A HIERARCHICAL APPROACH TO LOCAL FILE SYSTEM DESIGN FOR A TRANSPUTER-BASED MULTIPROCESSOR

by

Timothy J. Dudra

BSc. (Honours), Simon Fraser University, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

Simon Fraser University

December 1992

ISBN 0-315-91038-0

APPROVAL

Name: Timothy Joseph Dudra

Degree: Master of Science

Title of thesis: A Hierarchical Approach to Local File System Design for a Transputer-based Multiprocessor

Examining Committee: Dr. Ramesh Krishnamurti
Chair

Dr. M. Stella Atkins, Senior Supervisor

[illegible]

Dr. Wo-Shun Luk, Supervisor

[illegible]

Dr. Alan Wagner, Examiner

Date Approved: Dec 1 1992

Title of Thesis/Project/Extended Essay

A Hierarchical Approach to Local File System Design for a Transputer-based Multiprocessor.

Author: [illegible]
(signature)

Timothy Joseph Dudra
(name)

December 7, 1992
(date)

Abstract

The rate at which processor power is increasing is not being matched by similar increases in mass storage performance, nor is this disparity expected to be rectified. This problem is especially acute in multiprocessor architectures and has come to be known as the I/O bottleneck crisis. Vast quantities of data are required for full utilization of a multiprocessor and this data must be provided to that processor, either directly or indirectly, via high speed networks, from some mass storage medium.

Disk caching and prefetching have been proposed as good (possibly short-term) techniques for reducing the impact of the I/O bottleneck on multiprocessor throughput. An impressive amount of research has been invested in analyzing the effects of these techniques on the performance of tightly coupled shared memory multiprocessors; Application of these methods to loosely coupled multiprocessors has not been pursued with the same fervour. This thesis presents the design and performance of a disk cache system for use within networks of Inmos transputers. The intent of this system is to provide a local file system for the transputer network. This system is intended to be a building block in the development of a distributed file system for the transputer multiprocessor.

The Transputer Auxiliary Storage System supports advanced cache management features such as user directed prefetching, opportunistic write back, disk interleaving, multi-threaded server support and RAM disks. The design and implementation issues encountered during the creation of this storage system are discussed. Also, the performance of the system is analyzed for various workloads. The results confirm the positive effect of the system upon sustained I/O performance into and out of the transputer network.

Acknowledgements

Firstly, I would like to thank all of those persons who persevered with me through the seemingly eternal ages of this work. My family was perpetually supportive. My supervisor was probably more patient than I deserved. The SFU computing science department did not threaten me with expulsion if I could not show some progress (actually, near the end they hinted a little).

Credit has to be given to those persons who joined me during many evenings of inspired drinking and pseudo-philosophical discussion. The emotional and psychological cleansing associated with these rituals cannot be underestimated.

Lastly, I extend special thanks to my father, whose constant nudging during my youth finally, against all odds, convinced me that an education has value.

Table Of Contents

Section	Title	Page
	Approval	ii
	Abstract	iii
	Acknowledgements	iv
	Table of Contents	v
	Index of Tables	x
	Index of Figures	xi
1	Introduction	1
1.1	Apologies	1
1.2	Hardware Overview	1
1.3	Research Goals	2
1.4	Thesis Overview	5
2	Related Research	7
2.1	The Memory Hierarchy	7
2.1.1	Disk Performance	11
2.1.1.1	Buffering to Reduce Disk Delays	14
2.1.1.2	Disk Scheduling to Reduce Seek Times	15
2.1.1.3	Track and Sector Interleaving to Reduce Rotational Latency	16
2.1.2	Joining of Multiple Disks	19
2.1.2.1	High Bandwidth Disk Arrays From Low Bandwidth Disks	20
2.2	Cache Design	24
2.2.1	Cache Components	25
2.3	Caching Strategies	27
2.3.1	Placement Policy	28
2.3.2	Replacement Algorithms	30
2.3.3	Combined and Partitioned Caches	32
2.3.4	Write Policy	34
2.3.5	Prefetching	35
2.4	Data Access	36
2.4.1	Data Operations	37
2.4.2	Data Objects	38
2.4.3	Data Access Patterns	38
2.4.3.1	Categories of Data Access Patterns	38
2.4.3.2	Relevant Patterns	39
2.4.3.2.1	Sequential Global Irregular	40

2.4.3.2.2	Sequential Local Disjoint Irregular	42
2.4.3.2.3	Random Overlapped	42
2.5	Other Systems	43
2.5.1	RAID	44
2.5.2	DataMesh	45
2.5.3	A Parallel Interleaved File System	46
3	Design Overview	51
3.1	Hardware Environment	51
3.1.1	Transputer Architecture	51
3.1.2	Vendor Boards	53
3.1.3	SFU Transputer Environment	55
3.1.4	Vendor Supplied Software	57
3.2	TASS Design	59
3.2.1	Hardware Configuration for TASS Storage Nodes	63
3.2.2	Software Design	65
3.2.2.4	Client Interface Layer	67
3.2.2.4	File System Layer	67
3.2.2.4	Cache Layer	70
3.2.2.4	Disk Support Layer	72
4	The Disk Server	74
4.1	Considerations in the Design	78
4.1.1	Device Dependent Interface	78
4.1.2	Device Independent Interface	80
4.1.3	Server Message Protocol	81
4.1.4	Miscellaneous Considerations	82
4.2	Implementation of the First Disk Server Prototype	84
4.3	Implementation of the Disk Server 2 Prototype	87
4.4	Implementation of the RAM Disk Server Prototypes	91
5	The Cache Server	92
5.1	Layer Requirements	92
5.1.1	Disk Support Layer	92
5.1.2	Cache Layer	95
5.1.2.1	Overview	95
5.1.2.2	Memory vs CPU Cycles Trade-off	96
5.1.3	File System Layer	98
5.1.4	Server Message Layer	100
5.2	Implementation of the Cache Server 1 Prototype	102
5.2.1	Requestor Process Group	105
5.2.2	Acceptor Process Group	106

5.2.3	File Server Process Group	107
5.2.4	The Cache Manager	107
5.3	Implementation of the Cache Server 2 Prototype	110
5.3.1	Disk Support Layer	112
5.3.2	Cache Layer	115
5.3.2.1	The CacheManager Module	115
5.3.2.1.1	Get Lock	116
5.3.2.1.2	Release Lock	117
5.3.2.1.3	Insert New Entry	117
5.3.2.1.4	Start Write Back	118
5.3.2.1.5	Finish Write Back	119
5.3.2.1.6	Stop Cache Flush	119
5.3.2.1.7	Stop Cache Flush	120
5.3.2.2	The Cache Module	120
5.3.3	File System Layer	121
5.3.3.1	File Server Process Group	121
5.3.3.2	Write Back Process Group	122
5.3.3.3	Predictor Process Group	124
5.4	RAM Cache Prototype	125
6	The Client Interface	126
7	Device and Disk Server Performance	130
7.1	Approximating Data Access Patterns	131
7.1.1	Sequential Global Irregular	132
7.1.2	Random Overlapped	133
7.2	Device Performance Limitations	134
7.2.1	Host Interconnect	134
7.2.2	Transputer Link Throughput	135
7.2.3	CP-3100 Fixed Disk Throughput	136
7.2.4	T800 Based RAM Disk Throughput	136
7.2.5	Context Switch Time	136
7.3	Disk Server	137
7.3.1	Testing Environment	137
7.3.2	Test Results	138
7.3.2.1	Sector Interleaving	138
7.3.2.2	Track Interleaving	139
7.3.2.3	Sequential Access	140
7.3.2.4	Random Access	142
7.3.2.5	Processor Usage	145
7.4	Conclusions	145

8	Cache Server Performance	147
8.1	Testing Environment	147
8.2	Disk Support Layer	150
8.2.1	I/O Bandwidth	150
8.2.1.1	Optimal Workloads	150
8.2.1.2	Comparing DSL Results to Disk Server Results	157
8.2.1.2.1	Single Disk Systems	158
8.2.1.2.2	Multiple Disk Systems	159
8.2.1.2.2.1	Sequential Access	159
8.2.1.2.2.2	Random Access	160
8.2.2	Processor Cycle Usage	162
8.2.2.1	DiskLink Module	162
8.2.2.2	DiskLayer Module	165
8.3	Cache Layer	166
8.3.1	I/O Bandwidth	167
8.3.1.1	Optimal Workload	167
8.3.1.2	Sequential Access	171
8.3.1.3	Random Access	174
8.3.2	Processor Cycle Usage	174
8.4	Server and File System Layers	175
8.4.1	Processor Cycle Usage	176
8.5	Conclusions	176
9	TASS Storage Node Performance	180
9.1	Testing Environment	180
9.2	I/O Bandwidth	182
9.2.1	Read Access	182
9.2.1.1	Only Cache Hits	183
9.2.1.2	Sequential Global Access with Prefetching	183
9.2.1.3	Random Access	186
9.2.1.4	Comments	190
9.2.2	Write Access	190
9.2.2.1	Only Cache Hits	191
9.2.2.2	Sequential Global Access	192
9.2.2.2.1	Single Disk Systems	192
9.2.2.2.2	Multiple Disk Systems	194
9.2.2.3	Random Access	195
9.2.2.4	Start-up Performance	197
9.3	Processor Cycle Usage	200
9.4	Conclusions	201
10	Summary	204

11	Future Work	208
11.1	New Hardware	208
11.2	Increased Cache Capacity	209
11.3	Expanding Disk Server Functionality	209
11.4	Additional Disk Joinings	210
11.5	Removing the Implicit Prioritization of ProcAltList()	211
11.6	Non-RPC Client Interface	211
11.7	More Functional Local File Systems	212
11.8	Distributed File Systems for the Transputer Multiprocessor	212
11.8.1	DFS using Point-to-Point Communication	213
11.8.2	DFS for SFU Style T800-based Network	214
Appendix A	Glossary	217
	Bibliography	221

Index of Tables

Table	Description	Page
1	Categories of Parallel File Access Patterns	39
2	CP-3100 Specifications	55
3	File System Operations	69
4	Cache Layer Interface	72
5	Disk Support Layer Core Primitives	73
6	Interface to the Device Dependent Layer	79
7	Disk Server to Cache Server Communication Protocol	83
8(*i*)	Cache Server to Client Communication Protocol	101
8(*ii*)	Cache Server to Client Communication Protocol (continued)	102
9	The Procedures of the Client Interface	128
10	Transputer to Transputer SDTR	135
11	Read Access SDTR for Various Chunk Interleavings	140
12	Write Access SDTR for Various Chunk Interleavings	140
13	Disk Server SDTR Samples and Averages	141
14	RAM Disk SDTR in KBytes/second	141
15	Random Access SDTR over Entire Disk Address Space	143
16	Disk Layer Throughput - Sequential Global Irregular DAP	152
17	Disk Layer Throughput - Sequential Local Disjoint Irregular DAP	153
18	Disk Layer Throughput - Random Overlapped DAP	154
19	Disk Support Layer Throughput in KB/s by Joining & Workload	157
20	Processor Cycle Usage for DiskLink Module	163
21	Processor Cycle Usage for the Disk Support Layer	166
22	Cache Layer Throughput - Sequential Global Irregular DAP	168
23	Cache Layer Throughput - Random Overlapped DAP	170
24	Cache Layer Throughput in KBytes/second by Joining and Workload	171
25	Processor Cycle Usage for the Cache Layer	175
26	Processor Cycle Usage for the File System and Server Layers	177
27	Read Access Sustained Data Transfer in KBytes/second	182
28	Write Access Sustained Data Transfer in KBytes/second	191
29	Processor Cycle Usage for the TASS Client Interface	200

Index of Figures

Figure	Description	Page
1	Objective of Network Disk Support	4
2	A Simple Disk Device	12
3	Sector Interleaving	17
4	Track Interleaving	18
5	Joining Techniques	22
6	Sequential Global Irregular DAP	41
7	Random Overlapped DAP	43
8	The Transputer Processor Family	52
9	The SFU Transputer Network	56
10	TASS Storage Node Configurations	63
11	Distribution of Primary Layers over Component Processors	66
12	Components of The Disk Support Layer	75
13	Disk Access Pipeline	76
14	Disk Server Prototype 1 Process Heirarchy	85
15	Disk Server Prototype 2 Process Hierarchy	88
16	Cache Server Prototype 1 Process Hierarchy	104
17	Cache Server Prototype 2 Process Hierarchy	111
18	Random Access for Varying Sizes of Disk Region	144
19	The Cache Server Test Environment	148
20	The TASS Storage Node Testing Environment	181
21	Random Read Access Performance While Varying File Size	188
22	Random Write Access Performance While Varying File Size	196
23	Start-up Performance Under a SGI write DAP	198
24	A Distributed File System Environment	215

1. Introduction

This section overviews the hardware and software environment within which this research occurred. The goals that have been pursued during this research are direct consequences of conditions found within this environment. My goals are presented as a subset of the more extensive goals of the Simon Fraser University Transputer Research Group. The work presented here is targeted to a transputer based multiprocessor. However, the findings should not be considered valid only for that platform as they may apply, in some degree, to any loosely coupled multiprocessor platform.

1.1. Apologies

I tend to use acronyms. Being a person who hates people who use acronyms, I find myself perched precariously on the precipice of hypocrisy. To prevent a perilous plunge, I provide a glossary of acronyms with this document. Please feel free to use it. The previous sentences prompt the next apology. I tend to be too verbose. This would be excusable if I had any idea what half the words I use actually mean. Therefore, I humbly apologize for any faux pas regarding sentence structure and word usage.

1.2. Hardware Overview

The primary hardware used during this research is a 64 node transputer network, constituting a loosely coupled multiprocessor. The transputer is a family of reduced instruction set (RISC) processors which were introduced in 1985 by Inmos [Inm89]. Each transputer is equipped with four high speed bi-directional serial links which allow parallel processing networks to be built easily and economically. Additionally, each device is also equipped with 4 KBytes of on-chip memory and a memory interface allowing the chip to be augmented with external memory. Through the use of *direct memory access* (DMA) circuitry, the transputer allows communication and computation to occur simultaneously and autonomously. There is no hardware support for shared memory between any two nodes in a transputer network, thus the autonomous communication, via the serial links, is required to provide transputer networks with an efficient message passing model.

The SFU transputer network is subservient to a Sun-4 host and is connected to that device via the Sun's VMEbus (This is elaborated on in Section 3.1.3). The Sun-4 host is also connected to the SFU computing science research ethernet.

At the commencement of this work, mass storage support systems were not available for transputer networks. In most cases, information to be processed by the transputers had to be routed from disk into the transputer network via the host machine (in our case the Sun-4) and the research ethernet. As if in answer to this problem, transputer based device controllers conforming to the SCSI standard [ANS86] were appearing but there existed little in the way of software capable of turning these controllers into full mass storage systems.

1.3. Research Goals

One of the goals of the SFU transputer group is to study the feasibility of providing transputer networks with a distributed file system. The work presented in this thesis is our initial adventure within this realm of research.

Disk systems can be classified into two layers, the lowest layer being *Local File Systems* (LFS) and the highest layer being *Distributed File Systems* (DFS). Local file systems provide an interface between client programs and a group of storage devices[1], which are controlled, possibly indirectly, by a single processor. The traditional distributed file system does not physically control any disks, rather it is a collection of communication protocols, cache coherency algorithms, etc., which coerce a consistent pattern of behaviour from numerous distinct local file systems. For example, Russell Sandberg et al [San85] discuss the implementation of the Unix based Sun Network File System using varying numbers of file servers, each of which provides a Unix based local file system. Also, Peter Dibble [Dib90] discusses a parallel interleaved file system (PIFS) which is built entirely upon a group of local file systems. This style of DFS is further detailed in sections 2.5.3 and 3.2.2.

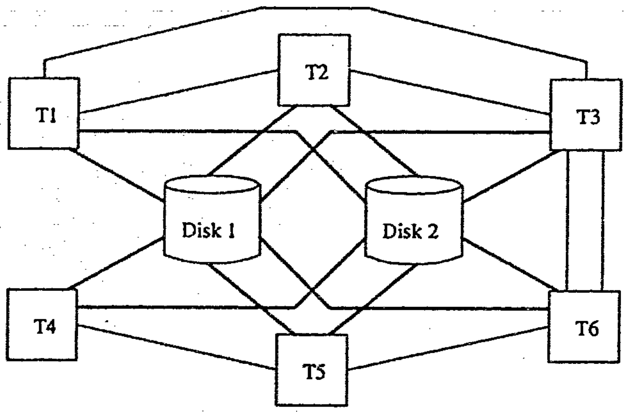
One of the nasty realities in developing a DFS is the need for supporting LFSs to build upon. Often these LFSs can be purchased from some computer vendor and the DFS built upon these pillars. For example, the Andrew system at Carnegie Mellon University was built upon generic Unix local file servers [How87]. Alternatively, an LFS can be simulated and used as the platform for the actual DFS. The simulation technique was used by Dibble in his PhD research. In our case no vendor-supported transputer-based LFS was available, therefore the first step in developing a DFS for the transputer network became the development of a LFS. It was our decision to implement an actual LFS, rather than simulating one, as was done by Dibble.

After great consideration I chose to call our LFS the Transputer Auxiliary Storage System (TASS)[2]. The primary goal of the TASS design is to provide a simple, efficient interface

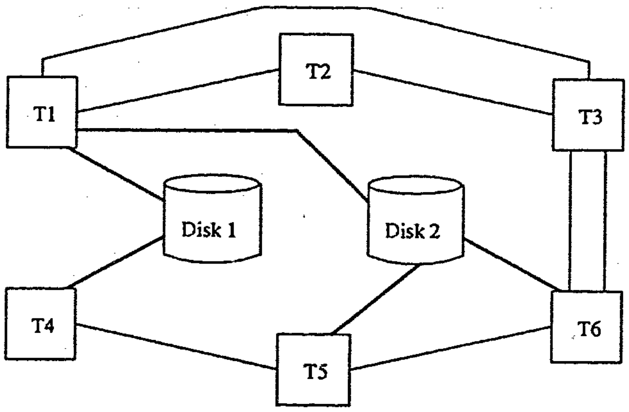
[1] The term *group* is meant to include groups containing only one storage device.

[2] To give some perspective to the time span of this thesis: Once upon a time, when this research began, there was a viable world power called the Soviet Union. It spanned a good percentage of Asia and Europe, including such countries as Russia, The Ukraine, Lithuania and Latvia. Within the Soviet Union lurked an information agency called TASS which performed admirably at the absorption of data and a little less admirably at

between mass storage and the transputer network. The system will not constitute a DFS, rather it will provide *pockets* of mass storage support (MSS) within the transputer network, in the form of LFS server nodes. These MSS pockets are to behave as if they were a single transputer node with access to large quantities of slow memory. This will allow the MSS pockets to be placed within the network wherever the system and/or program designer wishes, thus providing the system with good flexibility. This concept is illustrated in Figure 1. A fully distributed disk support system is shown as an example of what we are not trying to achieve and is compared to a network with two 'pockets' of mass storage. One pocket, Disk 1, provides storage support to transputers 1 and 4, while the other pocket services nodes 1, 5 and 6. Nodes 2 and 3 are not provided with any mass storage support at all. To improve



(i) Future Work: A DFS



(ii) The Objective: A LFS

Figure 1: Objective of Network Disk Support

the provision of that data to interested parties. To me, this sounded a little like a mass storage system (although closer to a tape based one than to a disk based one), so I thought it might be cute to choose some lofty title with the acronym of TASS. However, for the purposes of litigation, I must state that any similarities between this system and any persons, places, government agencies either living or dead is purely coincidental.

data throughput, a software disk cache will be implemented as a component of the TASS LFS design.

1.4. Thesis Overview

Chapter 2 overviews related work in the area of disk and disk cache system design. Also presented are some of the findings regarding the application of these techniques to parallel processing environments. Some similar systems, to the one proposed herein, are discussed.

Chapter 3 covers the design of the TASS system. This is a high level discussion dealing predominantly with issues of design modularity and system functionality. As a good percentage of the TASS design was dictated by the available hardware, an overview of the transputer multiprocessor platform used during this research is presented. TASS was designed as a rigidly layered heirarchy of modules. The functionality provided by each primary layer is discussed.

Chapters 4 through 6 cover the gory details of the software implementation. Chapter 4 covers the disk server design, focusing on the breakdown of the server into device dependent and device independent component modules. Chapter 5 covers the design of the cache server and file system. Chapter 6 overviews the interface between TASS and its clients. The techniques used within each software layer in order to provide the required services is discussed. A qualitative comparison between some of the techniques used and some possible alternative techniques is made.

Chapters 7 through 9 cover the performance of the TASS system. Chapter 7 is dedicated to the disk server performance. Similarly, Chapter 8 looks at the cache server performance and Chapter 9 analyzes the performance of the entire TASS system, as perceived by client processors. Performance issues considered include sustained data throughput between disk and clients. How the intervening layers of software affect data throughput is analyzed. The usage of processor cycles by each of the primary software layers is presented.

Chapter 10 presents the conclusions of the work.

Chapter 11 suggests future work, including design possibilities for a DFS.

The appendix consists of a glossary of acronyms, followed by the bibliography.

2. Related Research

There has been a great deal of research devoted to disk caching for multiprocessor environments, however most of this research appears to be grouped into the two categories of tightly-coupled (shared memory) multiprocessors and local area networks. The area of loosely-coupled multiprocessors seems to have been relegated to the technological revolution's back burner[3]. What research I have seen regarding disk systems in loosely coupled multiprocessors usually makes the assumption that point to point communication between all processing elements exists within the hardware architecture. Currently, the transputer environment does not support point to point communication and therefore much of the related research was not too relevant.

2.1. The Memory Hierarchy

Before discussing the concepts of caching and their application to the transputer environment, I will provide a brief discussion of the motivations behind data caches. Hwang and Briggs [Hwa84] describe computer memory systems as hierarchical levels of memory devices.

[3] This is my opinion and could just be a manifestation of my persecution complex which often leads me to assume that everything is a direct assault on my person. In other words, my inability to find suitable references may have led me to believe that no other persons are interested in the same things that I am, therefore, no other persons give a hoot in Hades about loosely coupled multiprocessors.

"The objectives [of a hierarchical memory structure] are to attempt to match the processor speed with the rate of information transfer or the *bandwidth* of the memory at the lowest level and at a reasonable cost" [Hwa84](page 52).

"[In other words,] the goal in designing an n-level memory hierarchy is to achieve a performance close to that of the fastest memory and a cost per bit [of storage] close to that of the cheapest memory" [Hwa84](page 56).

They further describe an example hierarchy involving, from highest to lowest layers, high speed (cache) memory, main memory, bulk memory, fixed head disk, moveable arm disk and tape devices. So, the objective of this hierarchy would be to provide data support to attached processors at or near the bandwidth of the fast cache memory, while providing an extremely large total memory capacity at the relatively inexpensive cost of tape and disk. The memory hierarchy concept was not originated by Hwang and Briggs, they just happen to be my favourite reference to it. It is also presented by Peterson and Silberschatz [Pet85], who date it back to the IBM 650 of the late 1950's, and by John Wilkes [Wil89].

The primary trait of each layer in the memory hierarchy is that the data transfer bandwidth provided by the associated device (the device at that layer) is larger than the bandwidth provided by devices at lower layers of the hierarchy. Additionally, the higher layers of the hierarchy almost always have significantly smaller total storage capacity than lower layers. Also, the size of the basic unit of transfer between the devices at each layer becomes smaller as the level gets higher. In other words, cache memory may allow transfers of single words of data between the processors and cache memory, whereas transfers between main memory and disk devices will usually be in the range of 1 to 100 KBytes.

The concepts behind Hwang and Briggs' memory hierarchy can be extended to distributed systems when augmented with additional layers such as local processor caches, local processor main memory, shared memory and network disk caches.

The operation of a memory hierarchy is similar to a 'recursive' procedure. Using the above hierarchy as an example, when a processor reads a piece of data, (eg: a specific word), the access goes through the high speed (cache) memory. If a copy of that word is currently stored in this high speed memory, then the word is transferred to the processor and execution continues. If that word is not resident in high speed memory, a process (either hardware based or software based) is invoked that will copy the specific word from slower main memory into the high speed memory. The first 'recursion' occurs at the level of main memory, where if the requested data is available, it is transferred to high speed memory. If the requested data is not available, the memory hierarchy will obtain it from the next lower level, namely bulk memory. This single data access may result in eventual accesses to disk and/or tape devices. At first, this process seems incredibly inefficient, since it may require access to every memory device in the entire memory hierarchy in order to read one 32 bit word. However, traditional data accesses exhibit what is known as *locality of reference*. The three basic flavours of locality of reference are *temporal, spatial and sequential* locality. Because these localities are discussed in some form by nearly every text and paper dealing with caching [Smi78],[Hwa84],[Pet85],[Sin88], etc., I will (and not be the first to) tag them with the lofty title of *principles*.

In the traditional software environment, both code and data segments of a program will exhibit temporal locality. What this means is that if a segment of code or piece of data has recently been used, then the odds are high that it will also be required by the program in the near future. This situation is a consequence of the fact that programs tend to be oriented around loop and procedure constructions which are often executed numerous times during any single program execution. Furthermore, programs usually involve temporary variables and stacks which are accessed often. A good example is the simple addition of a number **A** to some variable **B**. The contents of the memory location storing **B** usually must be copied into a register, the number **A** will then be added to it and **B** will be immediately returned to the same memory location. This accesses the same region of memory twice within a very short period of time.

Spatial locality is a phenomenon that occurs predominantly in the code segments of programs, however, it is not uncommon in data segments. The principle is that any reference to an element of memory will usually be followed, in the near future, by a reference to an element of memory that is spatially close to the previously referenced element. In other words, a reference to memory address **k**, will usually be followed by a reference to memory address **k**$\pm n$ where n is some small number. Examples are program branching and array processing.

Sequential locality is a specific form of spatial locality. It is mentioned separately since it leads into the discussion of prefetching (see section 2.3.5). In sequential locality, a reference to address **k** is usually followed by a reference to address **k**+1. Again, this occurs in both code and data segments, with examples being normal sequential execution of program regions and sequential processing of arrays.

As mentioned earlier, the memory hierarchy relies on these principles in order to provide high performance from a large amount of slow memory storage. Because most memory references exhibit temporal and spatial locality, a program can be executed with most of its current code and data requirements stored in high speed memory, which is the top layer of the memory hierarchy. Additionally, the entire program can often be stored in the slightly slower main memory. With programs that observe the locality of reference principles, the average time to access each word of memory should approximate the average time to read a word from the high speed memory, with only a slight degradation due to the occasional accesses that must be fetched from lower layers of the memory hierarchy. For example, 10000 data accesses at memory speeds of 30 ns per read coupled with one disk access at 5 ms per read will provide an average data access time of 499 ns per read, which is much closer to 30 ns than it is to 5 ms. In actual operation, the ratio of memory accesses to disk accesses will usually be much higher than the conservative 10000:1 ratio used here, further reducing the average access time. For example, 16 KBytes of data read during one disk access should require at least 4000 4 byte word transfers between processor and cache memory. Reuse of these bytes (during program loops or repetitive variable usage) while they remain in memory should increase this minimum of 4000 references.

The word *cache* is derived from the French word *cachet* which means "a supply of goods stockpiled for future use" [CSD79]. The concept of caching, as applied to computer systems, is derived from the memory hierarchy and the locality of reference principles. The locality of reference principles allow each layer of the memory hierarchy to act as a stockpile of data held for probable future use by any processors acting upon that memory. The high speed memory at the top layer of our previously specified memory hierarchy acts as a cache between the processors and main memory. The term cache seems to be becoming synonymous with this top level manifestation, however, I believe the word cache can and should be applied to lower levels of the hierarchy as well. For example, paging systems and buffering techniques, allow main memory to act as a cache of disk data. Caches of disk data within main memory are often referred to as *buffer pools*. Using tapes as backing store effectively makes disk packs a cache of the *current state* of the computer system, with the hopefully unneeded previous (backup) states of the system being stored in the less efficient tape library.

2.1.1. Disk Performance

Disks are manufactured as one or more disk *platters*, each having two *surfaces*[4].

[4] On some disk devices, one surface is not used. For example, the disk devices for the Apple II were im-

Surfaces are partitioned into *sectors* (similar to the partitioning of a cherry pie). Furthermore, each surface is further partitioned into *tracks*, much like the tracks of a record, except that disk tracks are separate circular entities, rather than one long spiral groove. Each sector/track combination is called a *block* and is capable of storing some fixed number of data bytes[5]. For example, each of the CP-3100 disk devices used within this research have 4 platters, giving a total of 8 surfaces. Each surface has 776 distinct tracks and is divided into 33 sectors. The 8*776*33=204864 blocks formed by these tracks and sectors each hold 512 bytes of information. An example disk is diagrammed in Figure 2. One additional term may

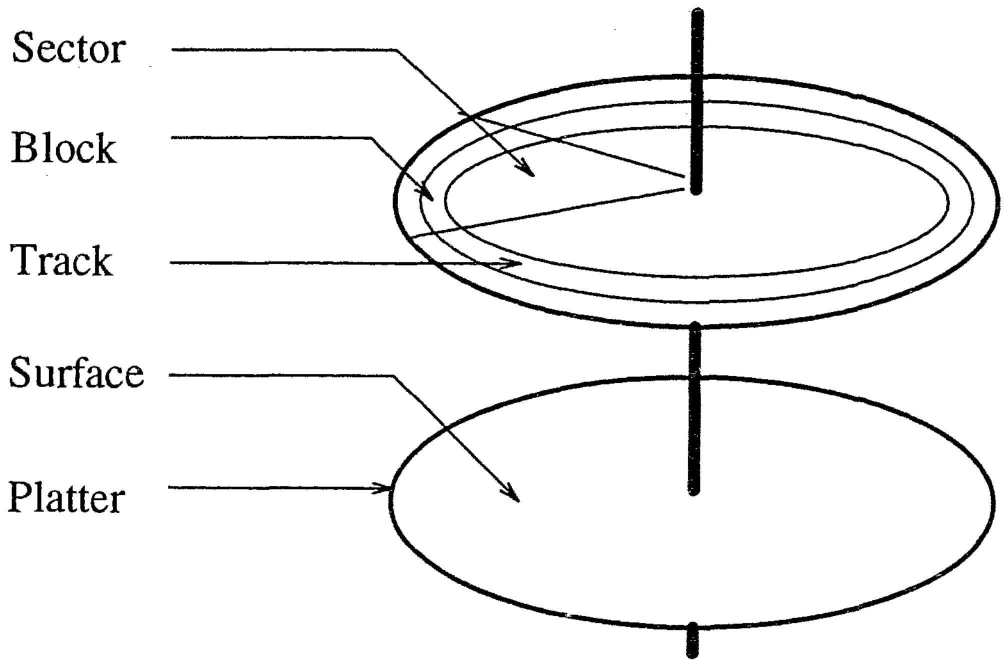


Figure 2: A Simple Disk Device

plemented with only one head, which was capable of reading only one surface of the standard 5 1/4" floppy disk.

prove useful, so I will define a *cylinder* to be the set of all tracks that are aligned vertically over all the surfaces. In other words, the outermost cylinder comprises all tracks that are outermost on their own particular disk surface. Each surface is equipped with a read/write head (RW head) that allows magnetically encoded data to be either read from or written to disk blocks.

