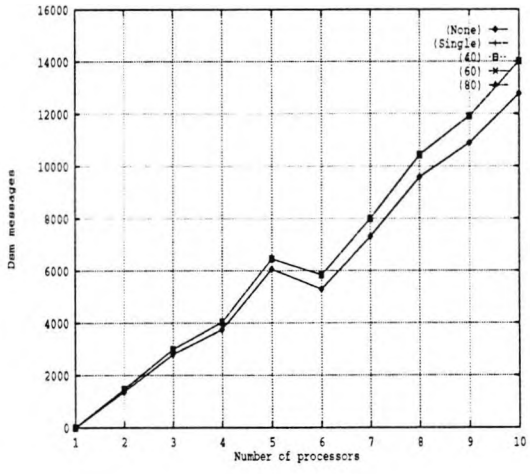
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

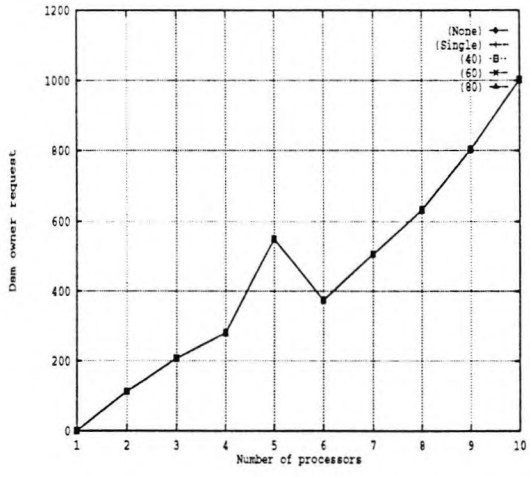


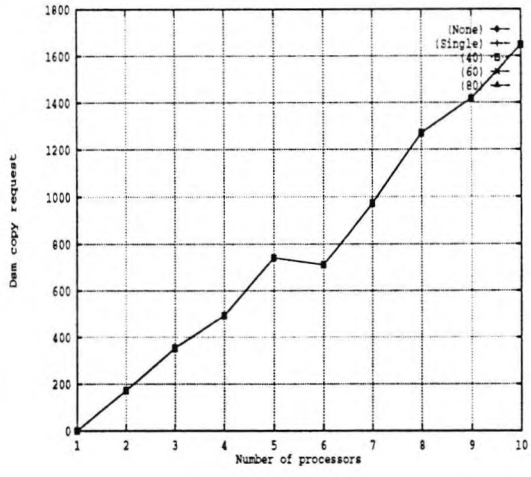

Write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

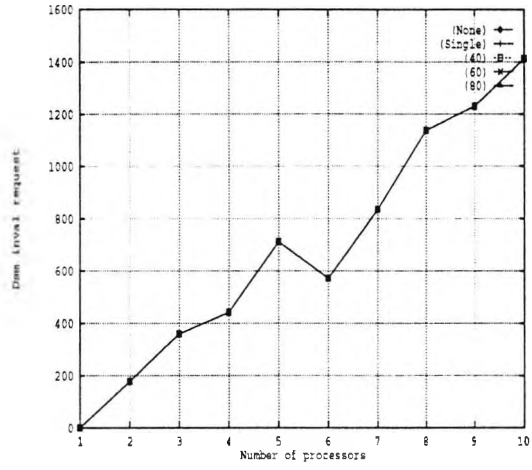

Copy-on-write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

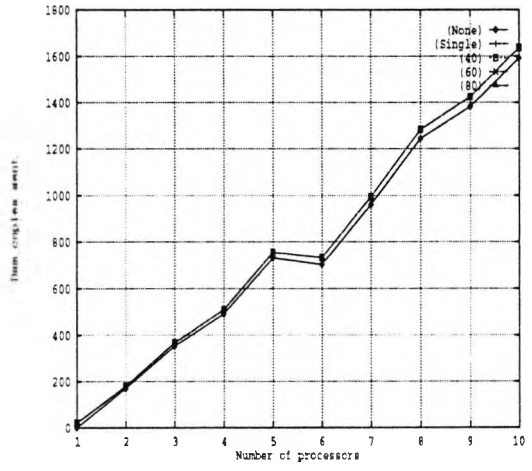

Accessed page additions
Number of processors
(None)
(Single)
(40)
(60)
(80)

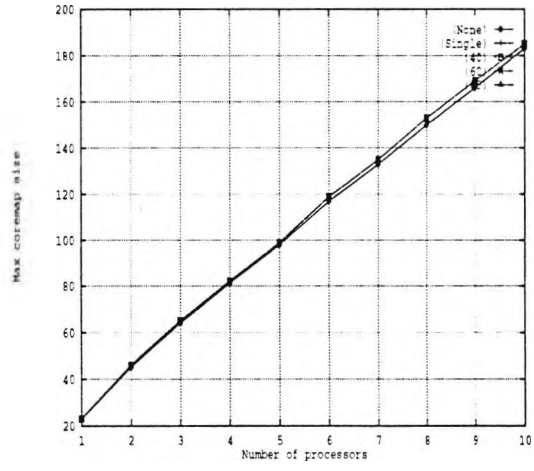

Dsm messages
Number of processors
(None)
(Single)
(40)
(60)
(80)

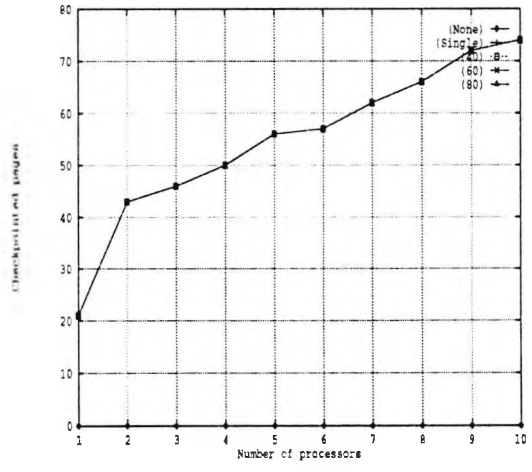

Dsm owner request
Number of processors
(None)
(Single)
(40)
(60)
(80)

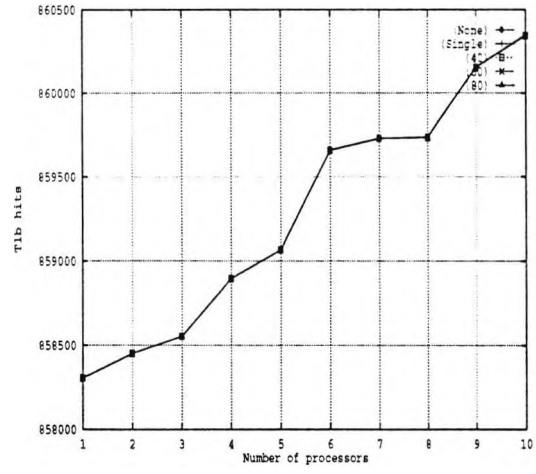

Dsm copy request
Number of processors
(None)
(Single)
(40)
(60)
(80)

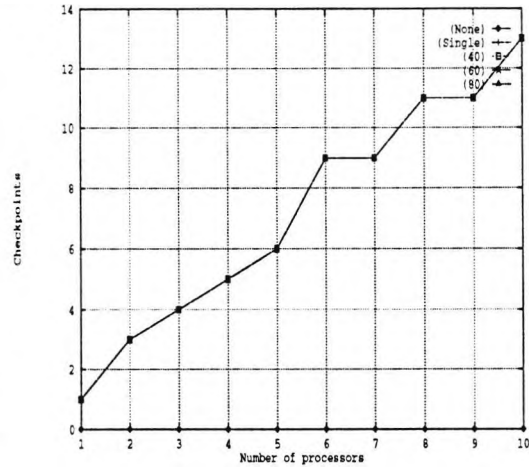

Dsm inval request
Number of processors
(None)
(Single)
(40)
(60)
(80)


Dsm copies sent
Number of processors
(None)
(Single)
(40)
(60)
(80)


Max coremap size
Number of processors
(None)
(Single)

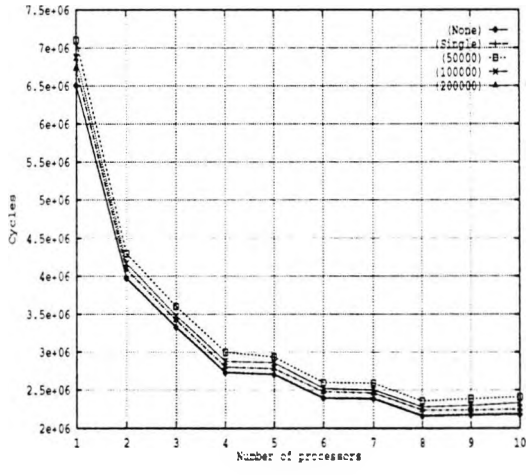

Checkpointed pages
Number of processors
(None)
(Single)
(40)
(60)
(80)

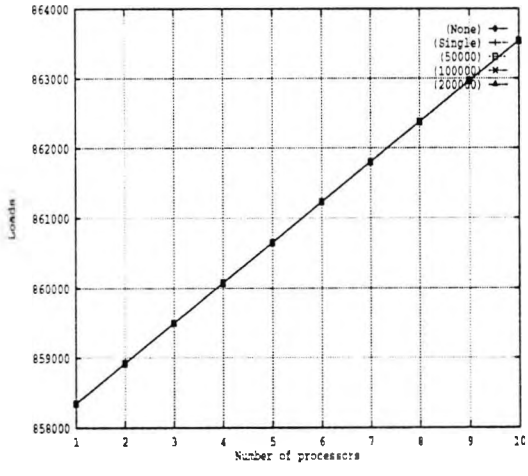

Tlb hits
Number of processors
(None)
(Single)

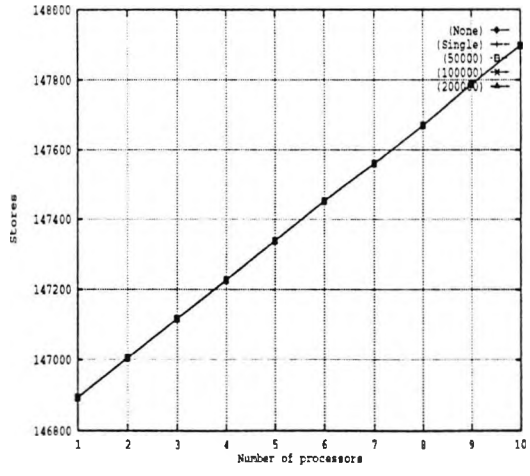

Checkpoints
Number of processors
(None)
(Single)
(40)
(60)
(80)

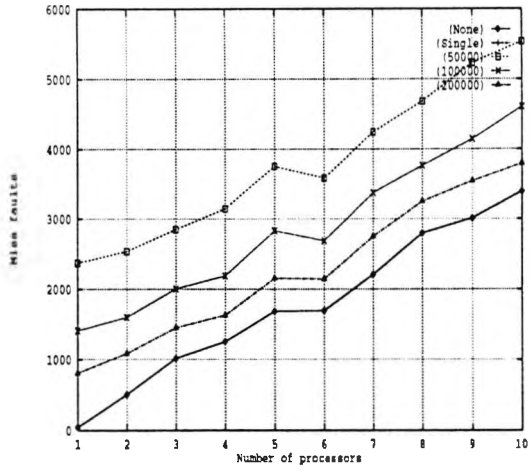
B.2.2 16 molecules – Time based

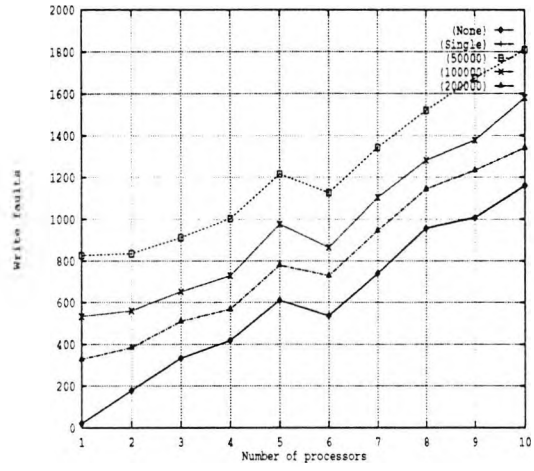
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

