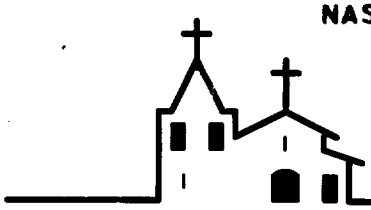


Santa Clara University

School of Engineering
Department of Computer Engineering
Santa Clara, CA 95053

Parallel Processing for Scientific Computations

Final Report

Overview of Research Project Accomplishments

NASA Agreement Number NCC 2–644

March 29, 1995

Submitted to
Mr. Ken Stevens, Jr.
NASA–Ames Research Center
NAS System Division, Mail Stop 258–5
Moffett Field, CA 94035
(415)604–5949

1. Introduction

The scope of this project dealt with the investigation of the requirements to support distributed computing of scientific computations over a cluster of cooperative workstations. Various experiments on computations for the solution of simultaneous linear equations were performed in the early phase of the p[roject to gain experience in the general nature and requirements of scientific applications. A specification of a distributed integrated computing environment, DICE, based on a distributed shared memory communication paradigm has been developed and evaluated. The distributed shared memory model facilitates porting existing parallel algorithms that have been designed for shared memory multiprocessor systems to the new environment. The potential of this new environment is to provide supercomputing capability through the utilization of the aggregate power of workstations cooperating in a cluster interconnected via a local area network.

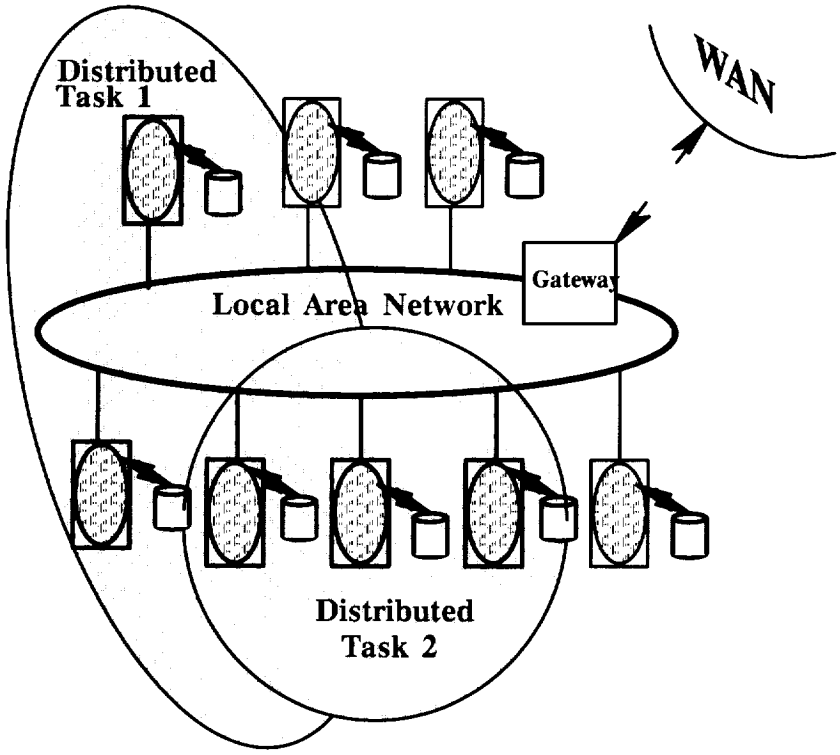


Figure 1
Clusters of Cooperating Workstations

The great majority of scientific applications require a fairly large amount of memory to execute a task. If a task is to be partitioned into threads (sub-tasks) that are executed in

parallel, memory sharing is very desirable since it allows sharing variables among threads within the same task. Shared memory multiprocessor systems have been the predominant platform selected for executing large scientific applications for these reasons.

Workstations, generally, do not have the computing power to tackle complex scientific applications, making them primarily useful for visualization, data reduction, and filtering as far as complex scientific applications are concerned. There is a tremendous amount of computing power that is left unused in a network of workstations. Very often a workstation is simply sitting idle on a desk. A set of tools can be developed to take advantage of this potential computing power to create a platform suitable for large scientific computations. The integration of several workstations into a logical cluster of distributed, cooperative, computing stations presents an alternative to shared memory multiprocessor systems. In this project we designed and evaluated such a system.

Attached to this report are three papers published or accepted for publication, resulting from this research project. These articles are:

1. Hasan S. AlKhatib, Qiang Li, Chi-Jiunn Jou, Tiekun Chen and Hassan Arafeh "DICE - a Distributed Integrated Computing Environment for Multi-threaded Parallel Processing", Proceedings of the Third International Systems Integration Conference, Sao-Paulo, Brazil, August 15-19, 1994, pp 612-621.
2. Chi-Jiunn Jou, Hasan S. AlKhatib, Qiang Li and Tiekun Chen "Coherency Protocol and Algorithm of the DICE Distributed Shared Memory", Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, October 6-8, 1994. pp 796-801.
3. Chi-Jiunn Jou, Hasan S. AlKhatib and Qiang Li "Two-Tier Paging and Its Performance Analysis for Network-based Distributed Shared Memory Systems", accepted for publication in the IEICE Transactions on Information and Systems.

2. DICE Overview

DICE is a computing environment for executing multi-threaded tasks on a cluster of networked workstations. In *DICE,* threads of a parallel task may run on separate workstations sharing the same virtual address space. Threads communicate with each other using shared memory. An overall system structure of *DICE* is shown in Figure 1.

DICE consists of three interactive subsystems: a distributed shared memory (*DSM*), a parallel scheduler (*PS*), and a distributed run-time subsystem (*DRS*). *DSM* provides mechanisms for sharing distributed memory among threads of a parallel task and hence supports the underlying computing and communication paradigm. *PS* provides tools to initiate both local and remote threads and to coordinate their execution over different workstations. *DRS*

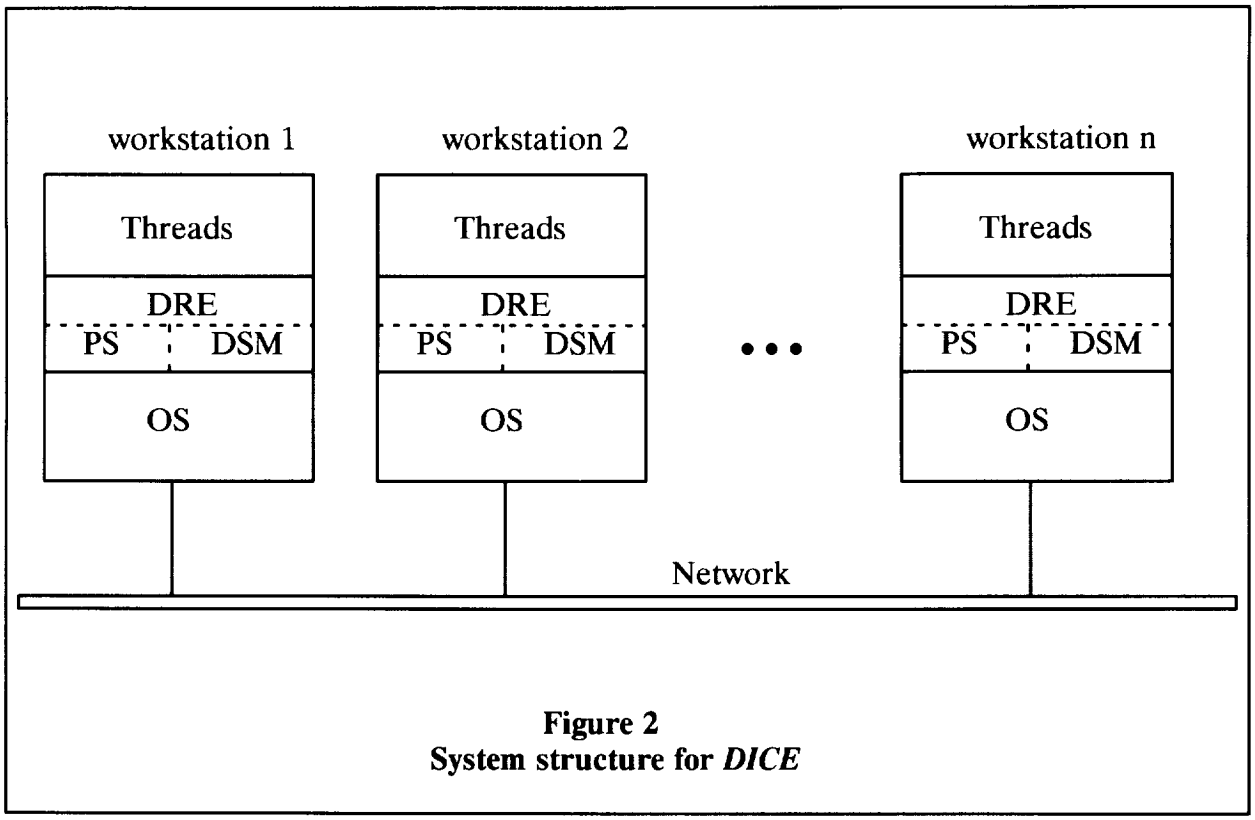


Figure 2
System structure for *DICE*

provides the programmers interface to develop parallel tasks as well as the run-time environment for their execution.

3. Distributed Shared Memory

In *DICE*, the physical memories of individual workstations in a cluster are treated as resources for the virtual space of a multi-threaded parallel task. Pages of the address space of a task can be shared among the threads of the same task. A task consists of multiple threads that can run on different workstations in a cluster simultaneously. The virtual memory of *DICE* is divided into private and shared spaces. Private space is local to a single workstation, and is not shared among threads. An example of private space is the stack of a thread. Shared space is global to all workstations, and is shared among all threads of a parallel task. Shared space is further divided into read-only code and read-write data spaces. The initial implementation of *DICE* will only support the shared data space.

DICE presents a new distributed shared memory design to attack the problems of false sharing and thrashing. False sharing may occur in a typical distributed shared memory system such as Ivy[1], since its consistency or access unit (eg. per word) is less than the sharing unit (per page). The single-write nature of its coherency protocol may cause a "ping-pong" behavior between multiple writers of a shared page, or the thrashing problem. To overcome

these problems, Mach[2] uses a shared memory server to perform the fault scheduling via a queueing mechanism[3]. Mether[4,5] avoids these problems through the use of the inconsistency. Clouds[6] avoid these problems by using a single-write-single-reader strict coherence semantics. Mirage[7] reduces the effect of these problems by using a time window scheme, in which the system guarantees that the writer of a page retains access to a page for a fixed period of time. Munin[8] minimizes these problems by using multiple type-specific coherency protocols.

To overcome these false sharing and thrashing problems, *DICE DSM* uses a hybrid memory granularity and supports multiple coherency protocols. Shared memory is structured as a two-layer paging system. The higher layer is a page, which is the same as the one in an existing system. The lower layer is a paragraph, which is a small fixed-sized memory region within a page. The memory sharing unit is a page, while the coherency unit is a paragraph. Each page in the shared address space is divided into several small equal-sized paragraphs. Each paragraph uses one and only one specific protocol at a time. The protocol used on a paragraph can be changed to adapt to new application requirement. The default protocol used on a paragraph is that of inconsistent memory, which only provides memory sharing without coherency. Other coherency protocols include write-invalidate, write-update, write-read-migrate, home-read-write, release-update, and entry-invalidate.

Write-invalidate, write-update, write-read-migrate, and home-read-write protocols provide a strict consistency on copies of a shared paragraph. They resemble the read-replication, full-replication, migration, and central algorithms in [9] respectively. Both release-update and entry-invalidate protocols provides weak consistency memory model on copies of paragraph. The weak consistency memory model is different from the strict consistent memory model in that it does not guarantee memory coherency without the use of explicit high-level synchronization operations. Parallel programs, therefore, would need to impose an ordering on accesses to shared memory by using synchronization operations. This protocol treats shared memory accesses differently from synchronization variable accesses. The model supports two types of synchronization accesses: *acquire* and *release*. Similar to the software release consistent protocol used in [8], release-update protocol ensures that all previously modified data is updated before the release is performed on a synchronization variable. Similar to the entry consistent protocol used in [10], entry-invalidate protocol ensures that a consistent copy of paragraphs are pre-fetched when the acquire or entry of a synchronization variable is performed.

DICE DSM is similar to Munin[8] system, since both of them use multiple type-specific coherency protocols. However, the kinds of protocol support and their designs are different

between them. More significant difference between them are the memory structure and granularity. *DICE DSM* uses fixed-sized paragraph flat memory space, while Munin uses variable-size object structure memory space. The advantage of using fixed-sized paragraph is that it allows the DSM to be implemented in hardware like MemNet [11]. This will improve the performance significantly, and is the final prototype of *DICE DSM*.

DICE separates synchronization mechanism from shared memory. It supports two kinds of synchronization variables locks and barriers. Whereas locks are used primarily for access control, that is, to resolve competition among parallel threads, barriers are used for sequence control, that is, to ensure correct timing among cooperating threads. Other kinds of synchronization variables can be built on top of them. *DICE* uses distributed queueing schemes for both lock and barrier synchronization protocols.

4. Parallel Scheduler

DICE PS is a self optimizing application specific scheduler. It is responsible for thread scheduling and synchronization. The *PS* is implemented as a thread within the parallel task. Each parallel task has one *PS* running on the workstation where the task initially start to run. This special thread is created during application load-time.

When an application needs to create another thread or to terminate itself by joining with other threads, it passes control of the execution to the *PS*. The *PS* will find the fastest way to run the application by using the information in the task execution dependence tree, which is created as an auxiliary file during the compiling of the source program.

The *PS* decides whether the local workstation has enough resources to run the different threads, which threads to send to remote workstations to run, and which remote workstations to send them to. It uses several tools to make intelligent decisions at run time. Those tools are: CPU load estimator, network load estimator, an intelligent database, and the bidding process.

The CPU load estimator runs on every workstation on the network and keeps track of the load on that workstation. The network load estimator monitors the traffic on the network, and helps the parallel scheduler in avoiding heavily loaded networks. A small and efficient database records thread performance on each workstations under different CPU and network load conditions. This database helps the bidding process by giving the workstations a reasonable estimate of the expected run times of various threads.

When the parallel scheduler decides that it is best to send some threads to a remote workstation to run, it needs a way to pick those workstations. Instead of forcing other,

possibly heavily loaded, workstations to take some of the threads, the parallel scheduler asks for help through the bidding process. It simply asks for help in running a given thread and tells the other workstations about the memory and CPU requirements of the thread. This information is found in the intelligent database. The detail design of *PS* is based on our previous work [12].

5. Distributed Run-Time Subsystem

DRS transforms the *DICE DSM* from a flat space into an object-oriented structured space. *DICE DRS* consists of a set of tools that implement the *DICE*. Application Programmer's Interface, API, provides users with programming tools to develop and execute *DICE* multi-threaded applications. The tools used during program development include a parallel language and its compiler, library interface functions, and a linker.

A new Object-Oriented Dataflow language(OODL) will be designed used as the parallel language used in *DICE*. One of the important features of object-oriented programming is information hiding and encapsulation [13,14]. It provides a higher level of data abstraction in modeling real world objects. Such concepts are helpful in designing parallel programs [13]. In general, parallel programs are difficult to design because the programmer must consider multiple execution threads instead of a single thread. All possible interactions among the threads must be considered. Also, parallel programs are hard to maintain because a simple change may affect the interaction pattern and result in global consequences. Information hiding helps in reducing possible interactions that need to be considered, while data encapsulation help in minimizing the maintenance effort when program changes are needed.

While the object-oriented model provides a high level of programming abstraction, it does not naturally exploit parallelism of applications constructed with objects. A dataflow model can expose and exploit the maximum amount of parallelism, as well as express data dependence from different levels of abstraction in a very natural way. The combination of the object oriented and dataflow concepts makes it easier for programmers to design large scale multi-threaded parallel programs, and to build re-usable concurrent software modules.

The OODL language, in *DICE*, will be an extension of the object-oriented programming language C++. Dataflow constructs will be added to allow programmers to express parallelism explicitly. The parallel compiler can be realized using a preprocessor to translate the extended source code into C++ programs, which in turn are compiled into object code using an existing C++ compiler.

The run-time library interface functions provide a collection of library routines that are linked with each parallel program. They are invoked to support the service requests made by

system processes at run-time. The OODL compiler will use these functions to realize the parallelism expressed in the application programs. These functions can also be used by the application directly.

6. Conclusions

The key results accomplished in this project include:

1. A design of a distributed shared memory system for distributed networked computing that solves the problem of false-sharing. The DSM employs a two-tier paging scheme and a set of management protocols and algorithms suitable for hardware support within the architecture of a workstation.
2. The DSM scheme was evaluated analytically. The results verify the validity of benefit of the two-tier paging scheme in solving the problem of false-sharing.
3. The DSM was alo simulated using the Block Oriented Network Simulator, BONeS, and was driven by a trace from a scientific application chosen from the Stanford's SPLASH benchmarks. The results of the simulation confirmed the results of the analytical work and also verified the utility of the use of the two-tier paging schem.

The papers attached to this summary report contain further details of the work performed under this project.

References

1. K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," In *Proceeedings of the 1988 International Conference on Parallel Processing*, pp. 94–101, August 1988.

2 A. Forin, J. Barrera, and R. Sanzi, "The Shared Memory Server," *Proceedings 1989 Winter USENIX Technical Conference*, Februray, 1989, pp. 229–244.

3. A. Forin, J. Barrera, and R. Sanzi, Design, Implementation, and Performance Evaluationof A Distributed Shared Memory Server for Mach, Technical Report CMU–CS–88–165, Carneigie–Mellon University, Computer Science Department, August, 1988.

4. R. G. Minnich and D. J. Farber, "The Mether System: A Distributed Shared Memory for SunOS 4.0," In *Useunix –Summer* 89, Usenix, 1989.

5 R. G. Minnich and D. J. Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

6. U. Ramachandran, M. Ahamad, and M. Khalida, "Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory," *Proceeedings of the 1989 International Conference on Parallel Processing*, pp. 160–169, August 1989.

7. B. D. Fleisch, G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989, pp. 211–222.

8. J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," *The 13th ACM Symposium on Operating Systems Principles*, October 1990, pp. 152–164.