Because of the architecture of disks, they are usually operated as *block transfer* devices, with information being transferred to and from them in either single block amounts, or multiple block amounts, often called *extents* or *chunks* [McV91],[Wil91],[Dud92]. The SCSI standard [ANS86] does allow for transfers of single bytes, but often the disk devices themselves prevent this style of access. In any event, such a technique would result in horrendous performance when looked at as an overhead:byte-transferred ratio.

To simplify the upcoming discussion, I will discuss read accesses only. If the material does not apply to write access, then I will qualify it. This will eliminate problem sentences like "data is copied from/to the disk to/from the controller". Disk platters are constantly being rotated. As a track on one platter passes under the head of the disk, the data can be transferred from the track and passed to any connected controllers. The time taken to perform this transfer is called the *transfer time*. Unfortunately, to begin reading a track the RW head must be positioned at the start of the data region to be transferred. The time taken to achieve this alignment is referred to as the *rotational latency* associated with a particular disk access. More often, this is referred to as the *average rotational latency* and is associated with a large sequence of disk accesses. Additionally, there is usually only one RW head per disk surface and this head must be positioned over the appropriate track, prior to beginning the disk access. The time required to position the RW head is called the *seek time*[5]. The I/O bandwidth of a disk device is the amount of data transferred divided by the sum of the rotational latency time, seek time and transfer time required to read that data. There is little that can be done to reduce the transfer time, since this is directly proportional to the rotational speed of the device which is in turn limited by the state of technology at the time the device was designed. Section 2.1.1.1 briefly covers a standard technique, known as *buffering*, which is intended to overlap computation with I/O, in order to reduce the delays suffered by processors that must wait for disk transfers to complete. Buffering does not alter the effective disk bandwidth, it merely tries to overlap it with computation. In order to improve the actual disk bandwidth, it is important to try and minimize the overhead associated with the rotational latency and the seek time. Fortunately, there has been a great deal of research devoted to minimization of the effects of both these times on system performance. Some known techniques for accomplishing just that are discussed in sections 2.1.1.2 and 2.1.1.3.

2.1.1.1. Buffering to Reduce Disk Delays

Perhaps the most common technique for dealing with disk delays is known as *buffering* and/or *double-buffering*.

> "The idea is quite simple. After data has been read and the cpu is about to start operating on it, the input device is instructed to begin the next input immediately. The cpu and input device are then both busy. With luck, by the time that the cpu is ready for the next data item, the input device will have finished reading it. The cpu can then begin processing the newly read data, while the device starts to read the following data" [Pet85](page 12).

[5] Fixed-head disk devices do exist. In such devices, each disk track has a dedicated head so no head movement is necessary, and hence the devices have no seek time. However, these devices are very expensive and therefore are not very common.

> "[The effect of buffering on system performance is] to smooth over variations in the time it takes to process a [unit of storage]. If the average speeds (in [units of storage processed] per second) of the cpu and the I/O devices are the same, then buffering allows the cpu to get slightly ahead or behind the I/O devices, with both still processing everything at full speed" [Pet85](page 15).

Effectively, buffering allows a system to proceed at the processing rate of the slowest device in the system, without imposing the additional delays that would result from activating all involved devices in a sequential fashion.

2.1.1.2. Disk Scheduling to Reduce Seek Times

The most prevalent technique for reducing seek times is to organize all outstanding disk access requests. The largest seek times for any disk device occur when the disk head must move across the entire disk surface, from the outermost track to the innermost, or vice versa. The time taken to seek to a new track is usually less when the tracks are closer together. If outstanding disk requests are organized so that they arrive at the disk controller in an order that minimizes the distance between any two subsequently accessed tracks, then the seek time observed between any two subsequent requests will be as small as possible, and the overhead imposed on the system, due to seek time, will be smaller than if the same requests are issued in an unordered fashion. There exist a number of *disk scheduling algorithms* that attempt to accomplish this task. However, any reordering of disk requests may have side effects on system performance. For example, an optimal disk scheduling algorithm is not fair if a disk request is starved out. Peterson and Silberschatz [Pet85](pages 257-268) provide a good overview of disk scheduling algorithms, and the interested reader is referred to their text. This software testbed used during this thesis does not implement any disk scheduling algorithm, rather the simple *First Come, First Served* (FCFS) method is utilized. Reasons for this will be discussed later in Section 4.1.1.

2.1.1.3. Track and Sector Interleaving to Reduce Rotational Latency

As mentioned earlier, rotational delay occurs when the disk controller has to wait for the disk head to line up with the start of the disk region to be accessed. Often, there is little that can be done to reduce this problem. For example, after a disk seek, the position of the desired track in relation to the disk head may simply not line up and since that track must be accessed next, there is nothing to do but wait. However, in some cases the penalty due to rotational delay may be reduced. It is important to note that these delays appear random to the user of the disk, but in fact are not random for particular instances. In other words, a seek beginning at track 5 and ending at track 23 will always take the same time. Similarly, since the disk is rotating at a fixed speed, it will move through the exact same number of rotations during that specific seek as it did during any previous seeks between tracks 5 and 23. More importantly, seeks between adjacent tracks will always take the same amount of time. Disk controllers can take advantage of this predictability in order to reduce the average rotational delay exhibited by the system.

Assuming that an entire disk track cannot be read during a single disk access, then the order in which blocks are addressed around that track becomes an important performance issue. Usually, the solution to this addressing issue is extended to sectors addressing around each surface of the disk platters. These addressing techniques are called *sector interleaving.*

Figure 3 shows two options for block address placement on a single track of a single disk surface. Figure 3 (*i*) shows what many people would at first consider to be the obvious addressing technique. However, if two consecutive reads to addresses *i* and *i*+1 were to occur (a common event during sequential access to disk data), then during the small but non-zero time taken to complete the access to block *i* and subsequently initiate the access to block *i*+1, the disk head would have moved past the starting point of block *i*+1 and the disk would have to wait one entire revolution before it could execute the second read. If the sector

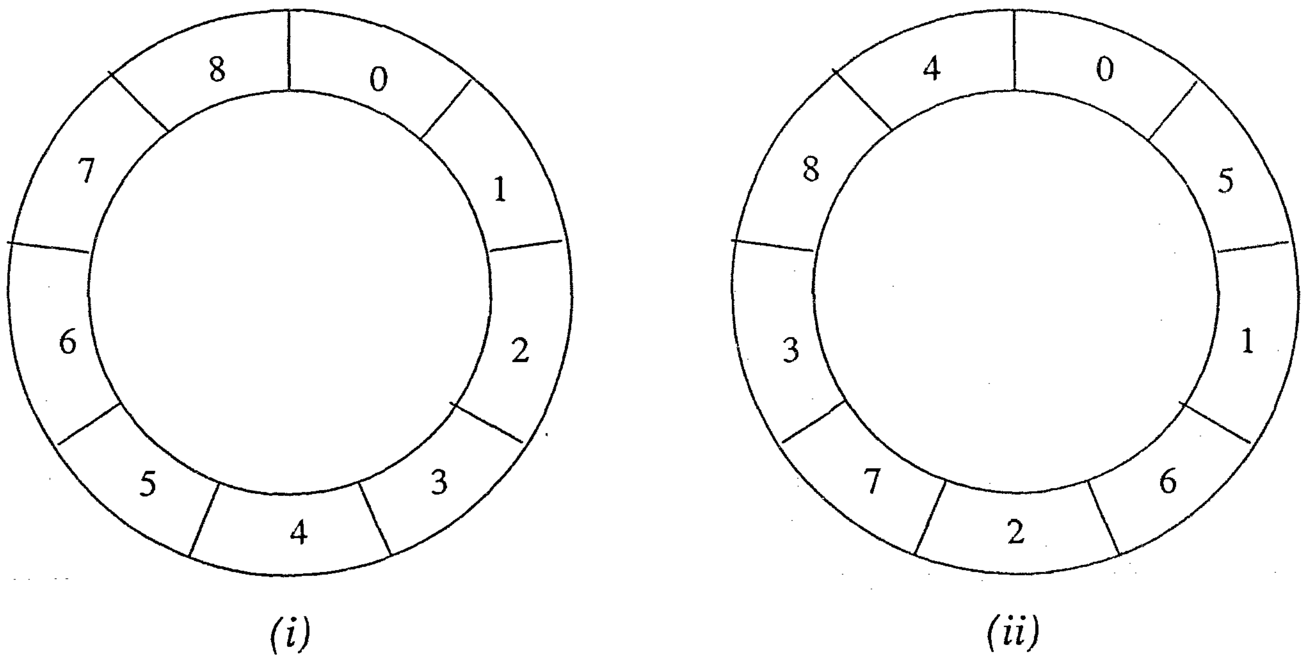


Figure 3: Sector Interleaving

interleaving shown in Figure 3 (*ii*) were used, then the time to finish one request and initiate the other should be less than the time taken to rotate to the next sector. Therefore, the amount of time spent waiting for the disk to rotate should be substantially less than one entire rotation. McVoy and Kleiman write that "If the I/O system is properly tuned, the I/O request will get to the disk as the appropriate block is moving under the head". They further discuss that unfortunately this arrangement, with every second sector being a 'hole' in the sequential ordering of blocks around the disk, "reduces the maximum transfer rate to half that of the disk rate" [McV91].

Alternatively, if an entire disk track can be read during a single disk access, then it has been shown, and this research will verify again, that the highest data throughput to/from disk occurs when the unit of disk access is equivalent to an entire disk track. The reason for this is that an entire track's worth of data of file data can be read in one [disk] rotation. The problem with this technique is that, after reading an entire track, there is nothing for the device to do except seek to a new track, or in the case of multiple disk platters, switch to a new RW head on a different platter. Although the seek time or head switch time may be significantly smaller than the rotation time for the disk device, an entire rotation may be required to align the RW head with the start of the new track. Therefore, each track read requires a second rotation to realign the RW head with the start of the track.

Some devices attempt to compensate for this alignment problem by staggering the start of tracks. For example, the disk in Figure 4 has finished reading the inside track and seeks to the outside track. If the seek time is less than 1/9th of the rotation time (nine total sectors, outside track is staggered by one sector) then the disk head will finish the seek prior to the start of the track arriving at the head's position. This is known as *track interleaving* since the optimal choice for how many tracks to seek over may not be a single track, depending on the

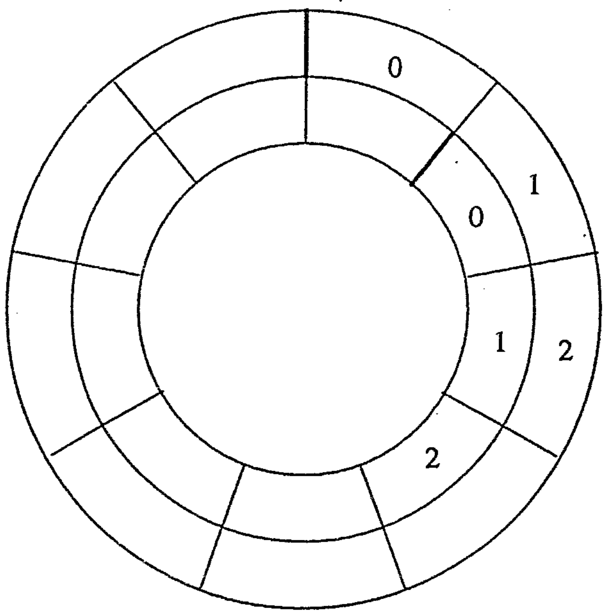


Figure 4: Track Interleaving

staggering of sectors around neighbouring tracks.

2.1.2. Joining of Multiple Disks

Traditionally, high performance disk devices have been built as *single large expensive disks (SLED)* and controlled individually or as groups of individual devices. Facilities that require large, high speed data stores will use large, high speed disk devices. The higher the facilities requirement for data storage size and I/O bandwidth, the higher the cost of the storage devices. Providing high I/O bandwidth with large disk devices usually leads to a high cost per megabyte of storage.

> "The performance of [SLED devices] has improved [during recent years] at a modest rate. These *mechanical* devices are dominated by the seek and the rotation delays: from 1971 to 1981, the raw seek time for a high end IBM disk improved by only a factor of two while the rotation time did not change... ... There is no reason to expect a faster rate in the near future" [Pat88].

"Personal computers have created a market for inexpensive magnetic disks. These lower cost disks have lower performance as well as less capacity" [Pat88]. However, as these smaller disks are targeted towards the general public, the need to make them affordable has been a primary driving force in their design. In other words, these smaller disks are also becoming quite inexpensive. In a 1989 paper by Garth Gibson, et al., they state that "The 5.25 inch form-factor drives may currently have the best cost per megabyte, and the 3.5 inch form-factor is expected to overtake it soon" [Gib89]. This significant advantage in cost per megabyte has prompted a new trend towards providing high capacity, high performance disk systems using multiple small inexpensive disks.

Research has burgeoned in the field of high performance disk devices composed of arrays of inexpensive disks. Primary examples are the numerous papers available on the concept of *Redundant Arrays of Inexpensive Disks* (RAID) [Gib89],[Pat88],[Sch88] as well as the more specific discussions of Hewlett-Packard's *DataMesh* prototypes [Wil89],[Wil91]. Most of the research involved in RAID prototypes involves the *joining* of multiple disks into a 'single' abstract disk which provide high performance as well as reliable data storage.

2.1.2.1. High Bandwidth Disk Arrays From Low Bandwidth Disks

The underlying factor in providing high I/O bandwidth using low bandwidth disks is that all disks in a group of disks can be accessed in parallel, whereas a single disk can only deal with one disk request at any given instant in time. Given that a SLED can provide data to its users at some bandwidth **B**, and an array is built using **n** disks each of bandwidth **B/n**, then the same data throughput should be obtainable from the multiple disks as is obtained using the SLED. By adding more disks to the array, even higher I/O bandwidths should be obtainable. Increasing the bandwidth of SLED devices would require improvements in device design which might require a higher level of technology than is currently available and/or an expensive upgrade of disk devices.

For this paper, I will define a *disk joining* to be any technique for combining multiple disks into a single abstract disk. From the point of view of performance, the abstraction of a group of multiple disks into an array of disks (which behaves like a SLED), can be done using at least three disk joining techniques. The three techniques that are relevant to this research are *concatenation*, *interleaving* and *striping*. I have been surprised to find that many papers claim that interleaved and striped disk systems are manifestations of the same monster. Other papers, notably [Kot91], differ on this point. I will agree with the latter group and claim that interleaving and striping are distinctly different.

For the purposes of this research, it will be sufficient to assume that all disks in any given joining are identical. The concepts presented can be extended to environments where the component disks are not identical, but this makes the discussion significantly more complex. All three techniques, concatenation, interleaving and striping, are shown in Figure 5. This figure does not attempt to describe joinings using generalized component disks. Rather, component disks are simple devices, each having three storage regions which will be called *chunks*. The joining of these simple component disks should be sufficient to illustrate the three join techniques described herein. The three techniques serve to combine a group of **N** component disks, $\mathbf{D}_n$, for $0 \leq n < \mathbf{N}$, into a single abstract disk, $\mathbf{D}_{Joined}$. Given that each component disk has a storage capacity of $\mathbf{C}_{Component}$ bytes, then the capacity $\mathbf{C}_{Joined}$ of the abstract disk will be $\mathbf{N}^*\mathbf{C}_{Component}$ bytes. An *extent* or *chunk* is a general unit of disk transfer whose size is selected to provide optimal disk performance during disk transfers. The term extents is used here instead of bytes, tracks or sectors, because the techniques are general and will apply to any choice of base storage unit. The choice of extent size is usually a performance issue and as such should not be specified at this point. Rather, I will define the size of each extent to be $\mathbf{S}_n$ for component disk n. The size of the extent for the combined disk will be denoted as $\mathbf{S}_{Joined}$. Furthermore, the component disks will each have $\mathbf{E}_{Component}$ extents, while the combined disk will have $\mathbf{E}_{Joined}$ extents. Each component disk has its own address space which I will refer to as $\mathbf{D}_x$[i] where $0 \leq \mathrm{i} < \mathbf{C}_{Component}$. Similarly, the abstract disk has an address space which can be described notationally as **D**[j] where $0 \leq \mathrm{j} < \mathbf{C}_{Joined}$. Additionally, **div** and **mod** will be used to represent the integer division and integer modulo operations, respectively.

For *concatenation*, each disk Di maps $\mathbf{E}_{Component}$ contiguous extents onto $\mathbf{D}_{Joined}$. For example, the first group of extents within $\mathbf{D}_{Joined}$ are stored on $\mathbf{D}_0$, the second group of extents within $\mathbf{D}_{Joined}$ are found on $\mathbf{D}_1$. All component disks, as well as the combined disk, will have the same extent size.

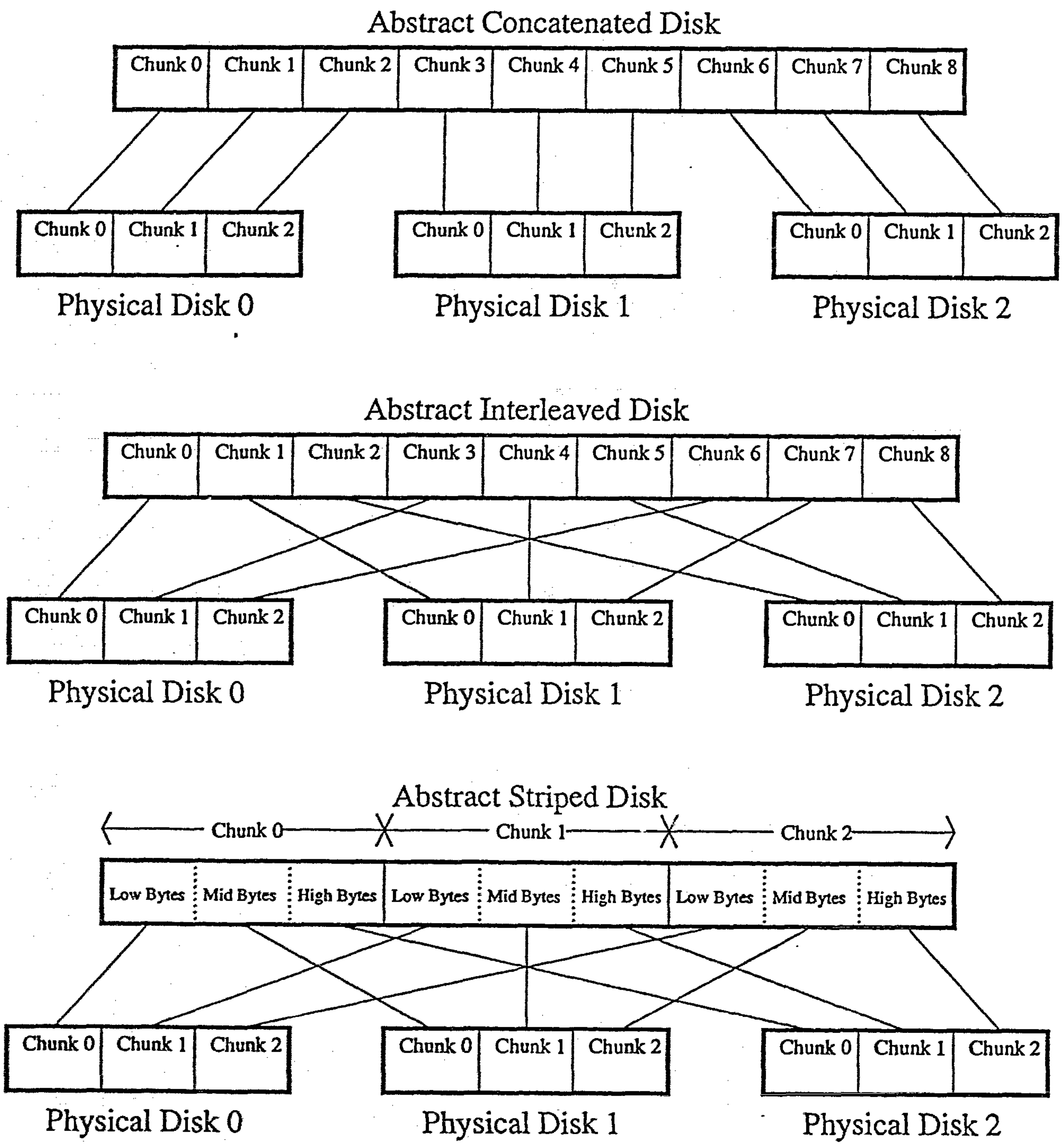


Figure 5: Joining Techniques

$$\begin{aligned}
\mathbf{D}[i] &= \mathbf{D}_{i \text{ div } \mathrm{N}}[i \text{ mod } \mathbf{C}_{Component}] \\
\mathbf{S}_{Joined} &= \mathbf{S}_n \text{ for all } n \\
\mathbf{E}_{Joined} &= \mathbf{N} * \mathbf{E}_{Component}
\end{aligned}$$

For *interleaving*, the disk extents from each real disk are alternated with disk extents from other disks in order to form the combined disk. The size of each extent on the combined disk is equivalent to the size of each extent on the component disks. Notationally we have,

$$\begin{aligned}
\mathbf{D}[i] &= \mathbf{D}_{i \text{ mod } \mathrm{N}}[i \text{ div } \mathbf{C}_{Component}] \\
\mathbf{S}_{Joined} &= \mathbf{S}_n \text{ for all } n \\
\mathbf{E}_{Joined} &= \mathbf{N} * \mathbf{E}_{Component}
\end{aligned}$$

In *striping*, extents k from each component disks are concatenated together to form one extent, which will be extent k on the combined disk. Therefore, the combined disk has an extent size which is the sum of the extent sizes from each component disk.

$$\begin{aligned}
\mathbf{D}[i] &= \text{Concatenation of all } \mathbf{D}_{Component}[i] \\
\mathbf{S}_{Joined} &= \mathbf{N} * \mathbf{S}_{Component} \\
\mathbf{E}_{Joined} &= \mathbf{E}_{Component}
\end{aligned}$$

Each technique has advantages and disadvantages. Concatenated disk systems do not generally provide as good of data support as do systems joined by the other techniques. However, when multiple independent processes are each accessesing a unique file (ie: each process reads/writes one file), often these files can be distributed across the n disks of the joining so that each process has its own dedicated disk, or only shares a disk with a small number of other processes. In these circumstances a concatenated disk system can perform better than a striped or interleaved disk system.

The striping and interleaving techniques generally multiplex the disk access across the component disks better than the concatenation technique does. However, this is only a substantial advantage when the sequence of disk accesses moves through the disk address space in a sequential fashion. Each striped access activates all component disks in the disk joining, effectively keeping all disks busy at all times. The interleaving technique activates one component disk, each access. All disks can be kept busy under heavy load, since there should always be at least one outstanding request that is targeted for each component disk. As discussed for concatenated disk systems, if multiple processes try to access multiple files in a striped or interleaved disk system, the result is often poor performance. Simultaneously accessing multiple files, which happen to be distributed across the disk address space, generates a sequence of disk accesses that appear random to the component disks in a striped or interleaved joining, even though those accesses are not random when viewed at a higher level of abstraction.

2.2. Cache Design

As with any feature of any computer, there are three primary ways of implementing caches, namely with hardware, software or a combination of the two. In some cases, the type of solution required is dictated by the environment. For example, high speed memory caches layered on top of slower main memory must be controlled by hardware, since software solutions would require execution of code segments for each memory access. Since executing a code segment requires memory access, the existence of a software solution would constitute a paradox[6]. Similarly, a main memory based cache of disk data may require a software solution if no supporting hardware was provided by the manufacturers of the target computer system. An example of a combination solution would be a paging system, where specialized hardware is required to detect page faults and trap them to the operating system, which then uses software routines to fetch the required pages into main memory. Paging is a fairly detailed concept and I don't wish to explore it further, as it does not signifantly apply to this thesis. Paging is discussed in most operating system textbooks and numerous papers, including [Hwa84],[Pet85],[Mor88]. Likewise, hardware based caching is not particularly relevant to this thesis and the interested reader is referred to [Hwa84],[Smi87],[Hil88]. The research within this thesis is entirely related to software based cache systems. I will try to note the places where hardware or combination based systems have either influenced the design of my software platform or motivated some avenue of research.

2.2.1. Cache Components

Hwang and Briggs state that

> "The cache memory generally consists of two parts, the *cache directory* (CD) and the *random-access memory* (RAM). The memory portion is partitioned into a number of equal-sized blocks called *block frames*. The directory, which is usually implemented as some form of associative memory, consists of block address tags and some control bits..." [Hwa84](page 98).

All physical data stored in the cache will be maintained in the block frames. The block address tags provide a mapping between the actual physical addresses of the data and the block frame where that data has been cached, if it is present. Block frames will alternatively

[6] This statement assumes that we are not talking about a micro-coded *software solution*. Personally, I lump micro-coding in with hardware, since micro-code is usually stored in a ROM which is dedicated to the micro-code engine.

be called cache buffers during much of this paper. The two terms are equivalent and interchangeable. The types of control bits required vary with the design selected. Some of the common bits are the *valid* and *dirty* bits. Additionally, various *protection bits* may be necessary to insure consistent cache operation.

- The *dirty bit* for each cache buffer designates whether or not the data maintained in that cache buffer has been altered since it was copied from some lower level of the memory hierarchy.
- The *valid bit* for each cache buffer designates whether or not the data maintained in that cache buffer constitutes a valid copy of the data for the indicated memory address.
- The primary use of *protection bits* are to prevent violation of the write one, read many (WORM) requirement, which states that only one process can have write access to a memory location at any instant in time. It also states that numerous processes may simultaneously read a memory location, assuming a write to that location is not currently underway.

A few basic terms are common in discussions of cache operation and cache performance. Since I can think of no better place to define them, I will do so now.

Definition: A *cache hit* occurs when an attempt to access some region of the disk address space finds a valid copy of the data from that disk region stored in the cache.

Definition: A *cache miss* occurs when an attempt to access some region of the disk address space fails to find a valid copy of the data from that disk region stored in the cache. A cache miss is the opposite of a cache hit. Each cache access results in either a cache hit or a cache miss.

Definition: The *hit ratio* exhibited by a cache is the ratio of the number of cache hits, to the number of cache accesses.

Definition: The *miss ratio* exhibited by a cache is the ratio of the number of cache misses, to the number of cache accesses. The sum of the miss ratio and the hit ratio is always 1.

2.3. Caching Strategies

There has been extensive research on cache techniques for both hardware and software cache environments. As I am fond of old growth forests, I do not want this thesis to be single handedly responsible for their destruction, therefore, I will give a highly abridged discussion of this research.

Regarding hardware based cache memories, Alan J. Smith writes

> "... their design is far from cut and dried. The designer has to make several choices and set various parameters. For example, decisions must be made on the algorithms that fetch, place and replace information, on the way the cache should be addressed, on the best size of the cache, given performance goals and design constraints, and on the best way to ensure consistency among several caches in a multiprocessor" [Smi87].

Most of these considerations will have to be dealt with by designers of software caches as well. These numerous design decisions I will cumulatively refer to as *caching strategies*.

There are a few primary conditions that must be handled by any cache system. Any design that does not correctly satisfy these conditions, will result in incorrect cache operation. The primary conditions are:

- The data stored in the cache must accurately reflect the data stored in the next lower level of the memory hierarchy. Any changes to the cached version of the data must eventually be executed to the copies stored at the lower level of the hierarchy. Furthermore, use of copies stored at lower levels of the memory hierarchy must be prevented when and if a modified version of that data is present in a higher level cache. I will call this requirement the *cache consistency criterion*.

- In parallel environments, it is often the case that multiple caches exist at the same level in the memory hierarchy. This environment will be discussed later. For now, it will be sufficient to state that changes to data stored in one cache must be reflected in all other caches at the same level of the memory hierarchy, before that data is used by other processes. For example, let processor **A** have a cache $\mathbf{C_A}$ and processor **B** have a cache $\mathbf{C_B}$ which are both at the same levels of the memory hierarchy. If **A** changes some shared variable **x** in $\mathbf{C_A}$, then **B** should not be able to access the same variable **x** from $\mathbf{C}_B$ until the change has been reflected in $\mathbf{C}_B$. This is known as the *cache coherency criterion*.

- As all caches must have a limited size, they will eventually become filled as programs reference sufficient code and data to exceed the caches capacity. Therefore, a technique for obtaining room by replacing old cached data with new cached data must be decided upon. I will refer to the selected technique as the *cache replacement algorithm* (CRA). Common variants on CRAs will be overviewed in section 2.3.2.