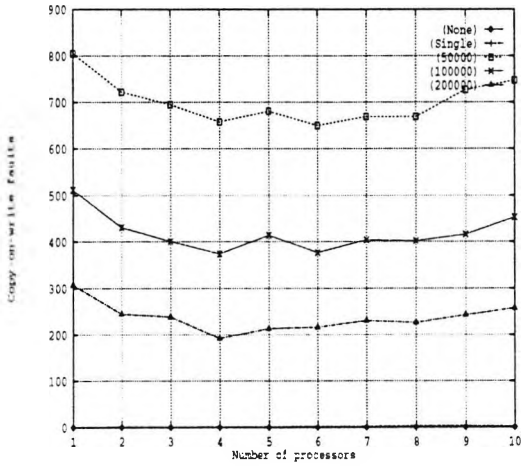


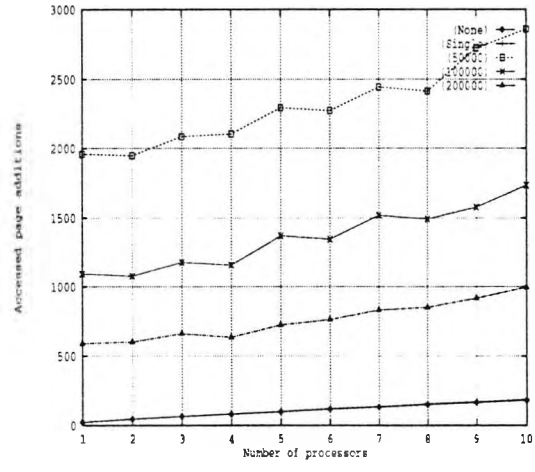


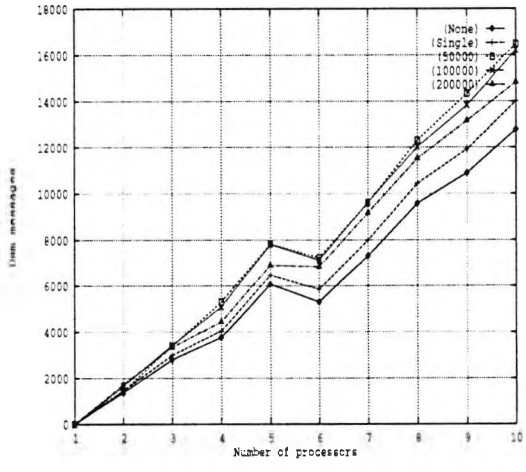


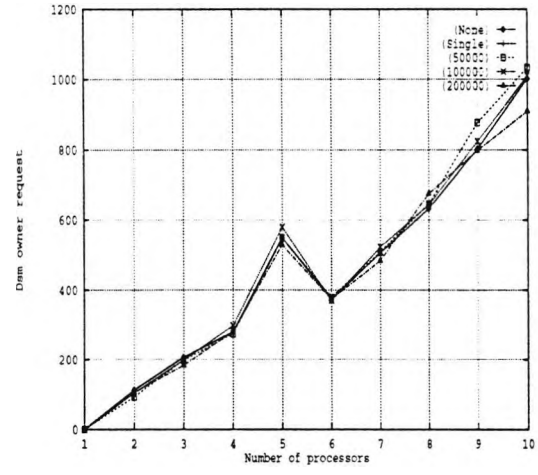


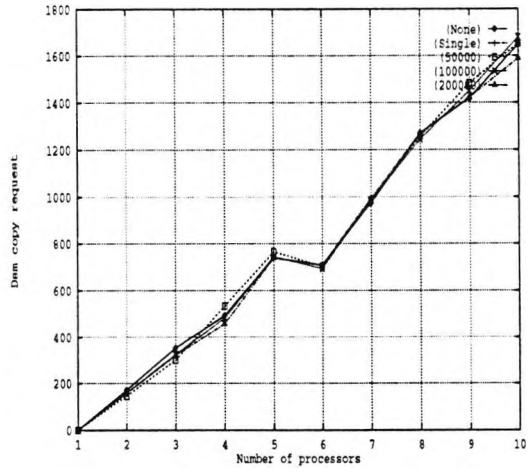

Write faults
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

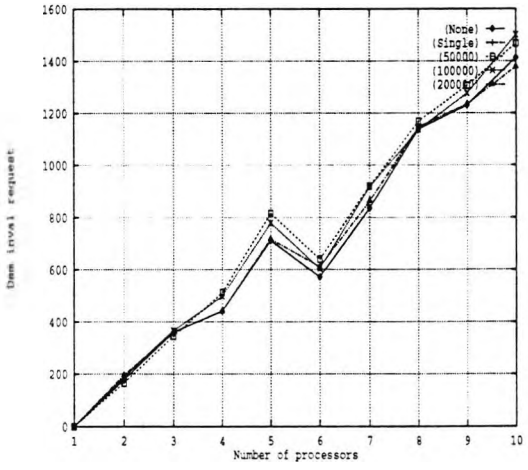

Copy-on-write faults
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

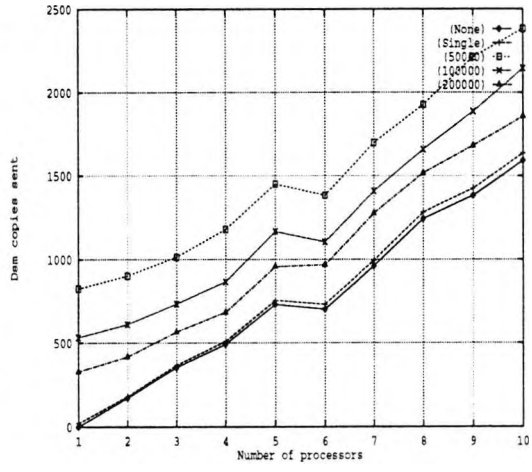

Accessed page additions
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

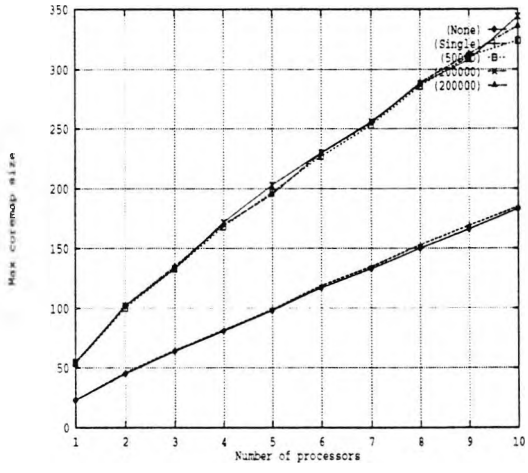

Dsm messages
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

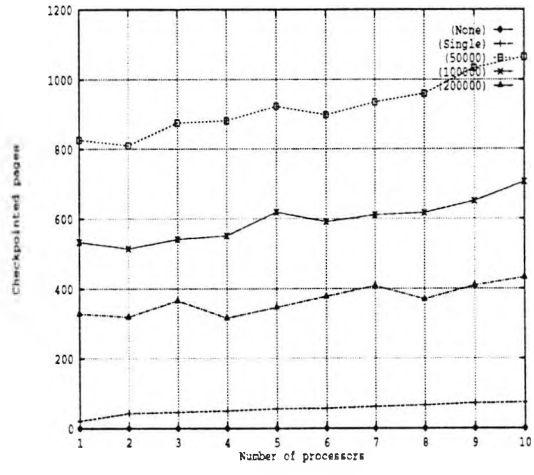

Dsm owner request
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

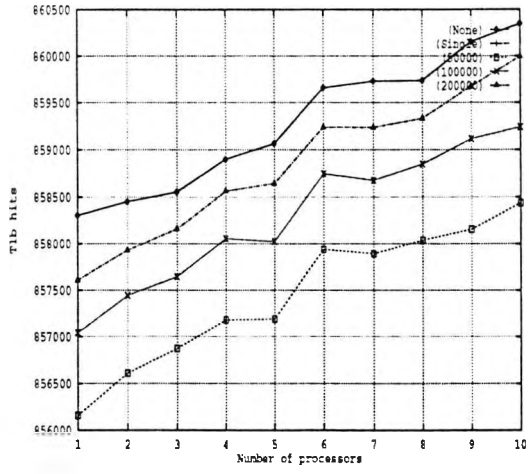

Dsm copy request
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

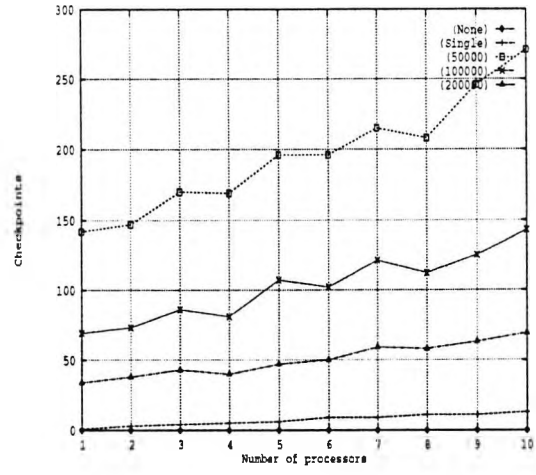

Dam inval request
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)


Dam copies sent
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)


Max coremap size
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

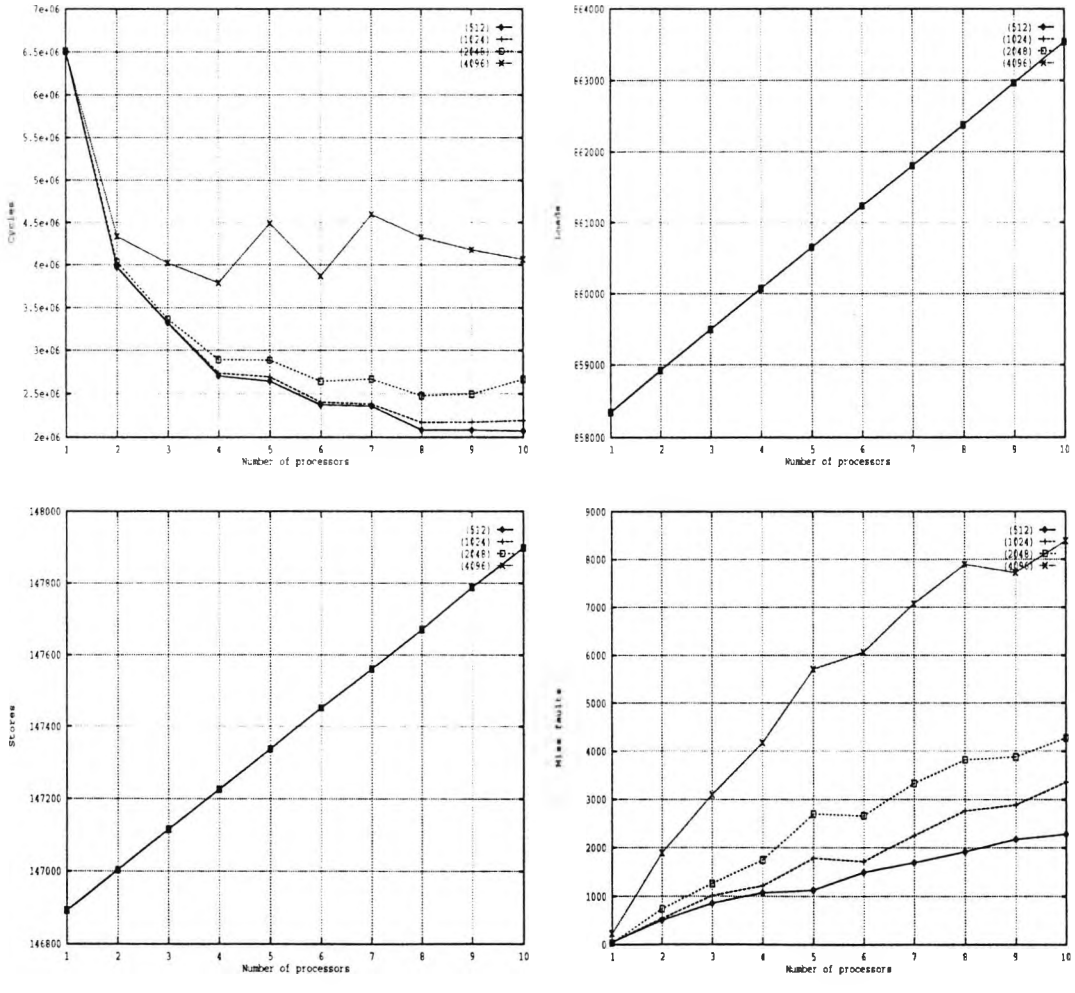

Checkpointed pages
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)


Tlb hits
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

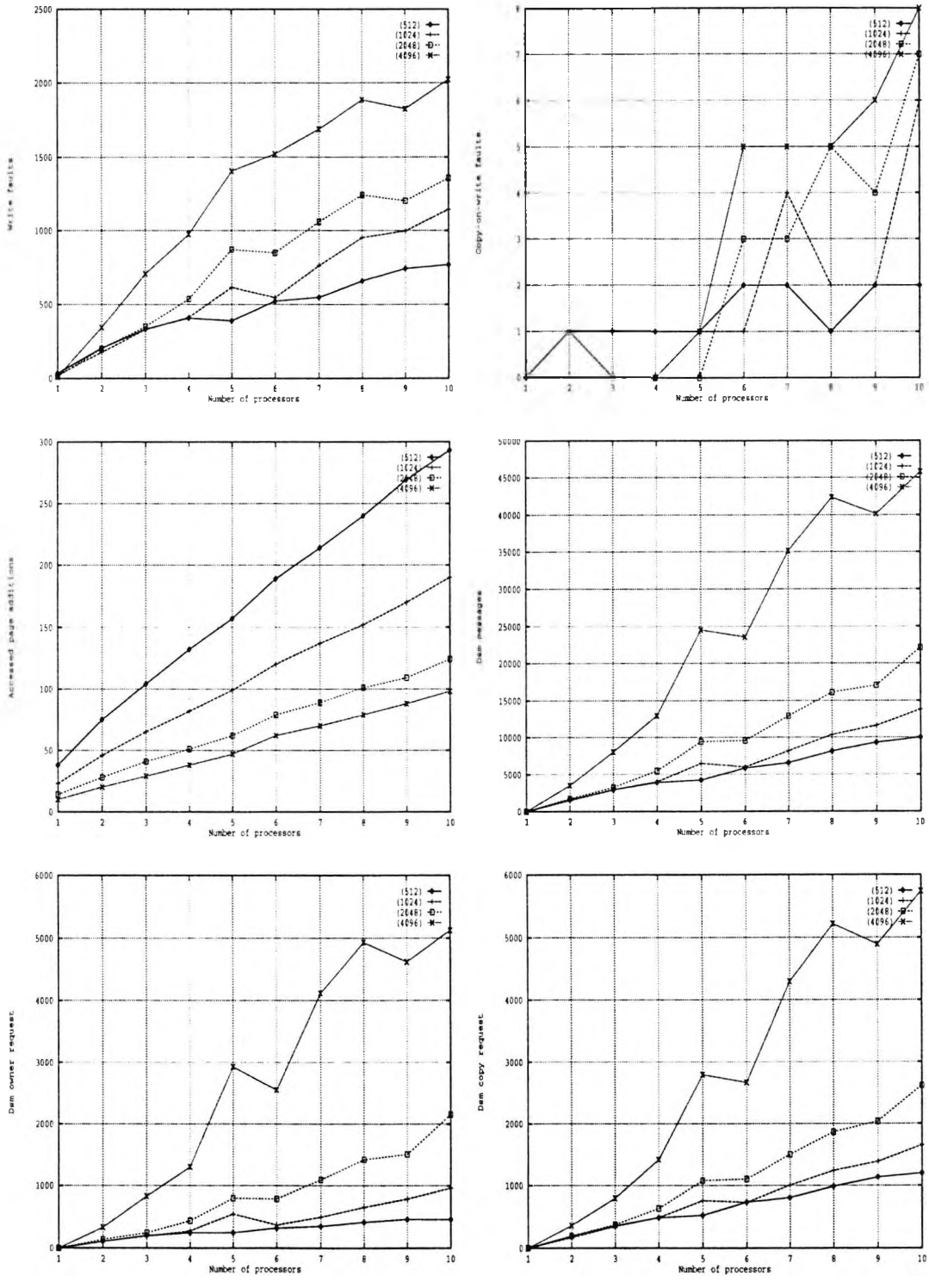

Checkpoints
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

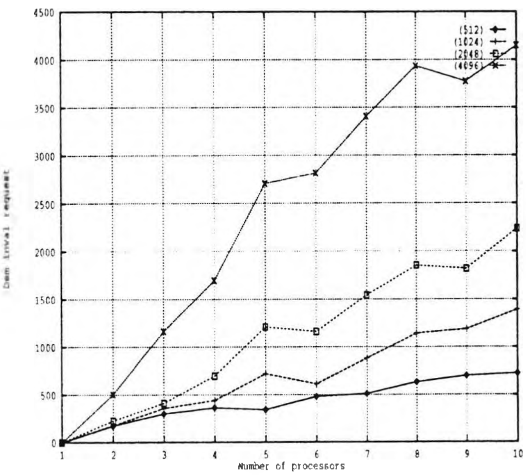
B.2.3 16 molecules – Page size

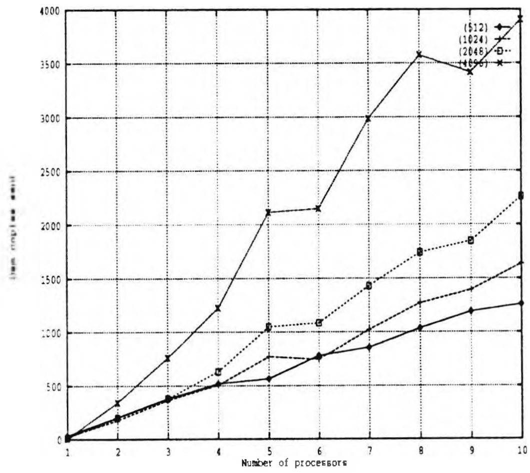
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