9. M. Stumm, and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, Vol 23, No. 5, May 1990, pp. 54–64.

10. B. N. Bershad, M. J. Zekauskas, Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Technical Report CMU–CS–91–170, Carneig–Mellon Univeristy, September 1991.

11. G. S. Delp, The Architecture and Implementation of MemNet: A High Speed Shared–Memory Compute Communication Network. Ph.D. Thesis, University of Delaware, Department of Electrical Engineering, Newark, DE, May 1988.

12. H. Arafeh and H. S. AlKhatib, and H. Barraclough, "MOPPS: A Scheme for Managing Parallel Scientific Programs in a Distributed Architecture," *Proceedings of COMPCON'90*, the Annual International Computer Conference of the IEEE Computer Society, February 25 – March 2, 1990, San Francisco, CA, pp 387–395.

13. B. Cox, Object Oriented Programming – An Evolutionary Approach, Addison–Wesley, 1986.

14. B. Stroustrup, "What is "Object Oriented Programming"?," *IEEE Software*, Vol 5, No. 3, May 1988, pp. 10–20.

15. Y. Wu, T. G. Lewis, "Parallelism Encapsulation in C+ +," In *Proceeedings of the 1990 International Conference on Parallel Processing*, pp. 35–42, 1990.

Proceedings of the ISCA
International Conference

PARALLEL AND DISTRIBUTED COMPUTING

SYSTEMS

Las Vegas, Nevada U.S.A.
October 6-8, 1994

A Publication of
The International Society for
Computers and Their Applications - ISCA

ISBN: 1-880843-09-9

Coherency Protocol and Algorithm of The DICE Distributed Shared Memory

Chi-Jiunn Jou, Hasan S. AlKhatib, Qiang Li, and Allen Tiekun Chen
Computer Engineering Department
Santa Clara University
Santa Clara, CA 95053

Abstract

DICE (Distributed Integrated Computing Environment) DSM (Distributed Shared Memory) is an experimental system, being developed at Santa Clara University, which supports the execution of multiple threads on a cluster of networked workstations. This paper presents the coherency protocol and algorithm of DICE DSM, which is a novel approach to the design of the virtual-memory based DSM. In DICE DSM, the shared memory uses a two-tier paging system. The first tier, *page*, is the common page used in an operating system. The second tier is called a *paragraph*, which is a smaller fixed-sized unit of memory contained within a page. The introduction of paragraphs improves system performance by reducing the probability of false sharing as well as the size of the unit of information transferred over the network for maintenance of memory coherency.

Keywords: coherency protocol and algorithm, distributed shared memory, local area network.

1. Introduction

A Distributed Shared Memory (DSM) system supports the sharing of a virtual address space among processes running on loosely-coupled processors. A number of DSM systems over LANs have been developed [8]. Among them, Ivy [5] is implemented on a network of Apollo workstations. The memory is paged, and copies of pages may be replicated in different hosts. Strict coherency semantics are used, and the memory coherency is maintained by a write-invalidate with dynamic ownership protocol. The owner of a page is located via either a centralized manager, fixed distributed managers, or an individual host which forwards the request. Ivy is used for applications employing multi-threaded tasks. All threads share the same virtual address space. False sharing may occur in this system, since its consistency or access unit (e.g. word) is less than the sharing unit (page). In addition, the single-write nature of its protocol may cause a "ping-pong" behavior between multiple writers of a shared page.

To overcome false-sharing and thrashing, some systems employ special schemes. Clouds [7] avoids them by using a single-writer-single-reader strict coherence semantics. Mirage [3] reduces thrashing is by using a time window scheme, in which the system guarantees that the writer of a page retains access to a page for a fixed period of time. Munin [2] handles it by using multiple consistency protocols and software release consistency. Mether [6] reduces false sharing and thrashing through the use of the incoherent memory.

DICE (Distributed Integrated Computing Environment)[1] presents a novel approach to handle the problem of false sharing and thrashing. The shared memory is structured as a two-tier paging system. The first tier, called *page*, is the page commonly used in an operating system. The second tier is called a *paragraph*, which is a smaller fixed-sized block of memory within a page. *Paragraph* is the coherency unit. The introduction of paragraphs improves system performance by reducing the probability of false sharing as well as the size of the unit of information transferred over the network for maintenance of memory coherency.

An overview of the DICE DSM architecture is given in section 2. Section 3 presents the memory coherency protocol. The algorithm for realizing the complete DSM protocol is presented in section 4. Section 5 discusses the expected system performance and concludes.

2. The DICE DSM Architecture

DICE is an experimental distributed computing system which aims at providing a computing environment for the execution of multi-threaded tasks. A parallel task may consist of multiple threads that can be scheduled to run on a cluster of workstations simultaneously. A *thread* is an active program entity that provides the notion of a computation. Threads on separate workstations also share the same virtual address space, and communicate with each other using shared memory. Synchronization of threads accessing shared resources is done using functions provided by a distributed run-time library.

Figure 1 shows the system structure of DICE. It consists of three interactive subsystems. *DRS* (distributed run-time subsystem) provides users with programming tools to develop and execute DICE multi-threaded applications. *DSM* (distributed shared memory) provides the underlying communication and computing paradigm for threads of a parallel task. *PS* (parallel scheduler) is a self-optimizing application-specific scheduler, and is responsible for thread scheduling and synchronization.

In addition to a host processor and memory, each node in *DICE* also has a network processor and a *Distributed Shared Memory Management Unit (DSMMU)*. *DSMMU* is an extension of the traditional MMU which supports paragraph valida-

"This work was supported by NASA-Ames Research Center grants number NCC 2-644 entitled "Parallel Processing for Scientific Computations".

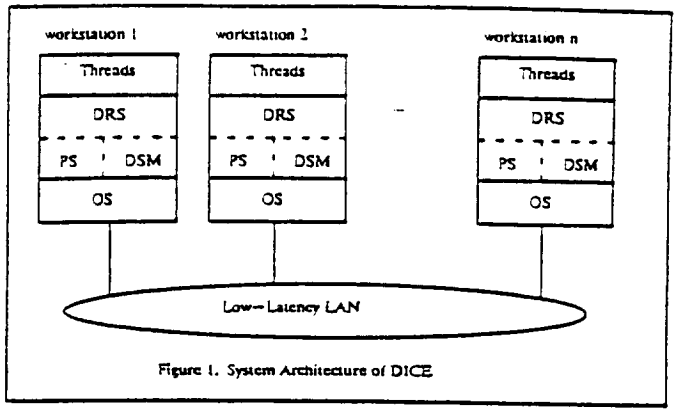


Figure 1. System Architecture of DICE

tion/invalidation to achieve efficient management of the DSM. When data is not available locally and needs to be fetched from a remote host, the *DSMMU* triggers a special access fault, otherwise, the *DSMMU* performs the traditional TLB operations.

3. Coherency Protocol

In DICE, a parallel task consists of multiple threads that run on a cluster of workstations (hosts), simultaneously. Shared data can be distributed and replicated on the physical memory of the members of a cluster. The DSM system supports the sharing of virtual pages, and maintains coherency among replicated data copies across the network. A parallel task has a *root host*, on which it was first loaded and executed. The root host maintains the state information for all shared pages used by the task. Other hosts in the cluster maintain the state information for the shared pages that are currently in their local physical memories.

In DICE each shared page of a parallel task has a *home host*. A home host maintains the state information for its pages, and ensures that the last copy of a page is not purged, and keeps track of all copies of the paragraphs of its pages. Other hosts in the cluster that have a copy of a page keep a pointer of the home host. When a thread makes an attempt to access a page for which it does not have a copy, it communicates with the home host of the respective page in order to complete the memory access transaction. When a host does not know the home host for a certain page, a *home–info* fault will be triggered and a *home–info* request will be sent to the root host. The root host replies with the information about the home host for the requested page. If the home host is not yet assigned, the root host will assign the first requesting host as the home host for the requested page. The root host will then update its database and send to the requesting host a reply informing this assignment.

The memory coherency of DICE DSM is maintained on the paragraph level. A paragraph can simultaneously be read by multiple hosts, but it can only be written by one host at a time. Access rights to a paragraph can be *read–write*, *read–only*, or *none*. An *owner host* is the most recent host that have *read–write* access to that paragraph. The ownership of a paragraph may be transferred from one host to another. There is no ownership when two or more hosts have *read–only* access rights to that paragraph. The Information about the ownership of a paragraph is maintained at the home host of the page containing the paragraph.

When a read operation is issued to a paragraph by a host with *none* rights, a *read–data* fault will be triggered and a *read–data* request will be sent to the paragraph's home host. When a write operation is issued to a paragraph by a host with *none* right, a *write–data* fault will be triggered and a *write–data* request will be sent to the paragraph's home host. When a write operation is issued to a paragraph by a host with *read–only* access right, a *write–access* fault will be triggered and a *write–access* request will be sent to the paragraph's home host. In each case the home host directly or indirectly responds with the requested information.

At initialization, a home host is the default owner host for all paragraphs within its respective pages. Any other host will send a remote request to the home host when it tries to access a paragraph of this page. If a *read–data* request is received, the home host will return a reply containing the most recent copy of the desired paragraph when it is the owner host or there is no owner host of that paragraph. The access rights of both home and requesting hosts are changed to *read–only*. If the home host is not the owner host, it will forward this *read–data* request to the owner host of that paragraph. The latter changes its access right to *read–only*, and then sends to both home and requesting hosts a reply containing the most recent copy of that paragraph. After it receives the reply, both home and requesting hosts changes their access right to *read–only*. Home host will also reset the owner host of that paragraph to *none*. If it is the requesting host, the home host will directly send the *read–data* request to the owner host. The latter changes its access right to *read–only*, and then sends back a reply which contains the most recent copy of that paragraph. Having received this reply, the home host changes its access right to *read–only* and resets the owner host of that paragraph to *none*.

If a *write–data* request is received, the home host will return a reply containing the most recent copy of the desired paragraph if it is the owner of this paragraph. If multiple valid copies exist, the home host will send *invalidate* requests to all hosts holding the copies, and wait for confirmations from all of them before returning the reply. Upon receiving the *invalidate* request, each host changes its access right of that paragraph to *none* and returns its confirmation to the home host. The access right of the home host is then changed to *none*, while the requesting host becomes the owner host and its access right is changed to *read–write*. If the home host is not the owner host, it will forward this *write–data* request to the owner host of that paragraph. The latter changes its access right to *none*, and then directly sends to the requesting host a reply containing the most recent copy of that paragraph. After receiving the reply, the requesting host changes its access right to *read–write* and sends a confirmation message to the home

host. Having received this confirmation message, the home host updates its database and records that the requesting host becomes the owner host of that paragraph. If the home host is the requesting host, it will directly send the *write–data* request to the owner host. The latter changes its access right to *none*, and then sends back a reply which contains the most recent copy of that paragraph. Having received this reply, the home host changes its access right to *read–write* and becomes the owner host of that paragraph.

If a *write–access* request is received, the home host will return the *write–access* confirmation when it is the owner of that paragraph. If multiple valid copies exist, the home host will send *invalidate* requests to all hosts (except the requesting one) holding the copies, and wait for confirmations from all of them before returning the confirmation message. Upon receiving the *invalidate* request, each host changes its access right of that paragraph to *none* and returns its confirmation to the home host. The access right of the home host is then changed to *none*, while the requesting host becomes the owner host and its access right is changed to *read–write*. If the home host is the requesting host, it will directly send the *invalidate* requests to all hosts (except the requesting one) holding the copies and wait for confirmations from all of them. Upon receiving the *invalidate* request, each host changes its access right to *none* and returns its confirmation to the home host. The home host then changes its access right to *read–write* and becomes the owner host of that paragraph.

Figure 2 shows the state diagram representing the location of a valid paragraph. This state diagram reflects the protocol described above. At any time, the location state of a valid paragraph is either *none*, *at home host*, *at owner host*, or *at multiple hosts*. The state is initially set to *none* when a home host has not yet been assigned. A *home–info* fault and request made by any host forces the root host to assign the requesting host to become the home host. The state is then changed to *at home host*. In this case the home host is the owner of the paragraph.

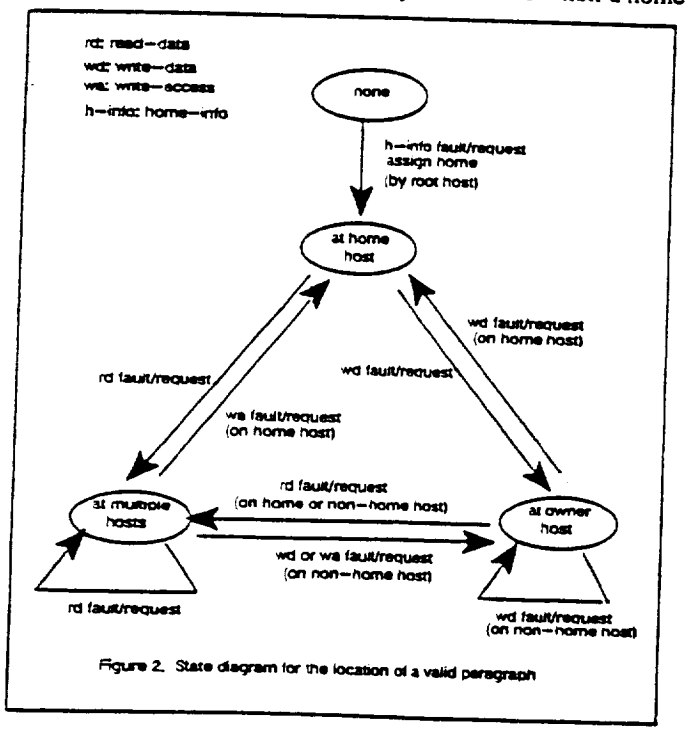


Figure 2. State diagram for the location of a valid paragraph

A paragraph will leave the *at home host* state when either a *read–data* or a *write–data* fault occurs. A *read–data* fault and request at any non–home host causes the paragraph to transit to the *at multiple hosts* state. In this case there is no owner host and multiple hosts have valid copies (with *read–only* access rights) of the paragraph. Note that these multiple hosts always include the home host. A *write–data* fault and request causes the paragraph to transit to the *at owner host* state, where the requesting host becomes the owner host of the paragraph.

The paragraph will leave the *at multiple hosts* state when either a *write–access* or a *write–data* fault occurs. A *write–access* or a *write–data* fault and request at any other non–home host causes the paragraph to transit to the *at owner host* state. A *write–access* fault and request at the home host causes the paragraph to transit to the *at home host* state. A *read–data* fault and request at any other non–home host will still keep the paragraph in the *at multiple hosts* state. Note that a *read–data* or a *write–data* fault will never occur at the home host, since a home host has a valid copy of the paragraph (with *read–only* access rights) in the *at multiple hosts* state.

The paragraph may leave the *at owner host* state when either a *read–data* or a *write–data* fault occurs. A *read–data* fault and request at any other host causes the paragraph to transit to the *at multiple hosts* state. A *write–data* fault and request at the home host causes the paragraph to transit to the *at home host* state. A *write–data* fault and request at any other non–home host causes a change of ownership, but the paragraph will still be in the *at owner host* state.

4. Coherency Algorithm

To support the above protocol, a *Page table* (*PT*) and a *paragraph table* (*ParT*) are used to maintain the state information about shared pages and paragraphs. Each DICE application maintains its own set of these tables. In addition to the address mapping information and flags, *PT* also maintains the information about the location of home host of each shared page. This location information is denoted by the *home host identifier* or *hid*. *ParT* maintains the information about the access rights to each paragraphs (*acc*). The *ParT* of the home host also maintains the location of the owner host of a paragraph (*oid*), and the set of hosts (excluding the home host) which have read–only copies of the paragraph (*copyset*).

The coherency algorithm handles various kinds of paragraph validation faults as described in section 3. These faults include *home–info*, *read–data*, *write–data*, and *write–access* faults. We divide the algorithm into four parts, corresponding to the four fault types. Each part of this algorithm consists of a fault handler and its server, as illustrated in Figures 3 to 6 for the respective fault type. Note that p and g, which are used

within the algorithm, denote the current page and paragraph numbers, respectively.

```
home-info fault handler:
        send home-info request to root host;
        receive home-info/home-assign reply from root host;
        create a ParT for p;
        IF ( home-assign confirmation is received)
          BEGIN
                PT[p].hid = myself;
                FOR paragraph i in p DO
                        ParT[i].oid = myself;
                        ParT[i].copyset = {};
                        ParT[i].acc = read-write;
                END DO;
          END
        ELSE /* home-info reply is received */
          BEGIN
                PT[p].hid = the assigned home host;
                FOR paragraph i in p DO
                        ParT[i].acc = none;
                END DO;
          END
        return;

home-info fault server:
        IF (p is not yet assigned with a home host)
          BEGIN         /* "none" state -> "at home host" state */
                PT[p].hid = requesting host;
                create a ParT for p
                FOR paragraph i in p DO
                        ParT[i].acc = none;
                send home-assign confirmation to requesting host;
          END;
        ELSE            /* "at home host" state - no state change */
          send home-info reply to requesting host;
        return;
```

Figure 3. The algorithm for handling home-info faults

```
Read_data paragraph fault handler:
        IF (I am home host)
          BEGIN  if      /* "at owner host" state -> "at multiple hosts" state */
                send read_data request to ParT[g].oid;
                receive read_data reply from ParT[g].oid;
                ParT[g].copyset = {ParT[g].oid};
                ParT[g].oid = none;
          END
        ELSE
          BEGIN
                send read_data request to PT[p].hid;
                receive read_data reply from owner host;
          END;
        update local copy of g;
        ParT[g].acc = read-only;
        unblock host processor;
        return;

Read_data paragraph fault server:
        IF (I am home host)
          BEGIN         /* "at multiple hosts" state - no state change */
                IF (no owner host)
                  BEGIN
                        send read_data reply to requesting host;
                        ParT[g].copyset=ParT[g].copyset+{requesting host};
                  END
                ELSE IF (I am owner host)
                  BEGIN             /* "at home host" state -> "at multiple hosts" state */
                        ParT[g].acc = read-only;
                        send read_data reply to requesting host;
                        ParT[g].copyset = {requesting host};
                        ParT[g].oid = none;
                  END
                ELSE /* owner host is not me */
                  BEGIN             /* "at owner host" state -> "at multiple hosts" state */
                        forward read_data request to ParT[g].oid;
                        block processing future requests for g;
                        receive read_data reply from owner host;
                        update local copy of g;
                        ParT[g].acc = read-only;
                        ParT[g].copyset = {Part[g].oui,requesting host};
                        Part(g].oui = none;
                        unblock processing future requests for g;
                  END
          END
        ELSE /* I am owner host but not home host */
          BEGIN
                Part[g].acc. = read-only;
                IF (requesting host is home host)
                        send read_data reply to home host;
                ELSE
                        send read_data reply to both home and requesting hosts;
          END
        return;
```

Figure 4. The algorithm for handling *read_data* paragraph faults

5. Discussions and Conclusions

We have presented the memory coherency protocol and algorithm of DICE DSM. The coherency protocol for this two-tier paging system is now being simulated in software. The performance of *DICE DSM* system has been studied using an analytical model [4], which derives an expression for *the speedup of the parallel part of an application* (or S_p). In this analysis, a high-speed and low-latency ATM LAN is chosen as the underline platform, and the queuing time on the network is assumed to be negligible. The memory access unit is assumed to be four bytes (or one word). Each page has P bytes and k paragraphs per page. An application is executed by N hosts, and uses M bytes of shared memory space. The behavior of an application is represented by the percentage of data memory accesses for total instructions (denoted by d); the probabilities of read and write faults (denoted by N_{rf} and N_{wf}, which are the number of read faults and write faults per 1,000,000 memory references per host); temporal locality (denoted by x_t , which is the number of times that the same paragraphs accessed continuously by a host); and spatial locality factor (denoted by x_s , which is the probability of a certain region of shared memory being accessed by a specific host). The temporal locality x_t is further represented by a step uniform distribution (with parameters N_0 , N_1 , and g, which are the starting pointer, ending pointer, and window size of this step), which approximates the bell-like normal distribution reflecting intuition that the chance of a memory location being accessed by a host decreases as the distance grows from the previously accessed location.