2.3.1. Placement Policy

Hwang and Briggs state that "One of the most important parameters in the design of a cache memory is the *placement policy*" [Hwa84](page 99). The placement policy for a cache at level i of a memory hierarchy establishes a correlation between cache addresses and corresponding memory addresses at the immediately lower level i-1 of the hierarchy. As this research regards disk caches, the remainder of this section will refer to placement policy for disk caches. Any discussion regarding other levels of the memory hierarchy will be explicitly indicated. The purpose of a placement policy within a disk cache is to establish a correlation between cache buffer addresses and the actual disk addresses whose data is stored in that buffer. In other words, a placement policy is a mapping between disk addresses and cache buffer addresses.

There are three primary placement policies, (*i*) *direct mapped*, (*ii*) *fully associative* and (*iii*) *set associative*. An exhaustive description of these mapping techniques is given in Hwang and Briggs [Hwa84](pages 102-107). I will only provide the briefest of descriptions of each technique and any interested readers are referred to Hwang and Briggs.

Direct mapping is the simplest of the placement policies. Given that there are **N** disk regions of size S, direct mapped caches use k cache buffers, also of size **S**, to cache those $N=m*k$ disk regions. If resident in the cache, disk region i, $0 \le i < n$, will always be stored in cache buffer i modulo k. The performance and implementation of direct mapped caches is well discussed by Mark Hill [Hil88].

Fully associative mappings are, "In terms of performance, the best and most expensive cache organization. The mapping is such that any block in memory can be in any block frame" [Hwa84](page 104). In other words, there is no correlation between disk address and cache buffer address, as any disk region can be stored in any of the cache buffers. Usually, this technique is too expensive to implement, especially in hardware based caches of main memory, since large *Content Addressable Memories* (CAM) are required [Hwa84](page 380). These CAMs usually require too much of the available silicon real estate. For software based caches, fully associative mappings are not limited by silicon real estate, but often the memory required to maintain cache control data structures is too large to justify the use of a fully associative mapping. The software test bed for this thesis utilized this technique with optimizations to reduce memory requirements. This is detailed in section 5.1.2.

Set associative mappings are a hybrid of direct mapped and fully associative mappings. Basically, the cache is divided into S sets, each of which contains some constant number b of the $\mathbf{B}=\mathbf{S}*b$ cache buffers. Disk regions are directly mapped to particular sets. In other words, disk region i will always be stored in a buffer within set i modulo **S**. However, within sets, the mapping is fully associative so that any disk region can be mapped into any of the b

cache buffers within set *i* modulo S. With some consideration it can be seen that direct, fully associative and set associative mappings are all set associative mappings, with direct and fully associative mappings being the opposite extremes[7].

2.3.2. Replacement Algorithms

As mentioned previously, there exists an irritating problem associated with caches, namely that they have a finite size and therefore will at some point fill to their capacity. Determining what information should remain in a full cache, and what information should be replaced with more immediately useful information, is a highly studied field. Numerous cache replacement algorithms (CRAs) exist, although almost all research has shown that one particular algorithm, known as *least recently used* (LRU) usually provides the best general purpose performance [Hwa84],[Pet85],[Smi87],[Sto89].

The research contained in this thesis used a purely LRU replacement policy, so most of the discussion on replacement algorithms will deal with LRU and its advantages and disadvantages. However, to provide some breadth on the topic, I will briefly mention some other algorithms that have been proposed as CRAs. My source for these is [Pet85] and although I do not use direct quotes, I have paraphrased extensively.

- **Optimal:** This algorithm is the best CRA. The policy used in the optimal algorithm is *Replace that entry which will not be used for the longest period of time*. This algorithm is impossible to implement as it requires detailed knowledge regarding the future, and any

[7] Direct mapping is a set associative mapping where each set contains only one cache buffer. Fully associative mappings are a set associative mapping where there is only one set, containing all of the available cache buffers.

programmer with such detailed knowledge would be a very rich gambler or stock broker and probably never touch another computer in their lifetime, let alone deal with something as mundane as CRAs. The optimal algorithm is used by Peterson and Silberschatz as a performance target for all other replacement algorithms.

- **Least Recently Used (LRU):** This algorithm replaces the entry that was used the longest time ago (or least recently). As expected, "the LRU replacement policy is good, but not optimal" [Sto89]. In the same paper, Stone states that "an LRU strategy is robust, near-optimal, and difficult to improve upon in practice".

- **Most Recently Used (MRU):** This algorithm replaces the entry that was used the shortest time ago (or most recently). At first glance this may seem to be a fairly absurd policy, but for certain access patterns it has been shown to be quite effective.

> "Considering large sequential I/O, we can see that the pages just brought in [to the cache] are recently touched and as such will not be candidates for page replacement [under LRU algorithm]. This has the side effect of using all of memory as a buffer cache for I/O pages. For limited I/O, this is generally a good policy, but for large (greater than memory size) I/O this is a poor policy since it will replace all, potentially useful, pages with I/O pages that are unlikely to be reused" [McV91].

The MRU algorithm attempts to bypass this problem by replacing the most recently used pages, which will usually contain the stream data that was just written out or read in.

Although choosing a replacement algorithm is an important decision in any cache design, "It has been shown that, in general, the effect of cache replacement algorithms on the performance of the cache is secondary when compared to the effect of the [placement policy] on performance" [Hwa84](page 118).

2.3.3. Combined and Partitioned Caches

The effectiveness of a cache at improving data throughput between levels of a memory hierarchy is often dependent upon the type of data access pattern encountered. A *data access pattern* (DAP) is the way in which memory accesses occur, relative to previous and subsequent memory accesses. For example, DAPs can be sequential or random. David Kotz provides a detailed description of sequential and parallel DAPs in his PhD thesis [Kot91]. I provide a summary of his work on this topic in the upcoming section 2.4.3. Often, cache performance can be tuned to the nature of the DAP that is expected to be encountered. For example, a database machine would be more likely to produce a random DAP than would a data acquisition system, which would most likely generate a sequential DAP. Also, multiprogrammed environments will present multiple processes competing for a shared cache space. Differences in the DAPs generated and in how they are cached by these processes can seriously affect system performance.

> "Traditionally, computers have been built with a single cache for both data and instructions. This is the simplest method -- one cache communicates with one main memory. Moreover, the CPU's components have only one unit to refer to, whether they are reading or writing data, or calling for instructions... ...But there are advantages to splitting data and instructions into two separate caches. Conflicts between simultaneous instruction and fetches and data reads and writes are eliminated..." [Smi87].

The approach of splitting caches has usually been kept to high speed caches of main memory data, as determining the differences between instruction and data fetches is easiest at this level. I am unaware of any attempts to perform this type of cache partitioning at the disk cache level.

Thiebaut and Stone discuss a cache partitioning for multiple disk environments.

"Because each physical disk has a particular set of resident files and the accesses to those files have characteristic [data access] patterns, a natural way to use the partitioning model for our purposes is to associate a distinct process with each physical disk. The cache-partitioning problem becomes a problem in partitioning a large cache among [N] competing processes, each associated with a distinct disk" [Thi89].

They also claim that "It is also possible to partition a composite stream of accesses into streams associated with operating system, other systems programs, and individual application programs" [Thi89].

Partitioning of caches into separate regions, one for each currently active process, has also been extensively researched. In a multiprogrammed environment, with one cache shared by all processes and utilizing LRU replacement would have the current process $\mathbf{P}_{Current}$ replace all instructions and/or data cached by all other processes before any of $\mathbf{P}_{Current}$'s data was replaced. In this fashion, when another process is swapped in and begins to execute, none of its recently used instructions and data will be cached, resulting in cache misses on all initial accesses, further increasing context switch time (or in the worst case, causing a page fault and subsequent context switch).

"The miss rate tends to be high during the early part of the [context switch and cache reload] transient, and then drops as the working set of the process becomes resident in cache ... ... For large caches a better replacement policy than LRU replacement is to increase the cache allocation of a running process until the marginal improvement in miss rate multiplied by the time remaining in the quantum drops below unity" [Sto89].[8]

In multiproccessor environments the partitioning of caches among competing processes is also under investigation. However, this is not relevant to this research and the interested

[8] The process' *quantum* is the amount of CPU time it has been dedicated during its current execution phase. In other words, it is the amount of time it will be allowed to execute between being swapped in and subsequently swapped out, barring an interrupt.

reader is referred to [Sto89],[Thi89].

2.3.4. Write Policy

A write to level *i* of the memory hierarchy only alters the copy of the target data item stored at that level. The method used to update the changes to the item at the next lower level *i*-1 of the memory hierarchy is called the *write policy* for level *i*. There are two schools of thought on write policies, these being *write-through* and *write-back* policies [Hwa84].

In write through, any write to a piece of data cached at level *i* of a memory hierarchy, will also generate a write to the copy of that data stored at the lower level *i*-1. Therefore, the operation writes 'through' the level *i* cache. The advantage of using a write through policy is that the copies of a data item stored at two adjacent levels of the memory hierarchy are always identical, therefore, during failures there is no worry that changes to data is not reflected at the lowest level of the hierarchy. The primary disadvantage of using write through is that "5 to 30 percent of all memory references are write operations, ... [and for writes to a full cache] the processor is blocked during the write-through" [Hwa84(page 114). Effectively, read operations execute at or near the speed of level *i* of the hierarchy, but write operations execute at speeds dictated by some or all of the lower levels of the hierarchy.

Write back policies execute write operations only to the cached copy of the data. Writing changes to lower levels of the memory hierarchy is postponed until the data item is explicitly flushed out of the cache or until the cache buffer in which the data is stored in gets targetted for replacement by the cache replacement algorithm. The advantage of write back is that write operations can proceed at the speeds of the higher levels of the memory hierarchy. The disadvantage of write back is that users of the system are not automatically guaranteed that their changes get executed directly to permanent storage. Generally, a write back policy is implemented as a *flagged* write back. A dirty bit is associated with each particular cache buffer and is set whenever the data item stored in the cache buffer is modified. Before a cache buffer can be used for other purposes (ie: contents replaced with new data as per replacement algorithm), the dirty bit must be checked. If the bit is set, the entry must be written back to the next lower level of the memory hierarchy before the cache buffer can be used for other purposes. If the bit is not set, then the data can be discarded as it has not been modified while in the cache and the copy stored at the next lower level of the memory hierarchy will be identical.

2.3.5. Prefetching

The technique of prefetching is often used as a method for increasing the data throughput between any two levels of a memory hierarchy. As this research concentrates on disk caching, so will the following discussion. Waiting for a program to reference some piece of data stored on disk will probably result in a *cache miss* within the disk cache, since that piece of data has likely not been used recently. However, if the computer system were able to accurately predict what regions of disk would be referenced in the near future, then those regions could be fetched from disk and copied into the disk cache, prior to the time at which the actual data access occurs. If this region is fetched prior to the physical access, then use of the data can occur at main memory speeds, rather than at the much slower disk speeds. Of course, being able to infallibly predict the future is not always possible, therefore, the effectiveness of prefetching is limited by the accuracy of the prediction technique. "Certain [data access] patterns are commonly found in memory referencing behavior of computer programs, and it is possible to use these patterns to attempt to predict which sections of a program's address space will be referenced next" [Smi78](page 7). Smith mentions that the sequential nature of most object code and data stores (eg: arrays and queues) makes prediction of data accesses relatively easy.

Smith discusses that "... prefetching is generally a better idea for larger memory sizes" [Smi78](page 11). Therefore, its use in a disk cache, where the size of the memories involved is usually fairly large, should provide an effective performance enhancement. "For large enough memory sizes, prefetching reduces the miss ratio to about 15 percent of its former level" [Smi78](page 11). However, a slight increase in disk transfers (of about 10 percent) may occur since not all predictions will be correct, therefore some prefetches will copy unrequired data into the disk cache.

The prediction technique presented by Alan Smith in the much referenced 1978 paper [Smi78] involves only prefetching within a sequential access environment. The technique is called *one-block look-ahead* since the only data prefetched is that data stored in the memory region which is immediately sequential to the memory region just accessed. In other words, a memory access to region *i* will read that region from disk into the cache memory, in order to satisfy the memory access, and then immediately afterward a prefetch request to read region *i*+1 from disk into the cache will also be issued. This technique can be generalized to *n-block look-ahead* where the *n* disk regions subsequently following the immediately accessed region are read into the cache. This generalized technique assumes that sequential access will occur over much larger regions of the disk address space.

2.4. Data Access

The data throughput of a file and/or mass storage system while under load varies with the nature of workload presented. Often throughput will vary drastically with the nature of the workload. Showing that a given system provides some set rate of throughput is a meaningless endeavor unless a qualification is made regarding the type of workload that was utilized while the performance tests were underway. Additionally, if a manufacturer's performance figures are provided for some specific workload and a new system's requirements dictate that the probable workload will be drastically different than the manufacturer's test load, then any throughput results provided by that manufacturer may be worthless to the system designers. Because of these factors, I believe a strict definition must be provided in order to classify most[9] workloads into logical groupings. There are at least three components required in any classification of data access. These are (*i*) the type of access, eg: whether data is written to the disk or read from the disk), (*ii*) the nature of object accessed (eg: fixed size regions as opposed to variable length regions) and (*iii*) the temporal and spatial locality of the data accesses.

2.4.1. Data Operations

There are three basic operations for accessing the data within a file, these are read, write and update. These three operations differ only in the operations executed between the times when the file is opened and closed. A file is *read* if the only operations performed are read operations. Similarly, a file is *written* if the only operations are write operations. A file is *updated*, if both read and write operations occur. When data stored on a disk lacks the high level distinction incorporated within concepts such as a file, then the idea of updating such data loses meaning.

I suggest that file activity is usually limited to read or write access. In support of this I reference [Flo86],[Kot91]. Therefore, this research deals primarily with read and write access, according less attention to update activity.

[9] With any classification system, an example can almost always be found that does not fit with any category of the classification. Adding an *all others* category does not work simply because the members of such a category are usually too disparate to give the category any real meaning.

2.4.2. Data Objects

The architecture of the physical disk device is usually responsible for the choice of data objects within a disk control system. In most environments, selection of fixed sized data objects for disk transfers is preferred. Using variable sized data objects forces the system to both allocate and release cache memory, since no fixed size of cache buffer can store all sizes of data objects. Constant allocation and release of memory leads to memory fragmentation problems which are thoroughly discussed in [Pet85]. As noted in [Wil91], an interface for controlling variable sized data objects can be easily built upon a system that provides constant sized data objects.

2.4.3. Data Access Patterns

Fortunately, much of the available research regarding *data access pattern* (DAP) categorization has been well summarized by David Kotz [Kot91](pages 14-17). Rather than discussing DAPs in detail, I will mention only those which are relevant to this research. Although Kotz deals with files, his categorizations are applicable to any data objects.

2.4.3.1. Categories of Data Access Patterns

David Kotz breaks accesses down into two primary categories, random and sequential. Each of these categories has additional sub-categories. The final breakdown is shown in Table 1. I provide acronyms to ease later references to particular DAPs. These acronyms should all be listed in the glossary, when you eventually start to reencounter them.

Briefly, *random* access patterns have no regularity, accessing file entries without any correlation between current accesses and previous or future accesses. These random accesses are further broken into two categories where, (*i*) multiple clients access *overlapping* regions

Table 1: Categories of Parallel File Access Patterns

Access Pattern	Acronym
Random Overlapped	RO
Random Disjoint	RD
Sequential Local Overlapped Regular	SLOR
Sequential Local Overlapped Irregular	SLOI
Sequential Local Disjoint Regular	SLDR
Sequential Local Disjoint Irregular	SLDI
Sequential Global Regular	SGR
Sequential Global Irregular	SGI

of the file and (*ii*) clients access *disjoint* file regions, never sharing the same file portions.

Sequential access patterns have some regularity with current accesses being related, in some fashion, to either previous or future accesses. The file regions spanned by these correlated accesses are called *sequential portions* of the file. An access pattern is said to be *locally sequential* when each individual client uses a sequential portion of the file. A contrast to that is *global sequential* access, where some group of cooperating clients use a sequential portion of a file.

As sequential portions of a file may be shared by multiple clients, the locally sequential access patterns can be broken down into overlapping and disjoint patterns.

Finally, sequential accesses may be either *regular* or *irregular*. A regular access occurs when all sequential portions contain a consistent number of file units (blocks, records, etc.). All other access patterns are considered to be irregular.

2.4.3.2. Relevant Patterns

Not all of Kotz's patterns are relevant to the research contained in this thesis. I have concentrated on **RO** and **SGI** DAPs. Some use of the **SLDI** DAP has been made. For more information on any DAPs that I do not elaborate upon, the interested reader is referred to [Kot91].

The two upcoming sub-sections utilize Figures 6 and 7 to describe relevant appropriate DAPs. In these two figures, the disk address space is layed out across the bottom of the figure. In each figure, three processes compete for disk services[10]. What these figures show is an example of each DAP and how three processes could interact with the disk while operating within the parameters of that DAP. If process 0 accesses disk region 0 at time 0 then a $\mathbf{t}_0$ will appear on the line corresponding to process 0 and directly above disk address 0. The figures assume that every disk access will occur at some distinct time interval. This assumption is reasonable since the chances of two disk accesses arriving at exactly the same instant in time is almost nil. Therefore, on each figure, disk accesses are indicated by the time of their arrival using $\mathbf{t}_i$ for $i \geq 0$.

2.4.3.2.1. Sequential Global Irregular

Figure 6 shows a SGI access pattern. The arrival time of each access occurs after all accesses to lower numbered disk regions and before all accesses to higher numbered disk regions. Therefore the access is sequential. The three processes cooperate to access the entire disk address space sequentially, but the pattern of access is irregular, in that one process may read address regions 0, 4, 8, 12, 13, 14.

A more concrete example of a SGI pattern would be the following. A program for approximating solutions for an NP-complete problem uses a breadth-first branch and bound

[10] The choice of three processes is arbitrary and does not imply that the DAP only occurs when three processes are competing for disk services.

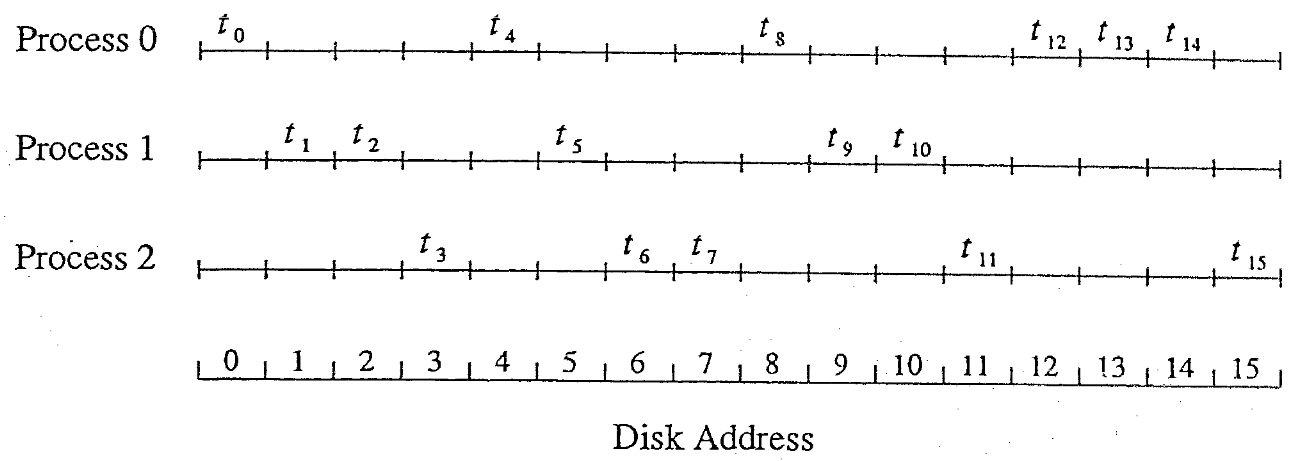


Figure 6: Sequential Global Irregular DAP

algorithm. This algorithm requires storage for long lists of partial solutions to the problem. As these partial solutions may be quite large and cannot all be simultaneously stored in main memory, they must be maintained on disk. Two distinct disk systems are to be used to store partial solutions. One disk system stores a sequential file of partial solutions for level l of the problem, while the other disk system stores the partial solutions for level l+1. Each processor reads a partial solution from the source disk system l, calculates a new partial solution based on that solution, and writes the new partial solution to the target disk system l+1. As each partial solution is 'completed' it is written to disk by whichever processor generated it. The writing of partial solutions exhibits a SGI access pattern, since solutions are sequentially laid down on disk as they are created, but there are no guarantees that generation of partial solutions will take the same length of time for each processor (in other words, over some period of time, one processor may write out twice as many partial solutions as any other processors). Similarly, reading of partial solutions will also display a SGI access pattern on the source disk system.

SGI access patterns are used in this research since I believe that they best represent the use of shared files for the environment in which we are working. Additionally, I believe that SGI patterns represent a best case situation for disk throughput performance tests, as access is always sequential and there is no potential for slow downs due to the process synchronization that would be required in a SGR access pattern.

2.4.3.2.2. Sequential Local Disjoint Irregular

This DAP is exemplary of concurrent access to multiple files. Effectively, it is multiple SGI DAPs, executed in parallel. In other words, there are *n* groups of processes, each group issuing a SGI DAP on one of **n** files. This DAP is not uncommon in disk systems and is presented since it is used as the best case DAP for concatenated multiple disk systems. The reason it is considered optimal for those disk systems is that each file can be placed on an individual disk, assuming $\mathbf{n} \geq \mathbf{D}$, where **D** is the number of disks. In that case, each group of processes issues a SGI DAP to one disk, which is not shared by any other group.

2.4.3.2.3. Random Overlapped

Programs accessing many types of database often exhibit random overlapped DAPs. An example of a RO DAP is shown in Figure 7. As a further illustration, accesses to a very large externally stored array may be randomly distributed throughout that database. Furthermore, there need not be any correlation between the client process issuing the request and the array indices specified within that request. In that event, the database regions accessed by one process may overlap the regions accessed by other processes sharing the database.

The SFU transputer group is associated with research projects concerning image analysis and reconstruction. The use of large databases, such as look-up tables, has been considered by some members of these projects. For this reason, the RO DAP is of

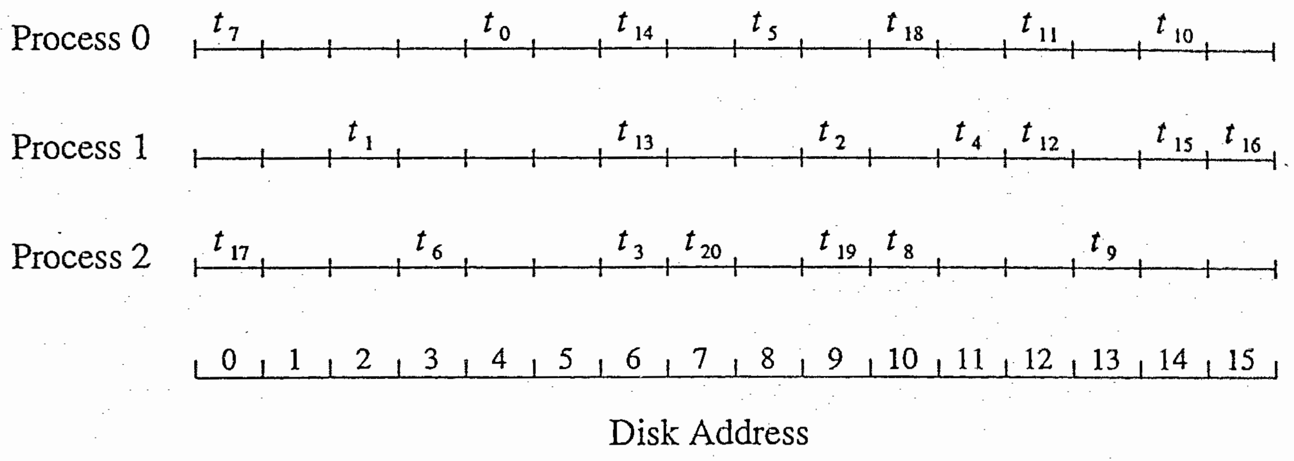


Figure 7: Random Overlapped DAP

considerable interest and will be included in the analysis of TASS performance.

Poor data throughput is usually the penalty for using random access patterns with mass storage devices. Average disk seek times are almost always larger for random access than for sequential access. This is because the disk head usually must travel extensively between subsequent random disk accesses. However, the range of head travel is limited by the size of the database region accessed. These regions can range from one byte up to the entire disk address space. It is expected that average disk seek times will be lower for smaller databases, hence the data throughput to them should be greater.

For the above reasons, random overlapping DAPs will be used to provide approximate lower limits on TASS data throughput.

2.5. Other Systems

This section presents concepts and systems that have influenced the TASS design.

2.5.1. RAID

A SLED of storage capacity **C** can be replaced using a simple disk array composed of **n** smaller inexpensive disks each having storage capacity of **C/n** bytes. The discussion in Section 2.1.2 hints that the cost of this disk array will be significantly cheaper than the cost of the SLED, since the cost per megabyte of storage is less for the smaller disks. Of course this does not reflect the cost involved in the cabling and packaging that would be required for such arrays. I have not seen any concrete estimates on these packaging costs for any particular arrays and the papers I have read give the impression that the costs should not be significant in comparison to the costs of the physical disk devices themselves.

The biggest problem with this simple approach is that multiple disk systems are, on average, less reliable than single disk systems. A valuable benchmark for disk devices is the *mean time to failure* (MTTF) or *mean time to data loss* (MTTDL)[11]. Patterson claims that the MTTF of the simple disk array as described above to be the MTTF of each disk, divided by the number of disks in the array. In other words, the chance of disk failure for an array of **n** disks is 1/**n** the chance of failure for each disk in the array. Given that the MTTF of a Conner-Peripherals CP-3100 disk is 30,000 hours, we can conclude that

> "The MTTF of [an array of] 100 CP-3100 disks is 30,000/100 = 300 hours, or less than 2 weeks. Compared to the 30,000 hour (> 3 years) MTTF of the IBM 3380, this is dismal. If we consider scaling the array to 1,000 disks, then the MTTF is 30 hours or about one day, requiring an adjective worse than dismal. Without fault tolerance, large arrays of inexpensive disks are too unreliable to be useful" [Pat88].

[11] Two terms for the same idea.

Providing fault tolerance in arrays of disks is the primary focus of the RAID papers I have referenced [Gib89],[Pat88],[Sch88]. The basic idea is to utilize additional disks in order to provide reliable storage of data in the presence of individual disk failures. Patterson gives

> "... a taxonomy of five different organizations of disk arrays, beginning with mirrored disks and progressing through a variety of [disk array organization] alternatives with differing performance and reliability. [He] refers to each organization as a RAID *level*" [Pat88].

I do not wish to delve into a discussion of RAID levels as my interest was limited to small groupings of disk devices, due to the hardware environment I was using (this will be further discussed in section 3.2). The only redundancy technique of interest to myself was RAID level 1, which is *Mirrored Disks*.

"Mirrored disks are a traditional approach to improving reliability of magnetic disks" [Pat88]. Basically, a copy of all data is maintained on two distinct but related disks. If one disk fails, all data should still be available from the other disk (assuming that a highly unlikely simultaneous failure of both disks does not occur). The problem with this approach is that the cost of data storage, in dollars per megabyte, is doubled since 2***n** disks are required to store what could otherwise be held by **n** disks. Furthermore, performance of two disks operating in tandem may be worse than the performance of a single disk acting alone.

> "When many arms seek to the same track then rotate to the described sector, the average seek and rotate time will be larger than the average for a single disk, tending toward the worst case times" [Pat88].

Readers interested in higher levels of RAID are referred to [Pat88].

2.5.2. DataMesh

Hewlett Packard's DataMesh project [Wil89],[Wil91] utilizes both disk joining and data redundancy techniques. Additional research avenues encountered in the DataMesh project include that of *SmartData* where data storage elements are coupled with processing power to provide more intelligent approaches to data access than available using the traditional options of read and write access. The goal of this project is to address the I/O bottleneck problem inherent in current disk technology by "... using parallel systems technology to consturct large, fast, highly-functional storage servers for groups of high-powered workstations serving the needs of technical professionals" [Wil91]. These DataMesh devices will behave in a fashion similar to the disk server devices employed in today's ethernet environments. For example, a DataMesh device could be used as a high performance replacement to the standard Sun disk servers found in local area networks (LANs) like that employed at Simon Fraser University.

2.5.3. A Parallel Interleaved File System

In his PhD thesis, Peter Dibble [Dib90] presents a *Parallel Interleaved File System* (PIFS) which is based on a group of 2 or more local file systems. These local file systems are combined, according to the technique layed out in Dibble's thesis, into a PIFS. Although Dibble does not state that each LFS should be interface and performance equivalent it seems apparent that this must be the case (as the system will exhibit behavior no better than that of a PIFS comprised of *n* LFS which are performance equivalent to the worst LFS in the actual system. As an example, if three LFS are used, having storage capacities 100M, 200M and 300M respectively, then the final PIFS will apparently have total storage capacity of 300M and utilize 100M from each file system[12]. This is because parallel files are interleaving over

[12] The PIFS comes with a set of *Tools* which actually allow access to these unused regions of disk. However, discussion of these tools is beyond the scope of this paper. The interested reader is referred to [Dib90].

the LFS' by placing the first record onto $\mathbf{LFS}_1$, the second record on $\mathbf{LFS}_2$, ..., the *n*th record on $\mathbf{LFS}_n$ and the *n*+1th record back on LFS_1, where *n* is the number of LFS in the PIFS A file of 30 records stored in a PIFS composed of 3 component LFS will be placed with 10 records per LFS. By extension, the smallest capacity LFS will eventually fill and prevent storage remaining on all the other LFS from being utilized. Different interleaving techniques could probably be applied to overcome these deficiencies, but this would be at the expense of the scalability of the PIFS, which Dibble states to be a primary goal of his design.