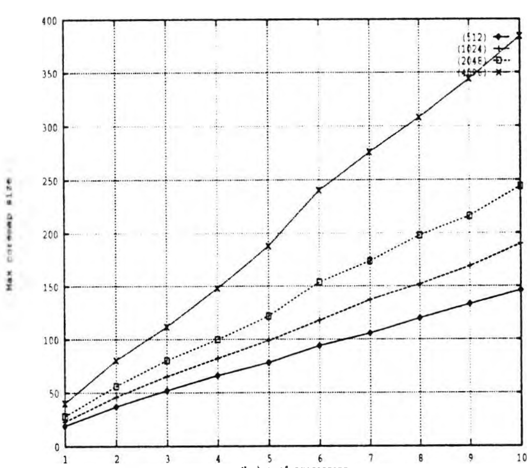


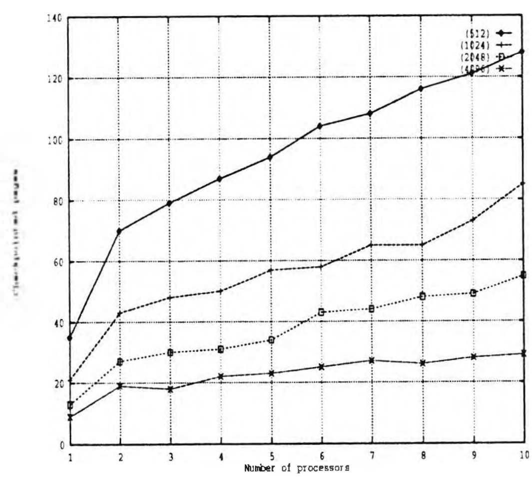
B.2. WATER

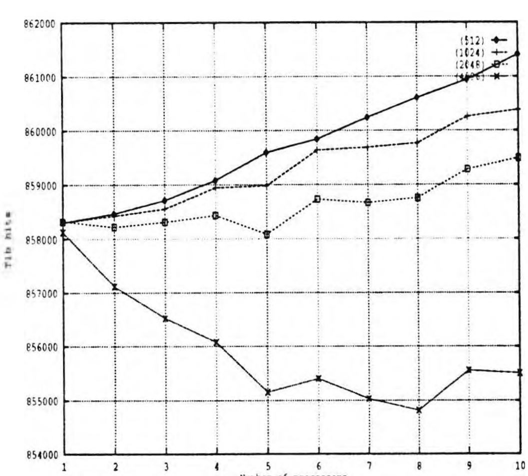

Write faults
Number of processors
(512)
(1024)
(2048)
(4096)
Copy-on-write faults
Number of processors
(512)
(1024)
(2048)
(4096)
Accessed page additions
Number of processors
(512)
(1024)
(2048)
(4096)
Dsm messages
Number of processors
(512)
(1024)
(2048)
(4096)
Dsm owner request
Number of processors
(512)
(1024)
(2048)
(4096)
Dsm copy request
Number of processors
(512)
(1024)
(2048)
(4096)

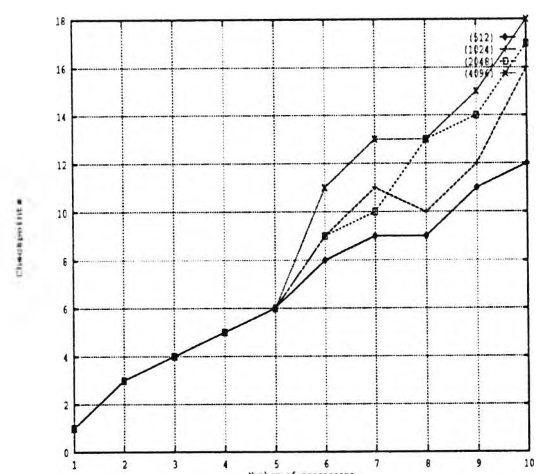

(512)
(1024)
(2048)
(4096)
Number of processors


(512)
(1024)
(2048)
(4096)
Number of processors


(512)
(1024)
(2048)
(4096)
Number of processors

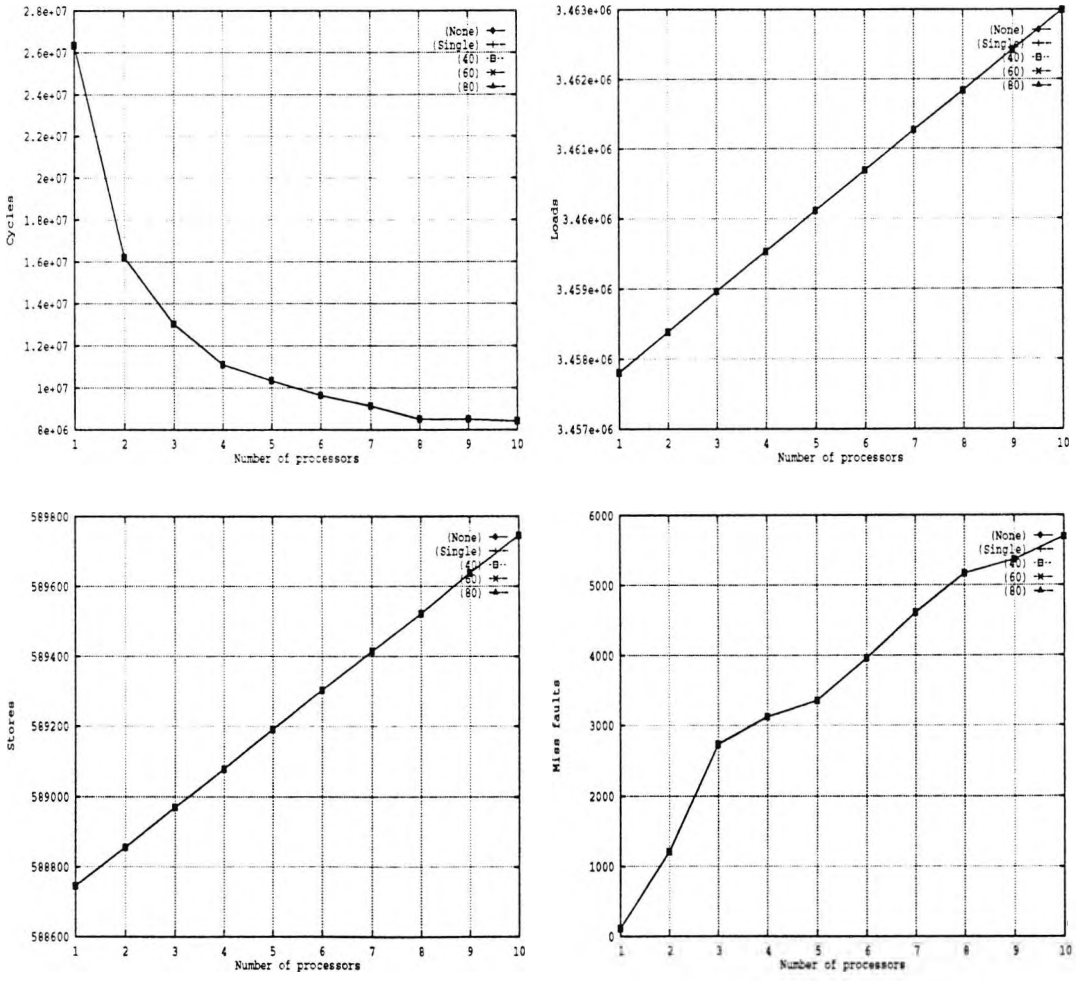

(512)
(1024)
(2048)
(4096)
Number of processors

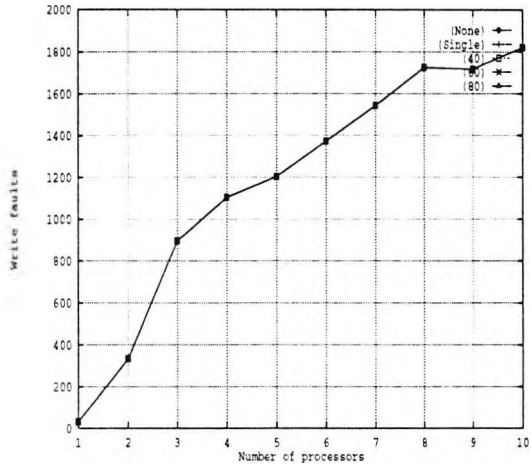

(512)
(1024)
(2048)
(4096)
Number of processors

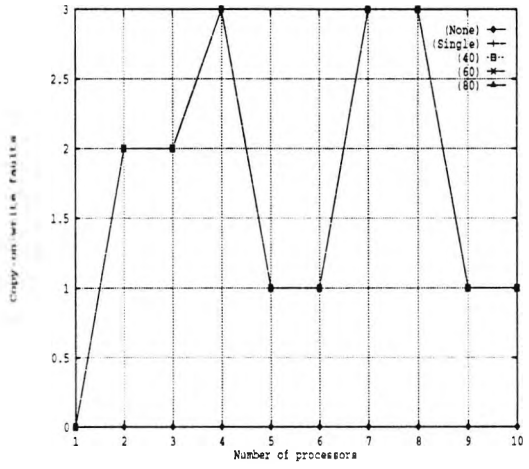

(512)
(1024)
(2048)
(4096)
Number of processors

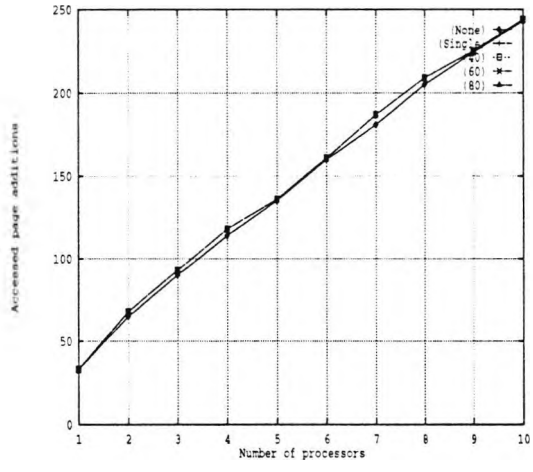
B.2.4 32 molecules – Access table size

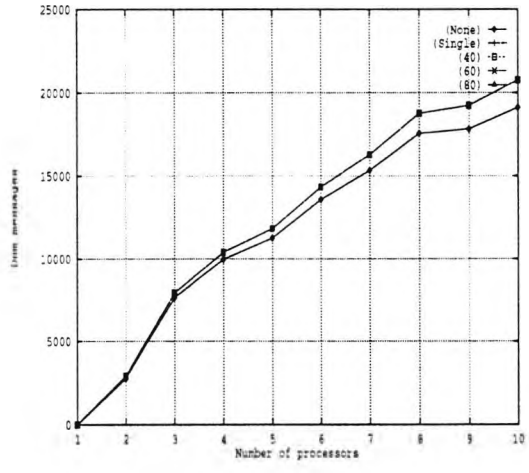
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

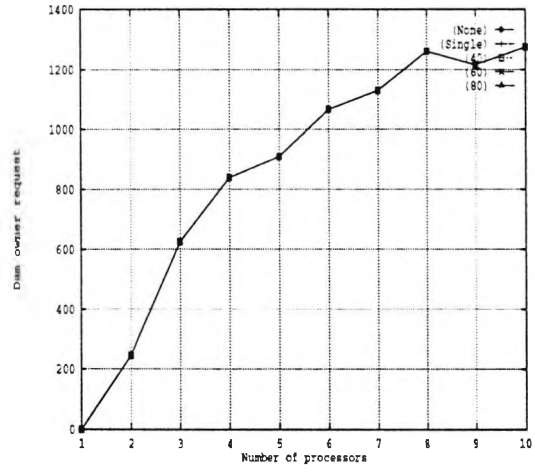


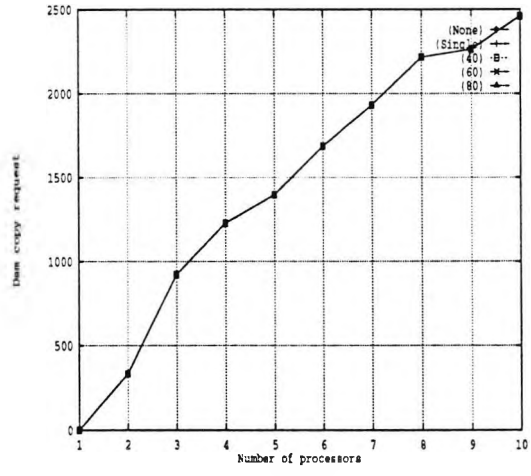

Write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

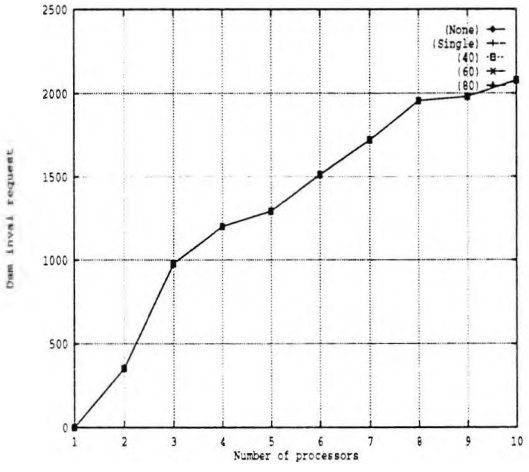

Copy-on-write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

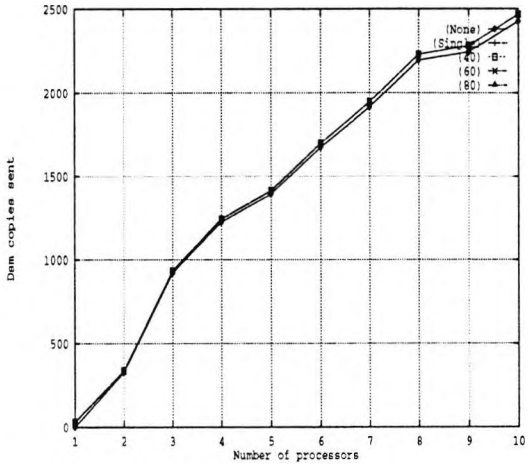

Accessed page additions
Number of processors
(None)
(Single)
(40)
(60)
(80)