The effects of changing S_p on system structure and application behavior has been studied, and some of these results are shown below. Figure 7 shows that the gain in S_p becomes smaller and smaller as the network data rate R_n increases. This may justify the above assumption that the queuing time on the network is negligible in high-speed and low-latency network. Figure 8 shows that S_p decreases as processor speed R_p increases. Note that the total execution time for an application will still be reduced as R_p increases, although S_p decreases.

Figure 9 shows that S_p increases as the number of paragraphs per page, k, increases up to a certain point. After that point, S_p slightly decreases as k further increases. Further-

```
Write_data paragraph fault handler:
    IF (I am home host)
      BEGIN        /* "at owner host" state -> "at home  host" state */
          send write_data request to ParT[g].oid;
          receive write_data reply from ParT[g].oid;
          Pard[g].oid = myself;
      END
    ELSE
      BEGIN
          send write_data request to PT[p].hid;
          receive write_data reply from owner host;
      END;
    update local copy of g;
    ParT[g].acc = read-write;
    unblock host processor;
    IF (I am not home host and reply is not from PT[p].hid)
          send write_data confirmation to PT[p].hid;
    return;

Write_data paragraph fault server:
    IF (I am home host)
      BEGIN
          IF (no owner host)
            BEGIN          /* "at multiple hosts" state -> "at owner host" state */
                send invalidation request to all hosts in ParT[g].copyset;
                block processing future requests for g;
                receive all invalidation confirmations;
                ParT[g].acc = none;
                send write_data reply to requesting host;
                ParT[g].copyset = {};
                ParT[g].oid = requesting host;
                unblock processing future requests for g;
            END
          ELSE IF (I am owner host)
            BEGIN          /* "at home host" state -> "at owner host" state */
                ParT[g].acc = none;
                send write_data reply to requesting host;
                ParT[g].oid = requesting host;
            END
          ELSE /* owner host is not me */
            BEGIN          /* "at owner host" state -- no state change */
                forward write_data request to ParT[g].oid;
                block processing future requests for g;
                receive write_data confirmation from requesting host;
                ParT[g].oid = requesting host;
                unblock processing future requests for g;
            END
      END
    ELSE /* I am owner host but not home host */
      BEGIN
          ParT[g].acc = none;
          send write_data reply to requesting host;
      END
    return;
```

Figure 5. The algorithm for handling *write_data* paragraph faults

```
Handle write-access paragraph fault:
    IF (I am home host)
      BEGIN        /* "at multiple hosts" state -> "at home host" state */
          send invalidation request to all hosts in ParT[g].copyset;
          receive all invalidation confirmations;
          ParT[g].copyset = {};
          ParT[g].oid = myself;
      END
    ELSE
      BEGIN
          send write_access request to PT[p].hid;
          receive access confirmation from PT[p].hid;
      END;
    ParT[g].acc = read-write;
    unblock host processor;
    return;
Write_access paragraph fault server:
/* "at multiple hosts" state -> "at owner host" state */
    send invalidation request to all hosts (except requesting host) in ParT[g].copyset;
    receive all invalidation confirmations;
    ParT[g].copyset = {};
    ParT[g].acc = none;
    send access confirmation to requesting host;
    ParT[g].oid = requesting host;
    return;
Invalidation server:
    ParT[g].acc = none;
    send invalidation confirmation to home host;
    return;
```

Figure 6. The algorithm for handling *write_access* paragraph faults

more, S_p is approximately the same for a fixed paragraph size, which is P/k. This behavior demonstrates usefulness of the use of a paragraph with a smaller granularity than a page. Figure 10 shows a similar behavior, for S_p in relationship with the number of hosts N.

The analysis of this performance model demonstrates the effect of using paragraph which has a smaller granularity than a page. This smaller granularity reduces the probability of false sharing and the amount of data to be transferred over the network. The performance of DICE DSM is also going to be evaluated by a trace–driven simulation model, which will take consideration of network queuing delay and give more realistic results.

The concept of using paragraph is different from that of using cache line or from the ones just using small page size. Cache–based DSM has been used in multiprocessor systems, which needs to build their own interconnected network interface and use their own message–based communication scheme. In contrast, paragraph–based or page–based DSM is used the systems over LANs, using the existing network interface with standard packet–based or cell–based network communication protocols. As compared with small page size, paragraph reduces the complexity of the shared memory management due to the use of small size of page table and the two layered hierarchical page/paragraph structure while allowing a host to continue using the larger size of page as the trends in current memory design in uniprocessor computer systems. This reduction of complexity is also due to the using of home hosts in the protocol, which allows easily to locate the desired memory unit while distributing the management of shared memory over the hosts on a LAN.

References

[1] H. S. AlKhatib, Q. Li, C Jou, T. Chen, and H. Arafeh, "DICE – A Distributed Integrated Computing Environment for Multi–Threaded Parallel Processing," to be appeared in the *Proceedings of International Conference on System Integration*, August 15–19, 1994, Sao Paulo, Brazil.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," *The 13th ACM Symposium on Operating Systems Principles*, October 1990, pp. 152–164.

[3] B. D. Fleisch, G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989, pp. 211–222.

[4] C. Jou, H. S. AlKhatib, and Q. Li, Performance Analysis of DICE Distributed Shared Memory System, Dis–

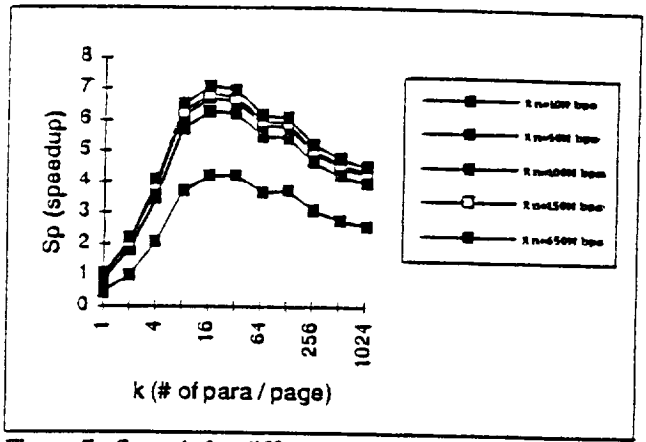


Figure 7. Sp vs k for different Rn. N=16, Rp=50Mips, M=64kbytes, P=4kbytes, d=0.4, Nrf=500, Nwf=10, Xs=0.5, N0=10, N1=100, g=100.

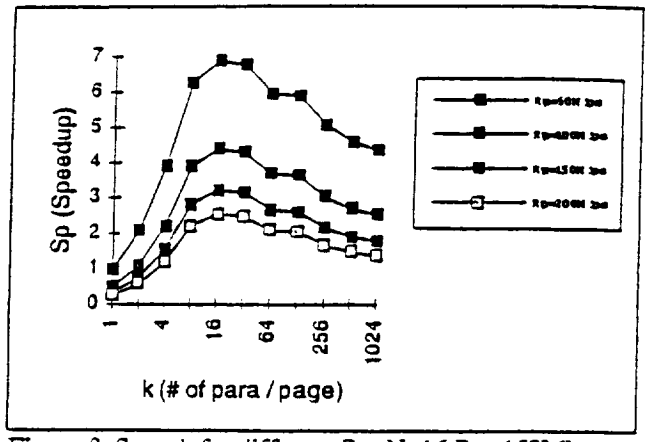


Figure 8. Sp vs k for different Rp. N=16,Rn=150Mbps, M=64kbytes, P=4kbytes, d=0.4, Nrf=500, Nwf=10, Xs=0.5, N0=10, N1=100, g=100.

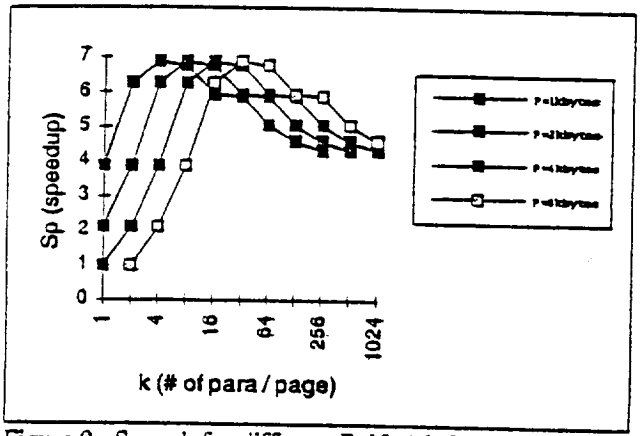


Figure 9. Sp vs k for different P. N=16, Rn=150Mbps, Rp=50Mips, M=64kbytes, d=0.4, Nrf=500, Nwf=10, Xs=0.5, N0=10, N1=100, g=100.

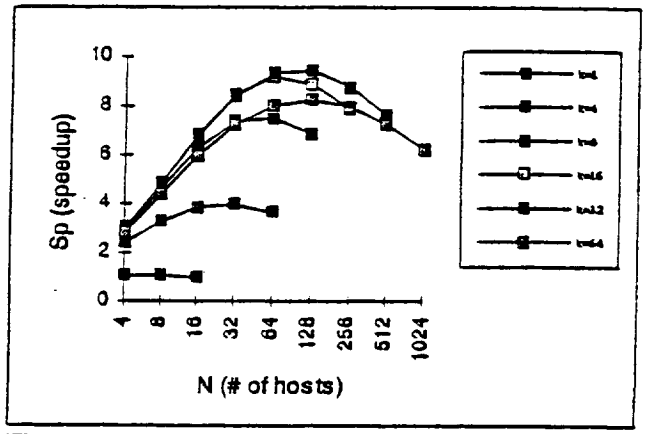


Figure 10. Sp vs N for different k. Rn=150Mbps,d=0.4, Rp=50Mips,M=64kbytes,P=4kbytes,Xs=0.5, Nrf=500, Nwf=10, N0=10, N1=100, g=100.

tributed Computing Lab Technical Report No. 03281994, Santa Clara University, 1994.

[5] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," In *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 94–101, August 1988.

[6] R. G. Minnich and D. J. Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

[7] U. Ramachandran, M. Ahamad, and M. Khalida, "Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory," *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 160–169, August 1989.

[8] M. Tam, J. M. Smith, and D. J. Farber, "A Taxonomy–Based Comparison of Several Distributed Shared Memory Systems," *ACM Operating System Review*, Vol. 24, No. 3, July 1990, pp. 40–67.

To appear in IEICE Transactions on Information and Systems.

A Two–Tier Paging Scheme for Network–based Distributed Shared Memory Systems

Chi–Jiunn Jou, Hasan S. AlKhatib, and Qiang Li

Abstract – Distributed computing over a network of workstations continues to be an illusive goal. Its main obstacle is the delay penalty due to network protocol and OS overhead. We present in this paper a low level hardware supported scheme for managing distributed shared memory (DSM), as an underlying paradigm for distributed computing. The proposed DSM is novel in that it employs a two–tier paging scheme that reduces the probability of false sharing and facilitates an efficient hardware implementation. The scheme employs a standard OS page and divides it into fixed smaller memory units called paragraphs, similar to cache lines.

An application address space is viewed as consisting of a shared data region, an unshared data region, a stack region and a code region. Code, stack and unshared data regions are handled by the OS in the standard manner without modification. The proposed scheme manages the shared data regions only. A hardware extension of a traditional MMU, Distributed MMU or DMMU, is introduced to support the DSM. Shared memory coherency is maintained through a write–invalidate protocol. An analytical model is built to evaluate the system sensitivity to various parameters and to assess its performance.

Keywords – distributed shared memory; false sharing; hardware support for distributed computing; memory coherency protocol; performance evaluation; networks of workstations.

1. Introduction

Despite the tremendous progress made in local area networking over the past decade and a half, the operating system and network protocol technologies have yet to address the main obstacle to distributed computing, namely the delay due to the network overhead. Network speed has reached several hundreds of Mbps, but the real issue is the network overhead latency in addition to sustained throughput.

*This work was supported by NASA–Ames Research Center grants number NCC 2–644 entitled "Parallel Processing for Scientific Computations".

The problem consists of a myriad of sub–problems, and is not simple to resolve. It requires a system–wide consideration on the full integration of networks into the operating system, and a re–examination of network protocols and the overall system architecture, including hardware support for both network protocols and the OS. This integrated view is underway in a project at Santa Clara University, called DICE, a Distributed Integrated Computing Environment [1]. DICE supports a distributed shared memory paradigm, DSM. This paper presents the design and performance of DICE DSM.

A number of DSM systems based on LANs have been developed over the past decade[18]. Among them, Ivy [13] is implemented on a network of Apollo workstations. The memory is paged, and copies of pages may be replicated in different hosts. A multiple–readers and–single writer strict coherency semantics is used on the page level. Memory coherency is maintained via a dynamic ownership protocol with a write–invalidate procedure. The owner of a page is located using either a centralized manager, a group of fixed distributed managers, or the individual host which forwards the request. Ivy is designed for multi–threaded applications. All threads share the same virtual address space. False sharing may occur in this system, since its consistency or access unit (e.g. word) is less than the sharing unit (page). In addition, the single–writer nature of its protocol may cause a "ping–pong" behavior between multiple writers of a shared page, leading to thrashing.

The problems of false–sharing and thrashing have been addressed by other DSM systems. Clouds [15] avoids them by using a single–writer–single–reader strict coherence semantics introducing instead significant blocking delays. Mirage [9] reduces thrashing by using a time window scheme, in which the system guarantees that the writer of a page retains access to a page for a fixed period of time, suffering again from blocking delays. Munin [3] handles it by using multiple consistency protocols and software release consistency, hence placing the burden on the user. Mether [14] eliminates false sharing and thrashing by ignoring memory coherency altogether, leaving its burden to the application software.

DICE represents a novel approach to handling the problem of false sharing and thrashing. The shared por-

tion of memory is structured as a two–tier paging system. The first tier is a normal *page*, and the second is called a *paragraph*, which is a smaller fixed–size block of memory within a page. Coherency is maintained at the level of a paragraph. The introduction of paragraphs improves system performance by reducing the probability of false sharing as well as the size of the unit of information transferred over the network for maintenance of memory coherency. A Distributed Memory Management Unit, *DMMU,* an extension of the traditional MMU, is designed to support the paragraph validation, and a special network controller is used to support the accesses to the remote memory and the maintenance of memory coherence.

Section 2 of this paper gives the overview of the DICE architecture. The design of the DICE distributed shared memory is described in section 3. An analytical model and the expected system performance are presented and discussed in section 4. Section 5 concludes this work and compares it to other approaches.

2. Overview of the DICE Architecture

DICE is an experimental distributed environment for executing multi–threaded tasks. A parallel task may consist of multiple threads that can be scheduled to run simultaneously on a cluster of workstations. Threads executing on separate workstations share the same virtual address space, and communicate with each other using shared memory. Synchronization of threads accessing shared resources is done using functions provided by a distributed run–time library.

DICE consists of three interactive subsystems. The *DSM* provides the underlying communication paradigm among threads of a parallel task. The *DRS* (distributed run–time subsystem) provides users with programming tools to develop and execute DICE multi–threaded applications. The *PS* (parallel scheduler) is a self–optimizing application–specific scheduler, and is responsible for thread scheduling and synchronization.