The PIFS provides three distinct interfaces to the user. Each interface provides a trade-off between ease of use and parallelism. A PIFS server provides the first two interfaces, whereas the third is not really an interface, but I refer to it as such. The three interfaces are:

(1) The *standard interface* provides no control of parallelism to the user. Some parallelism may be achieved within the file system due to the interleaving of files over many LFS. Thus, *n* records could be pre-fetched (or stored) in parallel. However, as the user would not have any control over this, this would simply be a sequential pre-fetch cache of size *n*. The standard interface provides eight basic functions, namely **Read, Write, Delete, Current (File) Position, End of File, Create, Open** and **Seek**. The standard interface stores files as *logical* records, rather than using the Unix convention of byte streams. This allows the PIFS to place one logical record on each LFS. The LFS can be modified to place records within sectors thus reducing the probability of sector faults during read (write) access.

(2) The *parallel interface* provides all the services of the standard interface, plus **parallel open, parallel read,** and **parallel write**. These parallel requests are made by a job controller on behalf of all processes in the job (the job controller may or may not be required to perform the duties of a worker processor. I assume it is required to perform these duties, rather than having special controller status). Included with the parallel

open request is a list of handles informing the PIFS how it is to route relevant data to each process within the group. Parallel reads (writes) require that all the target (source) processes within the job be ready to accept (transmit) the next record. This forces synchronization on all processors (assuming one process per processor) during each read (write) phase of the algorithm. This synchronization must be handled by the job controller, as the PIFS does not provide any. Incorrect synchronization may cause the PIFS to overwrite (accept) buffers upon which a processor has not finished operations. Again, it is not clear how Dibble utilizes the list of handles to route data. Is all data routed through the job controller, directly to each process, or through a forest of processors? He states that large amounts of data transfer can saturate the job controller. The parallelism exhibited is only as good as that of the worst processor and because of the synchronization, the multiprocessor is only as effective as the heavily worked job controller processor (multiplied m times where m is the number of processors in the job).

(3) The *tools interface* allows special programs (eg: sorttool) to have full access to the file system's internal parallelism. In other words, tools can access all the LFS directly, without being forced through the PIFS server. Tools must have knowledge regarding the storage (interleaving) scheme of the PIFS in order to access PIFS files. This requires that the storage mechanism be consistent over all files. Tools will access data through the LFS interfaces rather than those of the PIFS. Unlike users of the PIFS parallel interface, all parallelism and synchronization must be handled by the processes within the tool. Clearly, programming of tools will be very complex as compared to utilizing the more standard PIFS interfaces, but a higher degree of parallelism should be achievable. Also, performance differences between LFS could be utilized to improve parallelism and increase storage capacity.

Dibble discusses a PIFS implementation called Bridge. The LFS for Bridge is the Elementary File System (EFS) developed for the Cronus operating system. Bridge is implemented primarily as a single centralized process (the Bridge Server) which provides the following four functions:

(1) The server maintains the Bridge directory (directory of all files stored within the PIFS).

(2) Provides the standard and parallel interfaces.

(3) Provides EFS handles for use by any tools in the PIFS.

(4) Consistency checking and performance monitoring.

User processes and tools communicate with the server via a message passing protocol. Dibble emphasizes the fact that he did not implement message passing with the shared memory available on the Butterfly since he wanted to prove Bridge was extendable to multiprocessors without shared memory, without major performance sacrifices.

The Bridge directory keeps a list of all Bridge files and a list of EFS files comprising each Bridge file, along with which processors (EFS system) that the EFS file resides on.

Since Dibble did not have access to many disk packs, he simulated disks using some of the Butterfly's processors. By the end of all this simulation, he retained approximately 32 processors for actually running user and tool processes on the Bridge system. Consequently, I hesitate to believe his scalibility claims above 32 processors. Dibble's disk approximations are a little naive. He simulates disks by simply enforcing a sleep time between request arrival and servicing. This fails to capture disk timing problems like rotational latency and seek time. In this fashion he effectively prevents his system from dealing with short intervals of worst case behaviour, and assumes average performance response from the disks at all times.

There are many cases of overly optimistic simplifications of the type described above, and even with all this, the PIFS system performance is not outstanding (although the Butterfly upon which he based it can hardly be considered brisk as each processor runs at only 8MHz).

3. Design Overview

This chapter describes the high level design that is the basis for the *Transputer Auxiliary Storage System* (TASS) local file system. Before a detailed design can proceed, it is necessary to be familiar with the available hardware. This is because many decisions in any device oriented software design are direct consequences of hardware functionality. Therefore, I will first present a description of the transputer architecture. That will be followed by a brief description of the vendor supplied boards and some of the vendor software support that comprise the SFU transputer environment.

Any reasonable software design should be based on a rigidly defined hierarchy of software components. For example, the *International Organization for Standardization (ISO)* has developed the hierarchical OSI model for the design and implementation of communication protocols. This model incorporates seven increasingly abstract layers, namely the *Physical, Data Link, Network, Transport, Session, Presentation* and *Application* layers [Sta88](pages 9-14). The formation of a design hierarchy should be prompted by any obvious levels of functionality within the problem to be solved. However, there is always some flexibility in choosing where to produce layer boundaries. The last section of this chapter overviews the layered approach to disk system design that was proposed and implemented within this thesis.

3.1. Hardware Environment

3.1.1. Transputer Architecture

The transputer was briefly overviewed in Section 1.2. The basic architecture of the transputer family is shown in Figure 8. In addition to the features described there, the transputer

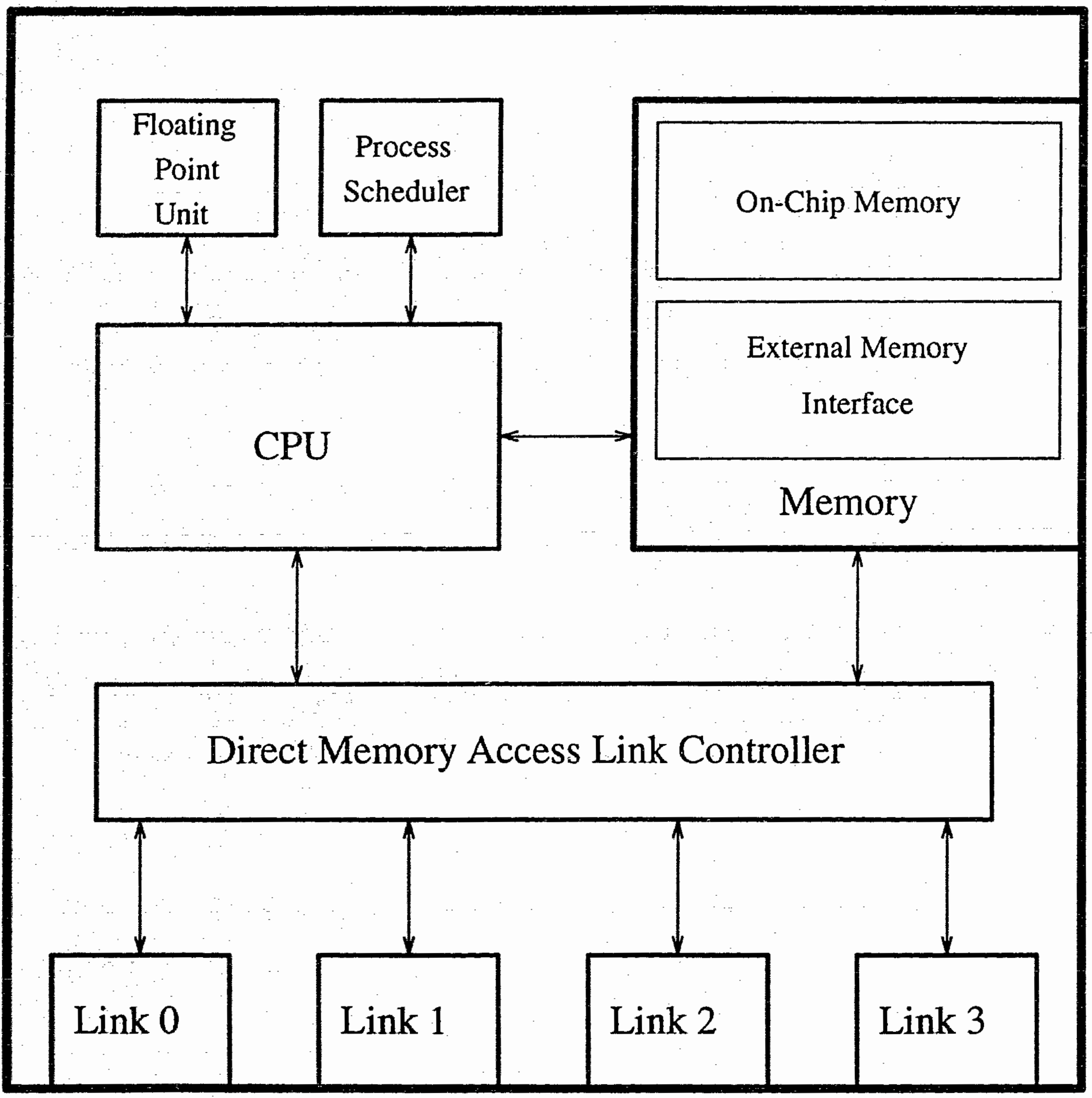


Figure 8: The Transputer Processor Family

provides a two priority level process heirarchy, coupled with a hardware based process scheduler. The scheduler provides rapid process context switches which, according to my tests (see Section 7.2.5) are usually in the neighbourhood of 4 microseconds. This architecture promotes the use of multiple process software environments, often implemented in the form of multi-threaded process groups. As there is no memory management provided by the device, all processes within one transputer share the same memory space.

There are two transputer variants discussed within this thesis, namely the 16 bit T222 and the 32 bit T800. Both provide all the features mentioned above, with the only notable difference being the limited 64 KByte address space of the T222.

3.1.2. Vendor Boards

All of our vendor supplied boards are products of Computer Systems Architects (CSA). To avoid numerous references to the same bibliographical entry, I will simply state that any facts regarding these boards are derived from their user manuals [CSA89], unless otherwise explicitly stated, There are four types of board used within the SFU transputer network, these are the *Part.6 T800 Transputer Board*, the *Part.7 Crossbar Switch*, the *Part.8 VMEbus Interface Board*, and *Part.12 T222 Based SCSI Controller.*

The Part.6 board consists of 4 20 MHz T800 transputers, each equipped with 2 MBytes of on-board RAM. The transputer serial links are all routed to the board edge for easy interconnection with other transputer-based boards.

Each Part.7 is a 16 by 16 crossbar which allows concurrent serial communication between two transputers over each of the sixteen currently connected wires. The crossbar is configurable using three additional serial links, called the *up*, *down* and *control* connections.

The Part.8 provides a connection between a VMEbus and 6 transputer style serial links. Access to the links occurs as memory mapped data transfers between the VMEbus address space and the serial links. Data transfers over the links can proceed both to and from the VMEbus.

The discussion of the Part.12 device will be slightly more detailed than that provided for the other CSA products since aspects of the device driver software design are dictated by the Part.12 architecture. The board consists of a 16 bit Inmos T222 transputer device, 64 KBytes of on-board RAM, and a memory mapped Western Digital 33C93A SCSI Bus Interface Controller (SBIC). "Four KBytes of internal RAM and about 52 KBytes of external RAM are available to the programmer. The rest of the address space is for memory mapping of the SBIC and other devices...". Data is transferred from the SCSI bus to the physical RAM using two 512 byte FIFO buffers. Data transfers between the FIFO buffers and disk, controlled by the SBIC, can overlap with access between main memory and the FIFOs. However, as the memory to FIFO transfers are much faster than the FIFO to disk accesses, there is not a great degree of performance gain due to this. CSA claim that "It is possible to transfer data directly out of a link from the FIFO or in from a link to the FIFO without going into transputer memory, reducing latency slightly at the expense of a slight bandwidth reduction". The device drivers provided with the Part.12 boards made use of this feature, and it's penalties and benefits are discussed later in Section 3.1.4.

Each Part.12 is equipped with a Conner-Peripherals CP-3100 disk drive. The CP-3100 is capable of storing 104 MBytes of data. The dimensions of the device were given in Section 2.1.1, but for the purposes of clarity they are tabulated in Table 2 along with other pertinent parameters.

It is possible to attach multiple disk drives to the SCSI bus of the Part.12 board. Any of these drives could be accessed by the T222 controller, however this would require the implementation of a SCSI bus disconnect/reconnect protocol. I have it on firm authority from my (ex) brother-in-law that this is no trivial matter. No disk server version was implemented for this multiple disk hardware configuration.

Table 2: CP-3100 Specifications

Diameter	3.5 inches
Capacity	104 MBytes
Platters	4
Surfaces	8 (2 per platter)
Tracks/Surface	776
Sectors/Track	33
Blocks/Sector	1
Bytes/Block	512
Bytes/Track	512 * 33 = 16896
Seek Times	
Track to Adjacent Track	8 ms
Average	25 ms
Maximum	45 ms
Revolutions/Second	60
Rotation Time	16.7 ms
Average Rotational Latency	8.4 ms
Mean Time To Failure	30,000 hours

3.1.3. SFU Transputer Environment

The SFU transputer network is connected to a Sun-4/110 host, *draco*, via a single Part.8 VMEbus interface. The environment is pictured in Figure 9. The Part.8 provides a communication link between the network host, *draco* and one T800 transputer, which will be hereinafter refered to as the *root transputer*. The Part.8 additionally provides a communication medium whereby *draco* can configure all of the Part.7 crossbars. The host, *draco*, is also connected to the SFU computing science research ethernet.

The SFU transputer network consists 15 Part.6 boards, providing a total of 60 T800 transputers, and 4 Part.12 boards, constituting 4 T222 transputers, each of which is augmented by a CP-3100 disk drive. All 64 transputers are interconnected using 8 Part.7 crossbar boards, which are in turn configured using a single Part.8 VMEbus interface. The size and number of crossbar switches used does not allow all the transputers to be interconnected in all possible ways, however, the implementation of the network provides a sufficient

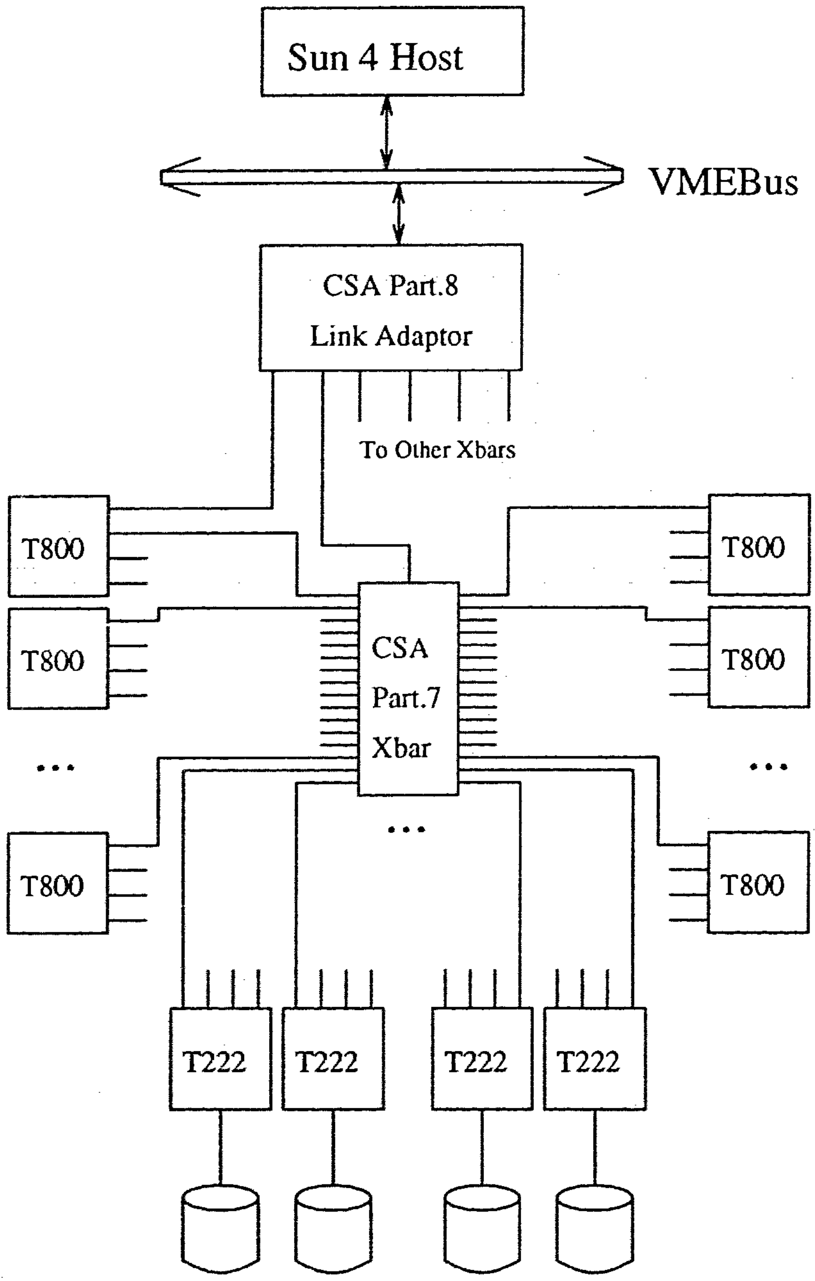


Figure 9: The SFU Transputer Network

subset of the possible interconnections to serve most research needs.

Software to be executed on the component transputers can be compiled on any Sun host within the ethernet. However, the object code can only be downloaded to the transputer network using *draco*. The software is downloaded to the root transputer, which acts as a conduit for downloading software to all other transputers within the network. The crossbars are configured at the time the network is booted and currently there is no ability to reconfigure the network during program execution. The transputer network constitutes a *multiple instruction, multiple data* (MIMD) system, so the code downloaded to one transputer need not be the same as the code downloaded to other transputers in the same network.

The purchase of the SFU transputer system occurred in incremental stages. As a consequence of these incremental enhancements, the actual Part.12 disk controllers (and their related disks) were purchased without firmly establishing their suitability to the SFU transputer group. It seemed likely that the hardware itself would be sufficient to provide data storage services within the transputer net, however, the effectiveness of the vendor supplied software was not known.

3.1.4. Vendor Supplied Software

All support software, for example, network loaders and debuggers, are CSA products. All software within the TASS system was implemented using the *Logical Systems C* compiler [CSA90]. Most of TASS's SCSI disk driver was translated to C from a vendor supplied Modula-2 implementation.

The support software and compiler were quite adequate for the purposes of TASS development. This section deals primarily with the vendor supplied Modula-2 SCSI driver, as it was determined, in the opinion of the SFU transputer users, to be inappropriate for our purposes. The reasons for this decision were:

- Software development on the SFU transputer network has predominantly been performed using C. Simultaneously booting a transputer network with code elements, some of which were compiled from C and some compiled from Modula-2, was not possible with the first release of the CSA Modula-2 system. Eventually, CSA rectified this problem, but using code produced from both languages was still more difficult than using code produced by just a single compiler.

- The operation of these CSA disk systems was severely limited in the fact that only one Part.12 disk controller transputer (called a *disk node* for now) could operate within any transputer network. In addition to that, it was required that the disk node be connected to the root transputer only. Therefore, at least one of the root transputer's links had to be dedicated to communication with the Part.12 board.

- The CSA Modula-2 disk system was targeted at transputer networks with MS-DOS configured IBM AT hosts. Because of that, the CSA disk system operated as an MS-DOS '*clone*', with all operations to/from disk having to be executed in that directory and file environment. This MS-DOS implementation was distributed between the Part.12's T222 and the T800 root transputer, with the majority of computational overhead foisted onto the T800. Any software requiring disk access that was to be executed on the T800 root transputer was required to link in this disk software, and at first this required the root transputer code to be compiled from Modula-2, and loaded using the Modula-2 system.

- In order to gain the benefits of reduced latency (mentioned earlier in section 3.1.2), CSA had implemented the T222 software as a minimalistic driver, acting as a router for disk block read/write/status requests from the T800 to disk and as a dumb controller for data transfers between transputer serial links and the Part.12 FIFO hardware buffers. Because of this, there was no disk abstraction at the T800 level of the implementation, therefore detailed knowledge of the particular disk used, and how to address it, was required at this higher level of the system.

- There was no attempt to allow parallel accesses to the disk system. No protection was provided for multiple processes accessing the disk support routines. Part of this problem was that the disk access primitives involved a complicated protocol between the root transputer and the T222 on the Part.12 board. Accesses had to be purely sequential, with every access having to complete, in it's entirety, before any other access could begin. Not even simple semaphore protection was provided by these data access procedures.

- It is a common technique to split device drivers into two halves, the lower one being device dependent and the upper half being device independent [Com84](pages 283-308). This basic software engineering technique had not been utilized in the CSA server, rather it had been hard-coded for handling CP-3100 drives.

It was our (the SFU transputer users) opinion that these points made the CSA disk system unusable for our computing requirements. We were hoping to use multiple disks within our transputer network. We wanted the flexibility to place those disks where they would be most useful, within the network configuration dictated by each specific application. We did not particularly wish to use an MS-DOS directory hierarchy, nor suffer the penalty of computational and data transfer overhead imposed on a system by such a rigid file environment. It was further noted that the implementation of the CSA system utilized no buffering of disk access requests and that it prevented any simple modification that would incorporate buffering. These issues prompted the first rewrite of the CSA disk system, which eventually led to the development of TASS.

3.2. TASS Design

As mentioned in Chapter 1, the goal of the TASS design is to provide a simple efficient interface between mass storage and the transputer network. This mass storage has been implemented as a rudimentary local file system (LFS) under the control of a group of transputers, cumulatively known as a *TASS Storage Node*. This storage node is based on the client/server model which was utilized extensively in the V system. In this model, disk devices are resources to be managed by *server* processes. "When a user or a process wants to use a resource controlled by such a server, it sends a request, thus becoming a *client* of that server" [Ber86]. Since the transputer architecture currently lacks point to point communication, the TASS storage node is only accessible by client processors that are directly connected to the storage node.

TASS reflects the fact that the transputer multiprocessor is a high performance computation platform. McVoy & Kleiman point out that some database users, in the pursuit of performance, actually get rid of the file system altogether. These sort of programs are implemented to access the raw disk itself. They also point point out that "This solution is an act of desperation. There is no file system, no file abstraction, etc." [McV91]. The overhead imposed by a high performance computing platform's file system should not be so great as to drive the platform's users to those frantic extremes. Therefore, the secondary intentions of TASS are to provide the highest possible mass storage performance. As such, the TASS design stresses performance over functionality. The performance parameters of interest are (*i*) high data throughput and (*ii*) low client processor and communication overhead.

To achieve high data throughput, we decided to implement TASS according to the following:

- Disk transfer size is chosen to maximize disk throughput. This indicates that all data transfers to and from disk are performed in fixed size units of transfer. Furthermore, the disk transfer data object is fairly large, in order to distribute the computation and communication overhead associated with each disk access across as many transferred bytes as possible. In other words, we try to minimize the amount of overhead per byte of data transferred. However, the system is not hard-coded for the size of the data object, rather it is designed to allow disk transfer object size to be varied.

- TASS includes a cache, allowing programs that issue temporally and spatially local data access patterns, to satisfy some of their data requirements using fast memory (the cache) rather than through accessing much slower disk devices.
- The system provides for parallelizing multiple disk devices within any single TASS storage node. Parallelization can occur using the concatenation, interleaving or striping techniques described in Section 2.1.2.1. The number of disks and the nature of the joining within each TASS node is easily selected at compile time. Also, if one TASS node utilizes one component disk, this does not prevent other TASS nodes within the same transputer network from being configured as multiple disk nodes.
- The system provides clients with the ability to prefetch into the cache any disk regions that they are likely to need in the near future. This allows clients with predictable access patterns to instruct the file system regarding what disk regions should be fetched into the cache.
- Buffering within the disk controller(s) is provided to more evenly parallelize the computation and physical I/O components of all disk accesses.
- The user of the system retains a high degree of control over system configuration. Many important performance issues are configurable at either run-time or compile-time, thus providing the user with the ability to select for system features that they require, without forcing users to be penalized due to the overhead imposed by redundant or superfluous system functions. In other words, the user retains a high level of control over what functions their

LFS provides.

To achieve low client processor overhead, the TASS client interface is designed with the following considerations:

- The object code size for linked LFS libraries is kept minimal. Programs utilizing the disk system are not forced to allocate large portions of their local memory, in order to access the large disk memory.
- The number of client processor cycles required to execute a single disk access is minimized. Additionally, the number of context switches required due to processes blocking during execution of the LFS library has been kept to a minimum.
- The system is implemented so that users having less stringent performance requirements can utilize more functional interfaces, without requiring alternate disk system control software.
- The system can adapt to alternative file systems, without requiring major overhauls in the lower layers of the code. The concept is similar to that of a plug-in feature, if the user desires a Unix file system (and someone has written one), then they are able to plug that file system in at the appropriate level of the TASS code and execute their programs using that interface. In other words, implementing an alternative file system, perhaps modelled on Unix or MS-DOS, does not require that the disk drivers and cache control be altered to support that file system.

The TASS design is broken down into both hardware and software components. The nature of the hardware configuration upon which TASS is built is central to the design of the software itself.

3.2.1. Hardware Configuration for TASS Storage Nodes

Since TASS is designed as a disk control system, it is expected that data transfers between memory and disk will be rather large. If possible, the transfer should be in the form of an entire track in order to maximize the amount of data transferred between disk seeks and rotational delays. The CP-3100 specifications in Table 2 provide a ballpark figure for the expected size of disk transfers, namely one track of 16896 bytes, which is a little over 16 KBytes. One primary drawback of the available hardware is the limited 64 KByte address space of the Part.12's T222 transputer. Implementation of a cache within such a small address space was not considered to be a very profitable adventure, since the size of the cache could be as little as three buffers. Implementation of a useful cache required a larger main memory, indicating that an additional transputer, of the 32 bit T800 variety, be utilized as a cache controller. Fortunately, the 64 KByte address space of the T222 is sufficient for double or even triple buffering of I/O requests, allowing the T222 transputer to be used as a disk controller. This reasoning led to the first concept of the TASS storage node, pictured in Figure 10 (*i*). The TASS storage node is configured as three hierarchical layers of processors,

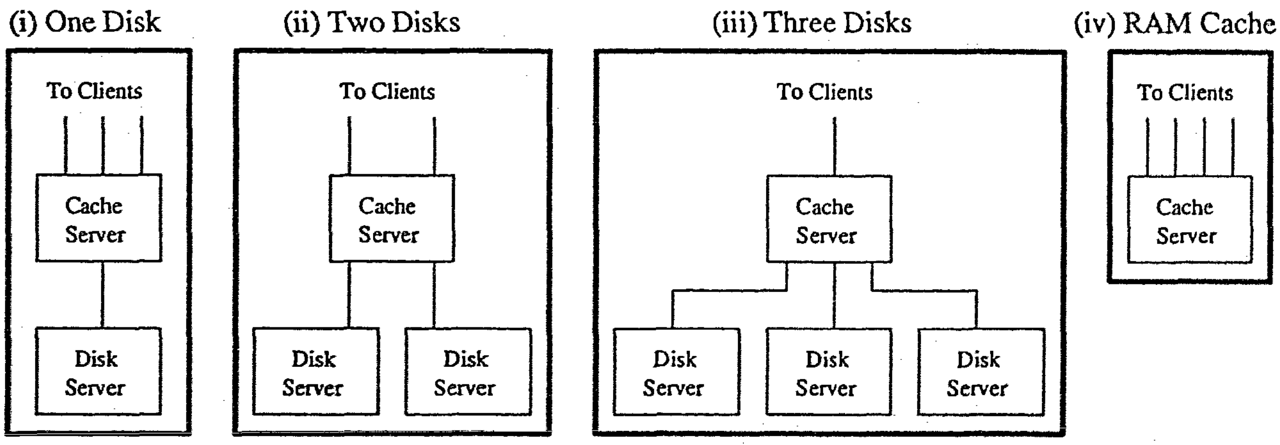


Figure 10: TASS Storage Node Configurations

namely, from highest to lowest, the *client processor(s)*, *cache server* and *disk server(s)*. Note that one serial link of the cache server is dedicated to communicating with the subservient T222 disk server. The three remaining serial links are available for communicating with client processors.

Using an extra T800 transputer for a cache controller is what prompted my interest in disk joining techniques. If a T800 were to be used as a cache controller connected to a disk controller, then why not parallelize disk access using concatenation, interleaving or striping? Alternatively, a more robust disk system could be built using mirrored disks. The fact that the T800 has only four serial links limited this configuration to a maximum of three subservient disks[13]. However, even using two parallel disks could greatly increase disk throughput for each storage node. Figures 10 (*ii*) and (*iii*) show these alternative configurations. Using a single T800 transputer, without subservient disks, could service as a RAM disk (or RAM cache). This configuration is shown in Figure 10 (*iv*). Hardware configurations similar to that of TASS are discussed as building blocks for Hewlett-Packard's DataMesh storage server architecture [Wil89],[Wil91].

Communication between neighbouring layers of processors is via the transputers serial links. This environment requires that software protocols are adhered to by neighbouring processor layers. However, "Most programmers do not understand message communication and prefer a simple procedural interface to services and facilities" [Che84]. Like the V system described in [Che84],[Ber86],[Che88], TASS provides a *remote procedure call* (RPC) abstraction to hide the communication protocols used over the interprocessor serial

[13] One serial link is required to communicate with the outside world. A disk system is not of much use if it is detached from all computers.

connections. An additional advantage of RPC implementations is that programmers utilizing an abstract system probably have enough problems of their own, without having to worry about whether or not they are correctly communicating with servers.

3.2.2. Software Design

The TASS software design is based on the premise that there are at least four primary layers within any local file system (LFS). Regardless of hardware architecture or device type, there must always be (*i*) a *client interface layer* (CIL), (*ii*) a *file system layer* (FSL), (*iii*) a *cache layer* (CL) and (*iv*) a *disk support layer* (DSL). This exact breakdown of LFS primary layers is also presented by Maurice Bach [Bac86](pages 19-21). The four primary layers are distributed across the component processors in the fashion described in Figure 11. The reasons for this distribution are detailed in Chapters 4 through 6. These chapters approach the design from the bottom up, beginning first with the disk support layer, and progressing on up through to the client interface layer. I feel that this is a more natural progression, since the design itself progressed in this fashion.