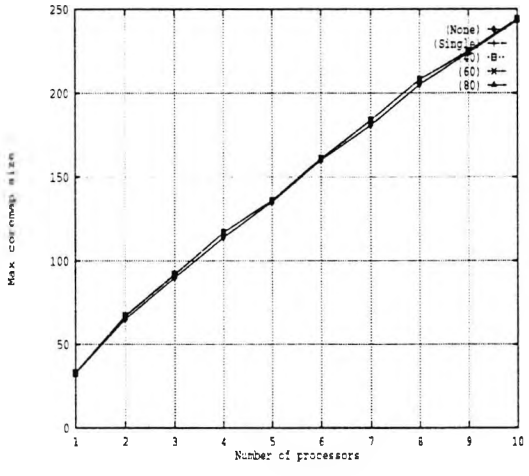

Dsm messages
Number of processors
(None)
(Single)
(40)
(60)
(80)

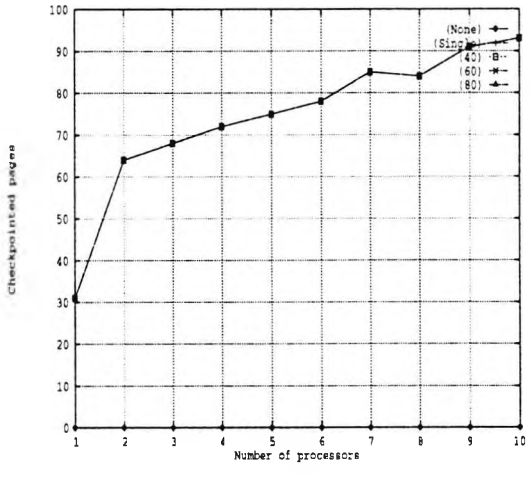

Dsm owner request
Number of processors
(None)
(Single)
(40)
(60)
(80)

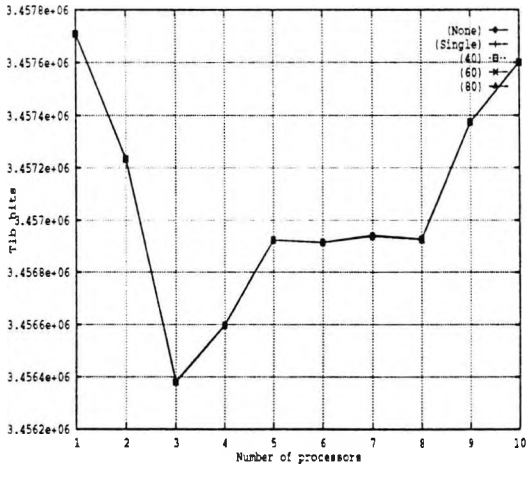

Dsm copy request
Number of processors
(None)
(Single)
(40)
(60)
(80)

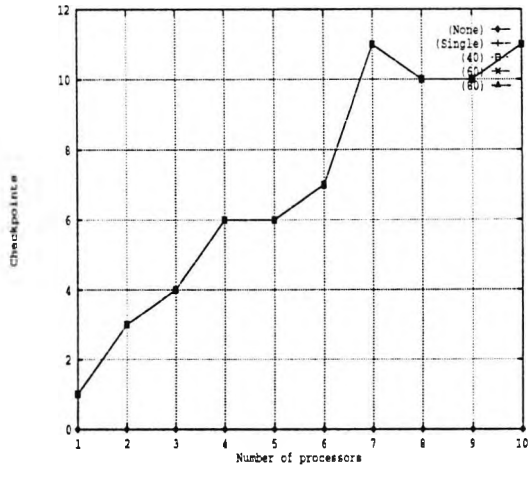

Dam inval request
Number of processors
(None)
(Single)
(40)
(60)
(80)


Dam copies sent
Number of processors
(None)
(Single)
(40)
(60)
(80)


Max commap size
Number of processors
(None)
(Single)
(40)
(60)
(80)

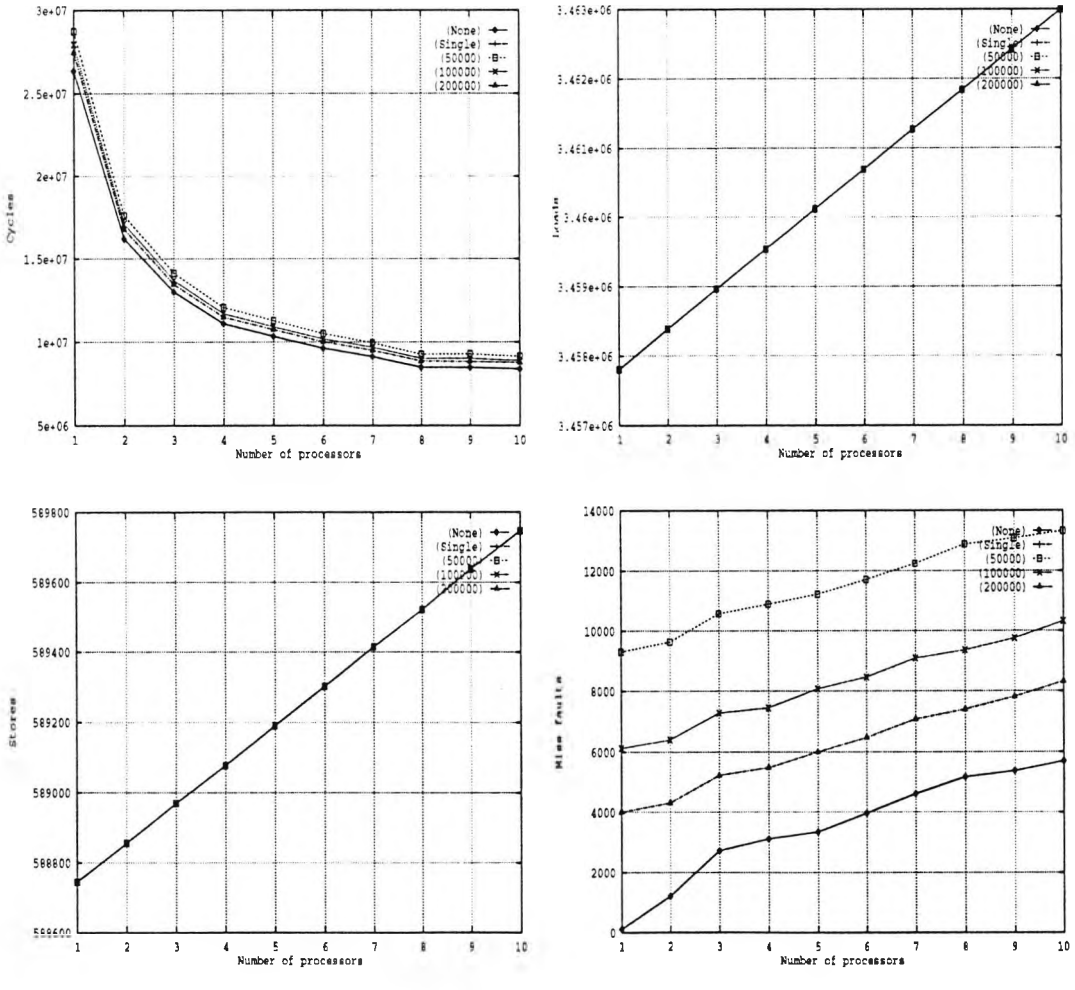

Checkpointed pages
Number of processors
(None)
(Single)
(40)
(60)
(80)

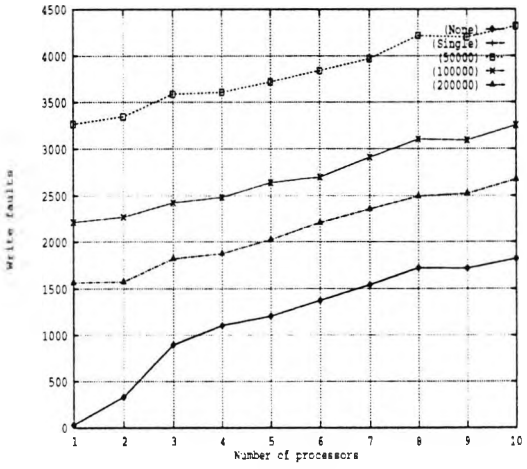

Tlb_hits
Number of processors
(None)
(Single)
(40)
(60)
(80)

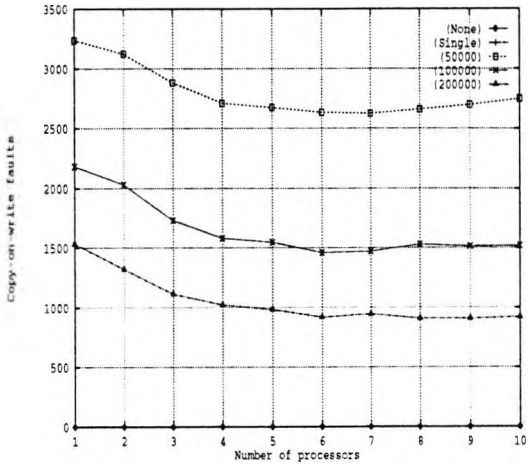

Checkpoints
Number of processors
(None)
(Single)
(40)
(60)
(80)

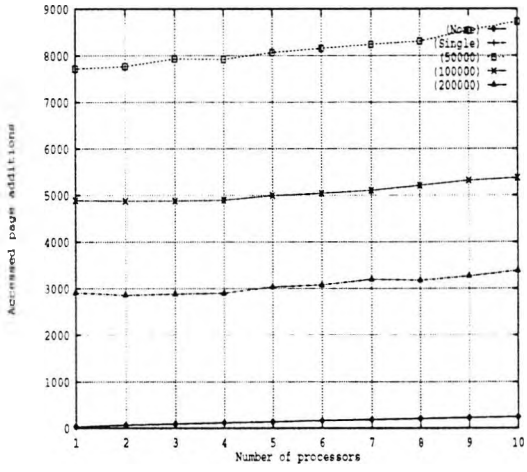
B.2.5 32 molecules – Time based

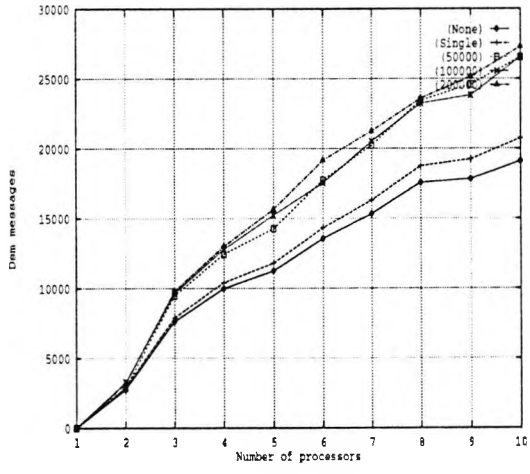
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

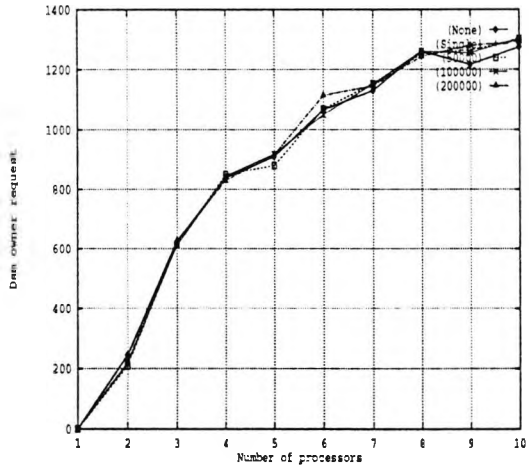


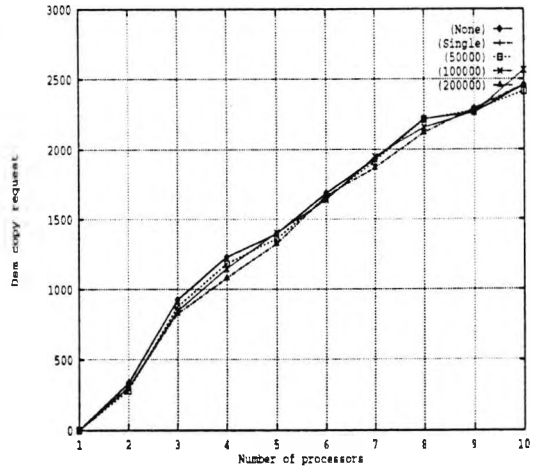

Write faults
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

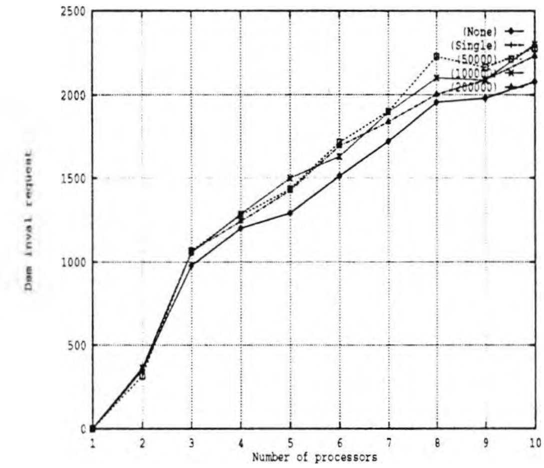

Copy-on-write faults
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

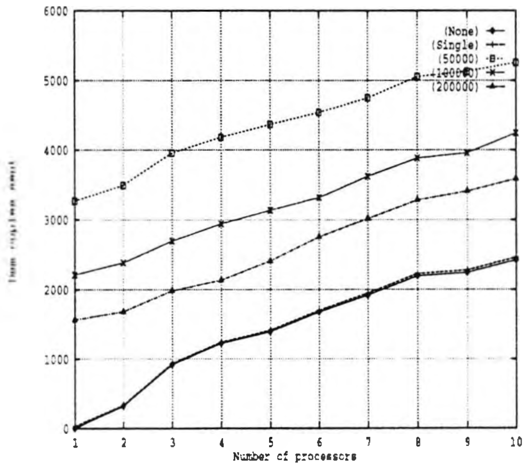

Accessed page additions
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

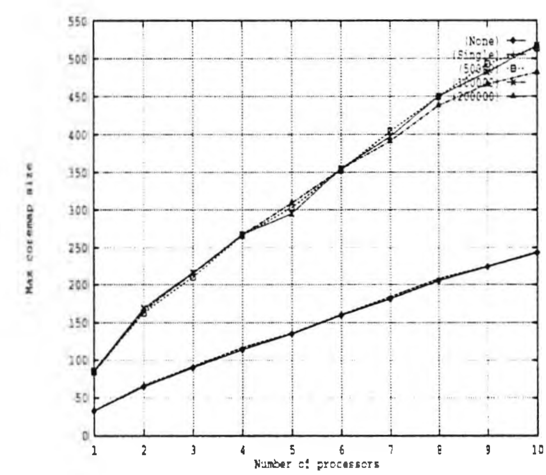

Dsm messages
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

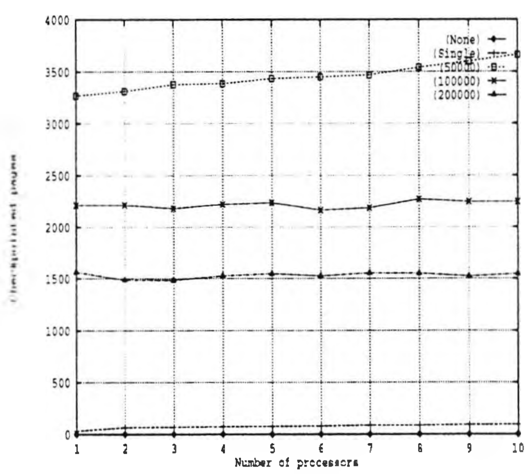

Dsm owner request
Number of processors
(None)
(100000)
(200000)