3. Design Issues of the DICE DSM

DICE DSM is designed for a cluster of workstations connected via a high–speed, low–latency local area network. The architecture of a node in a DICE system is shown in Figure 1. Each node consists of a host processor and a physical memory module. The traditional MMU is replaced by a DMMU. The network interface is attached directly to the memory bus and contains a network processor and a dual ported memory visible

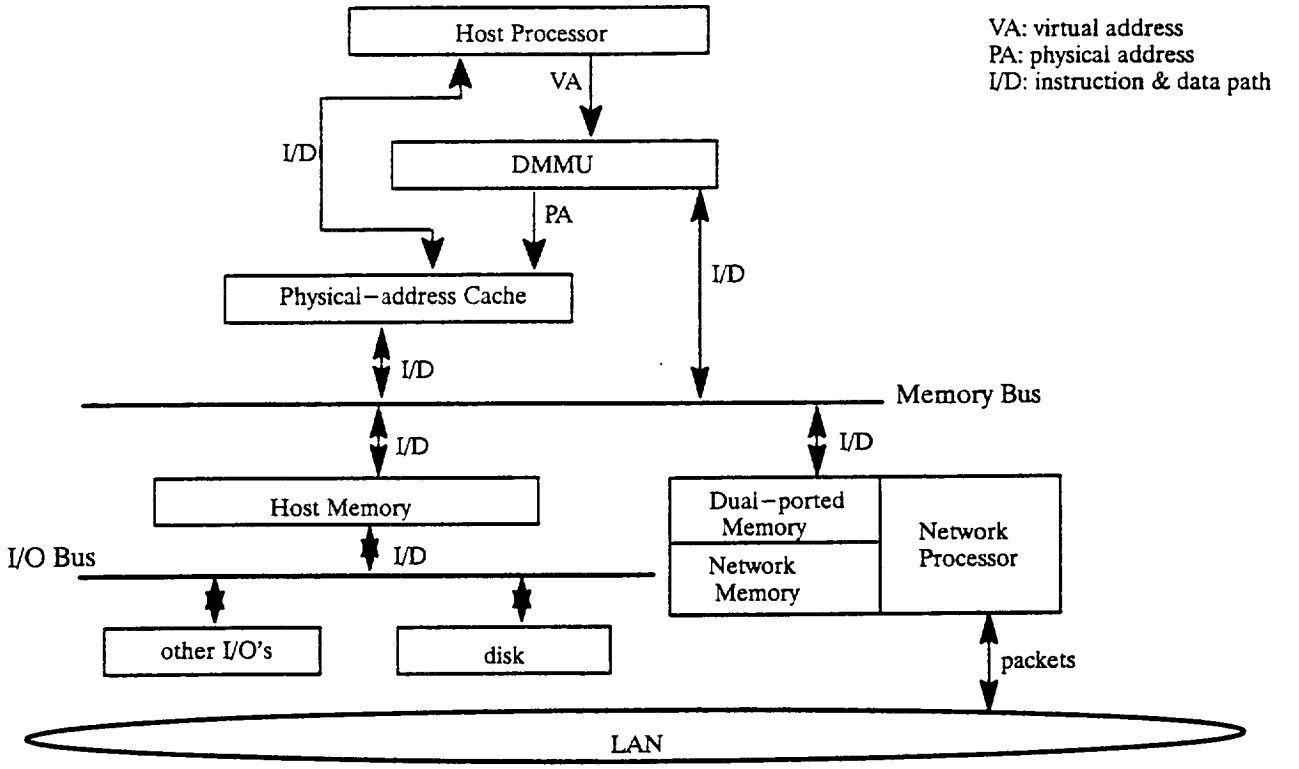


Figure 1. The Architecture of a DICE Node

both to the host and network processors, simultaneously. The dual ported memory holds data structures for managing the shared memory.

3.1. Programmer's View of DICE DSM Environment

In DICE, a parallel task consists of multiple threads that can run on a cluster of workstations (nodes), simultaneously. Memory pages required by each thread, whether code or data, are allocated physical memory blocks, at the respective node, where the thread is running. Shared data pages are distributed and replicated among the nodes as needed by the threads. The DSM system is designed to support the sharing of data pages. The DSM system also maintains the coherency among replicated data copies.

Each parallel task has a *root node*, on which it was first loaded and executed. The root node maintains state information for all pages, including shared pages used in the application, while other nodes maintain the state information for the pages that are loaded in their local systems.

Code and non−shared data pages of a thread are loaded in the physical memory of the node where

the thread is scheduled for execution. Shared data pages, on demand, are first loaded into the physical memory of the node. That node becomes the *home* node for the page. A home node maintains the complete state information for its pages. It ensures that the last copy of a page is not purged, and keeps track of all copies of paragraphs belonging to its pages. Other nodes in the cluster, that have a copy of a shared page, keep a pointer to the page's home node. When a thread makes an attempt to access a page for which it does not have a copy, it interacts with the home node of that page in order to complete the memory access. When a node does not know the home for a certain page, a *home–info* fault is triggered and a *home–info* request is sent to the root node. The root node replies with the information about the home node for the requested page. If a home is not yet assigned for the page, the root node assigns the first requesting node the status of home for that page. The root node then updates its table and sends the page to the requesting node. The requesting node, upon receiving the page and the assignment of home status, updates its page table and creates a paragraph map table for that page.

3.2. Coherency Protocol

The memory coherency of DICE DSM is maintained at the paragraph level. A paragraph can simultaneously be read by multiple nodes, but it can only be written by one node at a time. Access rights to a paragraph can be *read–write*, *read–only*, or *none*. An *owner* node of a paragraph is the node that has *read–write* access to that paragraph. The ownership of a paragraph may be transferred from one node to another upon demand. There is no owner for a paragraph, when two or more hosts have *read–only* access rights to that paragraph. The Information about the owner of a paragraph is maintained by the home node of the page containing the paragraph.

When a read operation is issued to a paragraph by a node with *none* rights, a *read* fault is triggered and a *read* request is sent to the paragraph's home. When a write operation is issued to a paragraph by a node with *none* rights, a *write–data* fault is triggered and a *write–data* request is sent to the paragraph's home. When a write operation is issued to a paragraph by a node with *read–only* access rights, a *write–access* fault is triggered and a *write–access* request is sent to the paragraph's home. In each case the home directly or indirectly re-

sponds with the requested information. The coherency of paragraphs is basically maintained through a write–invalidate protocol. The details of this protocol and its algorithm is shown in [11].

3.3. Management of Shared Memory

Page and paragraph tables are used to maintain the state information for shared pages and their paragraphs, respectively. Each DICE application maintains its own set of these tables. A *Page Table* (*PT*), similar to a traditional page table, provides the information about mapping the virtual addresses of pages to their corresponding physical addresses, at their respective nodes. A *Paragraph Validation Table* (*PVT*), maintains the information about the access rights of the page's paragraphs. Each entry of a *PVT* contains a 2–bit field maintaining the access rights of the local node to the respective paragraph. Note that there is no address translation for paragraphs. Each node keeps a *Page Table for Home information* (*PTH*), which maintains the information about the homes for its shared pages. Each home node of a page maintains a *Paragraph Table* (*ParT*) for that page containing a pointer to the current owner of each paragraph and a list of nodes with read–only copies of the paragraph. There is only one *ParT* for a page in the system. It is maintained by the home node of that page. The PT and PVT are maintained in the dual–ported memory, inside the LAN interface. They are used by both host and network processors. The PTH and ParT are maintained in the network subsystem, and are only used by the network processor. Figure 2 shows the data structures for these tables.

DMMU is an extension of the traditional MMU. It is designed to support paragraph validation for efficient handling of distributed shared memory. When data is not available locally and needs to be fetched from a remote node, the *DMMU* triggers special access faults via an embedded hardware unit, *PVLB* (Paragraph Validation Lookaside Buffer) – to validate the access rights of paragraphs. The *DMMU* performs the traditional TLB operations for all non–shared pages as well. When the *DMMU* does not find the entry it needs in its TLB, it fetches the entry from the appropriate PT in memory. When an entry is loaded from the PT into the TLB, all entries of its associated PVT (2 bits per paragraph) are simultaneously fetched and stored into the associated *PVLB*. When an entry of the TLB is replaced, all entries of its associated *PVLB* are also replaced. Note, there are no *PVLB*s for non–shared pages.

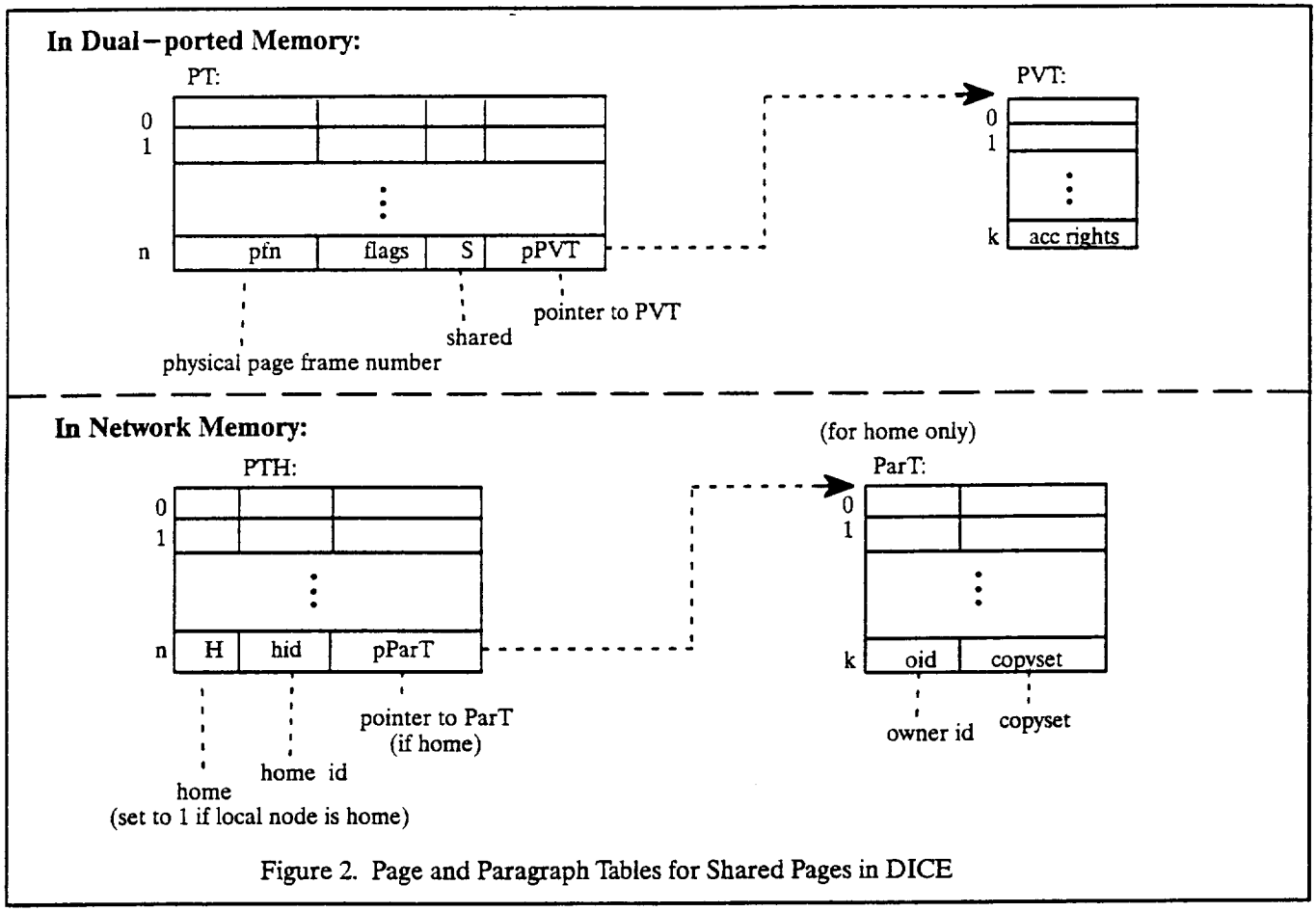
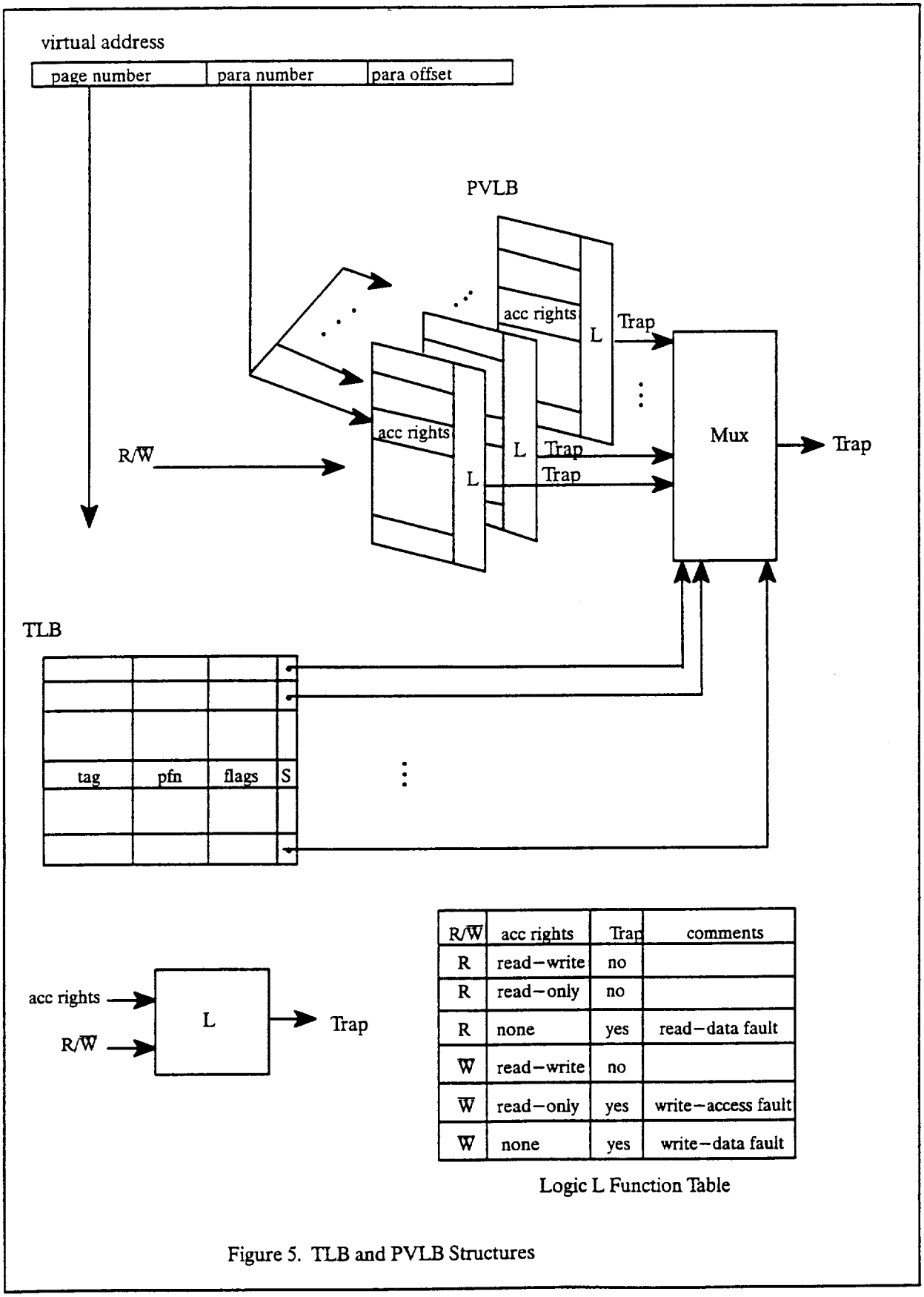


Figure 2. Page and Paragraph Tables for Shared Pages in DICE

Figure 3 shows the structure of the TLB and the *PVLB*. Each entry in the TLB contains an address tag, a physical page frame number, flags, and an *S* bit. The *S* bit is used to distinguish shared pages from non–shared pages. Each TLB entry of a shared page has an associated *PVLB*, which has *k* two–bit *access rights* fields, where *k* is the number of paragraphs within a page. The virtual address is grouped into three fields: a page number, a paragraph number, and a paragraph offset. The page number is used as a key to match the address tags in the TLB, while the paragraph number directly addresses the *PVLB* entries corresponding to the same paragraph number. The latter operation will simultaneously select *n* PVLB entries, where *n* is the number of PVLBs in the *DMMU*. Each PVLB has an associated logic *L*, which validates the access rights of the referenced paragraph. By checking the stored two–bit *access rights* field and the current memory access type $R/\overline{W}$, logic *L* generates a *Trap* signal. The *Trap* signal is *ON* when any paragraph validation fault occurs. The Trap causes a system trap and requires the software to distinguish the type of the current access fault and resolve it.



R/W	acc rights	Trap	comments
R	read–write	no	
R	read–only	no	
R	none	yes	read–data fault
W	read–write	no	
W	read–only	yes	write–access fault
W	none	yes	write–data fault

Logic L Function Table

Figure 5. TLB and PVLB Structures

If there is no Trap, the physical access to the paragraph proceeds without interruption. The function of logic L is shown in the table inside Figure 3. The S bit of the selected TLB entry is used as a gate to control the final selection of the *Trap* signal generated from the previously selected n PVLB entries. Note that the operations on the *PVLB* are executed in parallel with the operations on the TLB, except for the final selection of the *PVLB* output. Hence, if a memory reference does not generate a paragraph validation Trap, no significant extra delay will be suffered by going through this additional PVLB unit compared to a traditional MMU.

The control unit of the *DMMU* contains the logic to manage the retrieval of entries from the PTs and the PVTs in the dual–ported memory. It also controls the TLB and PLVB update operations, and handles other related activities. When the retrieval of the entries of the PVT fails, the DMMU triggers a *PVT* trap resulting into a *home–info* fault as described in section 3.1. Other paragraph validation faults are generated by the PVLB as described above.

4. Performance Analysis

The performance of a DICE DSM system is mainly affected by the delays encountered in handling different paragraph validation faults, which in turn depends on the execution delay of messages sent over the network to resolve paragraph faults. In the following analysis, a performance metric is first defined. The system and network model is presented. Thereafter, the application behavior model along with the protocol cost are described. Finally, the performance results for different combinations of system configurations and application profiles are shown and discussed.

4.1. Performance Metric

The performance of parallel systems is often measured in terms of *speedup*, which is the ratio of the execution time of a program run on a single processor to that run on a parallel system. We limit ourselves to the speedup for the parallel part of an application only. We define *the speedup for the parallel part of an application*, S_p , as the ratio of the execution time of the parallel part of an application running on a single processor to that running on a DICE DSM system.

Let us denote T_s and T_{dsm} to be the total execution time for the parallel part of an application by a single node and by N nodes in a DICE DSM system, respectively. Let the processor speed of a single node be denoted by R_p MIPS. Let the total number of instructions required to be executed in the parallel part of the application be denoted by I_a, and the average rate of shared data memory accesses per instruction be denoted by d_s. Then,

$$T_s = \frac{I_a}{R_p} \qquad \text{and} \qquad T_{dsm} = \frac{1}{N}\left(\frac{I_a}{R_p} + d_s I_a T_{pcost}\right) \tag{1}$$

where T_{pcost} denotes the average protocol cost per shared data memory access, and will be derived in the following subsections, using an analytical system model. The term $d_s I_a T_{pcost}$ represents the total overhead, when using the DICE DSM. The speedup for the parallel part of an application S_p is therefore:

$$S_p = \frac{T_s}{T_{dsm}} = \frac{N}{1 + d_s R_p T_{pcost}} \tag{2}$$

4.2. Network and System Model

In this analysis, a high–speed, low–latency ATM network is assumed to be the underlying local computer network. The queuing time on the network is assumed to be small enough to be neglected. (A future study is examining the effects of queuing delays.) The memory access unit is assumed to be one word (or four bytes). Each paragraph has G words. An application is executed by N nodes.