Each of the physical communication points (ie: between the client processor(s) and the cache server and between the cache server and the disk server(s)), will require splitting of the layer spanning those communication links into secondary layers. The CIL will be broken down into a client processor based *client communication layer* and a cache server based *server message layer*. The DSL will be broken down into a cache server based *client communication layer* and a disk server based *server message layer*.

To provide a concise division of layer boundaries, these layers are abstracted into a procedure call interface, either remote or local. This thesis proposes a technique for distributing these layers across multiple processor elements within a loosely coupled multiprocessor. The distribution technique is discussed in Sections 3.2.2.1 to 3.2.2.4

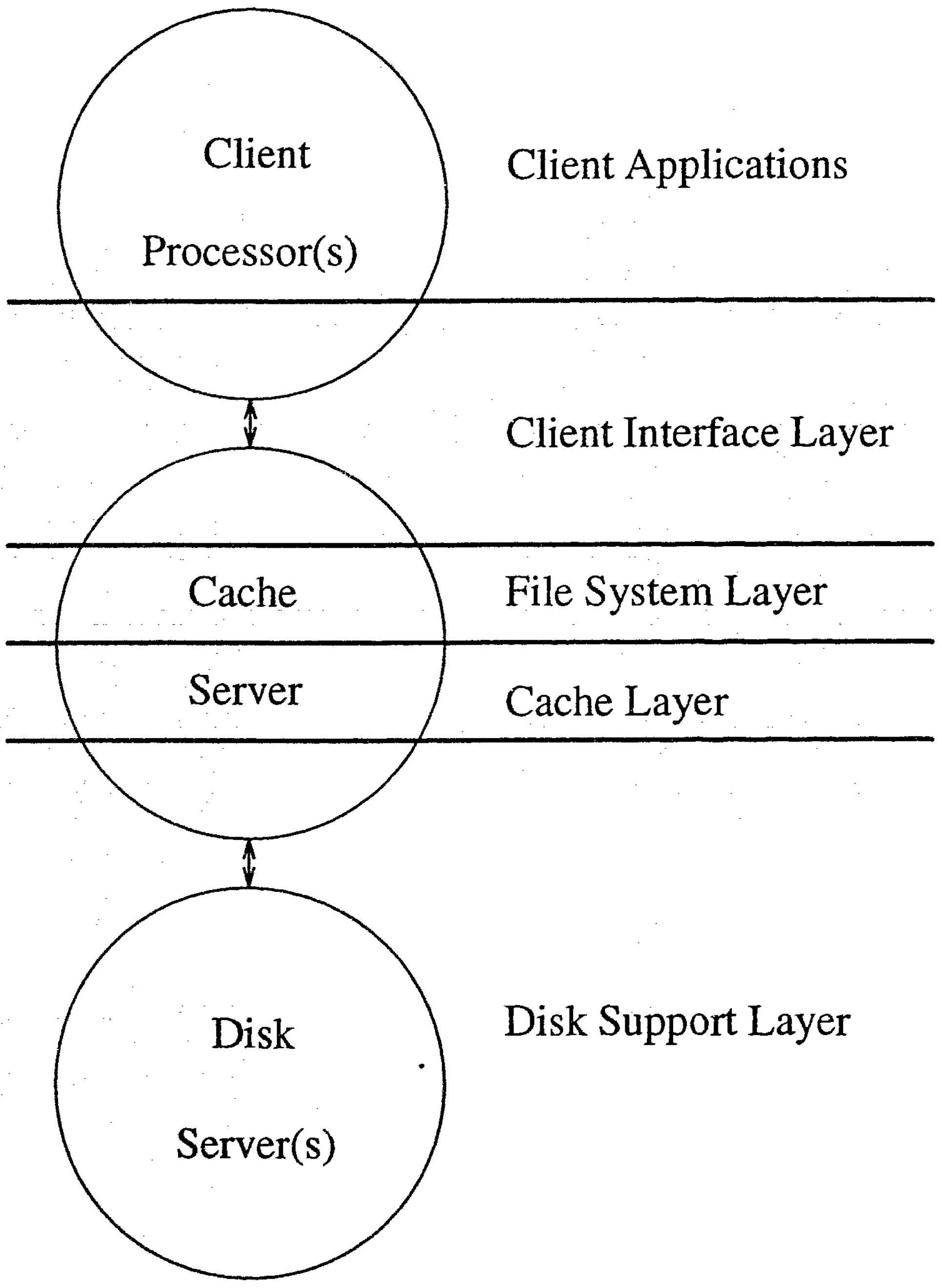


Figure 11: Distribution of Primary Layers over Component Processors

3.2.2.1. Client Interface Layer

The highest level of any LFS has the responsibility of communicating with client processes and/or processors. This is the *client interface layer*. The client interface should provide an efficient set of primitives which will allow file access to be effective and easily understood. These primitives should allow the client to utilize all file system features necessary for any application, without requiring undue knowledge of the file system operation nor of the communication involved in the interaction with the file servers. For example, client knowledge of the communication protocol used between the client interface layer and the file system should not be a prerequisite to using that file system. An effective client interface should provide high performance access while hiding the ugliness of implementation details. It should also provide a complete set of file system operations, without restricting the effectiveness of those operations, nor affecting the consistency of them.

In a distributed environment, this layer is responsible for relaying data between clients and the file system. It usually involves two components, which David Cheriton refers to as the *client communication interface* (CCI) and the *server message interface* (SMI) [Che84]. In a non-distributed environment, there is no need for distribution so the client interface layer becomes a null entity, effectively mapping directly onto the file system layer.

3.2.2.2. File System Layer

There are core operations that must be supported by any and all file systems, or the system will not function reasonably. Any file system must allow the reading and writing of data to/from the mass storage medium. Additionally, these read/write operations must behave in such a way that a client can control which data items they will operate on. For example, if a client writes ten files to disk, the client should be able to read back the fifth file and not have to worry that the disk system will provide the wrong file, or a jumbled combination of the ten files. Additionally, the system must maintain data integrity. In other words, data written by a client has to be read back without any alterations to the content of that data.

As mentioned earlier, the first implementation of TASS was not intended to provide a comprehensive and complete file system. It is intended to perform as a high performance local file system, only capable of distributing data to clients which are physically connected to the TASS LFS. As such, a complex interface was not required. We chose to provide what is herein called a *Chunk I/O* disk interface.

Definition: A *chunk* is a constant sized region of the disk address space of any TASS local file system. A chunk is roughly analogous to a disk track or sector and in fact the current TASS prototype uses a 1:1 mapping between chunks and disk tracks[14].

The TASS chunk I/O interface slices the disk address space into an array of chunks. Each chunk is a distinct entity within the mass storage address space, therefore chunks can be operated on individually, but not in groups nor in any manner that crosses chunk boundaries. The interface provides the core operations defined in Table 3. When reading about the core operations, it is relevant to consider the array of chunks abstraction. To utilize an array, it is necessary to be able to read from it, write to it and also to know the dimensions of the array, and of its elements.

This file system will be described in greater detail in Chapter 5. The selection of the chunk I/O interface does not prevent TASS from supporting an alternative file system. Rather it is intended to provide sufficient services in order to make TASS a useful support system. Implementation of a simple file system, in addition to the chunk I/O interface, was

[14] The term chunk was used, as opposed to words like page, track, sector, line, etc., in order to avoid confusing the chunk concept with the preconceived notions usually associated with those words. Other authors have used the term, notably John Wilkes [Wil91], to denote similar structures.

Table 3: File System Operations

Read Chunk	Copy a single chunk from the address space of the TASS LFS into the address space of the client processor.
Write Chunk	Copy a single chunk from the client processor's address space into the address space of the TASS LFS.
Prefetch Chunk	Copy a single chunk from the physical storage device within the TASS LFS into the cache storage space of the TASS LFS. The data is not copied to the client processor requesting the prefetch.
Flush Cache	Force all chunks resident in the TASS LFS cache to be written back to the physical storage devices within the TASS LFS.
Chunk Size	Obtain the size of the chunk. The chunk size is dictated by the physical disk device and disk joining technique used. This information is required by programs that need to allocate local storage buffers.
Number of Chunks	This primitive is required to inform client programs of the amount of data storage available on the storage node. Again, this information is dictated by the physical disk device, and must be made available.

considered, but these plans were tabled due to time constraints.

A second object type, the *TASS file* is occasionally referred to within this thesis. The term *TASS file* is used to reference a contiguous section of disk chunks. Within this context, files are neither defined nor maintained by the file system. The concept of a file is used only to provide a higher level of disk organization than would be available using the disk chunk abstraction.

> **Definition:** A *TASS file* of size **k** chunks, is a contiguous region of disk chunks beginning at disk chunk **i**, and spanning every chunk through to disk chunk **i+k**-1, inclusively.

To minimize disk seek times, many traditional and real-time systems attempt to allocate or preallocate files in this fashion [McV91],[Wil91]. Therefore, I believe this definition will provide reasonable approximation of real-time disk access activity.

3.2.2.3. Cache Layer

The cache layer is an autonomous module that converts the procedure call interface of the disk layer into a cache level interface for use by the file system layer. This cache level interface provides access to the cached data objects, without imparting special status to particular regions. A file system may be forced to allocate regions of disk for maintaining *file access tables* and *directory listings* but the appearance of these items to a non-partitioned cache should not be any different than the appearance of more mundane file data. Some systems incorporate caching directly into the file system, with file system style operations executing both above and below the actual cache. I believe choosing such a style of cache constitutes a poor design decision because

- In an intermixed environment, modification of the file system may affect cache operation, perhaps degrading performance or accuracy.
- Caching of data is a distinct concept. Caching responds to data access patterns and the operation of a file system on a disk is simply a complex DAP. The cache should be able to deal with these DAPs without specific information from the file system manager.

The TASS cache operations provide access to data chunks while hiding details such as:

- Whether the data is in the cache or on disk.
- On a cache miss, whether a free cache buffer is available to read (or write) the requested chunk into, or if a write back of a dirty buffer must first be performed.
- The cache abstraction hides whether or not the entry is currently being written to, or read from, and assures that two data accesses do not simultaneously access the entry in a fashion that would make the results of one or both operations invalid.

- The cache keeps track of which cache buffers are in use, in order to avoid that buffer being targeted for replacement prior to the cache client(s) finishing with the data stored therein.
- The cache locks entries in order to make the two above guarantees, and the file system should not have to be concerned with the protocol involved in these locks.

Briefly, the operations provided by the cache layer interface are shown in Table 4. The need for both lock and release primitives is to avoid internal copying of data. Memory copying is a time consuming process and transferring data between one buffer and another, within a single processor, is often a useless activity. Using two procedures, one to lock and one to release, allows the client processes to operate directly on the cache buffer and it avoids unnecessary copies and the coherency problems that occur from using them. Some of these operations operate simply as a 'relay' between the disk support layer and the file system layer.

Admittedly, many marketable disk systems do not incorporate a cache layer. For example, the Microsoft MS-DOS environment, prior to version 5.0, did not incorporate caching of any data read from or written to storage devices[15]. The disk support is provided in the BIOS layer and the client interface is provided within the DOS layer. The newest release of MS-DOS, namely version 5.0, has (finally) incorporated caching for storage devices, although Microsoft has been a little tight lipped about how this has been implemented. Designers of systems like MS-DOS which will not contain a cache layer should tailor their disk support interface to allow the easy and transparent insertion of a cache, when and if it is ever

[15] Actually, MS-DOS is more than a file system. In fact it provides other hardware oriented services that do not fit within a file system. However, there does exist (contrary to some rumours) an effective LFS within the MS-DOS environment.

Table 4: Cache Layer Interface

Get Read Lock on Chunk	This operation locks the cache buffer (that stores the target data item) and prevents concurrent accesses from writing to that buffer until the read lock is released. Simultaneous read operations are allowed since they do not modify the actual data stored in the buffer.
Release Read Lock on Chunk	This operation informs the cache that the read operation on the specified buffer has completed.
Get Write Lock on Chunk	This operation locks the cache buffer (that stores the target data item) and prevents concurrent accesses from reading from or writing to that buffer until the write lock is released. Simultaneous read operations are not allowed since they may copy the buffer's contents before the current write operation has completed, therefore reading an invalid version of the data.
Release Write Lock on Chunk	This operation informs the cache that the write operation on the cache buffer has completed.
Write Back Chunk	Allows the user of the cache to specify that a specific data chunk be returned to disk. This allows the file system to guarantee that data has been stored on disk.
Prefetch Chunk	Bring the indicated data chunk into the cache, pending later use.
Flush Cache	This operation returns all cache entries to disk.
Chunk Size	The size of data chunks is relevant to the file system layer, therefore this information must be made available as part of the cache layer abstraction.
Number of Chunks	The number of chunks available on disk defines the size of the data store. The file system will require this information and so it must be part of the cache layer interface.

required.

3.2.2.4. Disk Support Layer

The lowest level of any LFS is the *disk support layer.* This layer should provide a set of primitives allowing read (write) operations from (to) any region of the device's address space, regardless of the type of device (or number of devices) present. These core primitives must provide only essential services and should not incorporate any of the functionality that normally would be provided within a file system. In other words, the device support layer should be independent of the type of file system within which it will be used. Additionally, services that may help to enhance the performance of higher software layers should be provided, as long as they do not detract from the performance of the disk support layer itself and do not violate the independence criterion. Since the performance bottleneck in a LFS is usually the storage devices, streamlining the performance of this layer is paramount to an efficient LFS.

As with higher layers, a procedure call interface to the disk support layer should be used. Forcing client processes, namely those operating in the cache layer or higher, to handle communication protocols in order to utilize the disk access primitives can often lead to confusing and inaccurate code. With the selection of a procedure call interface, the required primitives can be selected. For TASS, the core primitives for the disk support layer are given in Table 5. These are not the only primitives that are provided by the layer, however those not listed are simply for accessing statistics accumulated during execution of the TASS storage node .

Table 5: Disk Support Layer Core Primitives

Read Chunk	Copy a single chunk from the physical disk into the buffer provided by the client process.
Write Chunk	Copy a single chunk from the buffer provided by the client process to the physical disk.
Chunk Size	Obtain the size of the chunk. The chunk size is dictated by the physical disk device and disk joining technique used. This information will be needed by the cache layer, in order to allocate cache buffers and control structures.
Number of Chunks	This primitive is required by the cache layer during allocation of control structures. As this is disk dependent information, it must be abstracted in this fashion to allow a device independent cache layer implementation.

4. The Disk Server

As mentioned in Chapter 3, all the software generated for the TASS disk server was written using Logical Systems C (version 89.1). Initially, the lowest levels of this driver were translated to C from the CSA Modula-2 disk driver product. The actual module translated was SCSIDisk.MOD. I have since modified the C version extensively. The disk server is built upon the CSA Part.12 board described in Section 3.1.2, but wherever possible, the code has been implemented in a device independent fashion.

The software comprising the *Disk Support Layer* (DSL) operates within at least two distinct processors, namely the disk server(s) and the cache server. There are five primary components of the DSL. Two of these components reside in the cache server processor, namely (*i*) the *disk services layer* (DSL) and (*ii*) the *client communication layer* (CCL). The remaining three components operate within the disk server processor(s) and are (*iii*) the *server message layer* (SML), (*iv*) the *device independent layer* (DIL) and (*v*) the *device dependent layer* (DDL). These components, and their intercommunication, are depicted in Figure 12. This chapter discusses the design of the three levels which are resident within the disk server (DIL, DDL, SML), postponing discussion of the remaining two levels to the cache server discussion in Chapter 5.

The actual hierarchy of the disk server based layers may be a point of consternation. If the SML communicates with the cache server based CCL, why is it lower in the hierarchy than the DIL? The reason for this is that the SML and CCL are hardware dependent layers since they deal with a physical communication medium, and the protocol used to communicate across it. Effectively, the SML and CCL are device dependent layers and therefore should be abstracted beneath the device independent layer. David Cheriton's TimeServer code example in [Che84](page 25) justifies this practice. The best way to consider the

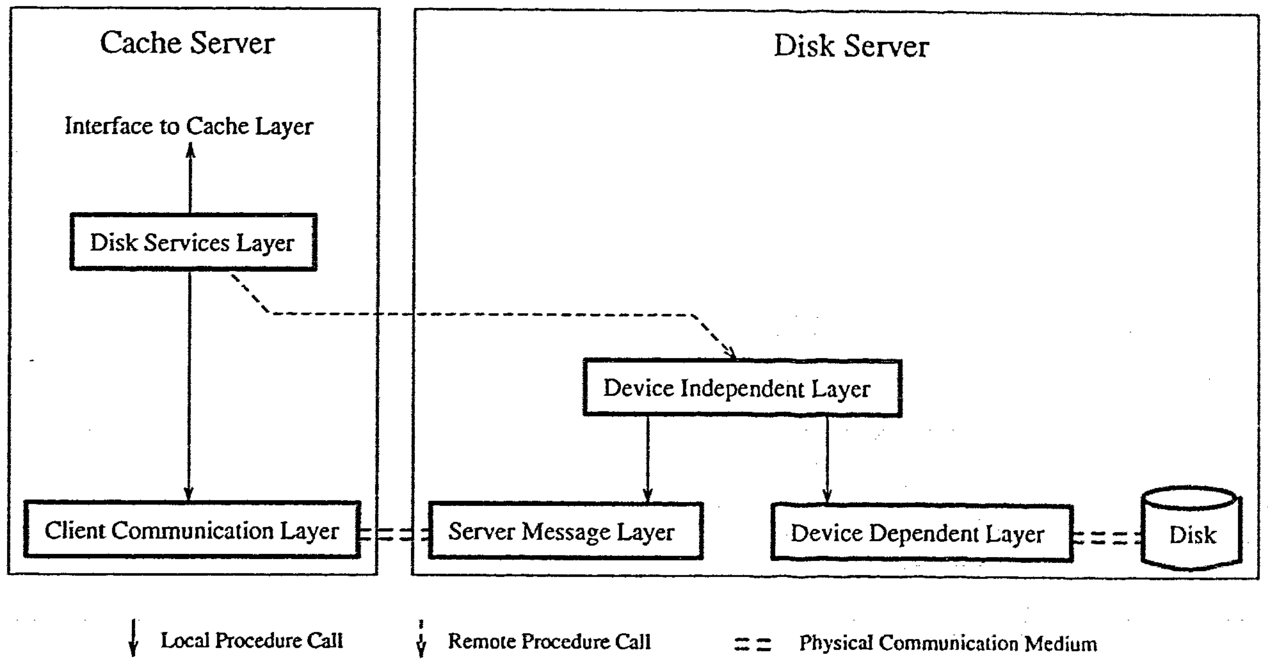


Figure 12: Components of The Disk Support Layer

hierarchy is to picture the DIL communicating with the DSL, each using the SML and CCL as subservient tools to achieve that communication.

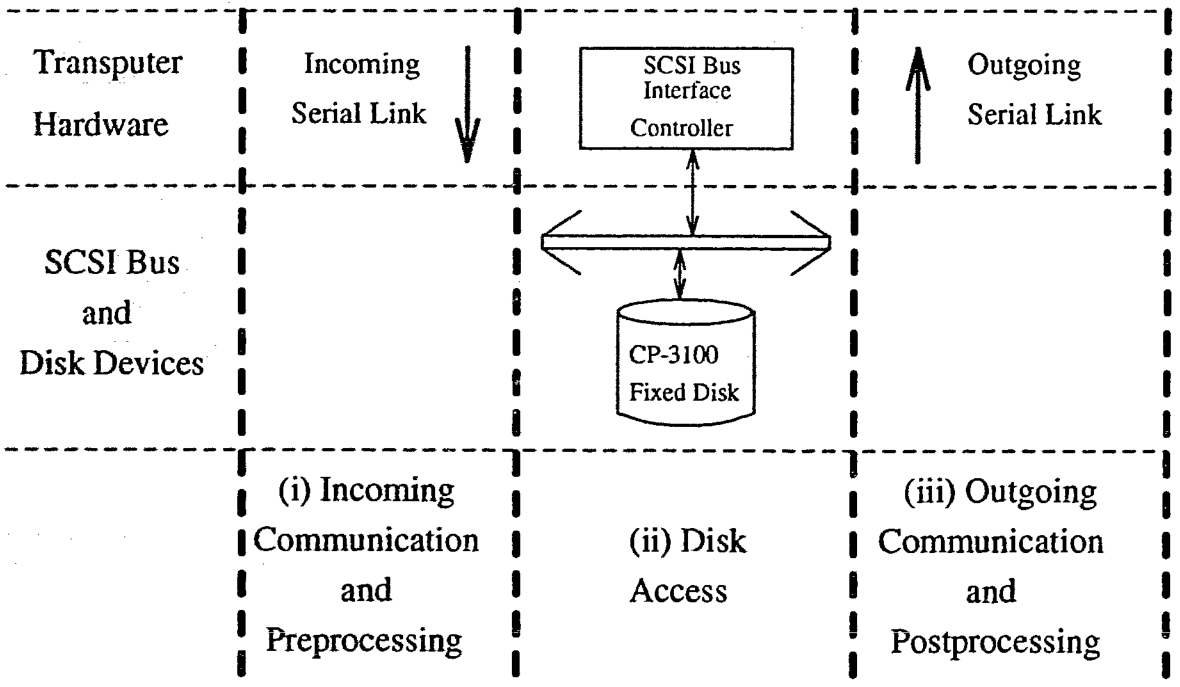
The operation of the disk server processor bears some consideration. Within the architectural constraints of a loosely coupled multiprocessor, any processing element controlling a disk device must perform the following three basic activities during the execution of a single disk request:

(1) Accept an incoming disk request, in the form of a message from a client. If the new request is a write operation, copy the data from the communication medium into a local memory buffer.

(2) Execute the indicated disk request, copying data between the local memory buffer and the physical disk in the appropriate direction (either read or write).

(3) Respond to the client, returning them an error status (informing them that the request succeeded or failed) and transferring to them, via the communication medium, any data read.

These three operational regions form a disk access pipeline, which is shown in Figure 13. Any process operating within some region of the pipeline must control specific hardware resources in order to operate. The figure indicates that these resources are, for the transputer processor, (*i*) the incoming communication link, (*ii*) the SBIC (and subservient disk device) and (*iii*) the outgoing communication link, for the three regions respectively. Furthermore, the operation of a process (and its acquired hardware) within any region of the pipeline, is independent of the hardware devices in other regions of the pipe. This allows for multiprocessing within the disk server transputer, with three or more server threads being able to concurrently operate on some phase of the pipeline.



Transputer Hardware
Incoming Serial Link
SCSI Bus Interface Controller
Outgoing Serial Link
SCSI Bus and Disk Devices
CP-3100 Fixed Disk
(i) Incoming Communication and Preprocessing
(ii) Disk Access
(iii) Outgoing Communication and Postprocessing

Figure 13: Disk Access Pipeline

One of the perceived drawbacks of the vendor supplied Modula-2 implementation was that no local memory buffers were used, hence there existed an overlap of required hardware resources within each region of the pipeline. This is because data transfer to (from) disk occurred concurrently with data transfer in from (out to) the communication medium. This single fact prevented the use of multiple processes within the disk server processor, prompting change to the driver code. In the Modula-2 driver, only one process was handling all aspects of server operation. This prevented the disk server from concurrently executing regions of the pipeline and subsequently reduced the data throughput of the disk server as a whole. The only solution to this problem was to implement an alternate driver that took advantage of the natural concurrency of the problem. The decision to re-engineer the disk driver software opened the door for the implementation of a multiple process disk server environment.

Section 4.1 describes some of the primary considerations in the design of the disk server code component. Target interfaces between software layers are defined and the communication protocol utilized between the disk server and the cache server processors is detailed. Sections 4.2 and 4.3 describe the implementation of two TASS disk server prototypes. The two prototypes are referred to as *Disk Server 1* (DS1) and *Disk Server 2* (DS2). Both prototypes are functionally equal, providing the same basic services and operating on fixed sized data chunks. The modularity and overall software quality of DS1 was not up to personal standards, mostly due to some poor initial design decisions. Additionally, DS1 did not provide acceptable performance, and for these reasons it was scrapped. Prototype DS2 provides significantly better disk transfer rates than DS1 and as such has been selected for use within TASS. Section 4.4 discusses the implementation of two RAM disk server prototypes.

4.1. Considerations in the Design

Each layer within a hierarchical design presents an interface to higher layers of the system. For the disk server design, both the device dependent and device independent layers provide a procedure call interface. The server message layer must provide a conversion from the services of the device independent layer into a communication protocol that can be used across the communication medium connecting the disk server to its clients. This section describes the interface or protocol chosen for each of the three software layers.

4.1.1. Device Dependent Interface

The device dependent layer is responsible for controlling the physical disk device. Contrary to its name, this layer must provide a set of device *independent* services that will give the user sufficient control of the device, without unnecessarily encumbering them with device control considerations. This is a trade-off, and as such there is never a 'right' solution. For the purposes of TASS, and the approach that I knew would be taken at higher levels of the design, a sufficient set of primitives would provide for device initialization, device formatting, allow reading and writing of fixed size data chunks, as well as providing certain pieces of device dependent information in an abstract way (for example, number of chunks, size of chunk, etc.)[16]. In addition, the clients of the layer would need their own distinct workspaces within the layer (to maintain information like which buffer they were using, etc.), so a client

[16] I do not claim that this interface is sufficient for all possible disk server environments, however, I gave considerable thought to its composition and I am still satisfied with it. More functionality would probably be required for a more robust interface, but time constraints limited the interface to providing those functions that were necessary for the TASS system itself.

registration primitive was included. The primitive set selected is given in Table 6. The two procedures GetBuffer() and RegisterClientThread() are dependent upon the amount of available memory, which is a board dependent issue. All other procedures, convert the intent of

Table 6: Interface to the Device Dependent Layer

Format	Parameters:	(*i*) Track interleave factor (*ii*) Sector interleave factor
	Results:	Re-formats the disk using the specified track and sector interleave factors.
	Returns:	Error, if any occurred.
Initialize	Parameters:	(*i*) Start of available memory (*ii*) Size of available memory (*iii*) Number of buffers to allocate
	Results:	The disk is reset and made ready for use. If there is sufficient memory available between the specified memory positions, then the procedure allocates the indicated number of buffers. Otherwise, the maximum number of buffers that will fit in the available space are allocated and this number returned in parameter *iii*.
	Returns:	Error, if any occurred.
RegisterClientThread	Parameters:	None.
	Returns:	Client thread i.d.
GetBuffer	Parameters:	(*i*) Client thread i.d.
	Returns:	Memory address of the buffer currently allocated to that thread.
ReadChunk	Parameters:	(*i*) Client thread i.d. (*ii*) Disk Address of the chunk to be read.
	Results:	Data from specified disk address is read into the buffer currently allocated to that client thread.
	Returns:	Error, if any occurred.
WriteChunk	Parameters:	(*i*) Client thread i.d. (*ii*) Disk Address of the chunk to be written.
	Results:	Data from the buffer currently allocated to the client is written to the specified disk address.
	Returns:	Error, if any occurred.
GetInformation	Parameters:	(*i*) Client thread i.d. (*ii*) Buffer to read information into.
	Results:	All device specific information that is relevant to higher levels of the system is transferred into the indicated buffer. This includes data like the number of disk chunks, the size of the disk chunk, etc.
	Returns:	Error, if any occurred.

the procedure and its parameters into a sequence of SCSI bus commands which are issued to the SBIC chip of the Part.12 board.

No disk scheduling algorithm was implemented in any TASS disk server prototypes since the address space of the Part.12's T222 controller is only 64 KBytes. This means that no more than 3 outstanding requests can reside in the disk server, assuming an optimized chunk size of 1 track which is 16896 bytes. Scheduling within a list of 3 outstanding requests was not considered profitable.

4.1.2. Device Independent Interface

The primary purpose of the device independent layer is in implementing the process environment that controls the flow of requests through the disk pipeline of Figure 13. According to David Cheriton, "The complexity of handling message communication is only imposed on programmers developing distributed programs and is overshadowed by the complexity of concurrent programming in general" [Che84](page 24). Implementing the process structures and their interaction is part of the concurrent programming problem within the disk server. Its logical grouping in the device independent layer is to make the process environment immune to changes in device dependent features like the type of disk controlled and the type of communication medium utilized.

The actual functionality of the device independent interface is very similar to that provided by the device dependent layer. In fact, the actual interface supports equivalent Read-

Chunk(), WriteChunk() and GetInformation() services[17]. The other services were not deemed necessary for the higher levels of TASS and, due to time constraints, were not implemented. One new function, Ping(), has been added. The purpose of the Ping() operation is to allow client processors to determine whether the disk server is active and if the disk device is functioning correctly.

4.1.3. Server Message Protocol

Communication between the disk server processor and its client, the cache server, occurs via a transputer serial link. This fact implies that intercourse between the two servers must occur using some form of communication protocol.

> [In order for processes on separate processors] to communicate successfully, they must 'speak the same language'. What is communicated, how it is communicated, and when it is communicated must conform to some mutually acceptable conventions between the [processes] involved. The conventions are referred to as a *protocol* [Sta88](page 10).

Defining the communication protocol for the serial link connecting the cache and disk server processors was one of the first considerations involved in the disk server code design. It had to be firmly established before the design could proceed. The protocol must convert the three selected disk services into a protocol that effects interprocessor communication. The communication takes place in two portions, namely the initial *request* and subsequent *reply*. Communication between the disk server and its clients can occur via any of the transputer's hardware links, as well as a internally allocated channel (soft link). Connecting the disk server software to the cache server software via an internal channel allows RAM disks to be

[17] With the exception that no client thread i.d. parameter is required, since that is a local phenomenon, used to indicate what thread is accessing the device dependent layer.

resident on the same transputer as the client it is servicing. The soft link connection was included to make the software compatible with any new disk device having an address space large enough to operate both the cache server and disk server software within the same transputer.