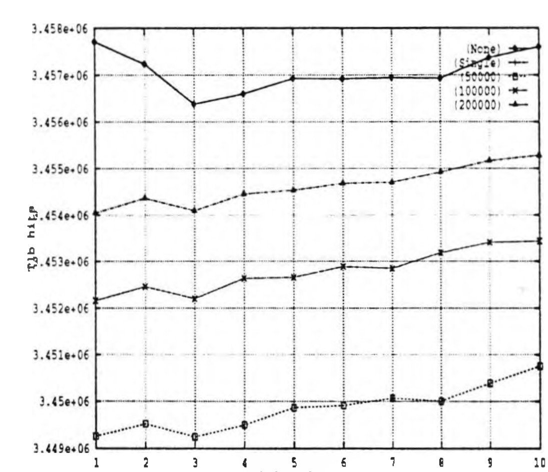

Dsm copy request
Number of processors
(None)
(Single)
(50000)
(100000)
(200000)

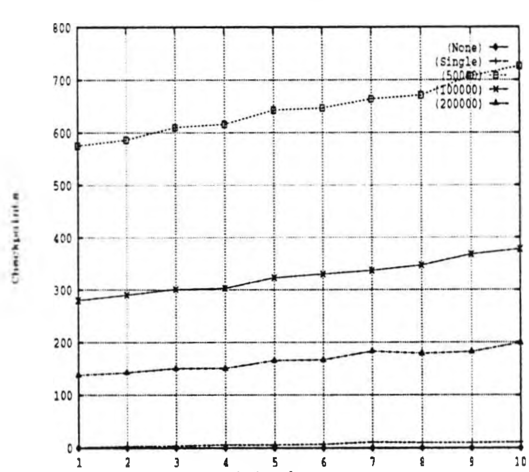

2500
2000
1500
1000
500
0
(None)
(Single)
(50000)
(100000)
(200000)
1 2 3 4 5 6 7 8 9 10
Number of processors


6000
5000
4000
3000
2000
1000
0
(None)
(Single)
(50000)
(100000)
(200000)
1 2 3 4 5 6 7 8 9 10
Number of processors


Max coremap size
550
500
450
400
350
300
250
200
150
100
50
0
(None)
(Single)
(50000)
(100000)
(200000)
1 2 3 4 5 6 7 8 9 10
Number of processors

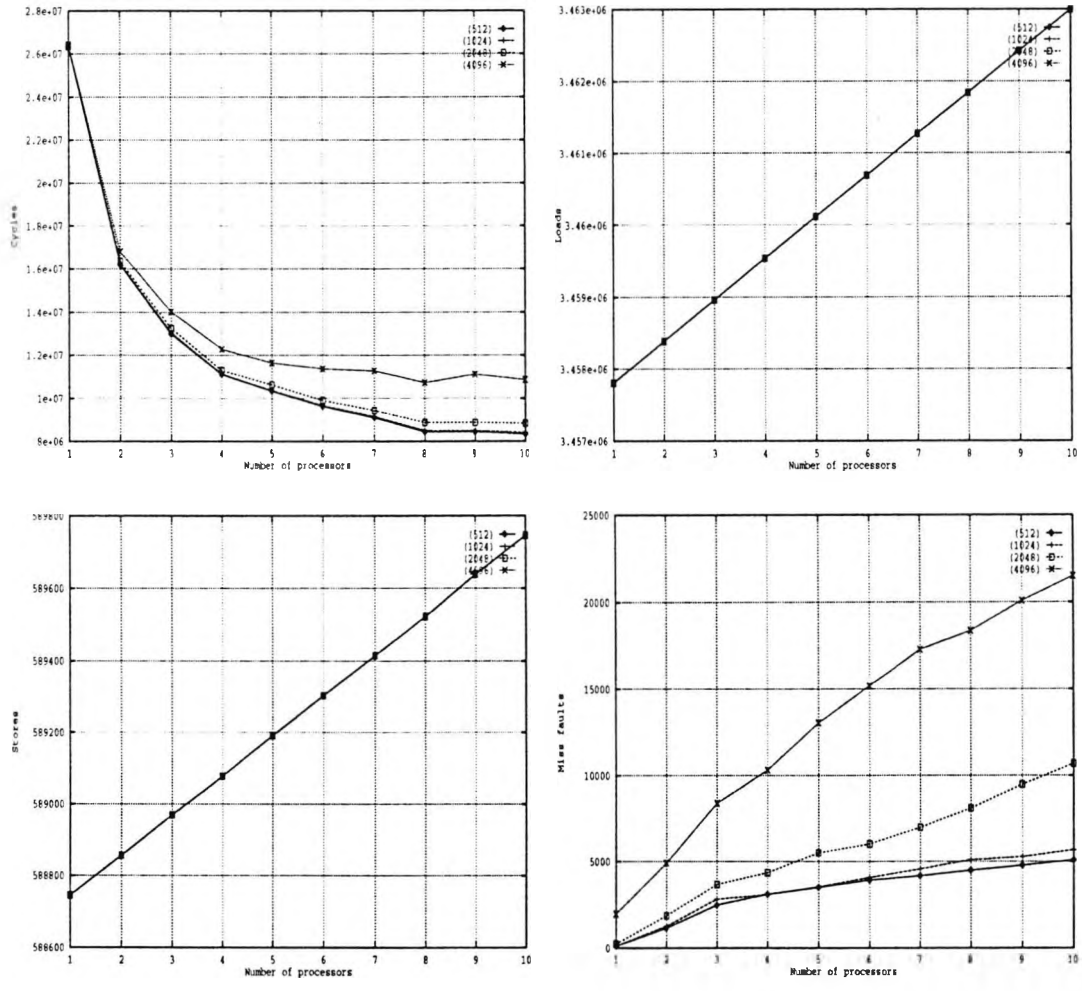

4000
3500
3000
2500
2000
1500
1000
500
0
(None)
(Single)
(50000)
(100000)
(200000)
1 2 3 4 5 6 7 8 9 10
Number of processors

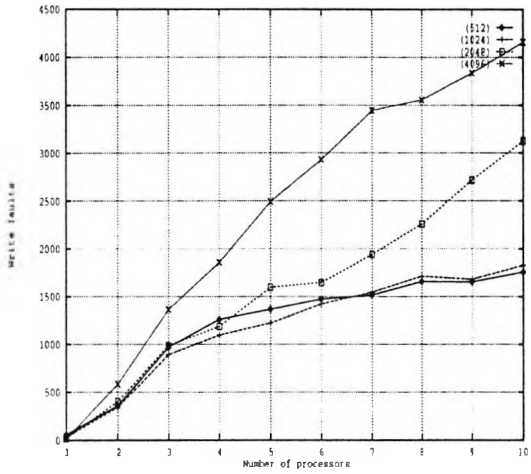

Tlb hits
3.458e+06
3.457e+06
3.456e+06
3.455e+06
3.454e+06
3.453e+06
3.452e+06
3.451e+06
3.45e+06
3.449e+06
(None)
(Single)
(50000)
(100000)
(200000)
1 2 3 4 5 6 7 8 9 10
Number of processors

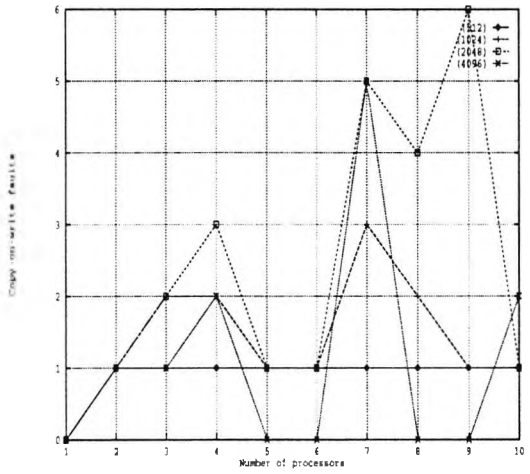

800
700
600
500
400
300
200
100
0
(None)
(Single)
(50000)
(100000)
(200000)
1 2 3 4 5 6 7 8 9 10
Number of processors

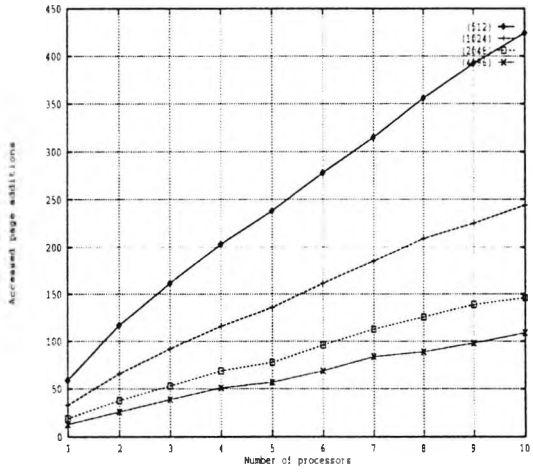
B.2.6 32 molecules – Page size

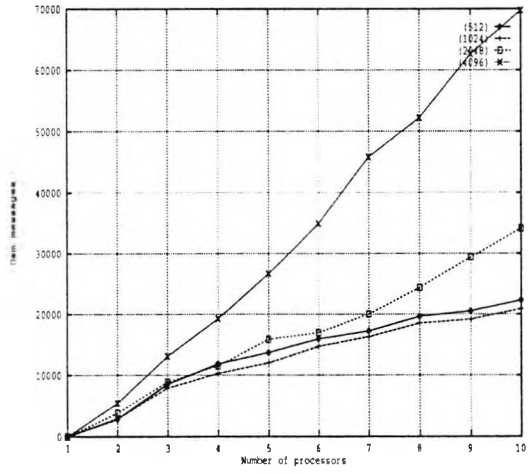
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

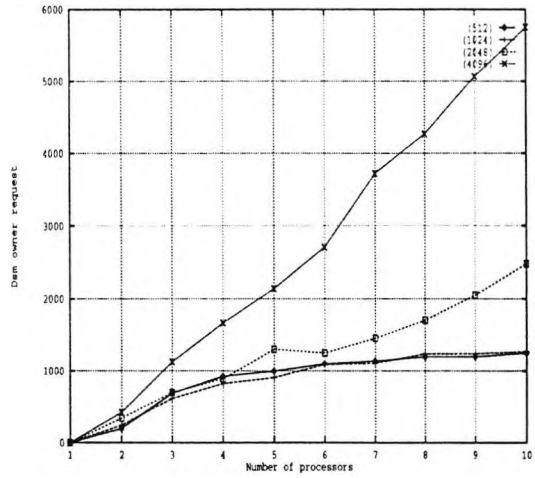


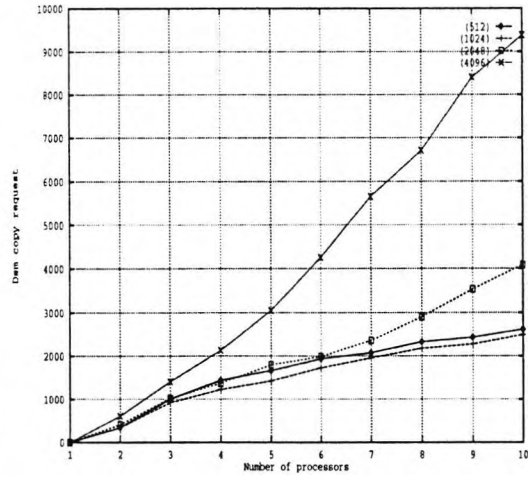

Write faults
Number of processors
(512)
(1024)
(2048)
(4096)

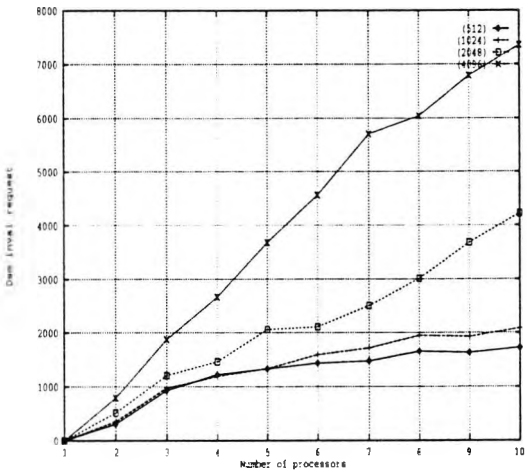

Copy-on-write faults
Number of processors
(512)
(1024)
(2048)
(4096)

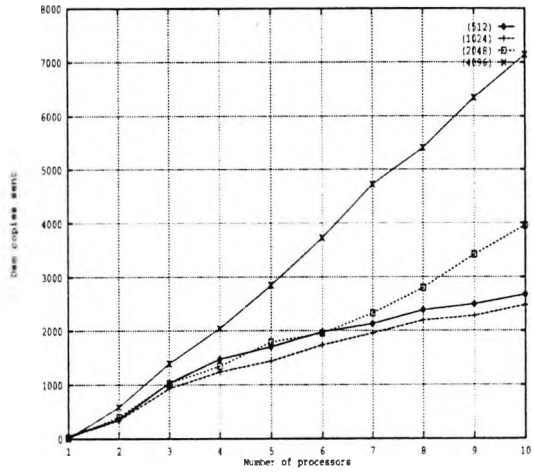

Accessed page additions
Number of processors
(512)
(1024)
(2048)
(4096)

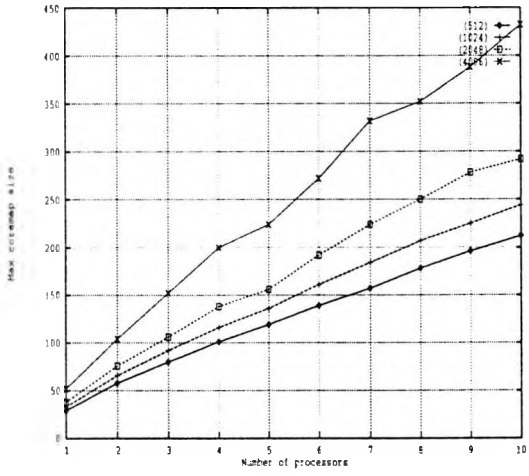

Dsm messages
Number of processors
(512)
(1024)
(2048)
(4096)