A typical ATM network consists of a set of nodes connected via a mesh of switches. In an ATM network, data is segmented into small fixed–length cells, routed, then reassembled at the destination using header information contained in the cells. Due to the efficient structure of ATM frames, the waiting time for accessing the network can be designed to be very short. In this model, each network message with length L_{msg} takes $t_{fix} + n_{cell}\, t_{var}$ processing time at the transmitting and the receiving nodes, $n_{cell}\, L_{cell}/R_n$ transmission time, and $n_{cell}\, t_{net}$ processing time through an ATM switch; where n_{cell} is the number of cells needed to transmit the whole message, or the ceiling of $L_{msg}/(L_{cell} - L_{hd})$; L_{cell} and L_{hd} are cell size and header lengths, respectively; t_{fix} and t_{var} are fixed and variable parts of processing delays in the communicating nodes, respectively; R_n is

the network data rate; t_{net} is the average network switch latency a cell goes through in a typical ATM network. Note that the processing time at the nodes includes the time for copying data between host memory and network buffer, network processor latency, interrupt handling on reception of frames, and segmentation/reassembly times.

The protocol cost is analyzed based on the time it takes for handling different kinds of paragraph validation faults. This analysis includes all but *home–info* faults, since they only occur when a page is accessed by a node for the first time. The fault handling time is expressed in terms of the total time for handling network messages, including required interrupt handling delays at the local and remote nodes.

The whole message for either *fault* request or *invalidation* request can fit into a single ATM cell. The messages for data reply will have the size of a paragraph, which may need one, two or more ATM cells depending on the size of the paragraph. The costs for these two different sizes of network messages, denoted by *request messages, msg–r,* and *data messages, msg–d,* are

$$t_{msg-r} = \frac{L_{cell}}{R_n} + t_{fix} + t_{var} + t_{net} \quad (3)$$

$$t_{msg-d} = \left\lceil \frac{G}{L_{cell} - L_{hd}} \right\rceil \frac{L_{cell}}{R_n} + t_{fix} + \left\lceil \frac{G}{L_{cell} - L_{hd}} \right\rceil (t_{var} + t_{net}) \quad (4)$$

From the memory coherency protocol, one can count the number of network messages involved in each kind of fault. This message count also depends on the home and owner node relationship, as well as the number of nodes within the copyset (the list of nodes with *read–only* copies of a paragraph), when a fault occurs. After examining the protocol, one concludes that the cost of message are as follows: $t_{msg-r} + t_{msg-d}$ for case *e1* and case *nrd*, $2t_{msg-r} + 2t_{msg-d}$ for case *e2*, $(2N_{set} + 1)t_{msg-r} + t_{msg-d}$ for case *nwd*, and $2N_{set}\, t_{msg-r}$ for case *nwa*. Here, N_{set} denotes the number of nodes within the copyset, when a fault occurs. Cases *e1* and *e2* represent the situation when a fault occurs while the *copyset* on *the home node* is empty. The former is the case when the owner is *the home*, or when the requesting node is the *home node*. The latter is the case when the owner is not *the home* and the requesting node is not the *home node*. Cases *nrd* and *nwd* and *nwa* represent the situa-

tions for a *read* fault, a *write–data* fault, and a *write–access* fault occurrence, when the *copyset* on *the home node* is not empty, respectively.

The average time spent for handling a paragraph fault depends on the probability of each of the above cases as well as the probability of the number of nodes within the copyset, when a fault occurs. These probabilities are estimated by simple probability models in this work. When a fault occurs, each node has equal probabilities of $1/N$ for having accessed and of $(1 - 1/N)$ for not having accessed this paragraph since the last time the copyset was empty. Hence, the probability that the copyset is empty, when a fault occurs, is the case that either none or any one node having accessed this paragraph. The probability that the number of nodes within the copyset is i, when a fault occurs, denoted by $p\{N_{set} = i\}$, is the case when any $i+1$ nodes have accessed the paragraph. Therefore, we have

$$P\{N_{set} = 0\} = \binom{N}{0}\left(\frac{1}{N}\right)^0 \left(1 - \frac{1}{N}\right)^N + \binom{N}{1}\left(\frac{1}{N}\right)^1 \left(1 - \frac{1}{N}\right)^{N-1} = \left(2 - \frac{1}{N}\right)\left(1 - \frac{1}{N}\right)^{N-1} \qquad (5)$$

$$P\{N_{set} = i\} = \binom{N}{i+1}\left(\frac{1}{N}\right)^{i+1} \left(1 - \frac{1}{N}\right)^{N-i-1} \quad \text{for } i = 1, 2, 3, \ldots, N-1 \qquad (6)$$

In the DICE DSM, it is expected that a paragraph is accessed by its home node most frequently. Let x_s denote the probability that a paragraph is accessed by its home node. Other nodes are assumed to exhibit a paragraph access probability that is uniformly distributed among all the non–home nodes with a total probability of $1 - x_s$. Note that x_s reflects the processor locality of parallel program behavior as described in [8].

The probability of each case is estimated by finding the conditional probabilities of each case, when either read or write access faults occur. The probability of case *nrd* fault is 1. The rest of the probabilities are

$$p_{e2} = \frac{(N-2)(1-x_s)}{2(N-1)x_s + (N-2)(1-x_s)} \quad , \quad p_{e1} = 1 - p_{e2} \qquad (7)$$

$$p_{nwd}(N_{set}) = \frac{(N-1-N_{set})(1-x_s)}{(N-1)x_s + N_{set}(1-x_s) + (N-1-N_{set})(1-x_s)} \quad , \quad p_{nwa}(N_{set}) = 1 - p_{nwd}(N_{set}) \qquad (8)$$

where p_{e1}, p_{e2}, p_{nwd} , and p_{nwa} are the probabilities of case *e1*, case *e2*, case *nwd*, and case *nwa*, respectively.

The average time spent for handling a paragraph read or write fault, denoted by t_{rf} and t_{wf}, can be obtained from Equations (3) through (8). After some simplification, we have

$$t_{rf} = (1 + P\{N_{set} = 0\}p_{e2})(t_{msg-r} + t_{msg-d}) \quad (9)$$

$$t_{wf} = [P\{Nset = 0\}(1 + p_{e2}) + \sum_{i=1}^{N-1} P\{N_{set} = i\}p_{nwd}(i)](t_{msg-r} + t_{msg-d}) + [\sum_{i=1}^{N-1} iP\{N_{set} = i\}](2t_{msg-r}) \quad (10)$$

4.3. Application Behavior Model and Average Protocol Cost

Torrellas et al. [19] proposed a model of sharing, which is classified into *true sharing* and *false sharing*. Based on this sharing model, we divide the average protocol cost, T_{pcost}, into two parts: one part is caused by *true sharing misses*, the other part is caused by *false sharing misses*. A miss is a true sharing miss, when a processor or node misses, because the word was previously used by another node. A false sharing miss is caused by multiple processors or nodes accessing different words within the same paragraph.

In this analysis, we first consider the application behavior independent of system architecture. The sharing misses are based on an access unit (word), as the same way in the work done by Eggers and Katz in [7], instead of a coherency unit (paragraph). Then, we integrate it with the effects of using a paragraph size consisting of multiple words.

True sharing misses are varied significantly for different parallel applications, since they inherently depend on the program behavior. True sharing misses are expected to increase as the number of processors or nodes increases, since the frequencies and degrees of sharing increase. Hence, we use a simple linear relationship to model this behavior. Let f_r and f_w denote the average rate of read faults per shared read data memory access and average rate of write faults per shared write data memory access, respectively. Then, we have

$$f_r = f_{r0} + f_{rx}N \quad \text{and} \quad f_w = f_{w0} + f_{wx}N \quad (11)$$

where f_{r0} and f_{w0} are the base points of f_r and f_w, respectively; f_{rx} and f_{wx} are the incremental rates of f_r and f_w,

when the number of nodes is changed, respectively. Note that f_r and f_w reflect the temporal locality of parallel program behavior.

When paragraphs, larger than a single word, are taken into account, the true sharing misses are expected to drop as the paragraph size increases. This is due to the spatial locality of a parallel program behavior, and the neighboring data having been prefetched before being used. Note that we consider the sharing misses only caused by the coherency protocol, and ignore those caused by insufficient physical memory to allocate space. We use the *ratio of miss ratios*, proposed by Smith in [16], to model the effects of this behavior. Let m_{r1} and m_{rr} denote the ratio of miss ratios when a paragraph size is G to that when a paragraph size is one word, and when a paragraph size is G to that when a paragraph size is $G/2$, respectively. Then, we have

$$m_{r1} = m_{rr}^{\log_2 G} \tag{12}$$

Several research results [2,6,18] indicate that false sharing will be increased, when either the number of processors or the coherency unit size is increased. Hence, we also use a simple linear relationship to model this behavior. Let ϵ_{fs} denote the probability of false sharing misses. Then, we have

$$\epsilon_{fs} = \epsilon_{fs0} + \epsilon_{fsx} N + \epsilon_{fsy} G \tag{13}$$

where ϵ_{fs0} is the base point of ϵ_{fs} ; ϵ_{fsx} and ϵ_{fsy} are the incremental rates of ϵ_{fs} , when N and G are changed, respectively.

Combining Equations (9) through (13), one can derive the average protocol cost T_{pcost} as

$$T_{pcost} = m_{r4}[(1 - w)f_r t_{rf} + w f_w t_{wf}] + \epsilon_{fs}[(1 - w)t_{rf} + w t_{wf}] \tag{14}$$

where w denotes the average rate of write operations per shared data memory access. In the above equation, the two terms on the right side represent the protocol costs caused by true sharing misses and false sharing misses, respectively.

4.4. Analytical Results

This section shows the effects of changing system structure and application profile on the speedup, S_p . A

typical value for each parameter is chosen to reflect a typical system architecture and a target parallel application profile. We analyzed the effects on S_p by only changing one or two parameters at a time and fixing other parameters to their typical values.

For program behavior parameters, the typical degree of sharing and access pattern are chosen to be 0.1 for both d_s and w. The typical fault rates are chosen to be 0.001 for both f_{r0} and f_{w0}, and 0.001 for both $f_{r\infty}$ and $f_{w\infty}$. The typical locality factors are chosen to be 0.6 and 0.5 for m_{rr} and x_s. Typical false sharing factors are chosen to be 0.0001 for ε_{fs0} and 0.00001 for both ε_{fsx} and ε_{fsy}. These typical values are intended to represent the suitable network–based DSM applications and to reflect the significant effects of localities as well as false sharing.

For system parameters, the lengths for an ATM cell and header are fixed to 53 and 5 bytes, respectively. Other parameters are varied to reflect the changes in of system technology and architecture. The typical system is chosen to have 16 nodes and 100 MIPS. The typical network data rate is chosen to be 150 Mbps. The typical ATM processing time is chosen to be 10 and 20 microseconds for t_{fix} and t_{var}, respectively. This is derived from the actual measurements of an ATM host–network interface in [4]. While, t_{net} is chosen to be 10 microseconds, which corresponds to the store–forward delay time of a single switch for an ATM LAN.

Figures 4 through 13 show the expected behavior of S_p, when the size of a paragraph, G, is changed. This behavior indicates that S_p increases as the paragraph size G increases up to a certain point. After that point, S_p starts decreasing as G further increases. The peak values of S_p is when the paragraph size G is between 32 and 256 bytes. This is less than the page size used in a common operating system. This behavior demonstrates the advantage of using a two–tier paging scheme. Note that the fixed small cell size (53 bytes) used in ATM networks leads to the abnormal dent at a granularity of 64 bytes shown in Figures 5 through 9 and 11.

Figures 4 and 5 show that S_p decreases as the average rate of shared data memory accesses per instruction d_s, and the average rate of write operations per shared data memory access w increases, respectively. Figures 6