The request primitives contain a *service operation code*, which dictates what disk service is to be invoked, and a (possibly null) set of *arguments*. Since all transputer link communication is synchronous, with sender processes being blocked until their messages are received, there is no need for disk server processes to acknowledge the receipt of a service request. Table 7 shows the components involved in both the request communication and the reply communication. The set of service operation codes is { PING, CHUNK_READ, CHUNK_WRITE, GET_DISK_INFORMATION }. A request consists of an operation code, followed by any arguments applicable to that operation. There is only one reply primitive, with subsequent arguments being passed in the indicated order, depending upon what service has completed and whether that service was error free or not.

4.1.4. Miscellaneous Considerations

The TASS disk server provides read and write operations for fixed size data chunk transfers only. Note that a *disk chunk* is the only data transfer size available within the disk server component of TASS, hence the term chunk will be used to describe data transfers throughout this paper. The choice of chunk size is not arbitrary but can be defined at compile time by alteration of a compiler constant. Ideally, chunk size should be optimized for the storage device used. The optimum chunk size for our CP-3100 disks will be shown (in section 7.3.2.2) to be 16896 bytes, the size of a single disk track. The actual size used is limited to any integer multiple of the disk device's block size and also by the amount of memory

Table 7: Disk Server to Cache Server Communication Protocol

(i) Requests

Operation:	PING
Arguments:	None
Purpose:	Allow the neighbouring processor to test if a neighbouring disk server is 'alive'.
Operation:	CHUNK_READ
Arguments:	(*i*) Chunk Address
Purpose:	Request that the disk server read from disk all the data stored at the indicated chunk address.
Operation:	CHUNK_WRITE
Arguments:	(*i*) Chunk Address (*ii*) Chunk Data
Purpose:	Request that the disk server write the provided data to the disk chunk specified by the indicated chunk address.
Operation:	GET_DISK_INFORMATION
Arguments:	None.
Purpose:	Request for a copy of all disk specific parameters, like the total number of disk chunks, disk chunk size, etc.

(ii) Responses

Arguments:	(*i*) Operation Responded To (*ii*) Chunk Address (*iii*) Error Code (*iv*) Length of Data to be Transmitted (*v*) Data Chunk or Information Block
Purpose:	Inform requesting client that the indicated operation on the specified disk chunk address has completed. Indicate any error that occurred and finally transfer any data requested by the client.

space available for creating buffers[18].

[18] The primary benefit of the TASS disk server design, as compared to the CSA design, is that multiple buffers are used within the disk server. If a chunk size is selected that is sufficiently large so as to prevent allocation of more than one data buffer, then many of the benefits of the TASS disk server are lost. Considering that the available memory on our Part.12 device is about 52 KBytes, this is not an unreasonable concern.

4.2. Implementation of the First Disk Server Prototype

I will briefly describe the DS1 prototype in order to provide an understanding of why the prototype was abandoned in favour of DS2. DS1 is designed as a hierarchically layered set of modules, each of which performs one basic task. David Clark states that

> "When implementing a system specified as a number of layers of abstraction, it is tempting to implement each layer as a process. However, this requires that communication between layers be via asynchronous inter-process messages. Our experience ... ... suggests that asynchronous communication between layers leads to serious performance problems" [Cla85].

Woe unto me! I fell prey to that temptation, consequently most of the DS1 modules allocated process groups to perform the operations for that module. The tasks were as follows:

- Accepting new requests from neighbour processor(s).
- Executing requests to disk.
- Issuing responses to neighbour processor(s).
- Controlling buffered requests/responses.

The process groups responsible for those four operations were labelled, *Receivers*, *Disk Controllers*, *Senders* and *Queue Managers*, respectively. The queue manager group was required to be single threaded. All other groups were capable of being multi-threaded. The process group interaction within DS1 is shown in Figure 14. Initially, the queue manager creates as many buffers as the available heap space allows. These buffers are allocated to the free buffer queue. The incoming and outgoing queues are initialized to empty lists. The routine operation of DS1 then proceeds as follows:

(1) A receiver obtains a free buffer from the queue manager.

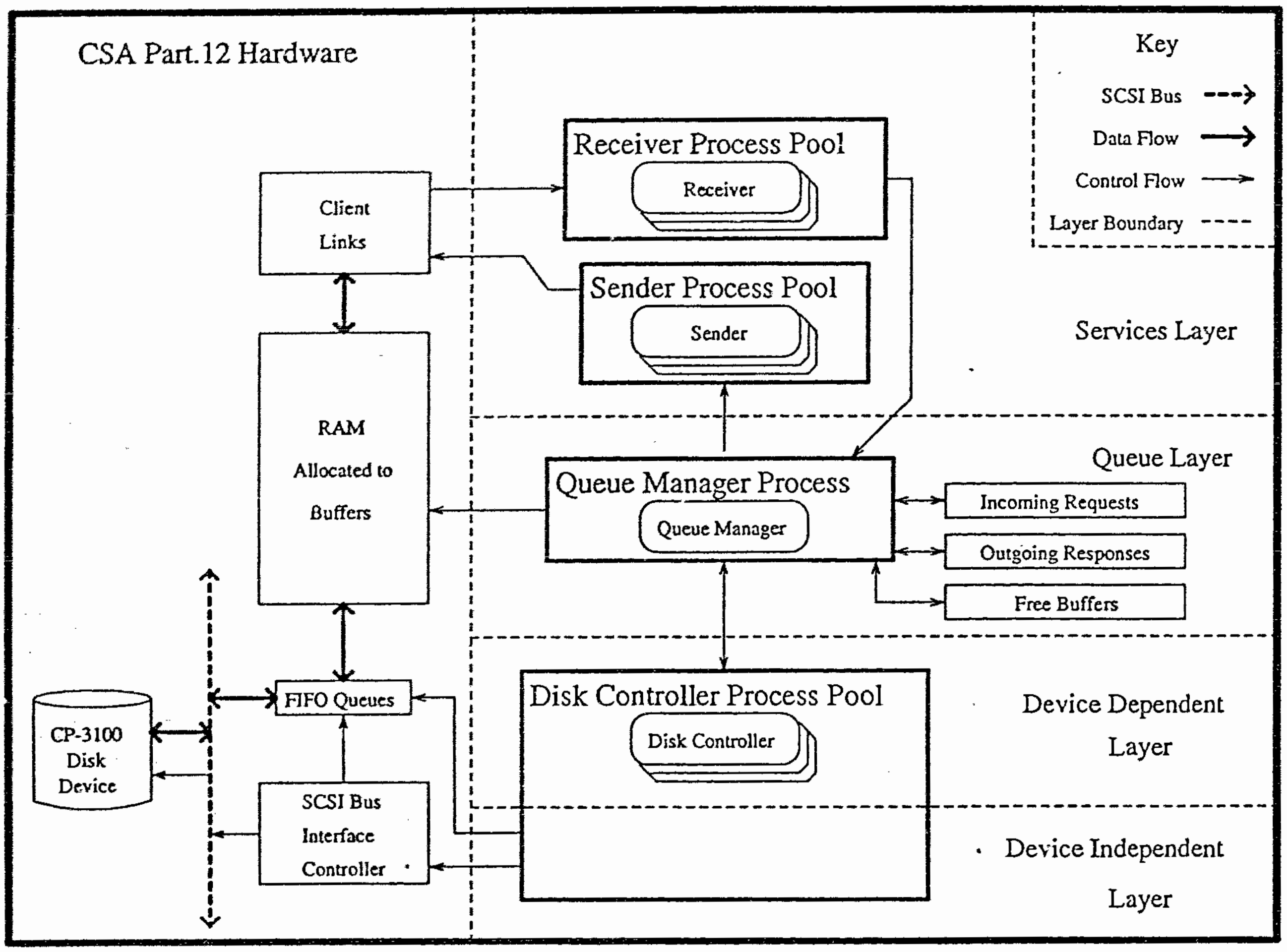


Figure 14: Disk Server Prototype 1 Process Heirarchy

(2) The receiver blocks on the incoming link until a request arrives. Then it reads the request, according to the link protocol, copying any incoming data chunk into the allocated buffer.

(3) The buffer is returned to the queue manager and the receiver returns to step 1.

(4) The queue manager places the buffer onto the incoming requests list.

(5) A disk controller requests a buffer from the incoming requests list, and when available, is given one.

(6) The request is issued to disk via the SBIC.

(7) Results from the SBIC are written into the buffer.

(8) The buffer is returned to the queue manager and the disk controller returns to step 5.

(9) The queue manager places the buffer onto the outgoing responses list.

(10) A sender process requests a buffer from the outgoing responses list, and when available, is given one.

(11) The response is transmitted over the hardware link.

(12) The buffer is returned to the queue manager and the sender returns to step 10.

(13) Queue manager places the buffer onto the free buffer list.

The reason for the failure of this initial prototype is, as Clark indicated, that there are too many communication points between the *queue manager* and all the other processes. Every interaction between two process groups requires a communication using a software channel construct. Additionally, these communications often involve the exchange of multiple data items, requiring multiple messages. Each message requires a context switch between sender and receiver, followed by another context switch once the receiver has obtained the component. This produces sufficient *communication overhead* to cause the DS1 prototype exhibits lower transfer rates than the DS2 driver when operating under light load, even though its sustained transfer rates are comparable to those shown by DS2. The reason for this is that, under heavy load, the communication overhead for one request can occur in parallel with the disk access component of other requests[19]. Therefore, n disk requests will take approximately the same time as is required to perform the n physical disk accesses. However, under light loads, the communication overhead is sequentialized with the physical disk access. This fact may reduce the observed disk bandwidth since some disk accesses may be delayed long enough for the start of target disk track to move past the read/write head, forcing an additional 17 millisecond rotation of the disk device prior to actually performing the physical disk access. The speculation that lead to the design of DS2 was that removal of these interprocess communication points would lead to enhanced performance.

4.3. Implementation of the Disk Server 2 Prototype

The functionality of DS2 is equivalent to that provided by DS1. Primarily, device dependencies are still isolated within low level software modules and the disk server still provides multiple buffering of disk requests. The failure of the first prototype was due to the overhead involved in queue management and process synchronization. To avoid these problems, DS2 was conceptualized as a vertical integration of all DS1 process groups, each performing a different function, into a single DS2 process group that handles all of those functions. This entails that multiple identical processes execute across all layers of the server design. The advantages of restructuring complex multi-level process designs into simpler vertically integrated designs have been discussed by Clark [Cla85] and Atkins [Atk88].

[19] Actually, the communication overhead overlaps with two other disk accesses, since communication for request **R** occurs both before and after the actual disk access for request **R**. So the 'before' part of the communication overlaps with the disk access for request **R**-1 and the 'after' part of the communication overlaps with the disk access for request **R**+1.

DS2 provides multiple buffering by generating 2 or more disk server processes within each disk controller. Figure 15 shows the threads of this process group in relation to the disk server hardware. Each of these processes has access to the following shared data structures:

(1) Memory addresses of the serial links.

(2) Mutual exclusion semaphores.

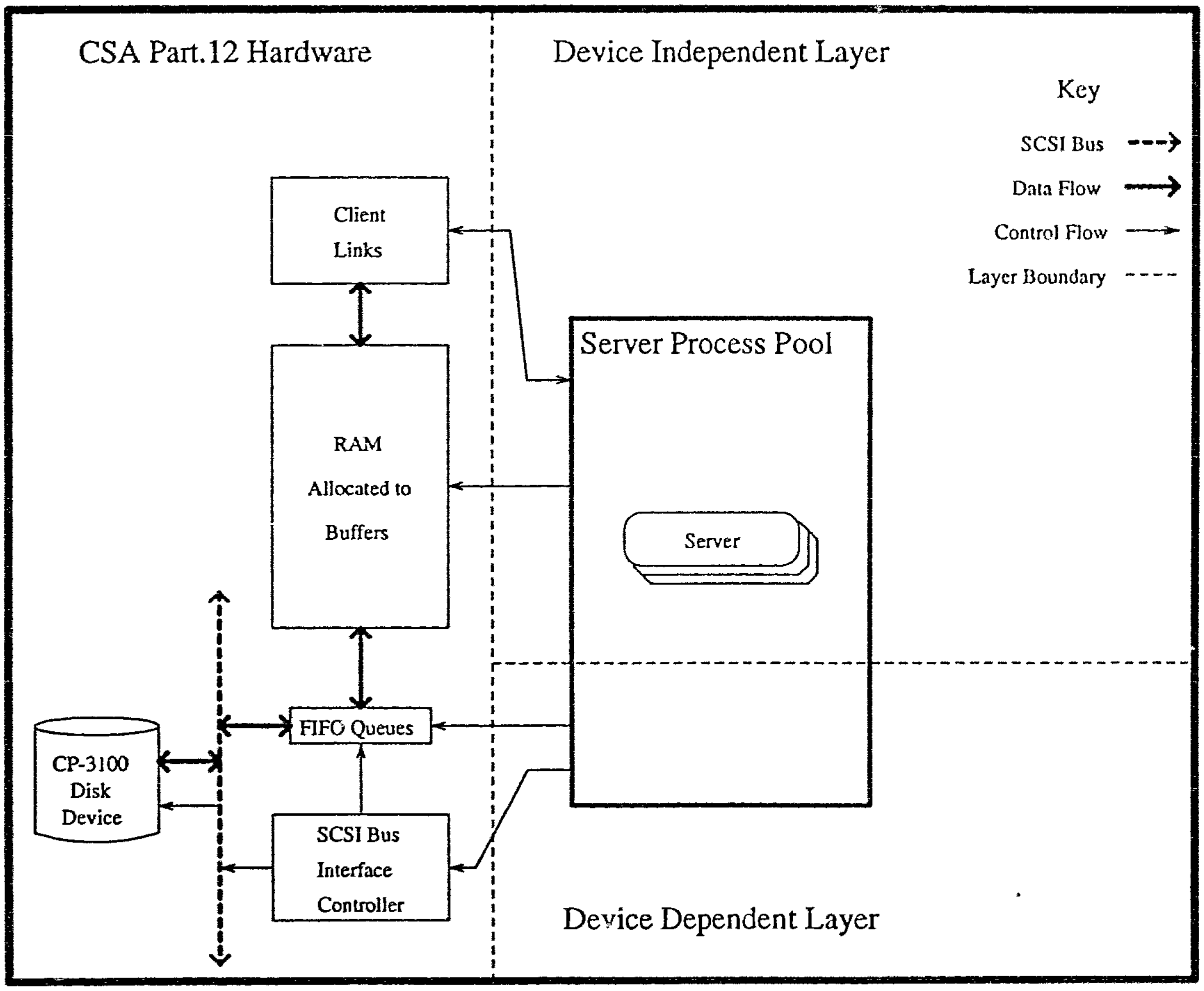


Figure 15: Disk Server Prototype 2 Process Hierarchy

Also, each process has exclusive access to an instance of the following unshared data structures:

(1) Storage buffer for holding one chunk worth of data.

(2) Command block array for transfers between process and SBIC.

(3) A stack.

The use of device dependent structures, namely the serial link addresses and the command block array are hidden with the device dependent SML and DDL, respectively.

Such a disk server, operating within our hardware environment, encounters three potentially blocking (process queueing) bottlenecks, which define the disk access pipeline shown in Figure 13. First, the request is accepted, along with any accompanying arguments, over a temporarily dedicated transputer hardware incoming link. Secondly, the disk operation is issued to, and a response accepted from the SBIC. Thirdly, the results of the disk operation are returned to the invoking processor via a temporarily dedicated transputer hardware outgoing link. These three sub-tasks must occur without interference from any competing processes, therefore each is treated as a critical section and mutually exclusive access to these sections is provided through the use of semaphores.

The basic operation of each DS2 server thread is as follows:

Critical Section: Incoming Hardware Links

(1) Gains control of the incoming hardware links by entering the incoming links critical section.

(2) Block until input is observed for one of the hardware channels (let it be link *i*).

(3) Read the disk operation code and any required parameters from link *i* into local storage. If a write operation is indicated then the data to be written to disk is DMA'd from link *i* into the local storage buffer.

(4) Exit the incoming links critical section. freeing those links for the use of other threads.

Critical Section: SCSI Bus Interface Controller

(5) Enter the SBIC critical section, gaining control of the SBIC and by default, the disk.

(6) Issue the disk request by writing the appropriate SCSI command codes to the SBIC's memory mapped addresses. If the operation is a write, then the chunk data is DMA'd to the SCSI FIFO buffers.

(7) Read the results from the appropriate memory mapped SBIC registers. If disk data has been read, then DMA it into the thread's internal storage buffer.

(8) Exit the SBIC critical section.

Critical Section: Outgoing Hardware Links

(9) Enter the outgoing links critical section, gaining access to all outgoing links.

(10) DMA the results, and any data that was read, out link *i* into the client processor's address space.

(11) Exit the outgoing links critical section.

(12) Return to step 1.

4.4. Implementation of the RAM Disk Server Prototypes

Two RAM disk server prototypes are also implemented. Both are based on the DS2 prototype. Both are intended to run on a T800 transputer, since the T222 has too little memory space to act as a RAM disk. Both prototypes use all of the available address space of the T800 as a RAM disk. The memory is chopped up into numerous disk chunks, with the size of those disk chunks defined at the time the RAM disk is compiled. The first prototype is implemented using the strict device dependent/device independent abstraction. However, due to the fact that server threads need to obtain a free buffer, prior to reading disk requests, it is impossible to implement the RAM disk device without using the bcopy() operation to transfer data between the RAM disk and the buffers. The second prototype streamlined the operation of the server by eliminating the need to use bcopy(), but at the expense of reduced modularity. The RAM disks are presented as a brief example of the modular design vs. performance trade-off. Some performance results for these two RAM disks are presented in Sections 7.2.4 and 7.3.2.3 indicating how much of a data transfer throughput penalty is imposed on the system through the use of the bcopy() operation.

5. The Cache Server

This chapter details the software design for the cache server processor which was shown in Figure 10. As shown in Figure 11, the cache server processor executes the highest portions of the *disk support layer* (DDL), the entire *cache layer* (CL) and *file system layer* (FSL), as well as the lowest section of the *client interface layer* (CIL). Since the body of code spans four distinct layers of the software hierarchy, Section 5.1 will briefly discuss the requirements of each layer prior to actually discussing prototype design. Section 5.1.1 will deal with the resident portion of the disk support layer, Section 5.1.2 will cover the cache layer, Section 5.1.3 presents the file system layer and finally, Section 5.1.4 overviews the *server message layer* (SML), which is the resident portion of the client interface layer.

As was the case for the disk server design, two cache server prototypes have been implemented. These are referred to herein as *Cache Server 1* (CS1) and *Cache Server 2* (CS2). The initial prototype, CS1, has been abandoned in favour of CS2 for qualitative, rather than quantitative reasons. In other words, the perceived failing of CS1 is in the quality of its design, rather than the performance provided by it, which is in fact comparable to that provided by CS2. Sections 5.2 and 5.3 provide descriptions of the design methodology used for each prototype.

5.1. Layer Requirements

5.1.1. Disk Support Layer

The lowest three levels of the disk support layer were discussed in Chapter 4. The higher two levels, namely the *client communication layer* (CCl) and the *disk services layer* (DSL) are presented in this sub-section. These two layers act as a relay, enabling the cache layer to directly invoke the disk services outlined in Chapter 4, without requiring knowledge of the underlying disk devices, the number of such devices, or even the fact that the disk controllers are actually separate processors. In this fashion, the cut between the disk support layer and the cache layer is the point in the TASS design where the abstraction of the disk device(s) into an array of chunks is complete.

As mentioned in Chapter 4, the cache server's CCL and the disk server's *server message layer* (SML) are simply tools used to facilitate communication between the cache server's DSL and the disk server's *device independent layer* (DIL). The CCL communicates directly with the SML, both conforming to the protocol outlined in Section 4.1.3. This protocol takes the form of a group of four request primitives and a group of one response primitive. Cheriton and Berglund have discussed [Che84],[Ber86],[Che88] that most programmers prefer to deal with procedure calls rather than communication oriented request/response primitive sets. For this reason, the interface between the *disk support layer* (DSL) and the *cache layer* (CL) is implemented as a procedure call, hiding the request/response nature of the protocol. However, the two prototypes CS1 and CS2 handle the call quite differently. The former using upcalls [Cla85] and the latter using a more traditional downward procedure call interface.

Although the actual transputer serial links are synchronous communication mediums, the fact that the disk server protocol is implemented as a set of request and response primitives, makes the protocol itself an asynchronous communication. This is because there are no guarantees provided by the protocol that response $\mathbf{Rsp}_i$ will match request $\mathbf{Rq}_i$ for all i. Therefore, each response must be matched to a particular request, or incorrect operation of the disk support layer will result[20]. It is entirely possible that after being issued to the disk server, request $\mathbf{Rq}_n$ may be delayed, perhaps due to a disk scheduling algorithm resident in the disk server, and the response associated with that request will therefore also be delayed. This problem becomes even more discernible when a group of **D** disks are being joined using disk striping. In that case, **D** requests must be issued, depending upon implementation, either sequentially or in parallel. A resulting response must be matched not only to the initial request, but also to the associated responses from the other disks. All of these considerations compound the asynchronicity of the communication medium.

Essentially, the problem faced by the disk support layer is how to provide synchronous communication primitives across an asynchronous communication medium or protocol. Once a synchronous communication environment is established, then conversion from a *remote procedure call* (RPC) to a synchronous communication, and vice versa, is a well documented technique, a good example being [Che84]. To achieve synchronous communication using the specified protocol, it was deemed necessary (by the author) to utilize two process groups, the *Requestor* process group is responsible for issuing requests, as instructed by some higher software level, and the *Acceptor* process group accepts all responses issued by the disk server, and relays those responses to the next higher software layer. A very similar technique is used by Yang and Qu [Yan92], who provide a transputer-based point to point communication service using *Sending-Threads* and *Receiving-Threads* as part of a *Commmunication Server* process group. The technique used in TASS to match disk server requests with related responses differs from prototypes CS1 to CS2, so that discussion will be postponed to Sections 5.2 and 5.3.

[20] Data or error return codes resulting from an access to address **A** may be matched up with a request issued to address **B**.

5.1.2. Cache Layer

5.1.2.1. Overview

The cache layer is responsible for abstracting (*i*) the disk services specified in Table 5 and (*ii*) the actual cache control mechanisms, into the interface described in Table 4. The TASS cache itself is implemented according to the following list of caching strategies:

- The cache uses a fully associative mapping.
- The cache uses the least recently used (LRU) replacement algorithm.
- When a prefetch occurs, the corresponding data will not be locked into the cache until it is eventually used, as has been proposed by some researchers [Smi78],[Kot91]. Holding of prefetched data until its eventual use requires the emplacement of an *erroneous prefetch detection* algorithm. The addition of any locks required to hold prefetched data within the cache has been foreseen in the design, and as such could be implemented with little difficulty. However, this would entail the addition of (*i*) an erroneous prefetch detector as part of the file system layer and (*ii*) additional cache layer functionality to allow that detector to signal the cache regarding incorrect prefetches. The execution of such a detector would require processor cycles (which might reduce system efficiency) and as such was excluded in the current TASS prototypes. Rather than using a prefetch locking mechanism, the TASS prefetch will simply be treated as an access to that data, and if the data is not present in the cache it will be brought in, or, if the data is present, then the cache entry holding the data will be moved to the MRU position within the cache.
- The cache can implement write-one, read-many access restrictions, or it can be compile time directed to allow write-many, read-many access, if the user so desires.

- The cache makes no distinction between the types of data cached, simply treating all data as information stored in the array of chunks abstraction.
- The cache is a delayed write-back implementation. Write backs can be (*i*) delayed until forced by the need for a free cache entry, (*ii*) explicitly performed on specified addresses (therefore allowing a write-through policy to be easily implemented), or (*iii*) can be performed on the least recently used dirty address, allowing a generic write back service to be implemented in a higher software layer (more on this in Section 5.3.3.2).
- The cache maintains information on cached data regarding whether that data is valid and/or dirty. Data within the cache that has not been modified need not be re-written to disk using the blind write-back method, rather write backs need only occur on dirty data.

5.1.2.2. Memory vs CPU Cycles Trade-off

The selection of some of these strategies, particularly the fully associative mapping, makes both processor cycle and memory usage demands on the cache server processor. These demands are effectively overhead to the operation of the TASS storage node. Regarding memory requirements, the only portion of the cache that is not overhead is the actual storage space allocated for cached data. All other information used to initiate and maintain a coherent structuring of that cached data constitues memory overhead. The cached data is stored in cache buffers. These cache buffers can be allocated as either an array of buffers or a linked list of buffers. The two TASS prototypes both use an array of buffers, maintaining these buffers in an LRU order using a separate linked list. This linked list structure requires, for each cache buffer, storage for *lru-neighbour* and *mru-neighbour* pointers, as well as storage for the actual disk address that the data in the buffer is a copy of. Also, a bit-set is provided for each cache buffer in order to maintain the *dirty*, *valid* and *protection* bits. This overhead amounts to between 6 and 10 bytes per cache buffer (depending on size of address buffers, number of protection bits, etc.). Since the size of chunks being cached is in the 4000 to 17000 byte range (depending upon disk type and configuration), 6 to 10 bytes of overhead per 4000 to 17000 bytes of storage is probably not significant. In the actual TASS implementations, the 2 MByte memory of the T800 cache server, coupled with the 16896 byte chunk size (proven to be optimal in Chapter 7), allows for a maximum of 124 cache buffers if no memory is used for code or overhead. In actuality, only 119 buffers can usually be allocated. Because of this, the *mru* and *lru-neighbour* pointers need only 1 byte. The 100 MByte CP-3100 disk, divided into 16896 byte chunks leaves a total of 6208 individual chunks, which can be addressed using only 2 bytes. 2 bytes are used for the bit-set, leaving a total memory overhead per cache buffer of 6 bytes/16896 bytes of storage, or 0.036%[21].

Most of the memory overhead comes from the fully associative cache mapping used. The cache mapping provides a means for the cache manager to (*i*) determine if the requested address A is present in the cache and (*ii*) determine what cache entry currently stores the data for address A, if it is present. The implementation of this mapping is a trade-off between memory overhead and processor cycle usage. One way to perform the mapping is to simply search the cache buffer list for every disk access. This is very expensive in processor cycles, especially considering that every cache miss requires a search of the entire list, in order to determine that the requested address is not held within. As the intention of TASS is to provide a highly efficient LFS, I chose to eliminate searching of the cache list, by keeping a bit set, with one bit for **every disk chunk**. Any time a chunk is brought into the cache that bit is

[21] In striped disk systems, the actual chunk size is larger, so for two striped disks there can be no more than 59 cache buffers with overhead of 6 bytes/33792 bytes. For three striped disks there can be no more than 39 cache buffers, with overhead of 6 bytes/50688 bytes.

set, and when it is removed from the cache, that bit is cleared. This bit-set requires very little in the way of memory (for example, our CP-3100 disks are each divided into 6208 chunks, which makes the size of this bit set 6208/8=776 bytes). Use of this bit set eliminates a great deal of processor overhead, at the cost of less than 1 cache buffer in memory overhead. This bit-set structure eliminates list searching on cache misses, however, list searching still occurs on cache hits. To eliminate all list searching, an index into the cache buffer array is maintained for each disk chunk. As mentioned above, the optimal size for the CP-3100 disk chunk is 16896 bytes, leaving room in the 2 MByte T800 memory for 119 cache buffers. Each index into the cache buffer array need only be 1 byte (or as much as 2 bytes if a T800 with a larger memory is used). Therefore, the 6208 disk chunks can be indexed with a 6208 byte mapping function[22]. This mapping technique eliminates all list searching, streamlining the processor usage, at the expense of between 1 and 2 cache buffers.

5.1.3. File System Layer

The file system is the highest level of the cache server's design heirarchy, since the *Cache Layer* (CL) and *Server Message Layer* (SML) are directly and the *Disk Support Layer* (DSL) is indirectly subservient to the *File System Layer* (FSL). The TASS chunk I/O FSL is provided predominantly as a hook into which any advanced file system can be inserted, assuming that one is designed and coded. This is because the TASS chunk interface (or file system), described in Table 3, maps almost directly to the cache layer described in Table 4. A few minor file system functions are required, for example, implementing the FSL

[22] This assumes that multiple disks are striped, or that interleaved or concatenated disk systems are only 1 disk. For multiple disk interleaved/concatenated systems, the size of this index is 6208 * the number of disks. Even in this case, no more than 2 cache buffers are lost in providing this mapping.

read/write chunk operations requires a simple sequence of subservient layer operations, namely a CL lock chunk, followed by a SML transfer of the physical data (between client processor and cache processor), finally followed by a CL release chunk operation. Additionally, some code has to exist to invoke the subservient CL and SML in an intelligent fashion, however, these details require very little code, so the FSL is almost a *null* entity, existing more as an abstraction than as a physical code module.

Since the file system services multiple clients using multiple communication links, it seems to be a design imperative that the cache server's file system be multi-threaded. Therefore, the file system is split into two component layers, the lowest of which provides the file system services described in Table 3, and the uppermost layer responsible for generating multiple server processes, which in turn invoke the file system services on behalf of client processes. This process group will be referred to herein as the *file servers* or just the *servers* for brevity. This will be elaborated upon in Section 5.3.3.1, however, knowledge of this multi-process environment is necessary in order to understand some of the upcoming discussion.

The only functionality added by the TASS FSL is the implementation of a prediction service. This service effectively controls prefetching within the TASS system. It is true that a prefetch operation is available as part of the TASS FSL, however, its use is limited to prefetching specific disk addresses. Often it is possible to utilize an algorithm to intelligently issue prefetch operations [Smi78],[Kot91]. This style of automated prefetching often takes the form of a *n*-block look-ahead algorithm. Currently, the TASS FSL provides for automated *n*-block look-ahead prefetching during read operations only. The value of *n* can be specified by the client (or the client can use a default value). Since sequential prefetching is a simple algorithm, no process group is required to implement it. However, more complex prediction algorithms try to determine what DAP is indicated by observing recent data accesses. This style of DAP analysis often requires a great deal of computation, therefore a dedicated process is usually required to perform this task. Therefore, the TASS FSL provides the ability to invoke the prediction module as either a simple procedure call, or as an interprocess communication (this will be discussed in Section 5.3.3.3).