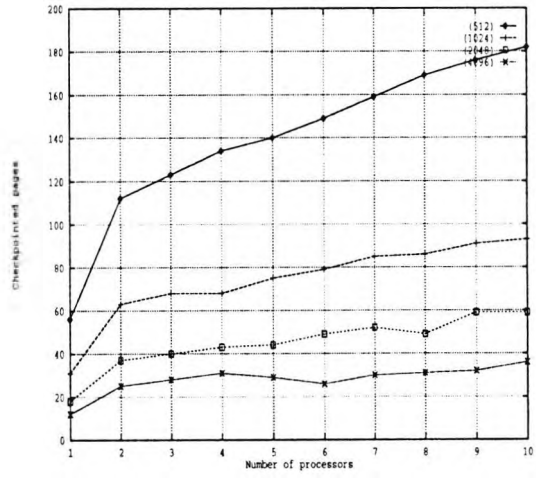

Dsm owner request
Number of processors
(512)
(1024)
(2048)
(4096)

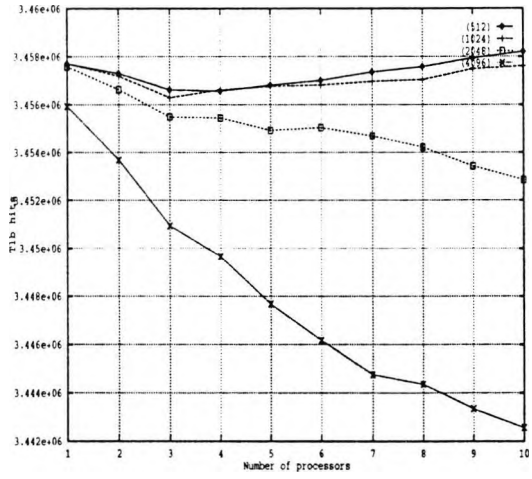

Dsm copy request
Number of processors
(512)
(1024)
(2048)
(4096)

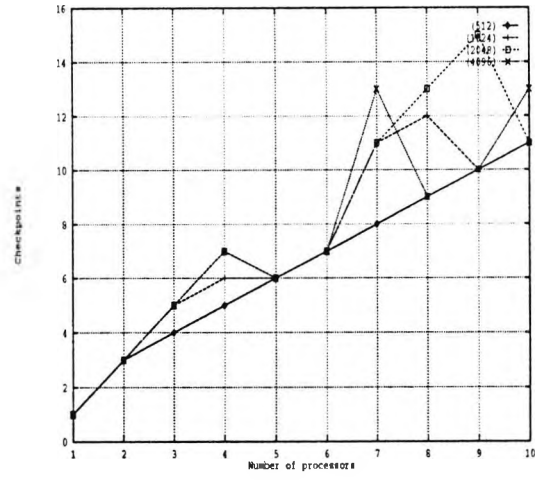

Dsm inval requests
Number of processors
(512)
(1024)
(2048)
(4096)


Dsm copies sent
Number of processors
(512)
(1024)
(2048)
(4096)


Max codemap size
Number of processors
(512)
(1024)
(2048)
(4096)

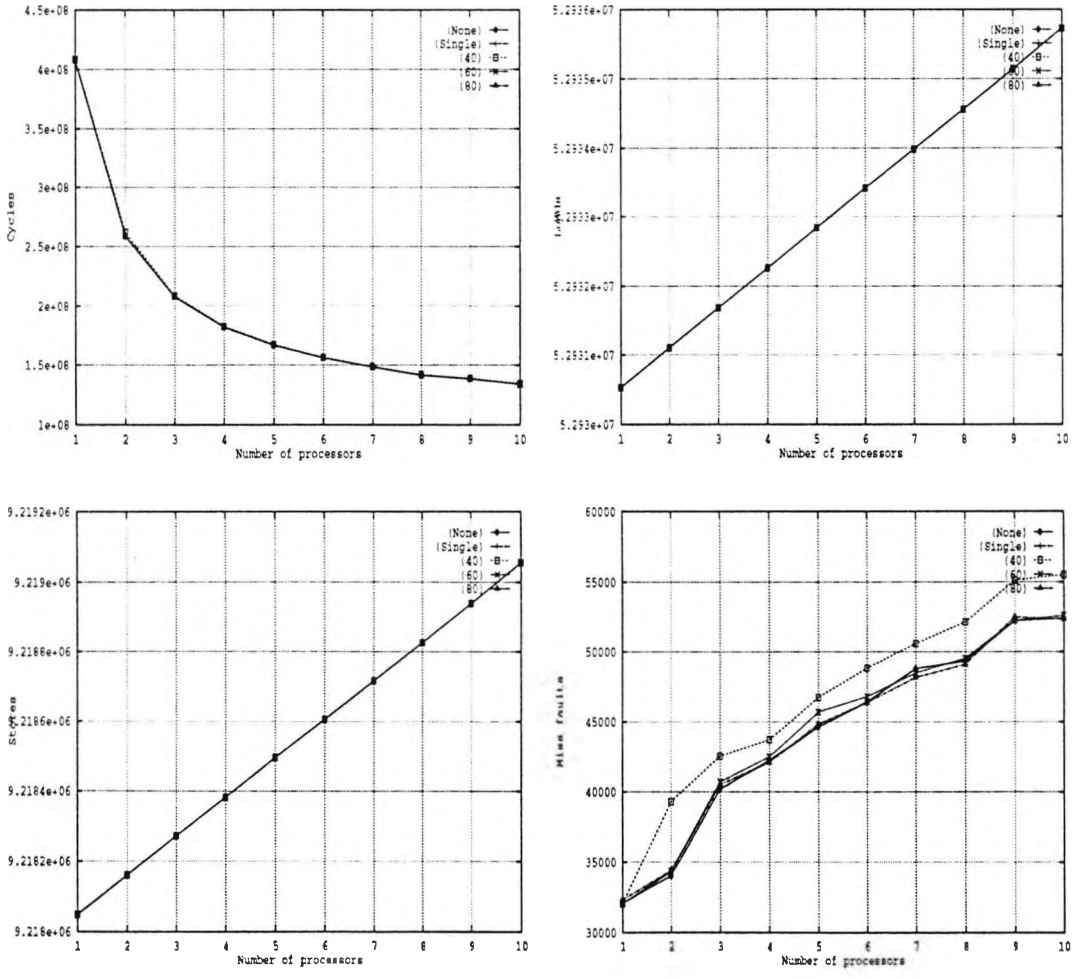

Checkpointed pages
Number of processors
(512)
(1024)
(2048)
(4096)

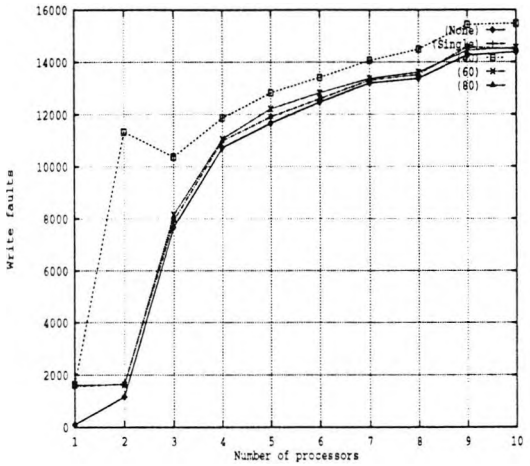

Tlb hits
Number of processors
(512)
(1024)
(2048)
(4096)

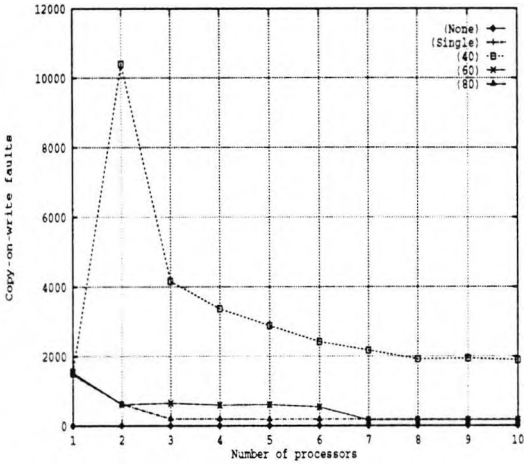

Checkpoints
Number of processors
(512)
(1024)
(2048)
(4096)

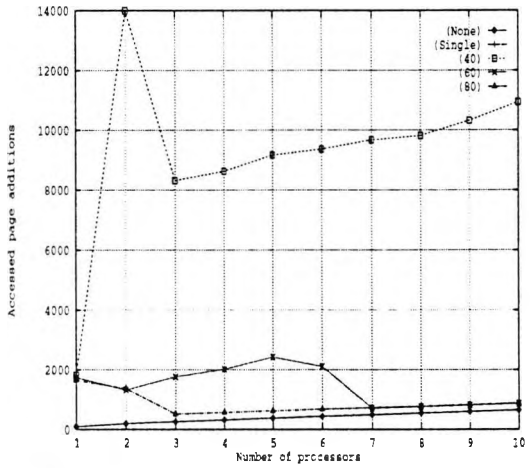
B.2.7 128 molecules – Access table size

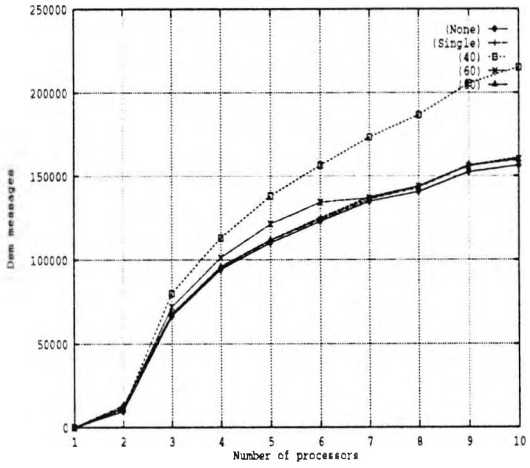
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

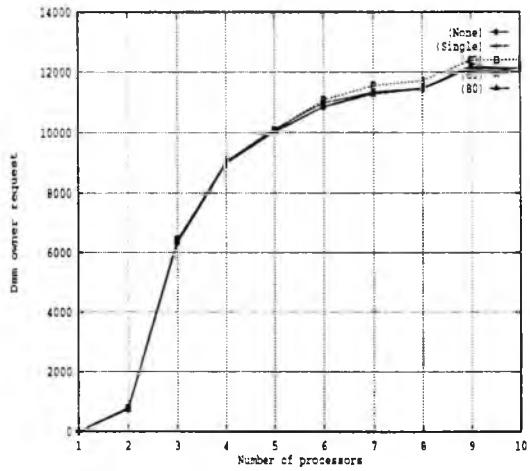


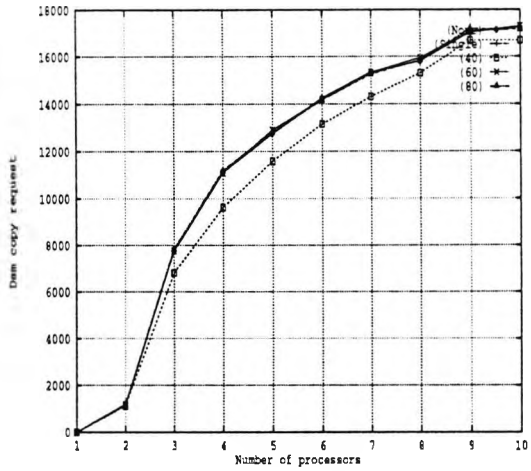

Write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

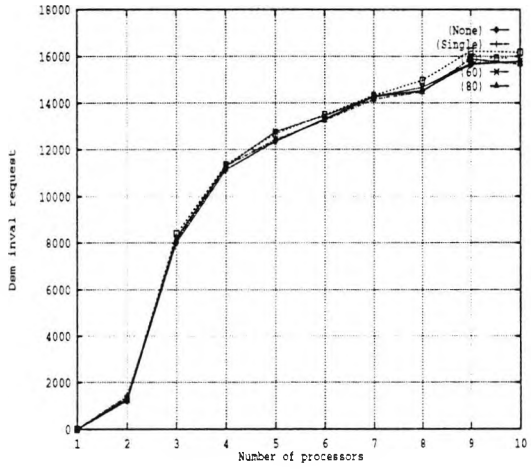

Copy-on-write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

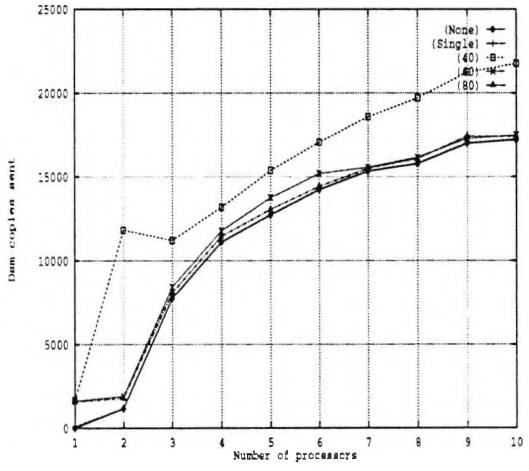

Accessed page additions
Number of processors
(None)
(Single)
(40)
(60)
(80)

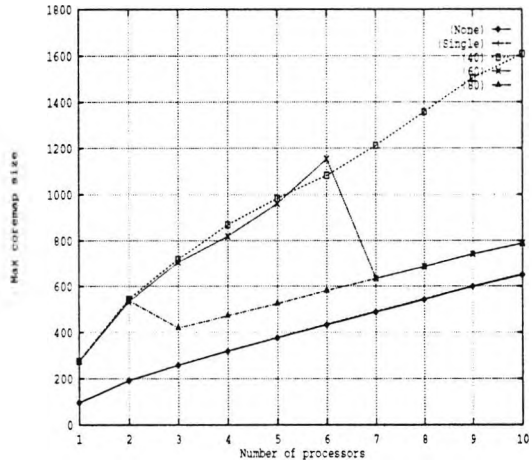

Dsm messages
Number of processors
(None)
(Single)
(40)
(60)
(80)

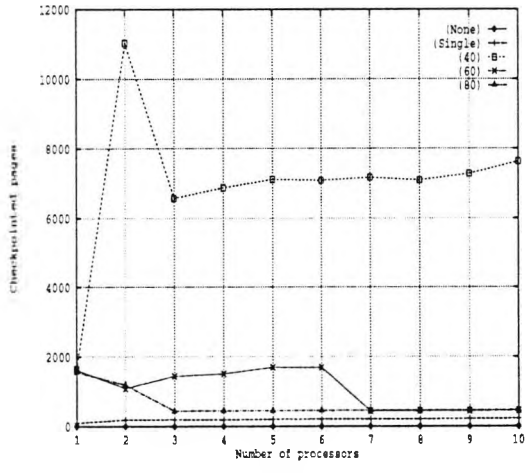

Dsm owner request
Number of processors
(None)
(Single)
(80)

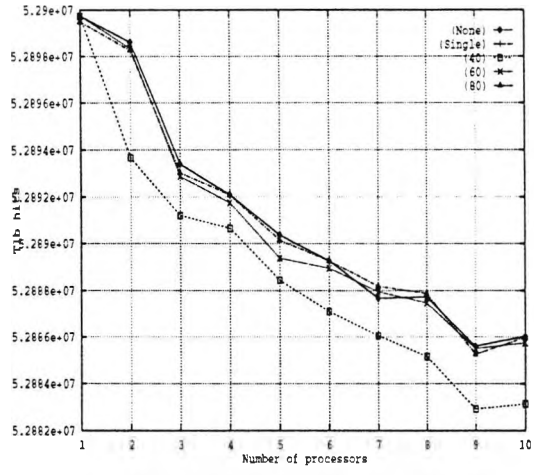

Dsm copy request
Number of processors
(40)
(60)
(80)