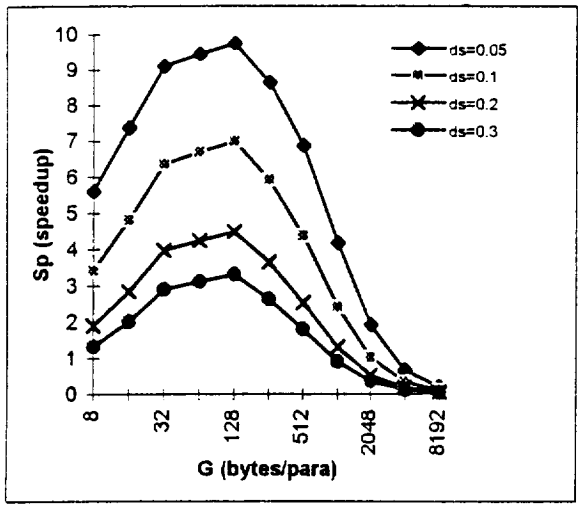


Figure 4. Sp vs G for different ds

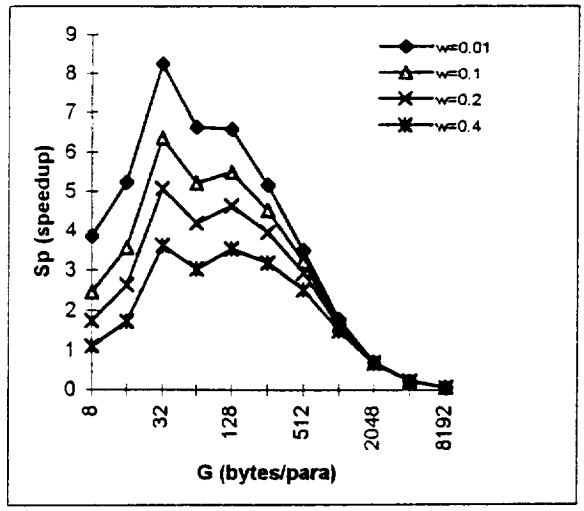


Figure 5. Sp vs G for different w

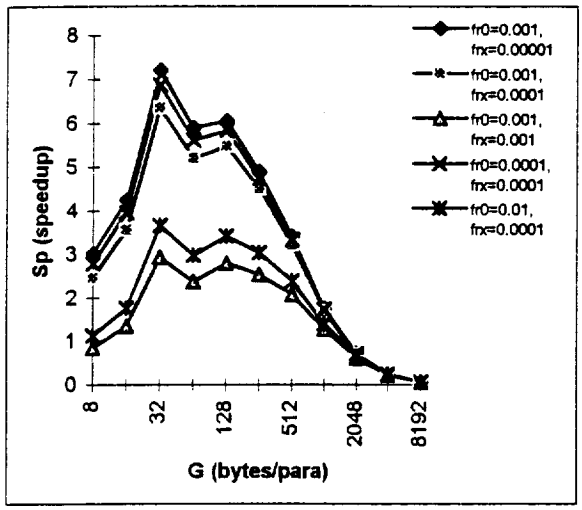


Figure 6. Sp vs G for different fr0 and frx

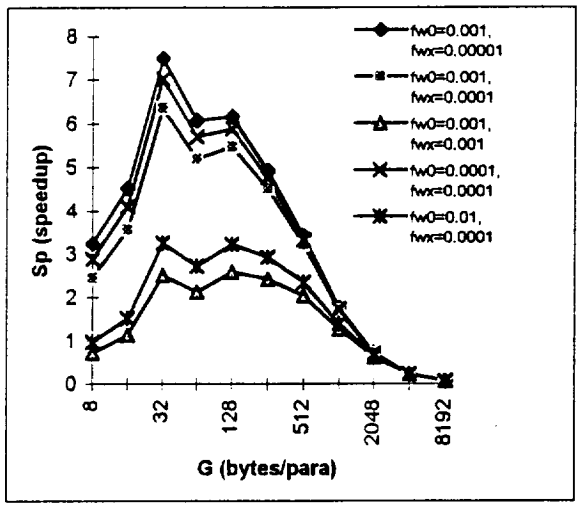


Figure 7. Sp vs G for different fw0 and fwx

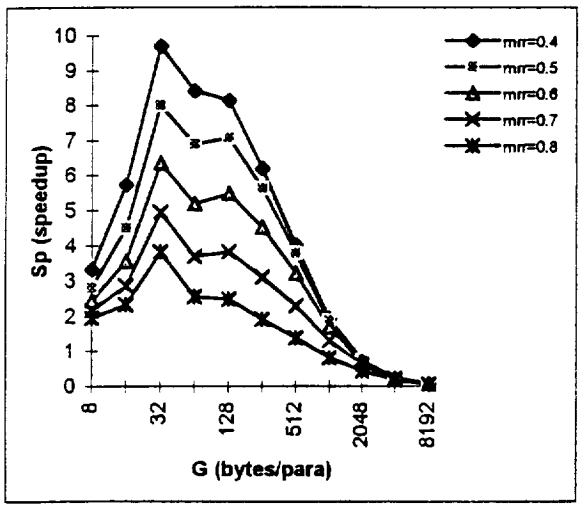


Figure 8. Sp vs G for different mrr

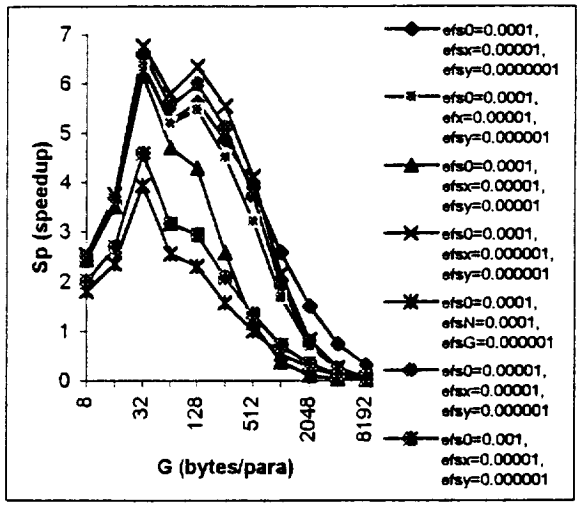


Figure 9. Sp vs G for different efso,efsx, and efsy

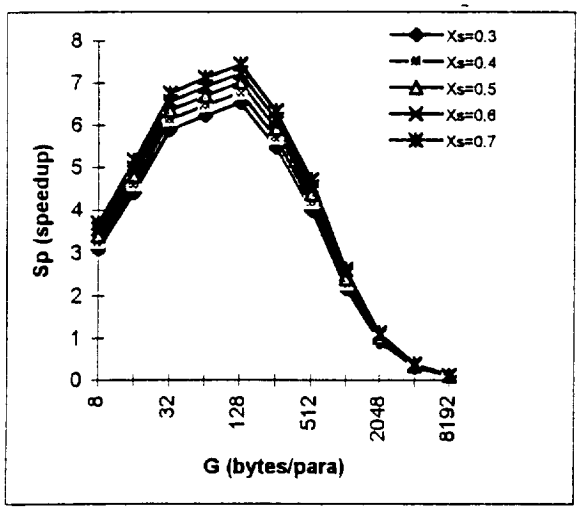


Figure 10. Sp vs G for different Xs

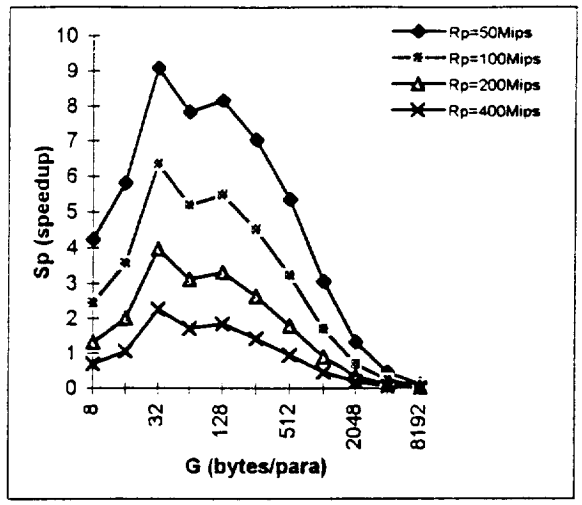


Figure 11. Sp vs G for different Rp

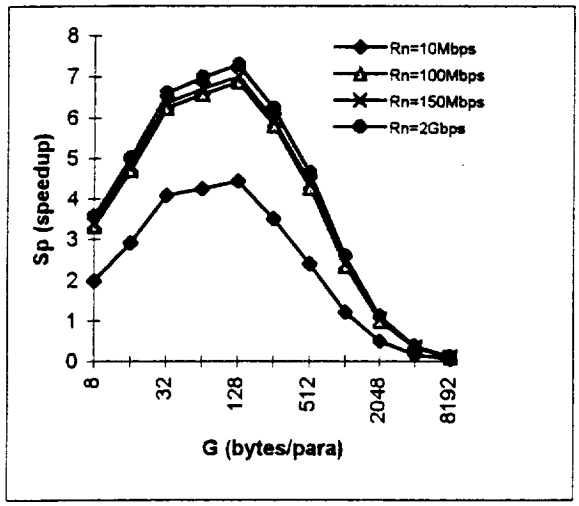


Figure 12. Sp vs G for different Rn

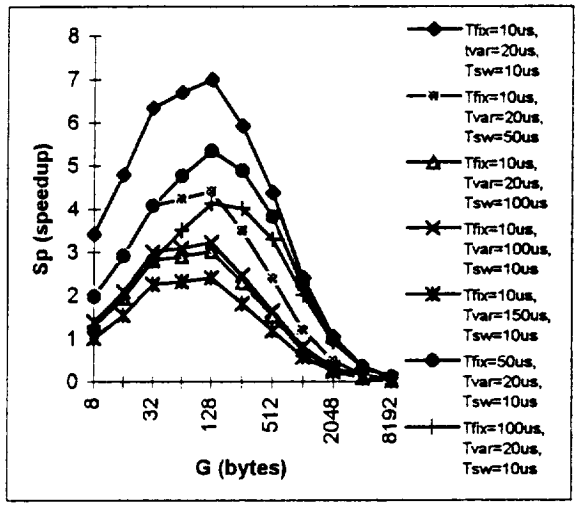


Figure 13. Sp vs G for different Tfix,Tvar, and Tsw

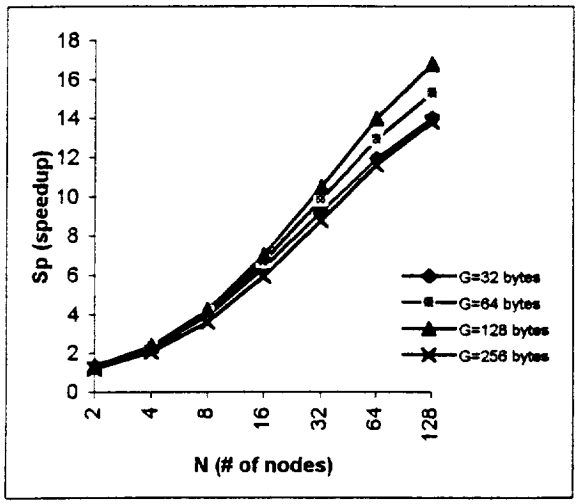


Figure 14. Sp vs N for different G

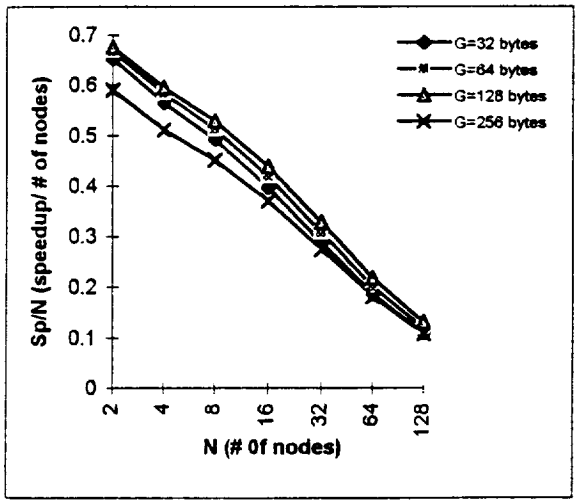


Figure 15. Sp/N vs N for different G

and 7 show that S_p decreases as the fault rate parameters (i.e. f_{r0}, f_{w0}, f_{rx}, and f_{wx}) increase. Figure 8 shows that S_p decreases as the ratio of miss ratios m_r increases. Figure 9 shows that S_p decreases as the false sharing parameters (i.e. ε_{fs0}, ε_{fsx}, and ε_{fsy}) increase. Figure 10 shows that S_p increases as the probability of a paragraph being accessed by its home node x_s increases.

Figure 11 shows that S_p decreases as the processor speed R_p increases, as the benefits of parallel processing diminish due to the increase in ratio of network overhead to execution time on each node. Note that the total execution time for an application will still drop as R_p increases, although S_p decreases. This asserts an important expected fact that as processor speeds increase, it is important to reduce network overhead in order to accomplish the same high level of speedup.

Figure 12 shows that S_p increases as the network data rate R_n increases, and that the margin of gain in S_p becomes smaller and smaller as the network data rate R_n increases. Figure 13 shows that S_p decreases as the ATM processing and switching times (i.e. t_{fix}, t_{var}, and t_{net}) increase.

Figures 14 and 15 demonstrate the relationship of S_p and S_p/N with the number of nodes for different paragraph sizes, respectively. S_p increases as N increases, and the margin of gain in S_p becomes smaller when N is large.

5. Conclusions

In this paper, we present the design of a two−tier paging system for distributed shared memory, where a paragraph, a much smaller memory unit than a page, is employed as the unit of coherency. The system is modeled and the analysis demonstrates the benefits of the multiple granularity memory management. The problem of false–sharing is alleviated, especially for systems with large page size and large objects. The network latency for coherence maintenance is significantly reduced, since only a small amount of data has to be transferred across the network for each remote memory access fault. Furthermore, the overhead of the coherency protocol processing is reduced by introducing hardware support.

The proposed two–tier paging scheme is different from the two–level paging method used in a uniproces-

sor system. The latter bears two levels of address translations. In our two–tier paging design, the page is the only address translation unit and the paragraph is the validation unit. There is no address translation for paragraphs.

The concept of using a second tier page, namely a paragraph, is different from that of using a cache line. The size of a paragraph is normally larger than a cache line. Although the paragraph coherency protocol and algorithm is similar to the one used in cache–based DSM multiprocessor systems, the design and implementation consideration are quite different. In a network based distributed shared memory system communication latency is significantly higher than that seen in a multiprocessor distributed shared memory system such as DASH [12]. Network based DSMs are implemented in software with hardware support, while multiprocessor based DSMs are implemented in hardware. Therefore, the allocation of and access to the coherency directories are quite different.

The use of paragraphs as opposed to using a small page size reduces the complexity of the shared memory management. If a small page size is used, very large page map tables will be required. By preserving the large page size and using paragraphs only for shared pages the page map tables stay small and additional paragraph map tables are needed for shared pages only. In addition to using the home node scheme we have distributed the management of paragraphs to the home nodes of the pages only. Hence, the root node acts as the clearing house for all application pages, and the home nodes act as the clearing houses for the paragraphs in their respective pages to which they are home.

A trace–driven simulation model that takes into consideration network queuing delays is under development. This simulation model will be used to validate the analytical model described in section 4. This simulation model is built with BONeS DESIGNER[5], and the traces are generated by Tango Lite[10] when running the parallel applications of Stanford SPLASH[16].

The current DICE DSM design is based on a strict consistency model and a write–invalidate coherency protocol. Extensions by using multiple consistency and coherency protocols are under consideration. In future version of DICE we plan to incorporate support for a relaxed consistency model to hide the large latency of

remote memory accesses by allowing buffering and merging.

Acknowledgements

The referees provided valuable comments on the contents of this work and the presentation of this paper.

References

[1] H. S. AlKhatib, Q. Li, C Jou, T. Chen, and H. Arafeh, "DICE – A Distributed Integrated Computing Environment for Multi–Threaded Parallel Processing," *Proceedings of the Third International Conference on System Integration*, Vol. 1, August 1994, pp. 612–621.

[2] W. J. Bolosky, and M. L. Scott, "False Sharing and its Effect on Shared Memory Performance," *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, September 1993, pp. 57–72.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," *The 13th ACM Symposium on Operating Systems Principles*, October 1990, pp. 152–164.

[4] A. T. Chandramohan, and H. M. Levy, "Limits to Low–Latency Communication on High–Speed Networks", *ACM Transactions on Computer Systems*, Vol. 11, No. 2, pp. 179–203, 1993.

[5] Comdisco Systems, Inc. BONeS DESIGNER User's Guides. Comdisco Systems, Inc., 1993

[6] S. J. Eggers, and T. E. Jeremiassen, "Eliminating False Sharing," *Proceedings of the 1991 International Conference on Parallel Processing, I – Architecture*, August 1991, pp. 377–381.

[7] S. J. Eggers, and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation", *Proceedings of the 15th International Symposium on Computer Architecture*, May 1988, pp. 373–382.

[8] S. Eggers, and R. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," *Proceedings of the Third ASPLOS*, April 1989, pp. 257–270.

[9] B. D. Fleisch, G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989, pp. 211–222.

[10] S. R. Goldschmidt, Simulation of Multiprocessors: Accuracy and Performance, Ph.D. Thesis, Stanford, 1993.

[11] C. Jou, H. S. AlKhatib, Q. Li, and A. T. Chen, Coherency Protocol and Algorithm of The DICE Distributed Shared Memory System," *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, October 1994, pp. 796–801.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, and J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessors", IEEE Computer, Vol 25, No. 3, March 1992, pp. 63–79.

[13] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," In *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 94–101, August 1988.

[14] R. G. Minnich, Mether: A Memory System For Network Multiprocessors, Ph.D. Thesis, University of Pennsylvania, 1991.

[15] U. Ramachandran, M. Ahamad, and M. Khalida, "Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory," *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 160–169, August 1989.

[16] A. J. Smith, "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, Vol. C–36, No. 9, September 1987, pp. 1063 – 1075.

[17] J. P. Singh, W.–D. Weber, and A. Gupta, SPLASH: Stanford Parallel Applications for Shared–Memory. Technical Report CSL–TR–91–469,Stanford University Computer Systems Lab, April 1991.

[18] M. Tam, J. M. Smith, and D. J. Farber, "A Taxonomy–Based Comparison of Several Distributed Shared Memory Systems," *ACM Operating System Review*, Vol. 24, No. 3, July 1990, pp. 40–67.

[19] J. Torrellas, M. S. Lam, and J. L. Hennessy. "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates," *Proceedings of the 1990 International Conference on Parallel Processing, II – Software*, August 1990, pp. 266–270.

Volume I

SYSTEMS INTEGRATION '94

PROCEEDINGS of the Third International Conference on Systems Integration

SÃO PAULO, SP, BRAZIL, AUGUST 15–19, 1994

EDITED BY :

PETER A. NG
New Jersey Institute of Technology

FUAD GATTAZ SOBRINHO
Instituto Internacional de Integração de Sistemas

C.V. RAMAMOORTHY
University of California, Berkeley

RAYMOND T. YEH
International Software Systems, Inc.

LAURENCE C. SEIFERT
Global Manufacturing and Engineering, AT&T

Sponsored by IISS

DICE - a Distributed Integrated Computing Environment for Multi-Threaded Parallel Processing*

Hasan S. AlKhatib, Qiang Li, Chi-Jiunn Jou, Tiekun Chen, and Hassan Arafeh
Department of Computer Engineering , Santa Clara University
Santa Clara, CA 95053

Abstract – *Often, the computing power of networks of workstations is left unused. The objective of this project is to develop a set of tools to take advantage of this potential computing power and to create a platform suitable for large scientific computations. This paper presents the architecture of a Distributed Integrated Computing Environment (DICE) consisting of a cluster of networked workstations. DICE consists of three interactive subsystems. DSM (distributed shared memory) provides the underlying communication and computing paradigm for threads of a parallel task to execute on a cluster of cooperating workstations. DRS (distributed run-time subsystem) provides users with programming tools to develop and execute DICE multi-threaded applications. PS (parallel scheduler) is a self optimizing application specific scheduler, and is responsible for thread scheduling and synchronization.*

1. Introduction

The majority of scientific applications require a fairly large amount of memory to execute a task. If a task is to be partitioned into threads (sub-tasks) that are executed in parallel, memory sharing is very desirable, since it allows sharing variables among threads within the same task. Also, software based on shared memory is more portable and machine independent as compared to that of distributed memory which is architecture dependent. The shared memory multiprocessor system has been more and more popular for executing large scientific applications for these reasons.

On the other hand, there is a tremendous amount of computing power that is left unused in networks of workstations. Very often a workstation is simply sitting idle on a desk. A set of tools can be developed to take advantage of this potential computing power to create a platform suitable for large scientific computations. The integration of several workstations into a logical cluster of distributed, cooperative, computing station presents an alternative solution to shared memory multiprocessor systems.

DICE (Distributed Integrated Computing Environment) is designed to meet these objectives. DICE employs virtual memory supported distributed shared memory(*DSM*) as its underlying computing and communication paradigm. It integrates *DSM* with a parallel scheduling as well as a parallel programming subsystem. In Figure 1, a distributed task '1' is running on four workstations, while a distributed task '2' is running on three workstations. These distributed tasks are independent of each other, and a workstation may have threads of two or more tasks running on it, concurrently.

This paper presents the *DICE* architecture in the following sections. Section 2 identifies the related work in this area. Section 3 describes the system architecture of *DICE*. It consists of three subsystems, which are described in sections 4 to 6, respectively. The interaction among these subsystems is delineated in section 7. The expected system performance is shown in section 8. Finally, section 9 gives a summary of the results accomplished with this work.

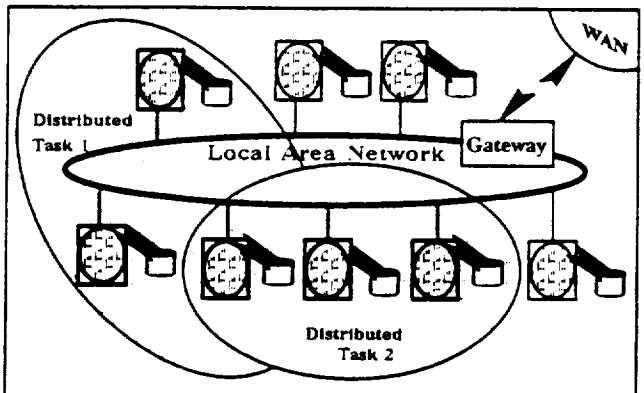


Figure 1. Clusters of Cooperating Workstations

*This work was supported by NASA-Ames Research Center grants number NCC 2-644 entitled "Parallel Processing for Scientific Computations".

0-8186-5502-X/94 $04.00 © 1994 IEEE

2. Related Work

There are several systems designed to utilize the processor power of idle workstations. These systems include Sprite [24], V system [33], NEST [1], Butler [23], Condor [20], REM [30], Stealth [17], and Sidle [16]. These systems provide remote execution or process migration facilities. In addition to these features, DICE provides the distributed shared memory (DSM) paradigm while using these idle workstations.

A number of DSM systems over LANs have been developed recently [31]. Among them, Ivy [18,19] is implemented on a network of Apollo workstations. The memory is paged, and copies of pages may be replicated in different hosts. Strict coherency semantics are used, and the memory coherency is maintained by a write-invalidate with dynamic ownership protocol. The owner of a page is located via either a centralized manager, fixed distributed managers, or an individual host which forwards the request. Ivy is used for applications employing a multi-threaded task. All threads share the same virtual address space. False sharing may occur in this system, since its consistency or access unit (eg. word) is less than the sharing unit (page). In addition, the single-write nature of its protocol may cause a "ping-pong" behavior between multiple writers of a shared page, or the thrashing problem.

To overcome false-sharing and thrashing, some systems employ special schemes. Mach [14] supports the DSM with a shared memory server. False-sharing and thrashing are handled by fault scheduling via a queuing mechanism [13]. Clouds [27,2] is an object-oriented distributed operating system where objects can migrate across processors. False sharing and thrashing are avoided, since Clouds uses a single-writer-single-reader strict coherence semantics.