5.1.4. Server Message Layer

The *client interface layer* (CIL) is responsible for conveying file system services from the FSL to TASS clients. It is important to note at this point, that the CIL is the lowest level of a distributed file system. Hence, this is where the division occurs between this thesis' target of providing a basic LFS (which can be used as a building block for a DFS) and actually implementing a DFS. Because the nature of this DFS is unknown, the design of the client side of the CIL, which is the *client communication layer* (CCL) cannot be firmly set. To test TASS, we needed some form of client interface and for this reason (which will be further elaborated on in Chapter 6), the CCL was implemented as a simple RPC interface. The cache server resident portion of the CIL is the *server message layer* (SML) which is a tool for converting the request/response protocol issued by the CCL into the file system services shown in Table 3. This protocol is shown in Table 8 (*i*) and Table 8 (*ii*). Unlike the protocol used in the disk server to cache server communication, all responses are not provided by a generic response primitive. Each individual request type has a specific response associated with it. Since the communication controlled by this protocol is a RPC, this implies that the client process that issued the request is blocked waiting for a response, and, since it knows the nature of the request, it will know and understand the nature of the response.

Table 8 (*i*): Cache Server to Client Communication Protocol

Operation:	**Set Links**	
	Request Format:	(*i*) Operation Code = SET_LINKS
	Response Format:	(*i*) TASS Chunk Size
		(*ii*) TASS Cache Size
		(*iii*) Number of Available Disk Chunks
		(*iv*) Error Code
	Purpose:	Allows the client processor to test if a neighbouring cache server is 'alive'. In addition to that, the primitive allows for the communication of relevant constants between the cache server and the client processor, eliminating the communication expense that would be involved in implementing operations such as get_cache_size() using a RPC protocol.
	Note:	This procedure is not specified in the File System Operations (Table 3). It is not really a file system operation, rather it is necessary to initiate the communication protocol over the link. It is also used to streamline protocol operation as well.
Operation:	**Read Chunk**	
	Request Format:	(*i*) Operation Code = READ
		(*ii*) Chunk Address
		(*iii*) Offset Within Chunk
		(*iv*) Number Bytes to Transfer
		(*v*) Predictor Argument
	Response Format:	(*i*) Error Code
		(*ii*) Data Transfer
	Purpose:	This format allows the client to request that the data stored at the indicated chunk address be copied to the client's local memory. The read need not transfer an entire chunk, rather partial chunk transfers are facilitated using the offset and number of bytes parameters. A predictor argument is also passed, enabling the client to direct the work of the predictor algorithm (if any) discussed in Sections 5.1.2 and 5.3.3.3.
Operation:	**Write Chunk**	
	Request Format:	(*i*) Operation Code = WRITE
		(*ii*) Chunk Address
		(*iii*) Offset Within Chunk
		(*iv*) Number Bytes to Transfer
		(*v*) Write Through Flag
		(*vi*) Data Transfer
	Response Format:	(*i*) Error Code
	Purpose:	Allows the client to request that the data contained in the data transfer component be copied into the indicated chunk address. As with the **Read** operation, partial chunk transfers are possible. In addition, the client can specify that the chunk be immediately flushed through the cache to disk.

Table 8 (*ii*): Cache Server to Client Communication Protocol (continued)

Operation:	**Prefetch Chunk**	
	Request Format:	(*i*) Operation Code = PREFETCH (*ii*) Chunk Address (*iii*) Number of Chunks
	Response Format:	None.
	Purpose:	Allow the client to request the prefetching of the indicated chunk address. In addition, the number of chunks argument allows for the prefetch of the specified number of sequentially subsequent chunks.
	Note:	The fact that there is no response makes the prefetch an asynchronous operation. The client will not be made aware of any errors that occurred during the reading of a chunk until an actual **Read** operation for that chunk is issued.
Operation:	**Flush Cache**	
	Request Format:	(*i*) Operation Code = FLUSH (*ii*) Invalidate Flag
	Response Format:	None.
	Purpose:	Indicates that all dirty chunks currently resident in the cache are to be written to disk. Additionally, the invalidate flag allows the client to request that all of the data stored in the cache be invalidated (forcing it to be re-read from disk the next time it is accessed).
	Note:	Like the **Prefetch** operation, flush is an asynchronous procedure.

5.2. Implementation of the Cache Server 1 Prototype

The two cache server prototypes only differ in the implementation of the DSL and CL. The FSL and SML implementations are not identical, but the CS1 versions of those layers are a sub-set of the CS2 versions. Because of this prototype being abandoned, those layers did not undergo the complete implementation intended by the design. Therefore, this section only discusses how the DSL and CL are implemented in the CS1 prototype. The discussion of the FSL and SML are postponed until the CS2 prototype discussion.

The initial attempts at generating a TASS cache design involved letting the *file servers* each access the cache directly. Wherever access to shared data structures, like the cache list, valid bits, look-aside buffers, etc., was required, a critical section would be established to monitor those accesses. Concurrent access to critical sections was prevented using the semaphore construction. That way any *file server* process operating in the cache server can share these important structures without fear that another server is already tampering with them. This design approach did not progress very far because the problems involved with these semaphores, namely the potential for deadlock and/or erroneous behaviour, was deemed too great. Circular dependency issues [Baw91] cropped up everywhere. For example,

(1) A *file server*, $\mathbf{P}_1$ trying to read disk address **A**, accesses the list of cache entries and obtains mutually exclusive access to that list.

(2) $\mathbf{P}_1$ finds the cache entry it is looking for, but the entry is currently being read from disk by another *file server* $\mathbf{P}_2$. Clearly, $\mathbf{P}_1$ cannot read the data yet, since it is not in the cache yet.

(3) $\mathbf{P}_1$ cannot go forward, since the data is not available, and it clearly cannot block, because it still holds mutually exclusive access to the cache list, which $\mathbf{P}_2$ will require in order to insert the data that $\mathbf{P}_1$ waits for.

(4) Even if $\mathbf{P}_1$ tries to release exclusive access to the cache list, then immediately block waiting for a signal from $\mathbf{P}_2$, it may be context switched out after releasing the cache list, but before placing itself on the blocked list for the desired cache entry. $\mathbf{P}_2$ could easily come along and insert the data into the appropriate cache entry, find no other process waiting for the data, and as a consequence $\mathbf{P}_2$, would issue no wake up call to $\mathbf{P}_1$ who would then be blocked indefinitely.

This is just an example, but illustrates one of the many problems involved with the implementation of a complex data structure that is to be shared by numerous concurrent processes. I decided that modularizing the cache control by creating a single process, the *Cache Manager*, to monitor all of these data structures was the only reasonable alternative.

Prompted by the perceived need for a *Cache Manager* prototype, I fell prey (as I had in the first disk server prototype) to the temptation of allocating a process group to perform each of the 'obvious' required tasks[23]. Eventually, this led to the process hierarchy shown in Figure 16. The *Cache Manager* is the central process, with all other process groups requesting

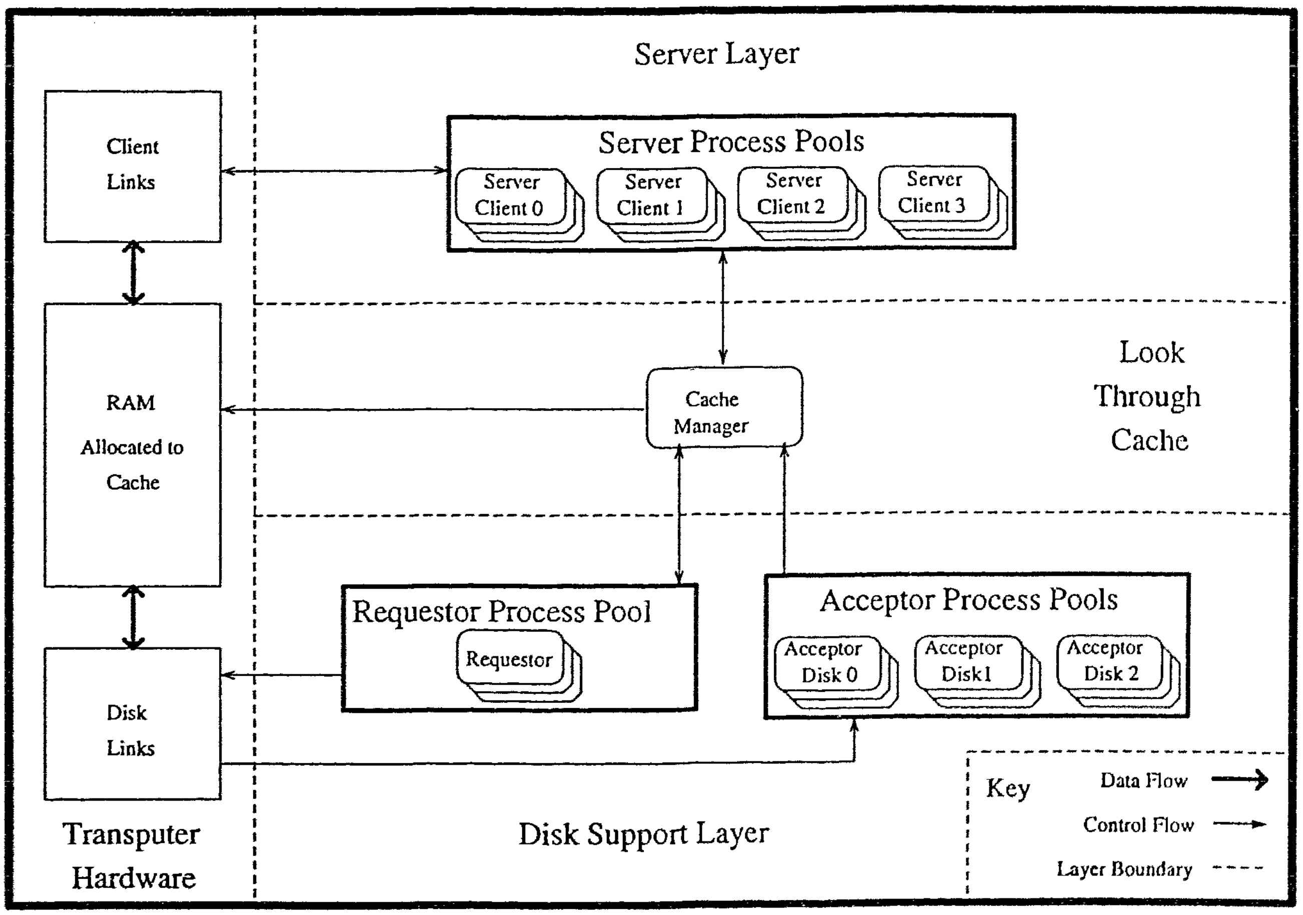


Figure 16: Cache Server Prototype 1 Process Hierarchy

[23] However, since the cache server code is significantly more complex than that of the disk server so it is much easier to justify imposing a little communication overhead to achieve a high degree of modularity.

service from that centralized source. In the case of the *File Servers* and the *Requestors*, this corresponds to the normal downward flow of client requests to the disk server. However, the asynchronous responses issued by the disk server, and accepted by the *Acceptors* contravenes this downward flow. The fact that the *Acceptors* 'depend on' the *Cache Manager* for service is an upward control flow.

This design uses an approach similar to that of the upcall technique presented by Clark [Cla85]. In Clark's method, the natural flow of control for the problem at hand, is mapped to the operation of the system's components, whether that flow of control is in a hierarchly downward or upward fashion.

> "... there are organizational and modularity reasons why this [hierarchical] downward flow of control is appealing in a layered system. However, the natural flow of control is not always downward. In a network driven environment, for example, most of the actions are initiated, not by the client from above, but by the network from below. The natural flow of control is thus upward, not downward. Especially where such an upward flow of control crosses a protection boundary, most systems do not permit this flow to be implemented as a procedure call. Instead, some more cumbersome and asynchronous mechanism must be used, such as an interprocess communications signal. Substantial inefficiencies and complexities can result from asynchronous upward flows. In our methodology, the system is organized so that the programmer has the choice as to whether an upward flow is implemented by procedure calls or asynchronous signals" [Cla85](page 171).

5.2.1. Requestor Process Group

The operation of the *Requestor* group for CS1 is very simple. Each *Requestor* process makes an upcall to the *Cache Manager* and (effectively) says *"I am idle. Give me work."*. When the *Cache Manager* needs a disk access performed, it issues the disk operation to one of the idle *Requestors* which then issues it to disk. After that, one further upcall is made from the *Requestor* to the *Cache Manager* in order to return the buffer within which the data was stored. As the *Requestor* knows what operation is associated with the buffer, it is its responsibility to inform the *Cache Manager* as to what must be done with the buffer. For a write operation, the buffer is returned to the cache (as the data in it is still valid). For a read operation, the buffer is returned to the free buffer list where it will eventually be filled with data from disk.

Any disk joining technique can be abstracted within this model as the *Requestor* process can map the disk address specified by the *Cache Manager* to the physical disk, and the address within that disk, which would correspond to the joining. However, since only one *Requestor* process is handling the request, a striped disk joining would have its component requests (one request per physical disk) serialized rather than parallelized.

5.2.2. Acceptor Process Group

Operation of the *Acceptor* process group is even more simple than for the *Requestors*. Each *Acceptor* must have a buffer available in case the operation it tries to accept is a read. If it has no buffer, it makes an upcall to the *Cache Manager*, which gives it a buffer from the free buffer list (the code is constructed so that there is always a free buffer available). The *Acceptor* then waits for any incoming disk server response. Once such a response is observed, the *Acceptor* reads it (and any incoming data is read into the available buffer) and makes an upcall to the *Cache Manager* to inform it as to the contents of that response.

As discussed with the *Requestor* process group, any disk joining could be implemented using this model. However, striped disks might pose a problem since all the responses from component disks would have to be assembled together into one combined response. This would require some form of bookkeeping structure, not currently incorporated into the model, in order to maintain the status of outstanding requests. The CS1 implementation was abandoned before the addition of striping to the multiple disk model, so I am not certain as to the problems that may have occurred in an actual striped implementation using the CS1 model.

5.2.3. File Server Process Group

Briefly, *File Servers* read requests from the clients via the serial links. These requests are then passed through the file system code which converts them into a sequence of cache accesses. These accesses are issued to the *Cache Manager* and appear to the issuing *File Server* to be synchronous events. For example, a *File Server* issuing a Write Chunk to address A will block until it is given an exclusive lock on A within the cache. The *File Server* will not notice any time spent waiting for the address to be read into the cache or for some other *File Server* to finish with address A before that lock is given to the requesting *File Server*. The *File Server* operation will be discussed in more detail in Section 5.3.3.1.

5.2.4. The Cache Manager

In the CS1 implementation, the *Cache Manager* performs the majority of the work involved in the operation of the system. Because of that, its operation is critical to the performance of the entire cache server. For this reason, the following rules are zealously adhered to.

- The *Cache Manager* can **never** be blocked while waiting for communication if there is another process, dependent on the *Cache Manager*, that would be starved due to that blockage. In other words, the *Cache Manager* cannot block while waiting for a disk access to complete if that blocking would prevent the *Cache Manager* from satisfying the request of some other process.

- Before entering into communication with any other process, the *Cache Manager* must be guaranteed that that communication can proceed to completion without requiring the *Cache*

Manager to indefinitely block during the communication.

The actual implementation of the *Cache Manager* requires that it maintain a list of communication channels, one per client process (**R** *Requestors*, **A** *Acceptors* and **F** *File Servers*, for a total of **R+A+F** channels). The *Cache Manager* can only block if none of these communication channels are active. Once one channel indicates an incoming communication, that communication must be completely handled in some fashion, thus allowing the *Cache Manager* to return to blocking on the list of channels.

For the *Requestors*, three possible communication types can occur, namely the "I am idle" message and the two buffer return message variants. The buffer return messages are simple to process to completion, since they require the *Cache Manager* to accept the buffer and place it on either the cache buffer list, or the free buffer list. The "I am idle" message is not as simple to handle. It is entirely possible that the *Cache Manager* has no need for a *Requestor* at this moment in time. It cannot leave the communication pending because, due to the nature of transputer interprocess communication, acknowledging the incoming message effectively begins the protocol for handling that message. So the communication with the *Requestor* must be completed. This requires that the *Cache Manager* maintains a separate structure to keep the status of all *Requestors*. In other words, the *Cache Manager* must maintain a list, in some form, of all *Requestors* that have signalled they are idle, and who have not been issued a request to handle since that idle message arrived.

There is no need to maintain any structure regarding the status of the *Acceptor* processes. The *Acceptor* processes simply act as relays, passing disk server responses to the *Cache Manager* (they are required since waiting on the disk server is an indefinitely long communication block, which the *Cache Manager*, by the above rules, cannot do itself). What the *Cache Manager* must maintain is a list of outstanding disk requests, since every incoming response must be matched to one of those requests. So even the correct operation of the

Acceptor process group requires the maintenance of status structures within the *Cache Manager*.

As for the *File Server* process group, there are many distinct communication operations that can occur, since a request for a *Read Lock*, *Write Lock*, *Write Partial Lock*, *Release Read Lock*, etc., can be issued. Some of these operations are inherently blocking in nature. For example, obtaining locks may require a *File Server* to block, since the data may not be in the cache and a disk access must be made to fetch it or the data may already be in the cache, but is currently locked, and the new lock violates the write-one, read-many (WORM) access criterion. Some other operations are *not* inherently blocking, since lock release operations can always be immediately satisfied. Because of this, the *Cache Manager* must accept incoming file server communications in order to determine if they are satisfiable. If they are immediately satisfiable then the *Cache Manager* can do that and all concerned parties are happy. However, if the request cannot be satisfied right away, then the *File Server* must be blocked until that request can be satisfied. This requires that the *Cache Manager* maintain a list of all blocked *File Servers*, as well as the reason why they are blocked. Every time another operation completes (for example a disk access completes and address **A** is placed in the cache, or a release lock operation is performed on address **A**), the *Cache Manager* must then check this list of blocked *File Servers* to see if their request can now be satisfied.

Furthermore, a distinction must be made as to whether a *File Server* wishes to write an entire chunk, or just part of it. If the whole chunk is to be overwritten, then there is no need to first read it from disk. However, if only part of a chunk is to be overwritten, then the chunk must first be read from disk. The addition of an alternate form of write locking, in the interests of avoiding unnecessary disk accesses, helped to complicate the software.

The primary reason for the perceived failure of this prototype was that during coding of the *Cache Manager* process, it became all too apparent that the concept behind the design was unwieldy. The *Cache Manager* process performed the majority of work within the system, maintaining cache integrity, issuing disk requests, processing disk responses, etc. Consequently, it was responsible for maintaining not only the cache data structures, but also status information regarding the progress of each *Requestor*, *Acceptor*, *File Server* and (effectively) of each client processor's outstanding requests. The *Cache Manager* software module rapidly became a king among monsters, its size rivaling Tyrannosaurus Rex, its complexity reminiscent of Fettucine Alfredo. For this reason alone, the decision to scrap the CS1 prototype was made, and the decision was made to proceed, hand in hand with all the knowledge gained in the CS1 experience, with the second cache server prototype, CS2[24].

5.3. Implementation of the Cache Server 2 Prototype

The concept behind the design of CS2 is to offload some of the work performed by CS1's *Cache Manager* onto the other processes in the system. In this fashion, the complexity of the CS2 *Cache Manager* is reduced to more manageable levels, at the expense of slightly more complex process structures for the *Requestors*, *Acceptors* and *File Servers*. Additionally, the problems inherent in disk striping under the CS1 model (Section 5.2) are considered and dealt with by the CS2 implementation. The process hierarchy is shown in Figure 17. Note that all the process groups in the CS1 prototype are also used within the CS2 prototype. There are two new process groups, namely the *Write Back* and *Predictor* groups. These will be discussed in Sections 5.3.3.2 and 5.3.3.3, respectively. Until those sections, it will be sufficient to simply think of the *Write Back* process group as being equivalent to the *File*

[24] This assumes that all the knowledge gained is actually retained. If you were fully aware of my filing system, you would scoff at this claim.

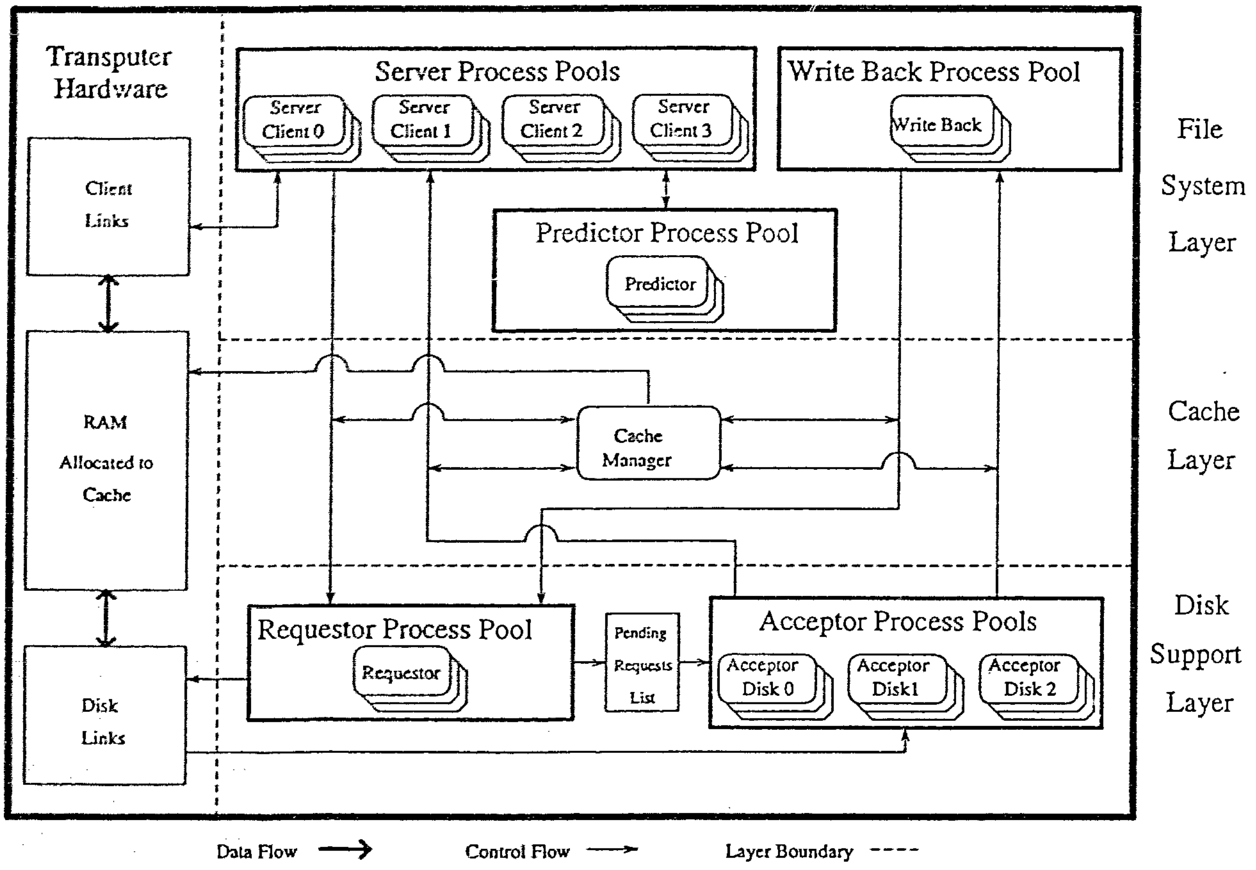


Figure 17: Cache Server Prototype 2 Process Hierarchy

Server process group. It should also be sufficient to not think of the *Predictor* process group at all.

The major changes in the operation of the cache server are:

- The upcall technique used for *Requestor* and *Acceptor* communication with the *Cache Manager* has been abandoned. Rather, the *Requestors* and *Acceptors* cooperate with each other to provide a procedure call interface that is concurrently accessible to multiple processes. This makes for a very clean modular cut at the disk support layer/cache layer interface.

- Rather than the *Cache Manager* being responsible for issuing disk accesses to the disk support layer, the *File Server* process group directly issues disk accesses to the disk support layer, when those accesses cannot be satisfied by the cache contents.

- The *Cache Manager* controls a look-aside cache, rather than the look-through cache of CS1. In other words, *File Servers* simply query the *Cache Manager* as to whether the required disk address **A** is resident in the cache. If **A** is resident, then the *Cache Manager* will either give the *File Server* access to the buffer holding the data from address **A**, or block the *File Server* if that access is not immediately satisfiable. If **A** is not resident, then the *Cache Manager* gives the *File Server* a cache buffer into which the data for **A** can be read (or written). Additionally, the *Cache Manager* tells the *File Server* whether the provided buffer holds a 'dirty' disk chunk **B**, and if it does, the *File Server* is responsible for returning the data for **B** to the disk, prior to using the buffer for address **A**. Once the data for **A** is copied into the buffer (either from disk, from the client, or both), the *File Server* returns that buffer to the *Cache Manager* requesting that it place address **A** and its corresponding data into the cache.

5.3.1. Disk Support Layer

This section describes the interaction between the *Requestor* and *Acceptor* process groups. The disk support layer core primitives are shown in Table 5. The conversion of an asynchronous protocol into a set of efficient remote procedure call stubs, and vice versa, was not as easy as I had expected. The most simplistic RPC implementation would be to have one process gain exclusive access to the disk server communication links and issue a request, holding the communication links until the associated response is obtained. This method gains its simplicity by sacrificing some of the inherent parallelism of the disk server access. As discussed in Chapter 4 and shown in Figure 13, execution of the disk server can be thought of as a pipeline. If every stage of this pipeline is not kept busy, then sub-maximal performance will be observed by the clients of that disk server. In Chapter 8 it will be shown that at least three simultaneously active requests are required to keep the TASS disk server and CP-3100 disk device busy, when a sequential data access pattern is being issued. When a random DAP is being issued, only two simultaneously active requests are required.

A less naive approach might be to have a process obtain mutually exclusive access to the send link, issue its request to the disk server and then obtain mutually exclusive access to the response link, reading its associated response when it becomes available. This assumes that the sequence of requests will identically match the sequence of responses, which we have already stated (in Section 5.1.1) will not be the case. Therefore, providing a device independent disk support layer cannot utilize this approach.

The TASS approach uses the *Requestor* and *Acceptor* processes as follows:

(1) The *Requestor* accepts a request from one of the disk support layer's clients, noting the client i.d. of that client[25].

(2) The request is preprocessed, in order to determine which subordinate disk it must be issued to (assuming multiple disk environment) and what address on that disk is actually being referenced.

(3) The request is issued to the subordinate disk. In the case of striped disks, the request is sequentially issued to all three component disks.

[25] In the TASS system, each process is given a unique process i.d. code upon generation of that process.

(4) The *Requestor* generates and places an entry on the *pending requests list* (PRL), which is maintained by the *Requestor* and *Acceptor* process groups. That entry contains the process i.d. of the requesting client, the physical disk involved (in a multiple disk environment), the type of disk operation, the physical disk address to which that operation is being issued, and the buffer into which any data transfer is to made.

(5) The *Requestor* returns to monitoring for new client requests.

(6) Initially, one *Acceptor* process is blocked on each incoming communication link from one of the disk servers. Once an incoming communication is observed, the *Acceptor* begins by reading the first portions of the response primitive (from Table 7), namely the operation and the disk address with which this response is associated.

(7) These two components of the response, coupled with what disk that response comes from, can be used to uniquely identify the corresponding entry on the PRL. If multiple duplicate requests for an address are on the PRL, then the assumption is that the response matches the first of those duplicate requests found. The entry is removed from the PRL and the client i.d. and buffer are kept by the *Acceptor*.

(8) Communication with the disk server completes when the *Acceptor* reads the error code, as well as any incoming data into the buffer.

(9) The *Acceptor* then uses the client i.d. in order to return that data to the appropriate client process, freeing it to continue its operation.

(10) Finally, the *Acceptor* returns to monitoring the link from the disk server.

In this method, the *Requestor* operation is implemented as a purely downward flow of control, from client process request through to the issuance of the actual disk access(es). The *Acceptor* operation is still an upcall, however, the fact that an upcall is involved is completely hidden from any clients of the disk support layer. This provides a better abstraction of disk services than is provided by the CS1 prototype.

5.3.2. Cache Layer

The *Cache Manager*, the *File Server* and the *Write Back* process groups all operate within the cache layer. The cache layer is divided into many distinct modules, however, there are only two primary ones, namely the *Cache* and *CacheManager* modules. All the other modules are subservient to the *CacheManager* module. As the name suggests, only the *Cache Manager* executes within the *CacheManager* module. The *File Servers* and *Write Back* processes execute the *Cache* module software.

5.3.2.1. The CacheManager Module

This is the central module to the cache server software. Even with the streamlining of the *Cache Manager* process that took place between the CS1 and CS2 prototypes, this module still remains quite large. However, its function is simpler, and, as a side effect, the code is a little less unwieldy and easier to comprehend[26]. In trying to simplify the design of the CS1 *Cache Manager*, I observed that there are seven basic operations that are required to completely service any cache request within the cache layer model outlined in Table 4[27].

---

[26] I have written two pieces of code in my life that I could not understand after 6 months without looking at it. These are the very first large program I wrote, which was in BASIC, and the CS1 *Cache Manager* software.

[27] This does not include many of the basic functions like **ChunkSize** and **NumberOfChunks**, as these do not even directly access the *Cache Manager*, rather they are routed by the *Cache* module directly to the disk support layer procedures of the same name. The only other procedure of note is **CacheSize**, which is easily implemented as a procedure call that returns the value of a variable cache_size which is in turn set by the *Cache Manager* immediately after the cache is initialized. In other words, none of these procedures require communication with the *Cache Manager*.

These seven procedures, which are discussed in the following sections, can be used to effectively execute any change to the state of the cache that might be required by any of the cache layer primitives. In the discussion of the primitives, the word server is used to mean either a *File Server* or a *Write Back* process.