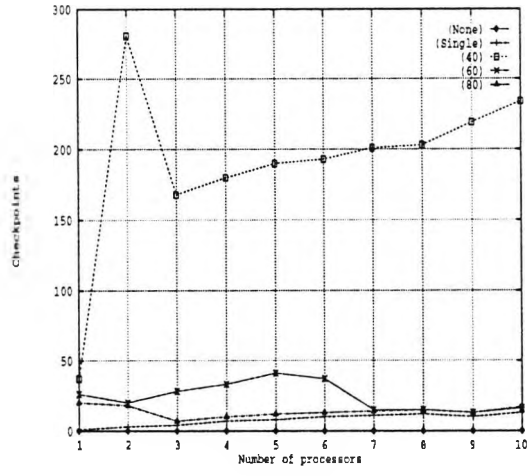

Dsm inval request
Number of processors
(None)
(Single)
(40)
(60)
(80)


Dsm copies sent
Number of processors
(None)
(Single)
(40)
(60)
(80)


Max coremap size
Number of processors
(None)
(Single)
(40)
(60)
(80)


Checkpointed pages
Number of processors
(None)
(Single)
(40)
(60)
(80)

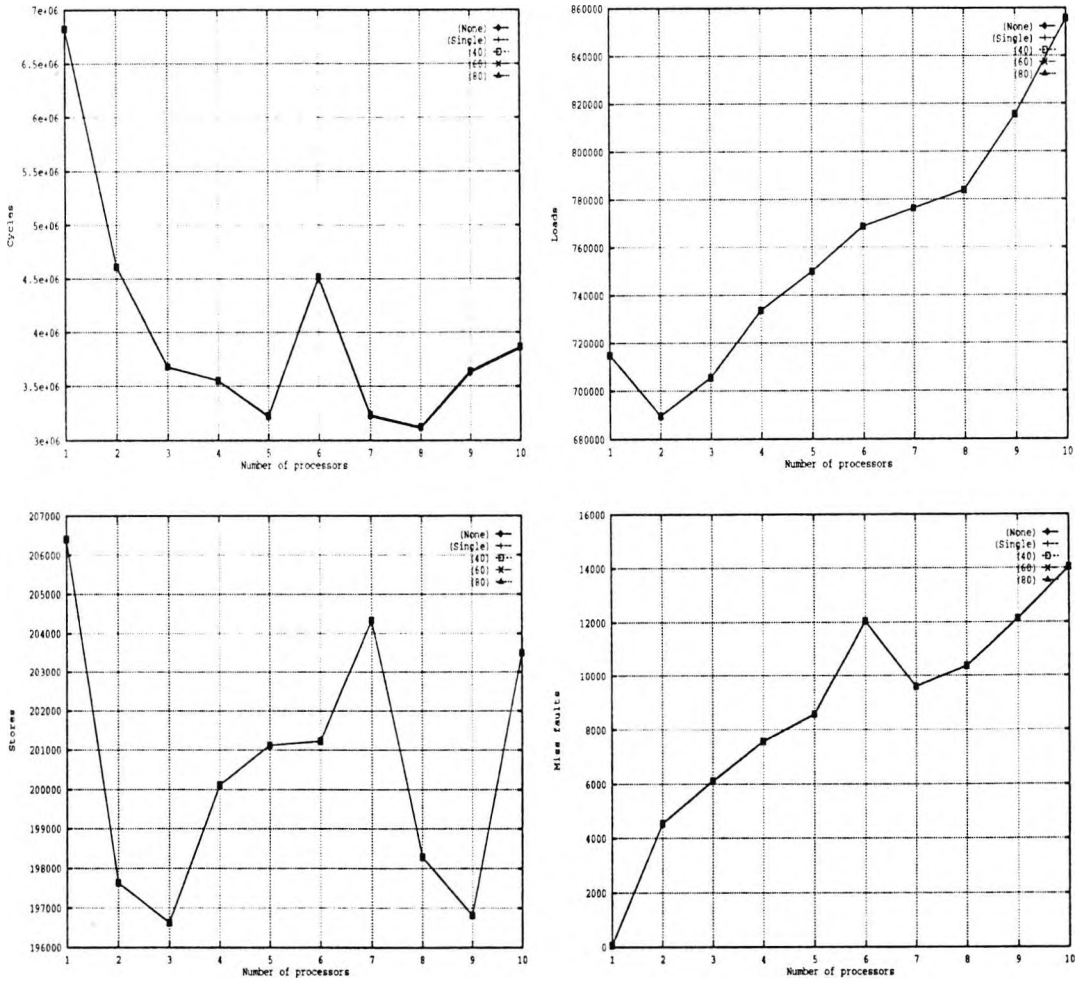

Tlb hits
Number of processors
(None)
(Single)
(40)
(60)
(80)

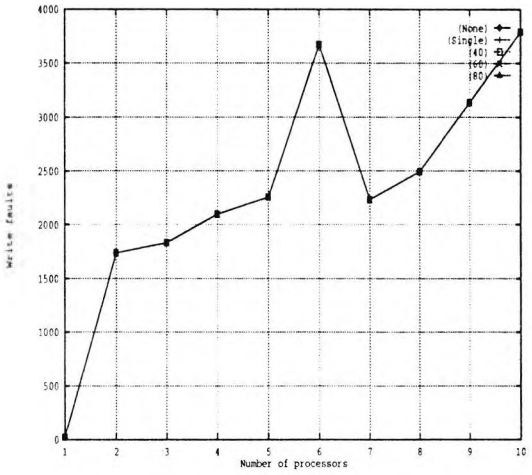

Checkpoints
Number of processors
(None)
(Single)
(40)
(60)
(80)

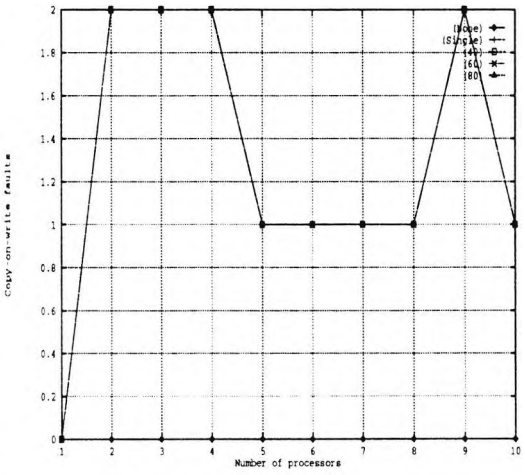
B.3 Barnes-Hut

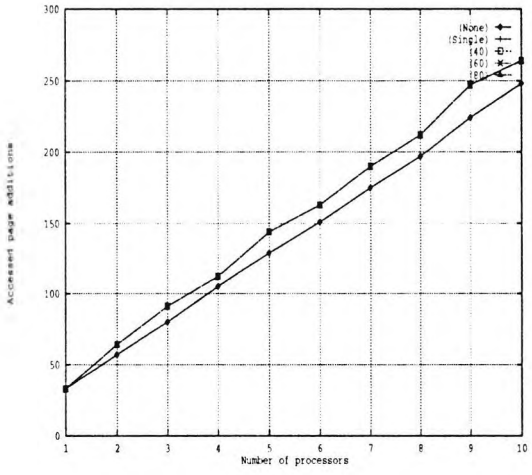
B.3.1 64 masses – Access table size

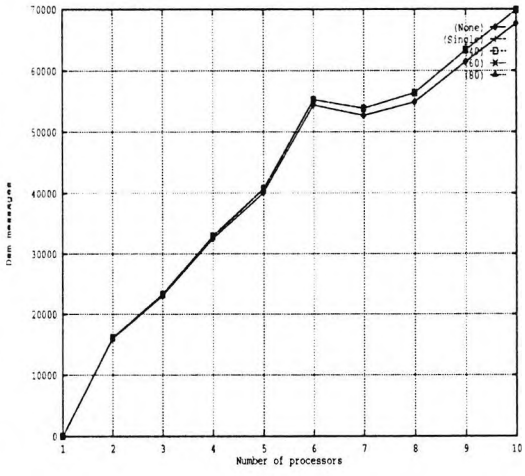
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

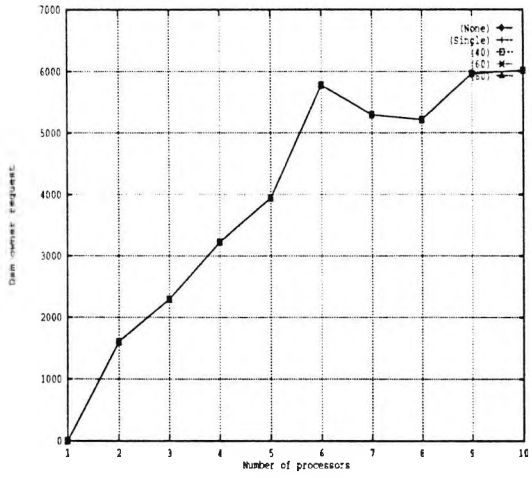


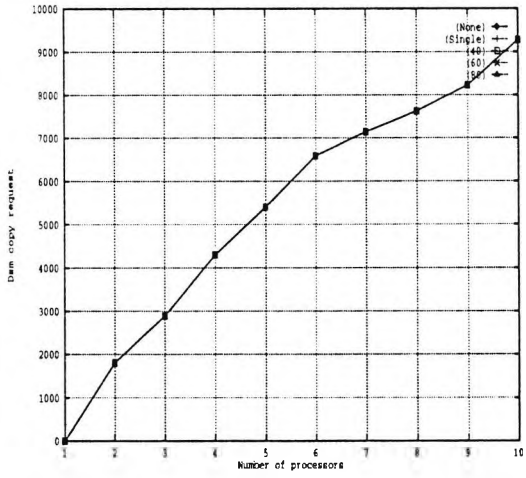

Write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)


Copy-on-write faults
Number of processors
(None)
(Single)
(40)
(60)
(80)

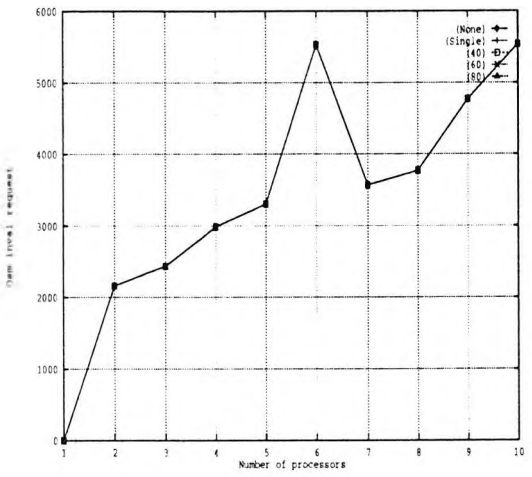

Accessent page additions
Number of processors
(None)
(Single)
(40)
(60)
(80)

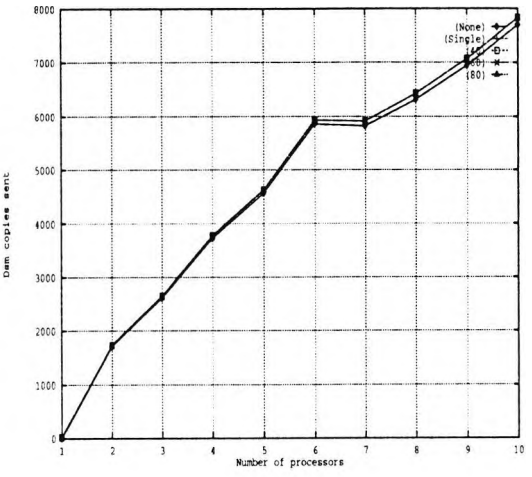

Dam messages
Number of processors
(None)
(Single)
(40)
(60)
(80)

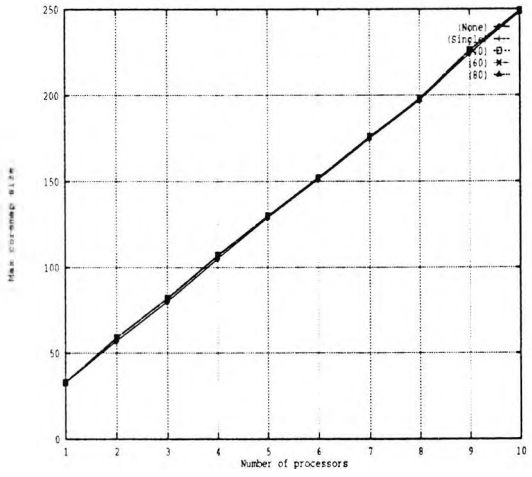

Dam owner request
Number of processors
(None)
(Single)
(40)
(60)
(80)

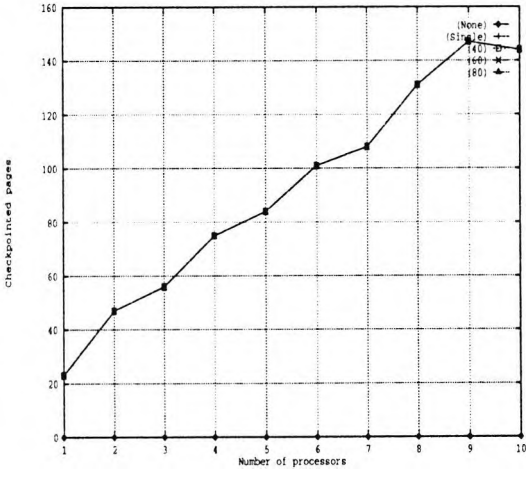

Dam copy request
Number of processors
(None)
(Single)
(40)
(60)
(80)