Mirage [12] is a DSM system implemented in the kernel of the Locus distributed system [34]. Thrashing is reduced by using a time window scheme, in which the system guarantees that the writer of a page retains access to a page for a fixed period of time. Munin [6,7] is a DSM system implemented on the top of the V kernel [9], which allows programmers to associate types with shared data. Hence, multiple consistency protocols can be used. A delay write update scheme is used for a read-mostly protocol. Hence, thrashing can be reduced by using different combinations of data types.

Mether [21,22] is a software DSM implemented on SunOS 4.0. It allows a process to access memory as either consistent or inconsistent, and only a subset of a page to be transferred. It also provides both demand-driven and data-driven semantics for updating pages. All of these operations are encoded in a few address bits in the virtual address. False sharing and thrashing is reduced through the use of the incoherent memory.

DICE presents a novel approach to handle the problem of false sharing and thrashing. The shared memory is structured as a two-tier paging system. The first tier is a *page*, which is the common page used in an operating system. The second tier is a *paragraph*, which is a smaller fixed-sized block of information contained within a page. The introduction of small paragraph size improves system performance, since it reduces the chance of false sharing and the amount of data needed to be transferred over the network.

Distributed run-time system, *DRS* is another part of *DICE*. A survey of object-oriented languages for parallel environments is presented in [36]. Other programming languages and systems developed for distributed systems are presented in [4]. Amber [8] and Orca [5,32] are two such systems.

Parallel scheduler, *PS* is the third part of *DICE*. Several approaches are taken by researchers at work on the problem of parallel scheduling. They range from centralized control where global knowledge of the system is maintained in one place [25,26], to distributed control where all nodes have equal knowledge of the system. Methods used vary from Baysian decision theory [28] to data flow graphs [10].

The parallel scheduler in *DICE* is an extension to our prior work done in MOPPS [3]. MOPPS is a self-tuning parallel scheduler. It partitions the given application into small tasks, schedules and coordinates these tasks among network resources, and maintains a balanced load between workstations without overburdening the communication network.

3. System Structure

DICE is an experimental system which aims at providing a computing environment for the execution of multi-threaded tasks. Figure 2 illustrates the system

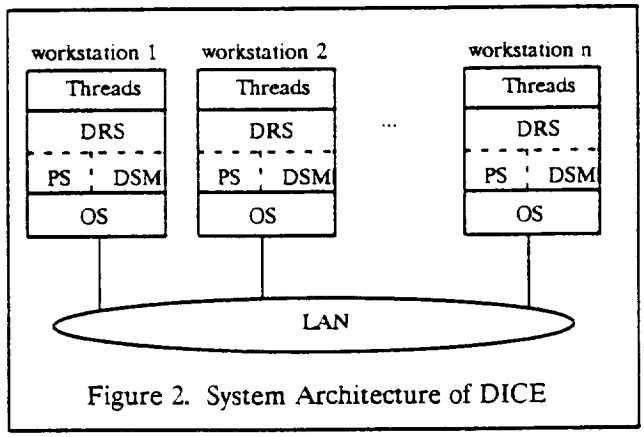


Figure 2. System Architecture of DICE

structure of DICE. A parallel task may consist of multiple threads that can be scheduled to run on a cluster of workstations, simultaneously. A *thread* is an active entity that provides the notion of a computation. Threads on separate workstations also share the same virtual address space, and communicate with each other using shared memory. Synchronization of threads to access shared resources is done using functions provided by the distributed run-time library.

4. Distributed Shared Memory

DICE DSM system consists of a cluster of workstations connected by a high-speed and low-latency local area network. Other than a host processor and memory, each node also has a network processor and a *Distributed Shared Memory Management Unit (DSMMU)*. *DSMMU* is an extension of the traditional MMU to allow *DSM* to handle shared memory efficiently. When data is not available locally and needs to be fetched from a remote host, *DSMMU* will trigger special access faults. Otherwise, *DSMMU* just performs the traditional TLB operations. An example of the architecture of a single host system is shown in Figure 3. Note that this example uses dual-ported memory, which allows both host processor and network processor to access the data structures for managing shared memory.

Each page of *DICE DSM* is the same as the common page in a typical operating system, such as the SunOS. Each page is further divided into several small equal-sized *paragraphs*. Paragraphs are used as the unit for coherency. Pages are used as the unit for sharing. Memory is allocated in a segment which may contain one or more pages. Figure 4 illustrates the hybrid nature of this memory structure.

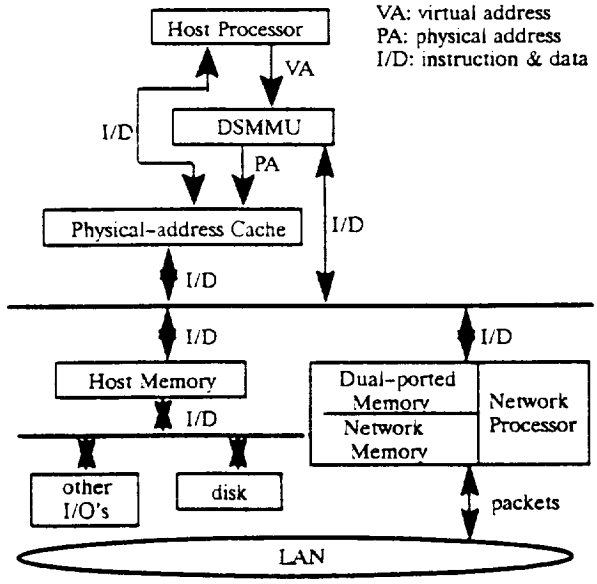


Figure 3. The system structure of a host system.

Coherency Protocol

DICE mainly provides the computing environment for the execution of multi-threaded tasks. A parallel task consists of multiple threads that are scheduled to run on a cluster of workstations, simultaneously. The shared data of memory pages are also distributed and replicated among these hosts. The DSM system supports the sharing of those pages, and maintains the coherency among replicated data copies. Each running application has a *root host*, on which it was loaded and executed. The root host maintains the state information for all shared pages used in the application, while other hosts maintain the state information for only the shared pages that are used in their local systems.

DICE is a home-based virtual DSM system, in which each shared page has a *home host*. A home host maintains the state information for its pages, ensures that the last copy of a page is not purged, and keeps track of all copies of the paragraphs within its pages. Other hosts only keep the information about the locations of the home host. A remote request for handling memory access faults is always sent to the home host of the target page. When a host does not know the home host for a certain page and tries to access it, a *home-info* fault will be triggered and a *home-info* request will be sent to the

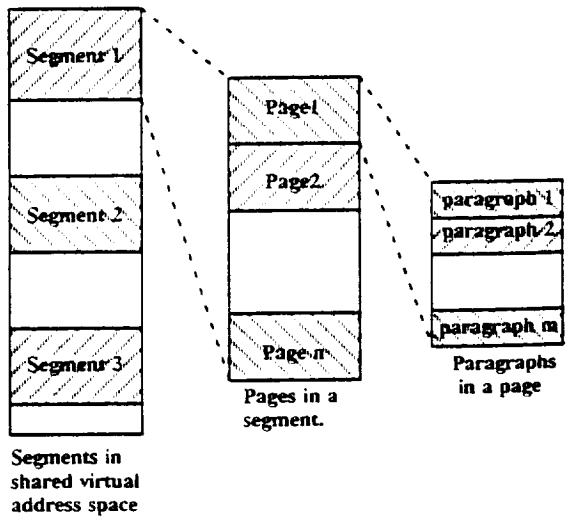


Figure 4. Segments, pages, paragraphs in *DICE DSM* structure

root host. If the home host is not yet assigned, the root host will assign the first requesting host to be home host of that page, update its own database, and send back a reply confirming this assignment. Otherwise, the root host simply sends back a reply giving the information of the home host for that page.

The memory coherency of DICE DSM is maintained on paragraph level. Each paragraph has an *owner host*, which has the ownership of this paragraph. An owner host always has an up-to-date copy of its paragraph, and is the only host which permits to write to the paragraph. The ownership of a paragraph may be transferred from one host to another according to the coherency protocol. Information about the current owner of a paragraph is maintained at the home host of the page containing the paragraph.

A paragraph can simultaneously be read by multiple hosts, but it can only be written by its current owner host. The access right of a paragraph for a particular host may be either *read-write*, *read-only*, or *none*. A host can acquire or upgrade its access rights by sending requests to the home host of the page in which the desired paragraph resides.

A host can immediately perform read and write operations on a paragraph if it has *read-write access* for that paragraph, or perform read operations on a paragraph if it has *read-only access* for that paragraph. When a read operation is issued to a paragraph with *none* rights, a *read-data* fault will be triggered and a *read-data* request will be sent to its home host. When a write operation is issued to a paragraph with *none* rights, a *write-data* fault will be triggered and a *write-data* request will be sent to its home host. When a write operation is issued to a paragraph with *read-only access*, a *write-access* fault will be triggered and a *write-access* request will be sent to its home host.

When a page is initialized, the home host is the default owner host for all paragraphs within this page. Any other host will send a remote request to the home host when it tries to access any paragraph of this page. If a *read-data* request is received, the home host will return back a reply containing the most recent copy of the desired paragraph when itself is the owner host of that paragraph. The access rights of both home and requesting hosts are changed to *read-only*. If itself is not the owner host, the home host will forward this *read-data* request to the owner host of that paragraph. The latter changes its access right to *read-only*, and then directly sends to the requesting host a reply which contains the most recent copy of that paragraph. After it receives the reply, the requesting host changes its access right to *read-only* and sends to the owner host an acknowledgement with the received reply. Having received this acknowledgement with reply, the home host also changes its access right to *read-only* and becomes the owner host of that paragraph. If itself is the requesting host, the home host will directly send the *read-data* request to the owner host. The latter changes its access right to *read-only*, and then sends back a reply which contains the most recent copy of that paragraph. Having received this reply, the home host changes its access right to *read-only* and becomes the owner host of that paragraph.

If a *write-data* request is received, the home host will return back a reply containing the most recent copy of the desired paragraph when itself is the owner host and no other host has a valid copy of that paragraph. If multiple valid copies exist, the home host will send *invalidate* requests to all hosts on which those copies are located, and wait for confirmations from all of them before returning back the reply. Upon receiving the *invalidate* request, each host changes its access right of that paragraph to *none* and returns its confirmation to the home host. The access right of the home host is then changed to *none*, while the requesting host becomes the owner host and its access right is changed to *read-write*. If itself is not the owner host, the home host will forward this *write-data* request to the owner host of that paragraph. The latter changes its access right to *none*, and then directly sends to the requesting host a reply which contains the most recent copy of that paragraph. After it receives the reply, the requesting host changes its access right to *read-write* and sends an acknowledgement to the owner host. Having received this acknowledgement, the home host updates its database and indicates that the requesting host becomes the owner host of that paragraph. If itself is the requesting host, the home host will directly send the *write-data* request to the owner host. The latter changes its access right to *none*, and then sends back a reply which contains the most recent copy of that paragraph. Having received this reply, the home host changes its access right to *read-write* and becomes the owner host of that paragraph.

If a *write-access* request is received, the home host will return back the *write-access* confirmation when no other host has a valid copy of that paragraph. If three or more valid copies exist, the home host will send *invalidate* requests to all hosts (except itself and the requesting host) on which those copies are located and wait for confirmations from all of them before returning back the *write-access* confirmation. Upon receiving the *invalidate* request, each host changes its access right of that paragraph to *none* and returns its confirmation to the home host. The access right of the home host is then changed to *none*, while the requesting host becomes the owner host and its access right is changed to *read-write*. If itself is the requesting host, the home host will directly send the *invalidate* requests to all hosts (except itself and the requesting host) which have valid copies of that paragraph and wait

for confirmations from all of them. Upon receiving the *invalidate* request, each host changes its access right to *none* and returns its confirmation to the home host. The home host then changes its access right to *read-write* and becomes the owner host of that paragraph.

5. Distributed Run-time Subsystem

DICE DRS transforms the *DICE DSM* from a flat space into an object-oriented structured space. *DRS* consists of a set of tools that implement *DICE*. Application Programmer's Interface, API, provides users with programming tools to develop and execute *DICE* multi-threaded applications. The tools used during program development include a parallel language and its compiler, library interface functions, a linker, and other system services.

A new *Object-Oriented Dataflow Language* (*OODL*) is being designed as the parallel language used in *DICE*. One of the important features of object-oriented programming is information hiding and encapsulation [11,29]. It provides a higher level of data abstraction in modeling real world objects. Such constructs are helpful in designing parallel programs [35]. In general, parallel programs are difficult to design because the programmer must consider multiple execution threads instead of a single thread. All possible interactions among the threads must be considered. Also, parallel programs are hard to maintain because a simple change may affect the interaction pattern and results in global consequences. Information hiding helps in reducing possible interactions that need to be considered, while data encapsulation helps in minimizing the maintenance effort when program changes are needed.

While the object-oriented model provides a high level of programming abstraction, it does not naturally exploit parallelism of applications constructed with objects. A dataflow model can expose and exploit the maximum amount of parallelism, as well as express data dependence from different levels of abstraction in a very natural way. The combination of the object oriented and dataflow concepts makes it easier for programmers to design large scale multi-threaded parallel programs, and to build re-usable concurrent software modules.

The *OODL* language, in *DICE*, is an extension of C++. Dataflow constructs are added to allow programmers to express parallelism explicitly. The parallel compiler can be realized using a preprocessor to translate the extended source code into C++ programs, which in turn are compiled into object code using an existing C++ compiler.

The run-time library interface functions provide a collection of library routines that are linked with each parallel program. They are invoked to support the service requests made by system processes at run-time. The *OODL* compiler will use these functions to realize the parallelism expressed in the application programs. These functions can also be used by the application directly.

The linker will create a standard execution file such as a.out and an execution dependency tree called a.tree. The information kept in the dependency tree includes the names of the parallel threads; information about the resources of the threads, such as starting address and memory requirements; and the predecessors and successors of each thread. This information will be used by the parallel scheduler to create and allocate shared memory segments, and to schedule threads on different workstations at run-time. The linker will arrange shared variables into shared segments, to simplify the management of shared memory by the *DSM* subsystem.

DRS also provide services for executing applications at load-time and run-time. These services include the use of the *DICE* daemon(s), as well as the automatic creation of a root process and alias remote processes for a parallel task.

For each workstation that participates in *DICE*, a daemon process has to be present. This daemon is responsible for invoking *DICE* alias processes on remote workstations. Each *DICE* application creates a root process when it starts. The workstation where the root process is running is referred to as the root workstation. A *DICE* application may have zero or more alias processes. An alias process is created by the root process on a remote workstation through a *DICE* daemon as needed.

The root process is a multi-threaded process which runs on the root workstation. It is created when the parallel task is submitted to the system. In *DICE*, the thread is the unit of execution, while a process is the unit of resource allocation. Each process contains one or more threads. The root process provides the virtual address and system resources for threads running on the root workstation. The root thread is the first thread of a parallel task. It is responsible for creating the parallel scheduler and *DSM* manager threads before any application threads start to run. It then becomes the first application thread running on the root workstation. The root process terminates when the parallel task is done.

An alias process is a reincarnation of the root process on each remote workstation. An alias process is created when a thread is scheduled to run on a remote workstation for the first time. The alias process supports the same virtual address as the root process and system resources for threads running on its workstation. These threads include an alias primary thread, *DSM* manager, and application threads. An alias primary thread is re-

sponsible for creating its local *DSM* manager as well as the first application thread running on its local workstation. This alias primary thread, then, listens to thread-create requests coming from the network. Subsequently, it creates these requested threads of its own parallel task on its local workstation. The alias primary thread and *DSM* manager of a remote workstation will remain when all of its application threads are terminated. The alias primary thread waits for thread-creation requests from the parallel scheduler, while *DSM* manager waits for memory access requests from other workstations. When the root process is done, the parallel scheduler sends out a termination signal to all the alias processes of that particular task. This is to ensure that all alias processes are terminated before the termination of the root process.

The *DICE* daemon process is a server that is responsible for invoking alias processes on a remote workstation. After invoking an alias process, the daemon process will have nothing to do with this application task. It will go back to listening to requests from the network. If a workstation does not want to participate in *DICE*, it can simply terminate this daemon process. A *DSM* manager is an active entity on each workstation responsible for handling memory access faults. Each *DSM* manager maintains a memory mapping table that maps each memory page to its local workstation or other remote workstations.

6. Parallel Scheduler

DICE PS is a self-optimizing application-specific scheduler. It is responsible for thread scheduling and synchronization. *PS* is implemented as a thread within the parallel task. Each parallel task has one *PS* running on the workstation where the task initially starts to run. This special thread is created during the task load-time.

When an application needs to create another thread or to terminate itself by joining with other threads, it passes control of execution to the *PS*. The *PS* will find the fastest way to run the application by using the information in a *Task Execution Dependence Tree*, which is created as an auxiliary file during the compilation of the source program.

The *PS* decides whether the local workstation has enough resources to run the different threads, which threads to send to remote workstations to run, and which remote workstations to send them to. It uses several tools to make intelligent decisions at run time. Those tools are: a *CPU load estimator*, a *network load estimator*, an *intelligent database*, and a *bidding process*.

The *CPU load estimator* runs on every workstation on the network and keeps track of the load on that workstation. When the time comes to run a thread on the local CPU, *PS* looks at the *CPU load estimator* for information about the load on the local CPU. Similarly, when a bid arrives at a workstation, the decision whether to accept the bid or not depends partially on the readings taken by the *CPU load estimator*.

The *network load estimator* monitors the traffic on the network. The *network load estimator* gives *PS* an up-to-date reading of the network traffic. Smaller partitions that takes a relatively short time to execute can become too expensive to ship if transmission times become too sever. In that case, it might be better to keep them on the local workstation, defer shipping them, or combine two or more into larger partitions.

The *network load estimator* has the responsibility to provide *PS* with real time network traffic information. The *network load estimator* can be as simple as a bus monitor which continually updates a register (interpreted as an integer) signify network utilization levels of high, medium, or low.

A small and efficient database records thread performance on each workstations under different CPU and network load conditions. This database allows the *bidding process* to generate a reasonable estimate of the expected run time of a thread on a particular target workstation.

The *intelligent database* is designed to categorize different higher level operations of modules and parameterize their computational and communication time requirements. The contents of *intelligent database* are tailored to the installation where it resides. The database is initiated with the types of applications being run, and its contents are updated as new applications are introduced.