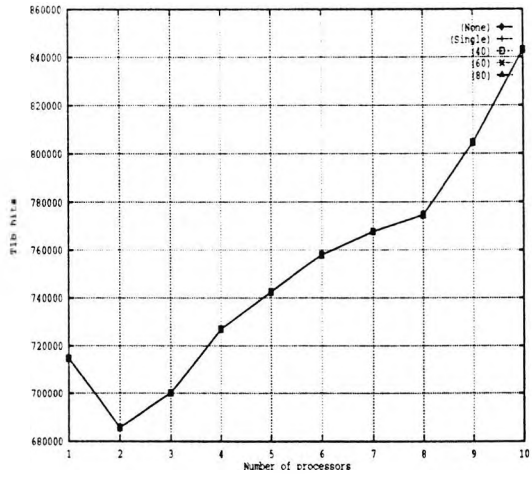
B.3. BARNES-HUT

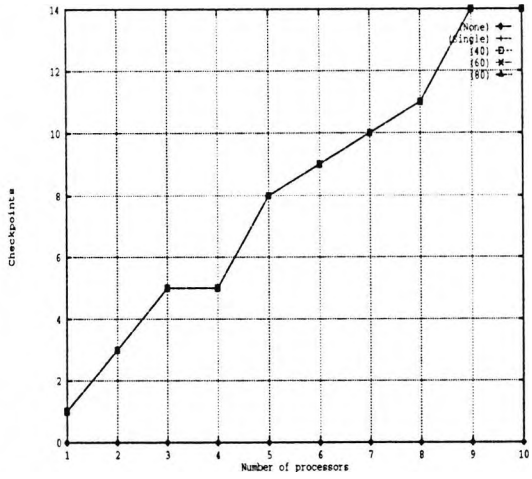






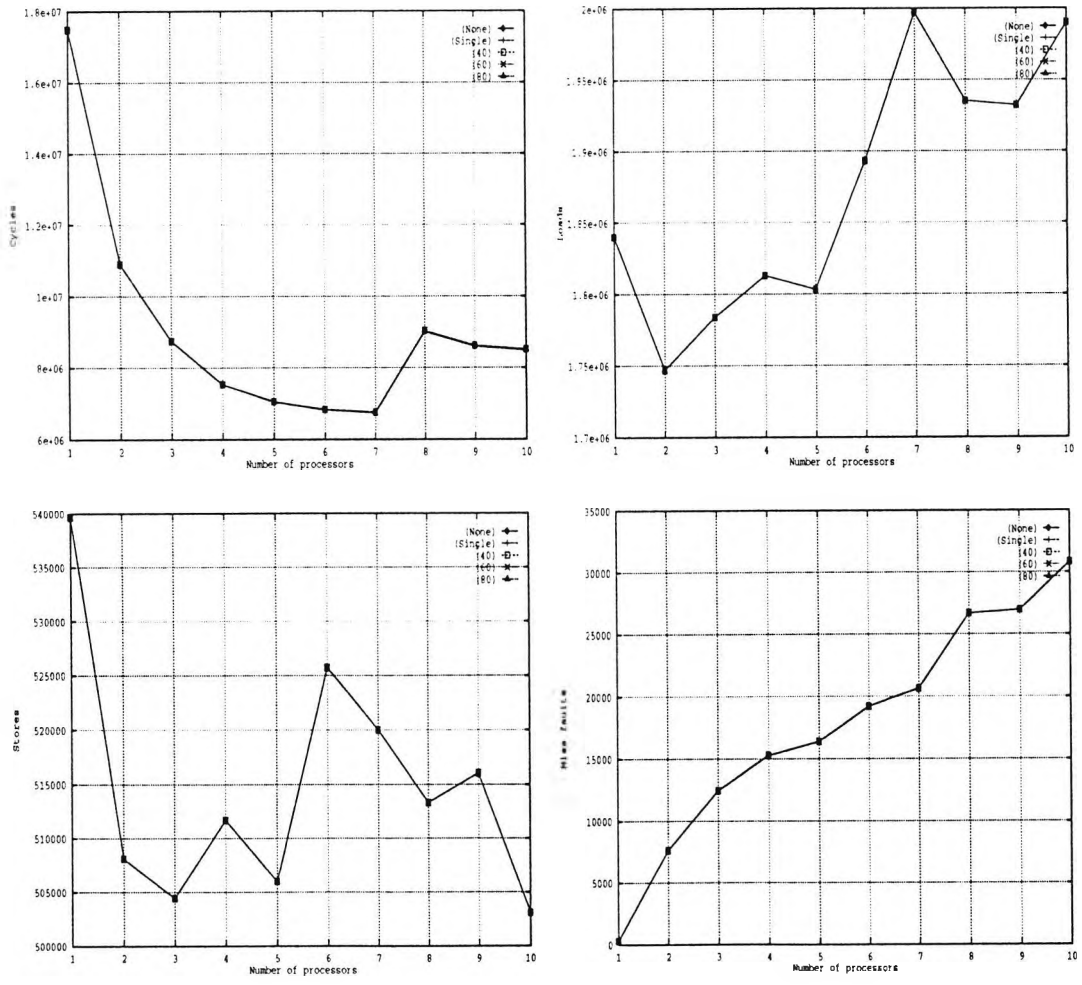




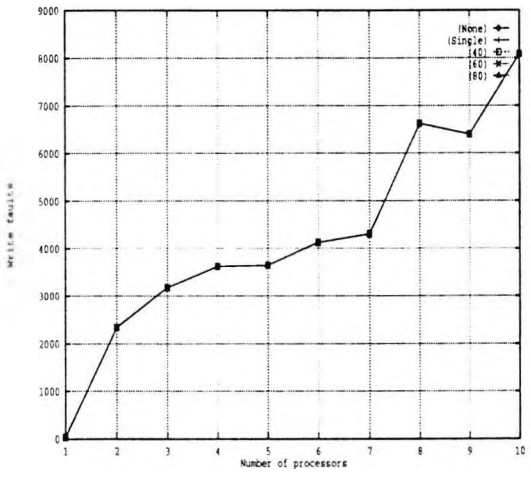


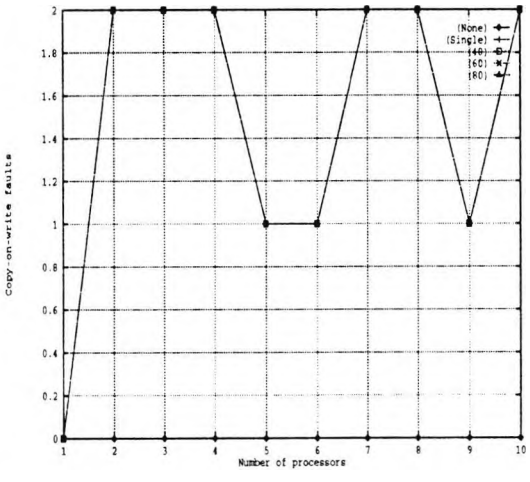
B.3.2 128 masses – Access table size

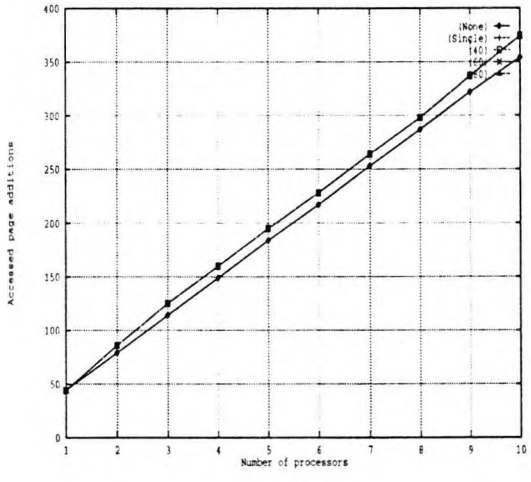
One to ten clusters,
One processor per cluster,
1K page size,
32 entry Pte cache,
50 cycle miss and write fault penalties,
30 cycles copy-on-write fault penalty (excluding copy),
100 cycles Dsm message delivery time (excluding page data).

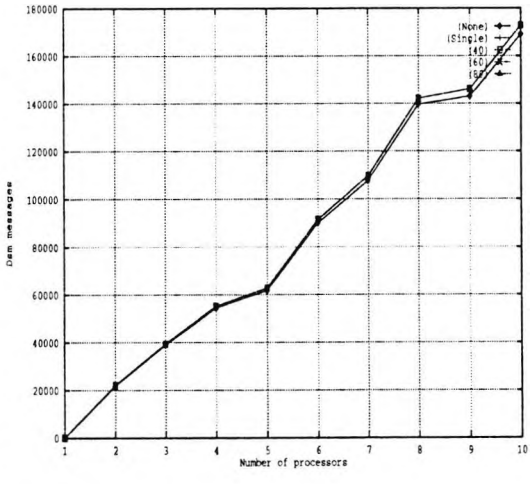


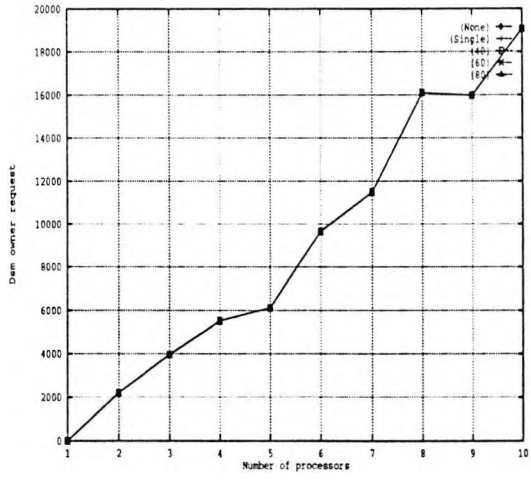
B.3. BARNES-HUT

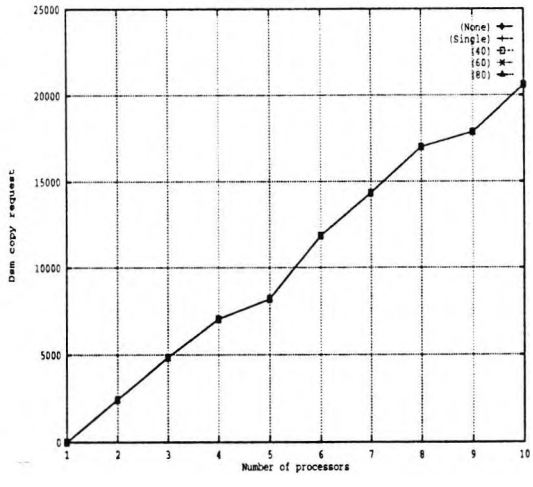


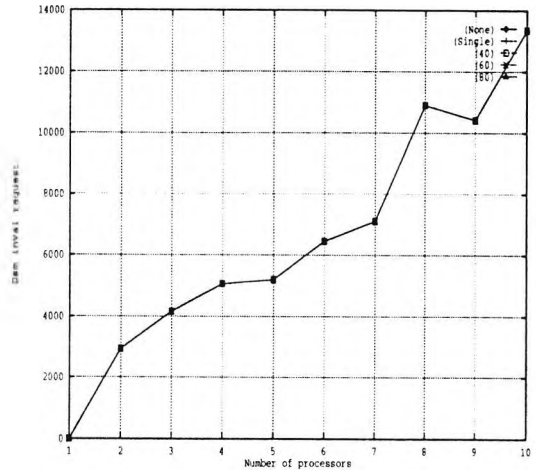


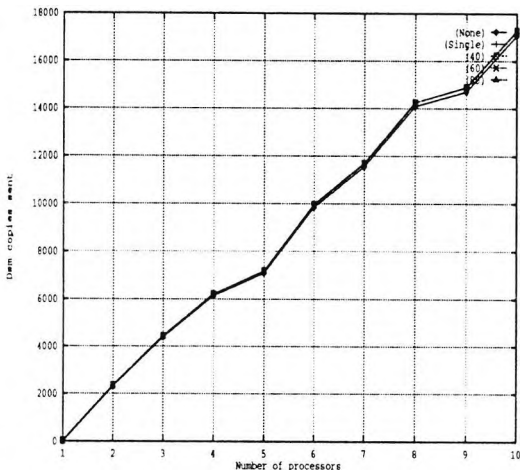


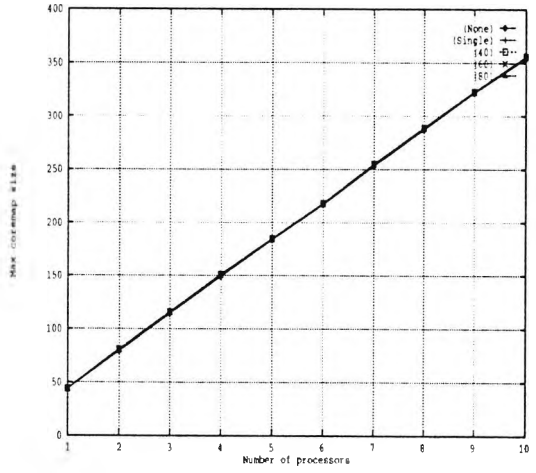


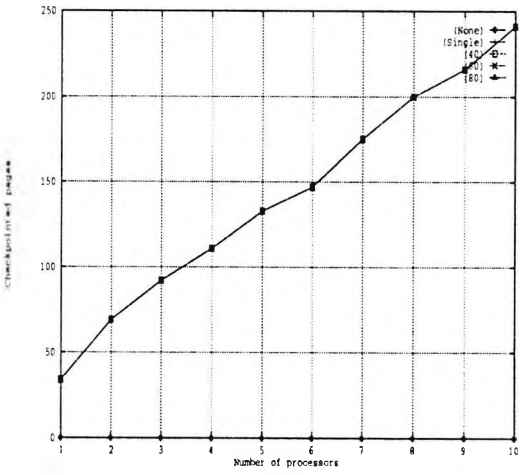


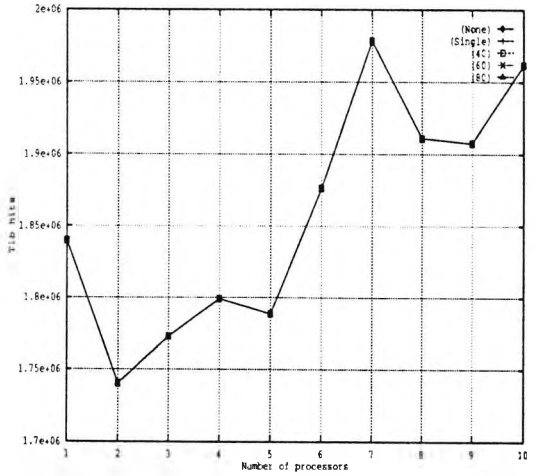


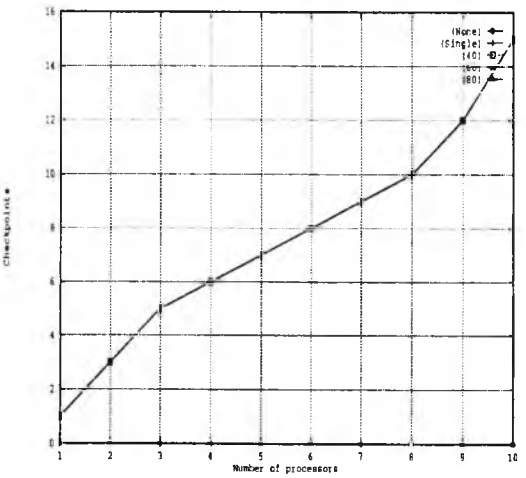

Dam inval requests
Number of processors
(None)
(Single)
(40)
(60)
(80)


Dam copies sent
Number of processors
(None)
(Single)
(40)
(60)
(80)


Max coremap size
Number of processors
(None)
(Single)
(40)
(60)
(80)


Checkpointed pages
Number of processors
(None)
(Single)
(40)
(60)
(80)


Tlb hits
Number of processors
(None)
(Single)
(40)
(60)
(80)


Checkpoints
Number of processors
(None)
(Single)
(40)
(60)
(80)