When *PS* decides that it is best to send some threads to a remote workstation to run, it needs a way to pick those workstations. Instead of forcing other, possibly heavily loaded, workstations to take some of the threads, *PS* asks for help through the *bidding process*. It simply asks for help in running a given thread and tells the other workstations about the memory and CPU requirements of the thread. This information is found in the *intelligent database*.

Upon each task completion, the *intelligent database* is updated to reflect the most current experience. When no data is available about an application, we can run it the first time with gross overestimates, or underestimates, and let *intelligent database* learn about it. Simulation may also be used to obtain initial estimates.

It is essential that *intelligent database* be queried and updated quickly as it would be a system bottleneck and might slow down the entire system if not properly designed. Ultimately *intelligent database* can be implemented in hardware as a content addressable memory.

In the bidding algorithm, *PS* weighs execution time versus shipping and management time for each resident module. If execution time is greater than shipping and management time and the loads on the local workstation is higher than a predefined threshold, the parallel scheduler broadcasts a global message through the network asking for help. This "help wanted" message includes enough information about the module to be sent enabling other workstations to determine if they can offer their help. The information includes the estimated module execution time, memory and disk requirements, and any other information that is useful in making the decision.

Those workstations which can potentially bid to accept the module for processing will examine this workload information and determine whether it is feasible to bid. If a workstation is capable of assisting, it will return a message stating its availability, and will commit to this bid for a period long enough for the asking workstation to receive the return message and act on it. Through this process, workstations that bid for help and are not accepted will waste little time before considering later "help wanted" messages.

Each workstation will monitor the network before sending its reply to determine if any other workstations have responded to the bid and will not send it reply if any workstation did respond. It is assumed that the first workstation to reply will get the job, and there is no need for others to do so. *PS* sends the module to the first workstation that replies to the request.

PS repeats the help wanted messages for a given task until either it receives a response or the task is at the point where it has to be executed in order not to delay the rest of the tasks.

As a thread is scheduled on a remote workstations to run, its respective virtual address space segments are allocated physical memory blocks on the same workstations. *PS* takes the consideration of available memory resource on a workstation when scheduling a thread over there.

7. Interactions and Integration

DSM, *DRS*, and *PS* are three separate subsystems of *DICE*. They interact with each other to provide an integrated environment and to cooperatively work to provide the distributed computing paradigm for a parallel task.

After a parallel task is compiled and linked, a task execution tree file a.tree is created. *PS* uses this tree to perform the parallel thread scheduling. When a thread is to be created, the root process will transfer execution control to *PS*. The latter will use a.tree file to schedule it on a local or remote workstation, and then transfer execution control back to the application. Similarly, the execution control will be transferred to *PS* when a thread terminates itself by joining other threads. Figure 5 shows the overall interaction between *DRS* and *PS*. The parallel compiler and linker create the image of virtual memory segments and the task execution tree. *PS* is invoked when a thread needs to fork or join with other threads.

Furthermore, the root thread of *DRS* is responsible for creating *PS*. The alias thread on each remote workstation listens to remote thread creation requests sent by *PS*, and creates threads locally.

Similarly, the active entity of the *DSM* subsystem *DSM* thread is created by the root thread or alias threads on different workstations. In the meantime, the data structures needed by *DSM* thread are also created and initialized.

The efficiency of handling shared memory by *DSM* subsystem is significantly affected by the layout of shared variables on *DSM* memory segments and the allocation of physical memory on different workstations by the parallel programming subsystem and parallel scheduler. Figure 6 shows an example of the run time behavior of the *DSM* subsystem.

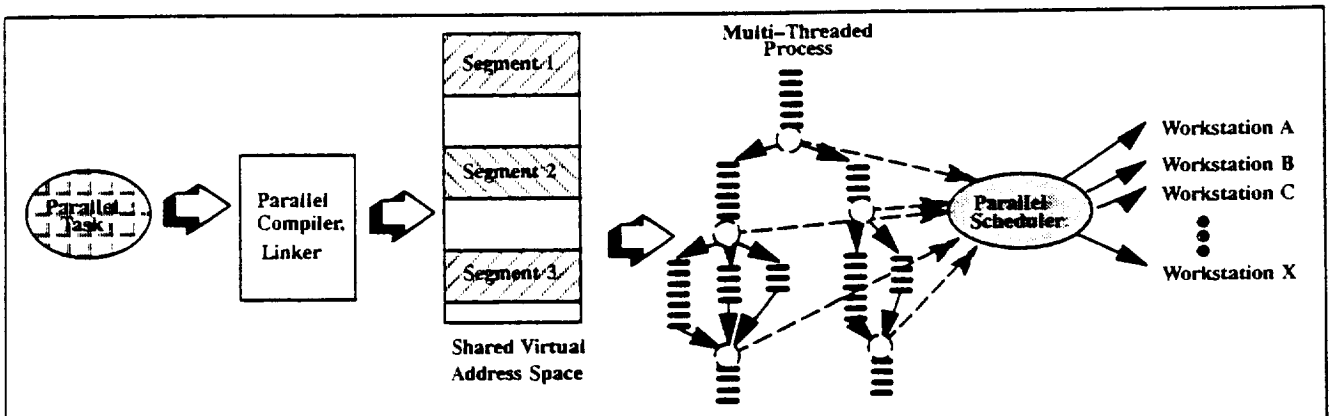


Figure 5. Scheduling of a Parallel Task on a Cluster of Workstations

In Figure 6, the virtual address space of the parallel task is on the left side. Each shadowed paragraph within the virtual address space represents a single virtual memory segment. The physical address spaces on different hosts are on the right side. The shadowed paragraph within a host denotes a block of a physical memory, and the other structure represents the segment map table. The paragraphs with arrowheads represent the corresponding mappings between the memory segments and the physical memory blocks on different hosts.

8. Performance and Discussion

The performance of *DICE DSM* system has been studied using an analytical model, which derives an expression for *the speedup of the parallel part of an application* (or S_p). The effects of changing S_p on system structure and application behavior is shown and discussed in [15]. Some of these results are shown in this section. The system and application parameters used in this model are summarized in Table 1 in Appendix.

High-speed and low-latency ATM LAN is assumed in this model. We also assume that queuing time on the network is negligible. This assumption is justified by the results shown in Figure 7 (Appendix), which indicates that the gain in S_p becomes smaller and smaller as the network data rate R_n is increased. Figure 8 (Appendix) shows that S_p decreases as processor speed R_p increases. Note that the total execution time for an application will still be reduced as R_p increases, although S_p decreases.

Figure 9 (Appendix) shows that S_p increases as the number of paragraphs per page, k, increases up to a certain point. After that point, S_p slightly decreases as k further increases. Furthermore, S_p is approximately the same for a fixed paragraph size, which is P/k . This behavior demonstrates usefulness of the use of a paragraph with a smaller granularity than a page. Figure 10 (Appendix) shows a similar behavior, for S_p in relationship with the number of hosts N.

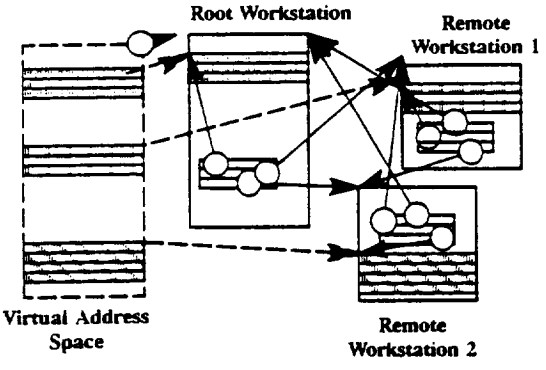


Figure 6. Distributed Shared Virtual Memory

9. Conclusions

In this paper, we presents the architecture of a distributed computing environment *DICE*, which integrates distributed shared memory with parallel scheduling and distributed run-time management. The analysis of performance model demonstrates the usefulness of the use of a paragraph with a smaller granularity than a page in DICE system. This smaller granularity reduces the chance of false sharing and the data size needed to be transferred over the network.

The coherency protocol for this two-tier paging system is also being simulated in software. The performance of DICE DSM is also being evaluated using a simulation model, which takes into consideration network queuing delay. The Object-Oriented Dataflow Language and self-tuning Parallel Scheduler are under development.

The current DICE DSM design is based on the strict consistency model and write-invalidate coherency protocol. This design is intended to be extended by using multiple consistency and coherency protocols. Multiple protocols will be used to tailor broader application requirements. DICE will incorporate the DSM design with a relaxed consistency model to hide the large latency of remote memory accesses by allowing buffering and merging.

References

[1] R. Agrawal, and A. K. Ezzat, "Location Independent Remote Execution in NEST," *IEEE Transactions on Software Engineering*, Vol 13, No 8, 1987, pp. 905-912.

[2] R. Ananthanarayanan, S. Menon, A. Mohindra, and U. Ramachandran, "Experiences in Integrating Distributed Shared Memory with Virtual Memory Management," *ACM Operating System Review*, Vol. 26, No. 3, July 1992, pp. 4-26.

[3] H. Arafeh and H. S. AlKhatib, and H. Barraclough, "MOPPS: A Scheme for Managing Parallel Scientific Programs in a Distributed Architecture," *Proceedings of COMPCON'90*, the Annual International Computer Conference of the IEEE Computer Society, February 25 - March 2, 1990, San Francisco, CA, pp 387-395.

[4] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Sys-

tems." *ACM Computing Surveys*, September 1989, pp. 261-322.

[5] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language For Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, Vol 18, No. 3, March 1992, pp. 190-205.

[6] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Adaptive Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 168-175.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," *The 13th ACM Symposium on Operating Systems Principles*, October 1990, pp. 152-164.

[8] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989, pp. 147-158.

[9] D. R. Cheriton, "The V Distributed System," *Communication of the ACM*, Vol 31, No. 3, pp. 314-333, 1988.

[10] W. W. Chu, and L. M-T. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Transactions on Computers*, Vol C-36, No. 6, June 1987, pp. 667-679.

[11] B. Cox, Object Oriented Programming - An Evolutionary Approach, Addison-Wesley, 1986.

[12] B. D. Fleisch, G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989, pp. 211-222.

[13] A. Forin, J. Barrera, and R. Sanzi, Design, Implementation, and Performance Evaluation of A Distributed Shared Memory Server for Mach, Technical Report CMU-CS-88-165, Carneigie-Mellon University, Computer Science Department, August, 1988.

[14] A. Forin, J. Barrera, and R. Sanzi, "The Shared Memory Server," *Proceedings 1989 Winter USENIX Technical Conference*, February, 1989, pp. 229-244.

[15] C Jou, H. S. AlKhatib, and Q, Li, Performance Analysis of DICE Distributed Shared Memory System, Distributed Computing Lab Technical Report No. 03281994, Department of Computer Eng.ineering, Santa Clara University, 1994.

[16] J. Ju, G. Xu, and J. Tao, "Parallel Computing Using Idle Workstations," *Operating System Review*, July 1993, pp. 87-96.

[17] P. Krueger, R. Chawla, "The Stealth Distributed Scheduler," *Proc. 11th ICDCS* 1991.

[18] K. Li, Shared Virtual Memory on Loosely Coupled Multiprocessors, Ph.D. Thesis, Yale, September, 1986.

[19] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," In *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 94-101, August 1988.

[20] M. T. Litzkow, "Condor- A Hunter of Idle Workstations," *Proc. 8th ICDCS* 1988, pp. 104-111.

[21] R. G. Minnich and D. J. Farber, "The Mether System: A Distributed Shared Memory for SunOS 4.0," In *Useunix -Summer* 89, Usenix, 1989.

[22] R. G. Minnich and D. J. Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

[23] D. A. Nichols, "Using Idle Workstations in a Shared Computing Environment," *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, December 1987, pp. 5-12.

[24] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System", IEEE Computer, February 1988, pp. 23-36.

[25] J. Pasquale, Knowledge-Based Distributed Systems Managements, Report No. UCB/CSD 86/295, UC Berkeley, Computer Science Division, June 1986.

[26] J. Pasquale, Using Expert Systems to Manage Distributed Computer Systems, Report No. UCB/CSD 87/334, UC Berkeley, Computer Science Division, January 1987.

[27] U. Ramachandran, M. Ahamad, and M. Khalida, "Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory," *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 160-169, August 1989.

[28] J. A. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling", *IEEE Transactions on Computers*, Vol C-34, No. 2, February 1985.

[29] B. Stroustrup, "What is "Object Oriented Programming"?," *IEEE Software*, Vol 5, No. 3, May 1988, pp. 10-20.

[30] G. C. Shoja, "A Distributed Facility for Load Sharing and Parallel Processing Among Workstations," *Journal of System and Software*, Vol 14, No. 3, pp. 163-172.

[31] M. Tam, J. M. Smith, and D. J. Farber, "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems," *ACM Operating System Review*, Vol. 24, No. 3, July 1990, pp. 40-67.

[32] A. S. Tanenbaum, M. F. Kaashoek, andH. E. Bal, , "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, Vol 18, No. 3, August 1992, pp. 10-19.

[33] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for V-System," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, December 1985, pp. 2-12.

[34] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the 9th Symposium on Operating System Principles 17*, 5 (November 1983), pp. 49-70.

[35] Y. Wu, T. G. Lewis, "Parallelism Encapsulation in C++," In *Proceedings of the 1990 International Conference on Parallel Processing*, pp. 35-42, 1990.

[36] B. B. Wyatt, K. Kavi, and S. Hufnagel, "Parallelism in Object-Oriented Languages: A Survey," *IEEE Software*, November 1992, pp. 56-86.

Appendix

parameters	meanings
N	the number of hosts executing an application
Rn	network data rate
Rp	processor speed
M	the total bytes of shared memory space for the running application
P	page size
k	the number of paragraphs per page
d	the percentage of data memory accesses for total instructions
Nrf	the number of read faults per 1,000,000 memory referenced per host
Nwf	the number of write faults per 1,000,000 memory referenced per host
Xs	spatial locality factor
No, N1, g	temporal locality factors

Table1. System and application parameters in the performance model.

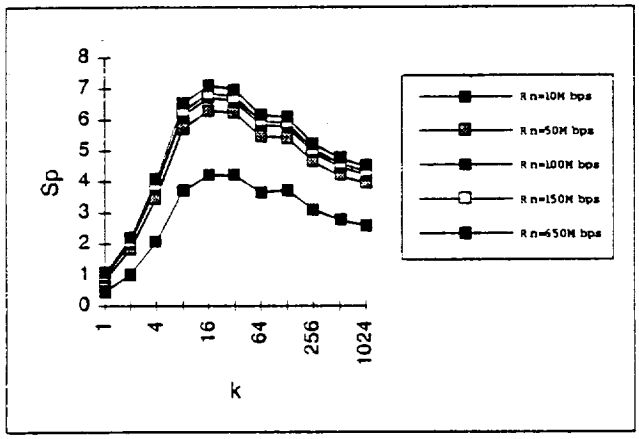


Figure 7. Sp vs k for different Rn. N=16, Rp=50Mips, M=64kbytes, P=4kbytes, d=0.4, Nrf=500, Nwf=10, Xs=0.5, N0=10, N1=100, g=100.

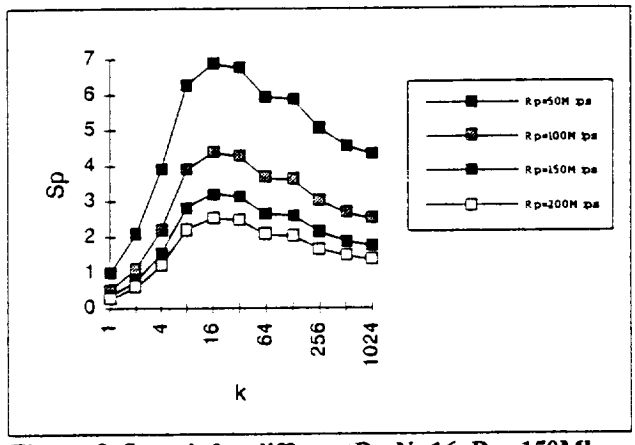


Figure 8. Sp vs k for differentRp.N=16, Rn=150Mbps, M=64kbytes, P=4kbytes, d=0.4, Nrf=500, Nwf=10, Xs=0.5, N0=10, N1=100, g=100.

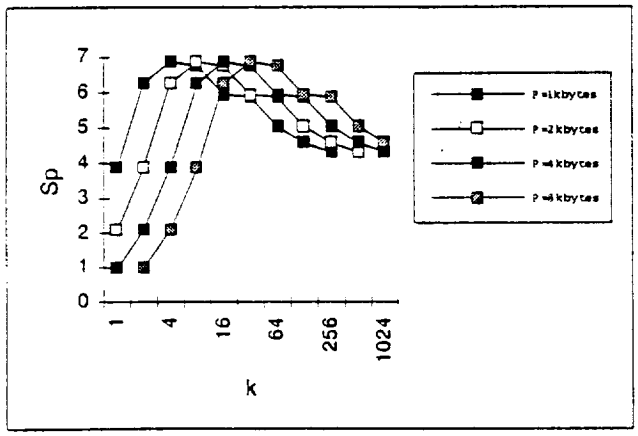


Figure 9. Sp vs k for different P. N=16, Rn=150Mbps, Rp=50Mips, M=64kbytes, d=0.4, Nrf=500, Nwf=10, Xs=0.5, N0=10, N1=100, g=100.

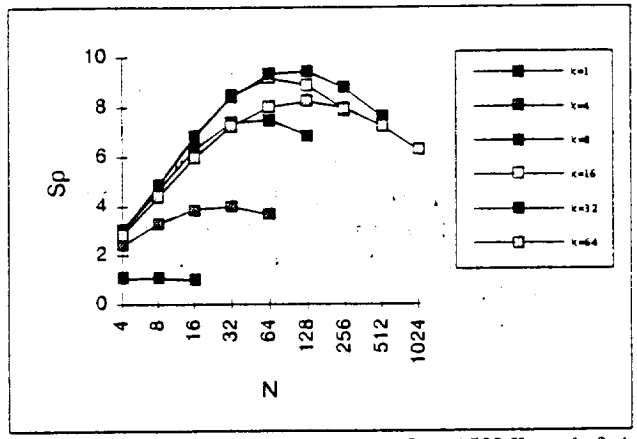


Figure 10. Sp vs N for different k.Rn=150Mbps,d=0.4, Rp=50Mips,M=64kbytes,P=4kbytes,Xs=0.5, Nrf=500, Nwf=10, N0=10, N1=100, g=100.