5.3.2.1.1. Get Lock

Arguments:	(*i*) Server i.d
	(*ii*) Requested Address
	(*iii*) Type of Lock Requested
Results:	(*i*) Obtained Address
	(*ii*) Pointer to Buffer Holding Data
	(*iii*) Dirty Flag
	(*iv*) Error Code

The server requests that the *Cache Manager* place a lock, from the set of lock types, on the requested address. The set of lock types is {Read,Write,None}. If the requested address is in the cache, then the *Cache Manager* can grant the lock, if possible, or block the requesting server until the lock is possible. If the requested address is not in the cache, the *Cache Manager* need only give the server a buffer and tell it to go fetch the requested address from disk. However, if the buffer provided has valid and dirty data in it, then the server must return that data to disk first. For this reason, the *Cache Manager* provides an obtained address field which tells the server which disk address it has received. If the obtained address does not match the requested address and the dirty flag is set, then a forced write back is necessary to clear the data in the buffer. The obtained address field also informs the server as to the disk address that the dirty data belongs in. Prefetch operations are simply Get Lock operations which use the 'None' lock type option. Note that unlike the CS1 prototype, no alternate write lock (to deal with partial writes) need be provided since the server process can choose to read the data from the disk prior to the access, if that is applicable.

5.3.2.1.2. Release Lock

Arguments:	(*i*) Server i.d
	(*ii*) Locked Address
	(*iii*) Type of Lock Requested
Results:	(*i*) Error Code

The server informs the *Cache Manager* that it has completed its operations on the locked address and that the type of lock specified can be removed. This operation only applies to Read and Write locks and is not necessary for the None lock variety.

5.3.2.1.3. Insert New Entry

Arguments:	(*i*) Server i.d.
	(*ii*) Address
	(*iii*) Lock Type
	(*iv*) Make MRU Flag
	(*v*) Is Valid Flag,
	(*vi*) Pointer to Data Buffer
Results:	(*i*) Error Code

This allows the server to complete a disk access by inserting the newly read address (and its data) into the cache. Also, since the server may still need a lock on that entry, it can specify that by using the Lock Type argument (thus giving the server that fetched the data first crack at using the data). Also, to allow a greater degree of control, the procedure was also implemented with flags allowing the server to select if the entry is placed in the MRU or LRU position of the cache, or even to indicate that the data in the buffer is not valid.

5.3.2.1.4. Start Write Back

Arguments:	(*i*) Server i.d.
	(*ii*) Requested Address
Results:	(*i*) Obtained Address
	(*ii*) Pointer to Buffer Holding Data
	(*iii*) Error Code

The server informs the *Cache Manager* that it wishes to write a chunk back to disk. The server can specify the address, in which case it will either be given that address and the buffer holding the corresponding data, or informed (via the Error Code) that the address is not dirty or not currently in the cache. Alternatively, the server can specify an invalid address, in which case the *Cache Manager* will give it the least recently used dirty address, if one is available. This last option is used by the *Write Back* process group.

5.3.2.1.5. Finish Write Back

Arguments:	(*i*) Server i.d.
	(*ii*) Address
Results:	(*i*) Error Code

This allows the server to inform the *Cache Manager* that the indicated address has been successfully returned to disk and that its dirty bit can be cleared.

5.3.2.1.6. Stop Cache Flush

Arguments:	(*i*) Client i.d.
	(*ii*) Invalidate Flag
Results:	(*i*) Error Code

Allows a server to inform the *Cache Manager* that a cache flush has been requested. Cache operations are to be suspended while all dirty data in the cache is cleansed by individual **Start/Stop Write Back** calls. Additionally, the server can inform the *Cache Manager* to invalidate all the entries within the cache, once they have been returned to disk (if they were dirty).

5.3.2.1.7. Stop Cache Flush

Arguments:	(*i*) Client i.d.
Results:	(*i*) Error Code

Once one of the **Start Write Back** operations returns an error code indicating that there are no dirty entries left in the cache, the server can use this procedure to inform the *Cache Manager* that the flush has completed and that normal cache operations can proceed.

5.3.2.2. The Cache Module

By now the modularity purists are screaming, "How can those procedures be justified when they require a higher layer of software to provide consistent lock values and operate in a consistent fashion, issuing start and stop operations in the correct order, etc., etc., etc.?" How I propose to justify those primitives is by adding an additional module layered on top of the *CacheManager* module. This module will be part of the cache layer, but its code will be in-line procedures executed by the clients. What the *Cache* module accomplishes is to provide a set of procedures that will call the *CacheManager* module in a consistent fashion. Furthermore, it allows for the client processes, namely the *File Server* and *Write Back* processes to directly access the disk support layer, effectively eliminating the *Cache Manager*'s dependence on that layer.

5.3.3. File System Layer

As mentioned earlier, in 5.1.3, the file system layer of TASS is effectively a null entity. The interesting component of the file system layer is the implementation of the three resident process groups, namely the *File Servers*, *Write Backs* and *Predictor*. Also of interest is how they interact with each other and the cache layer and server message layer.

5.3.3.1. File Server Process Group

The file server process group is the workhorse of the TASS system. It is responsible for (*i*) accepting requests from the serial links that connect TASS to its clients, (*ii*) converting those requests into a sequence of file system layer procedure calls, (*iii*) querying the *Cache Manager* to see if the request can be satisfied by the current contents of the cache, (*iv*) if not satisfiable by the cache, then the server group must execute the request to disk by initiating a sequence of calls to the disk support layer and (*v*) finally, return the results of the operation to the client via the same serial link upon which the request arrived. Admittedly, some of these operations ((*iii*) and (*iv*)) are hidden from the file server process group by the cache layer abstraction, but the fact that the *File Server* threads must actually do the operations makes the importance of the group paramount.

At compile time, the number of *File Server* threads is selected through the use of five constants. These are, from the *Process.h* header file PROCESS_SERVERS_PER_LINK, which defines how many file server threads will be allocated to the pool for each client link and four constants from the *ConfigureLinks.h* header file, CONFIGURELINKS_CLIENT_ON_LINK_*i*, where *i* ranges from 0 to 3. Basically, the client on link constants are booleans, defining what links are used for communicating with client processors and what links communicate with disk servers. Because of this, each client has a dedicated pool of servers, rather than all clients sharing the same pool. There are both advantages and disadvantages to this approach.

To exemplify a disadvantage, consider the following. A TASS storage node services two clients. One client only issues one disk request at a time and uses no prefetching or other services that tie up *File Servers*. Another client is executing using a highly predictable DAP and wants to prefetch extensively. However, the fixed size of that client's *File Server* pool limits the degree to which the client can control prefetching, even though there are idle *File Servers* sitting in the other clients pool.

The advantages to this technique are in simplicity of design and minimizing of shared data structures. Each pool is dedicated to a single serial link, therefore the threads in that pool don't have to worry about any other client links, therefore the code for those processes is simpler. Additionally, there is no need to share multiple client links, hence there is no need to implement a critical section that encompasses all client links. The design of such an encompassing critical section, within the constraints of the Logical C and transputer communication environment is not easy[28]. It is significantly simpler to use a single semaphore to guarantee any thread within a pool mutually exclusive access to the one link serviced by that pool.

5.3.3.2. Write Back Process Group

The write back process group is a compile-time optional component of the TASS system. Its objective is to keep the cache clean without degrading system performance. The

[28] Numerous methods were attempted. Either (*i*) they failed to correctly operate, leading to deadlock or (*ii*) they provided atrocious performance due to excessive communication overhead required to circumvent the problems that led to failure (*i*).

ideas prompting the inclusion of this process group are:

- In a delayed write back cache environment, altered data for disk address A stays within the cache until the buffer holding it is targetted for replacement. At this point, the modified data has to be written back to disk, prior to being replaced by any new data for disk address A'. When this occurs, the client waiting for the data from A' must wait for two disk accesses, first the write back of A and then the read of A'. This degrades the response time observed by the client processor.

- During periods of low disk activity, it is possible to slip a few write backs in to the disk access stream, opportunistically flushing the buffers (and possibly eliminating the need to force a *File Server* to write back those buffers).

The initial plan for the TASS write back process group was to have it monitor the disk accesses. When a lull appeared, indicated by a small list of pending requests or a fixed time passing since the last issued disk request, a *Write Back* thread would be dispatched to perform an opportunistic write-back disk access. Like I said, that was the initial plan. During the implementation of the TASS system, I needed to put the write back process group in place to test its operation. I had not yet determined the method whereby opportunisitic activity would occur. Being a bit of a hacker, I simply made the *WriteBack* threads operate at a lower priority than the *File Server* threads. No opportunistic check was made, the rule being *any time's a good time for a write-back, as long as no real cache accesses are currently underway*. Performance was atrocious. The *Write Back* threads quickly cleared the cache, and then proceeded to use even the slightest gap in *File Server* requests to tie up the *Cache Manager* with useless '*Anything to write back?*' queries. Even with the *Write Back* threads operating at low priority, the overhead imposed on the *Cache Manager* was sufficient to drastically degrade performance. As you may be aware, early failures never dissuade a good hacker. I thought, well, if I can prevent these streams of useless queries, then I can get on with testing the code for bugs. So, I simply forced each *Write Back* thread to sleep for a short period of time after being told the cache contains no dirty entries[29].

The performance obtained with the *Write Back* threads using this 'sleep' period is dramatically better than the performance observed when they were not using it. In fact, as will be discussed in Chapter 9, performance results using this method are so good that I have never felt the need to properly implement an opportunistic decision algorithm. Consequently, the write back process group still operates using the 'hack' described above. One very substantial advantage to this simple hack is that it is almost without processor overhead. A little communication overhead is imposed by the communications between the *Cache Manager* and the *Write Back* threads, but it does not appear to affect the I/O throughput of the system as a whole. In fact, the use of the write back process group will be shown to substantially improve TASS performance during sustained periods of write activity.

5.3.3.3. Predictor Process Group

A 'hook' is provided into which a *Predictor* process group could be inserted. This hook provides the basic communication service that would be required for the use of such a process group. Therefore, TASS users are able to design their own prediction algorithms, and hook them into the TASS system without worrying about communication considerations. Exhaustive testing of this component of TASS has not been performed.

[29] The time selected was approximately the time required for three disk accesses to be issued and completed. The reason for this selection was sort of random, chosen simply because it seemed like a good period of time. Some tests I have run have shown that I am either (*i*) incredibly intuitive in matters disk physical, or (*ii*) as lucky as an Irishman at a convention of leprechauns.

Due to the simplicity of the *n*-block look-ahead prediction algorithm, it was not implemented as part of an autonomous *Predictor* process, rather it is provided as an in-line code procedure call. This eliminates the inter-process communication that would be required between the *File Servers* and any *Predictor* processes. All tests performed on the TASS system are made while using either no prediction service, or this simple in-line procedure call variant. The interface provided by the procedure call variant is identical to that provided by the communication stubs (or 'hook').

5.4. RAM Cache Prototype

The CS2 prototype can be utilized as a RAM disk, or RAM cache (to differentiate it from the disk server versions). This is a compile directed option and is selected by choosing the RAM cache option from the list of disk joining options. The implementation is fairly straigtforward, with the only significant difference being that the after the *Cache Manager* has allocated all of the cache buffers, it must make a call to the disk support layer to set the number of disk chunks to the number of cache buffers allocated[30].

[30] The **SetNumberDiskChunks** operation is not normally part of the disk support layer interface. However, when the RAM Cache version is selected at compile time, this procedure is added. Additionally, the disk support layer's **ReadChunk** and **WriteChunk** procedures are replaced by do nothing procedures that simply return with a '*no error occurred*' error code.

6. The Client Interface

This chapter will be refreshingly brief. The TASS client interface provides a set of RPC stubs that allow processes operating within any processor directly connected to a TASS storage node, to access the mass storage system provided by that TASS node. This set of procedure calls are given in Table 3. As discussed in Section 5.1.4, the client interface, as it exists on the client processors, is the first region of the TASS system that is part of a distributed file system. Since this research is oriented to local file system design, the client interface does not fit within the scope of this thesis. Regardless of that fact, it is included here for reasons discussed in the upcoming paragraphs.

In Section 5.3.1 it was discussed that implementation of a simplistic RPC stub, where the client process obtains mutually exclusive control of both the outgoing and incoming communication links and maintains that control for the duration of the communication, sacrifices the inherent parallelism of the environment. In spite of this, that very technique was utilized for the client interface RPC stubs. There are three primary reasons for this decision,

(1) Some form of client interface is necessary to test the TASS system for both correct function and performance measurement. Also, other SFU transputer users require the TASS system to support their parallel sort research testbed [Atk92]. This technique presented the easiest path for providing a suitable interface.

(2) The client interface, although a component of the TASS system, is not a component of the TASS LFS. Therefore, a complex implementation is beyond the scope of this thesis.

(3) As discussed in Chapter 5 it is practical (and maybe necessary) to use multiple processes to implement an efficient RPC on top of an asynchronous communication protocol.

The reason that the third point is an issue stems from the requirements specified in Section 3.2. These state

- The object code size for linked LFS libraries (in other words the client interface RPC stubs) is kept minimal. Programs utilizing the disk system are not forced to allocate large portions of their local memory, in order to access the large disk memory.
- The number of client processor cycles required to execute a single disk access is minimized. Additionally, the number of context switches required due to processes blocking during execution of the LFS library has been kept to a minimum.

Implementing a set of RPC stubs using a method similar to that used in the cache server's disk support layer would violate both of these conditions.

In an effort to offset the penalties imposed by such a simplistic RPC implementation, the use of client directed prefetching has been heavily stressed. The interface has been tailored to easily provide sequential prefetching, often hiding it within the normal **Read Chunk** operation. This assumes that the cache server software has been compiled using the sequential access prediction service mentioned in Section 5.3.3.3. Because of this, the actual interface which is shown in Table 9, provides numerous variants on the basic read and write operations. There is no explanation of the procedures given, as the actual names are quite long and hopefully completely descriptive of the function. Each of the tass_read() procedures is equipped with a *PredictorArgument* parameter which is passed to the sequential prediction service and is used to indicate the value of *n* which is to be used for *n*-block look-ahead prediction. The interface also allows for the client to easily choose between the delayed write back and forced write through caching strategies. Any operation that communicates with the cache server processor will return an error code indicating what, if any, error occurred during the execution of the request. The predefined type TASSError (which is just

Table 9: The Procedures of the Client Interface

TASSError	tass_set_links()
int	tass_number_chunks()
int	tass_cache_size()
int	tass_chunk_size()
TASSError	tass_read_chunk()
TASSError	tass_read_chunk_sequentially()
TASSError	tass_read_chunk_sequentially_by_n()
TASSError	tass_read_partial_chunk()
TASSError	tass_read_partial_chunk_sequentially_by_n()
TASSError	tass_write_chunk()
TASSError	tass_write_partial_chunk()
TASSError	tass_write_through_chunk()
TASSError	tass_write_through_partial_chunk()
TASSError	tass_prefetch()
TASSError	tass_prefetch_n_chunks()
TASSError	tass_flush()
TASSError	tass_flush_and_invalidate()
TASSError	tass_shutdown()
TASSError	tass_cache_statistics()
TASSError	tass_detailed_cache_statistics()
TASSError	tass_disk_statistics()
TASSError	tass_detailed_disk_statistics()

an unsigned integer) is used for these errors.

Basic operation of the client interface involves first issuing a tass_set_links() operation, specifying the addresses of the in and out components of the serial link[31] which connects the client processor to the TASS storage node. This operation can be issued as often as desired

[31] For example, LINK0IN and LINK0OUT in Logical Systems C. All serial links in Logical C are accessed using the defined constants, LINK*i*IN and LINK*i*OUT where *i* indicates which of the four serial links, from 0 to 3, is to be accessed.

(although once per link is all that is necessary), and for as many serial links as are connected to TASS storage nodes, allowing one client processor to access as many as four TASS storage nodes (although, barring shared memory interfaces, that would effectively cut the client off from the rest of the world). The tass_set_links() operation returns the code for any error that was observed, plus a TASSLink indicator. Subsequent calls to the interface use this indicator to determine which of the connected TASS storage nodes is to be accessed. This means that the client need not constantly specify the serial link addresses, which serves to eliminate some of the problems that might occur if that were the case[32].

To help preserve data integrity, the client can issue cache flush instructions. An additional option to the cache flush is tass_shutdown(), which firsts issues a cache flush and then causes program execution within the TASS cache server to halt.

For performance analysis, it is necessary to monitor cache activity. For this reason, TASS has been modified to tabulate various statistics, for example, the number of cache hits and misses, the number of disk read and write accesses, etc. Statistics tabulation is a compile directed option. In other words, if the statistics are not required for the users application, the code and data structures required to amass those statistics are not generated within the cache server. This helps to both optimize TASS performance as well as maximize the memory available for the cache.

[32] It is fairly easy, I have found, to mistype LINK0IN and put something inappropriate like LINK0OUT or LINK1IN. It is also fairly difficult to find that sort of error, since it is the communication protocol and not the compiler that objects. Run-time errors, ya' gotta love 'em.

7. Device and Disk Server Performance

The primary motivation for this research (section 1.3) is to implement an efficient local file system for use within the transputer multiprocessor. The previous chapters overviewed the design of a LFS, discussing qualitative issues involved in the implementation of prototype versions of this LFS. However, the efficiency of the resulting system has not been analyzed. How much data throughput can the system provide to a network of transputers? How much throughput is lost due to software overhead? Are the processors controlling the LFS taxed to the point where a more elaborate file system would saturate the processing power of the system? These are all questions regarding the efficiency of the LFS and cannot be answered in a qualitative fashion, rather they require a more quantitative analysis.

The quantitative analysis (or performance) of the TASS LFS can be approached in terms of (*i*) *sustained data transfer rates* (SDTR) and (*ii*) *system software processor use* (SSPU).

Definition: The *sustained data transfer rate* (SDTR) of a mass storage device is the number of data bytes that can be transferred each second between the device and the main memory of a processor. Furthermore, that data transfer must be sustainable over prolonged periods of time[33].

Definition: The *system software processor use* (SSPU) within a processor operating as a component of a software system is the percentage of the available processor cycles consumed in the act of executing that software system.

The analysis of these two issues shows whether the limiting factor to TASS performance is

[33] It should be noted that all SDTR results between a transputer and a device (possibly another transputer) are determined without utilizing the transputer's on-chip RAM. Also, a prolonged period of time is usually interpreted as a small (integral) number of seconds.

the design of the system, or the physical hardware itself. Assuming that the bottleneck is the hardware, the analysis indicates what degree of hardware upgrade will be possible before the system becomes the limiting factor. If the system is the limit to performance, then the qualitative analysis indicates what regions of the software, and under what conditions, are the cause of the performance degradation.

Section 7.1 discusses the techniques used to approximate the data access patterns (DAPs) used within the performance testing. These approximation techniqes are defined here, and used throughout this chapter and the subsequent two chapters. Section 7.2 discusses the hardware limitations which indicate the upper bounds on TASS performance. Section 7.3 concentrates on the SDTR performance of the disk server software and compare those results to the hardware limitations described in Section 7.2.

7.1. Approximating Data Access Patterns

Techniques are presented for producing approximations of the two predominant data access patterns (DAPs) used in performance analysis of the TASS prototype. These patterns are Sequential Global Irregular (SGI) and Random Overlapped (RO). In every level of analysis that requires a particular data access pattern, I have attempted to utilize the same pattern generation technique without any modifications. Where the technique is modified, a brief overview of the modifications is provided, accompanied by a discussion regarding the effects those modifications have upon results obtained using the data access pattern.

For all access patterns, SDTR results are determined using S generated samples, each sample involving N read (or write) disk chunk operations. In situations displaying negligible variance in results, S may be kept small, possibly as low as 1 sample per test. The number of chunks operated on, N, will usually be large, in the neighbourhood of 200 to 1000 operations. The actual SDTR figures presented are determined by timing how long it takes to execute the

N disk chunk operations. Timing begins immediately prior to the first disk request being issued and stops immediately after the last response from the disk server is accepted. Let this length of time be *t*. The data throughput is then calculated as (**N** chunks * *B* bytes/chunk) / *t* seconds, giving the SDTR in bytes/second. All SDTR figures are actually given in KBytes/second, so this result is additionally divided by 1000 bytes/KByte.

7.1.1. Sequential Global Irregular

The sequential access SDTR figures for each of the TASS components is determined using SGI data access patterns. The SGI pattern minimizes average disk seek times since each disk access follows an access to the sequentially previous chunk. It is discussed in Section 7.3.2.2 that the chunk interleave factor and chunk size used with all the TASS disk servers is chosen to provide maximal data throughput for purely sequential access patterns. Therefore, any other access patterns can only exhibit larger average seek times, providing lower SDTR results.

Each of the **N** contiguous disk addresses accessed during each sample are operated on by one of the **P** worker processes. A process p, $0 \leq p < \mathbf{P}$ initiating a new operation accesses address n+1, where n was the previous address upon which an operation was initiated.

During each sample, the nth operation issued accesses disk chunk address n+a, where $0 \leq n < \mathbf{N}$ and a is the address of the first disk chunk accessed in this sample. In each of the **S** samples, the values of a are (approximately) evenly distributed across the entire disk address space. In other words, at the start of each sample, the first chunk accessed is not always chunk 0, rather it might be chunk 2000 for the second sample, then chunk 4000 for the third sample, etc.

To implement the above strategy, each worker requires access to a shared resource. This resource informs workers as to which disk address is next in global sequential order. This shared resource is implemented as either a shared procedure or an autonomous process, depending upon whether workers are resident on the same or separate processors, respectively.

7.1.2. Random Overlapped

The use of the RO DAP shows the minimum sustained data throughput that can be achieved using TASS[34]. This provides an indication of the lowest I/O bandwidth that each TASS storage node can provide to clients within the transputer network.

There are two stages involved in the generation of an approximate RO DAP. Firstly, a RO DAP is supposed to emulate random access to a file, therefore some contiguous region of the disk address space must be selected to be that 'file'. Often, the region chosen will be the entire disk address space, however, sometimes the number of chunks N in the *TASS file* (see Section 3.2.2.2) is chosen to be somewhat less than size of the entire disk. Once the *TASS file* size is chosen, it must be randomly distributed within the disk address space by randomly determining the first chunk (in the group of contiguous chunks)[35].

[34] Assuming that the availability of requests saturates the server, forcing it to access the disk constantly. Also, this assumes a reasonable disk access pattern. Some access patterns could maliciously choose disk accesses that would maximize the time spent performing disk seeks and display a lower SDTR than that displayed using the RO DAP.

[35] It would be poor scientific practice to test a random access to the same region of disk for all samples. It is possible that performance would be worse when the region accessed is varied. This may not be likely, but then again, it is not hard to stagger where the contiguous region begins.

Once the region of disk to use is determined, individual disk accesses to that region are made in a random fashion. Each address to access is generated using the formula

$$\mathit{CurrentAddress} = (\,\mathrm{rand}()\ \mathrm{modulo}\ \mathbf{N}\,) + \mathbf{F}$$

where **F** is the first chunk address in the disk region.

The rand() function does not provide a random sequence of integers, rather each time it is used it will generate the same sequence of pseudo-random integers. To provide more realism, every sample generated will utilize a different sequence of pseudo-random numbers by using the srand() seed initialization function[36].

7.2. Device Performance Limitations

7.2.1. Host Interconnect

Data transfer between a Sun host and the root of a transputer network has been documented in [Atk91]. Results given in that paper were obtained using a serial link interconnect which was driven by a Sun resident buffered polling device driver. That paper indicates the current I/O throughput between a transputer network and a Sun-3 or Sun-4 host is 634 or 699 KBytes/second, respectively.

[36] The srand() function is initialized using a new prime number each time. I don't believe prime numbers are required, but I took a number theory course and I have to apply it somewhere.

7.2.2. Transputer Link Throughput

There are many active users of the Simon Fraser University transputer environment. Since the transputer network must support the interconnections required by each user, each serial link on each transputer is connected to a CSA Part.7 crossbar switch, allowing arbitrary transputer interconnections to be made when each user's program is booted.

Sustained uni-directional transputer to transputer data transfers are expected to be no higher than 1.74 MBytes/second [Inm89]. Those results, however, are based upon transfers between each transputer's on chip RAM. The relatively small size of on chip RAM prompted the addition of an external 2 MByte, 3 cycle RAM to the CSA transputer boards. The SDTR between two transputers' external RAM is detailed in Table 10. The transfer rates given are for transfers through direct wire connections and also through CSA Part.7 crossbar switches.

The degradation of performance due to the crossbars is a consequence of CSA equipping each crossbar link with line driver hardware. The line drivers, intended to provide increased noise immunity, impose a (pessimistic) 45 ns delay on each serial communication [CSA89],[AMD81]. Coupling that with the crossbar delay of 1.75 bits/byte [Inm89] (which works out to approximately 88 ns), a combined delay of 133 ns occurs during each communication. As two signals, send and acknowledge, are required for every serial transfer, the total delay is in the neighbourhood of 266 ns. This approximation agrees reasonably with the

Table 10: Transputer to Transputer SDTR

Connection Medium	Sustained Data Transfer Rate (KBytes/sec)	Average Time to Transfer 1 Byte (nanoseconds)
Wire	1,666	600
Crossbar	1,243	805

805-600=205 ns difference shown in Table 10 for one byte transfers through both wires and crossbars.

Since we are utilizing crossbar switches within our hardware configuration, any SDTR results involving a transputer link have an upper limit of 1,243 KBytes/second. Where possible, relevant performance results using wire interconnections have been included to augment the results obtained using the crossbars.

7.2.3. CP-3100 Fixed Disk Throughput

The CP-3100 device can transfer data no faster than one 16896 byte track per rotation of the device. Since a rotation occurs every 16.8 ms (from Table 2), this means the absolute best performance obtainable would be about 1 MByte/second. This is a very optimistic estimate and the actual limit is probably much lower.

7.2.4. T800 Based RAM Disk Throughput

The maximum transfer rate for the RAM disk device is effectively limited by the serial link connecting the disk device with its clients. The previous section demonstated that this limit is 1,243 KBytes/second through the crossbar switches, and 1,666 KBytes/second through interconnecting wires. The actual performance of these devices is documented in Section 7.3.2.3.

7.2.5. Context Switch Time

The time required to execute a context switch within a T800 transputer is determined using two processes, each incrementing a counter and then forcing a context switch to the other process. Results from this software indicate that 7.9 million context switches can occur in a 30 second period. This means that the average time for a context switch is approximately 3.8 microseconds. These results are for two processes operating within the transputer's off-chip memory, rather than using the on-chip memory.

7.3. Disk Server

Disk server performance results given herein are for both the CSA Part.12 physical disk device and a T800 RAM disk device. Due to the simplicity of the RAM disk device, testing will concentrate on the physical disk device.

7.3.1. Testing Environment

Within SFU's transputer environment, it is impractical to determine the SDTR between disk and main memory of the Part.12's T222 transputer. Connecting a 16 bit T222 as the root node of the network would seriously inconvenience other users. Furthermore, debugging of test software *without* the T222 as root node would seriously inconvenience myself.

Data can be relayed from T222 memory into a connected T800's memory faster than that data can be read from disk. Since the transputer DMA serial link interface allows communication to oocur in parallel with computation, I allege that SDTR results obtained between disk and T800 main memory accurately portray the SDTR between disk and T222 main memory. Therefore, disk server SDTR is determined for transfers between disk and a client T800 transputer directly connected to the T222 disk controller.

The test software utilizes all the modules within the disk server software as well as the lowest module within the cache server software. This module, *DiskLink*, abstracts the link communication protocol from Table 7 into a set procedure calls that allow for requests to be issued and responses to be accepted. The overhead from these DiskLink procedures is considered to be insignificant since the DiskLink code is small and contains no computationally intensive regions. Two client processes operate above the DiskLink layer. One process, the *Requestor*, issues disk access requests while the other process, the *Acceptor*, obtains disk access responses[37]. Tests, which are not discussed herein, verify that one process handling each of these tasks is sufficient to provide maximal performance. An explanation for this is that requests are temporally spaced by at least the time taken for the disk to rotate once, which is 16.7 ms[38]. The processing time required by the disk server threads and the *Requestor/Acceptor* processes is insignificant when compared to those limits. Therefore, one process can issue disk access requests as rapidly as possible, while one process can completely handle any disk response before subsequent responses become available.

7.3.2. Test Results

Each of the following sub-sections details one particular aspect of disk server performance. Both read and write access results are presented for each of these tests.

7.3.2.1. Sector Interleaving

Varying the sector interleaving of the CP-3100 has no effect on performance, since the CP-3100 can only be configured with 1 block interleaving, where each block follows its

[37] These two processes are not to be confused with the *Requestor* and *Acceptor* process groups discussed in Section 5.3.1. Their function in communicating with the disk server is similar to that of the process groups, and they use identical communication protocols, however, they do not operate in the same fashion as the process group versions.

[38] The disk can transfer data no faster than one track per rotation, since it only has one RW head per platter and only one head can be active at any given time.

immediate predecessor[39]. This configuration is shown in Chapter 2, Figure 3(i). According to the SCSI protocol [ANS86], each manufacturer is required to provide 1 block interleaving and a default interleaving. Apparently, these two interleavings can be the same because the CP-3100 device, which conforms to the SCSI standard, has been implemented in this fashion. Because of this, there is no benefit to modifying the sector interleaving even though both the SBIC and CP-3100 will allow you to do so.

7.3.2.2. Track Interleaving

Track interleaving is implemented in the device dependent layer (DDL) of the disk server code. The degree of interleaving is controlled by a user-definable compile time constant.

Tables 11 and 12 show the performance of the TASS disk server, while controlling a CP-3100 disk device. Both the chunk size and the track interleaving are varied. Testing is performed for both read and write accesses, each test performing 200 of those operations during the timing samples. Disk chunk accesses are sequential, meeting the SGI data access pattern requirement[40]. The SDTR, in KBytes/second was determined in each case. For both read and write accesses, the highest SDTR is obtained when the chunk size is 16896 bytes and the track interleaving is 3. These results indicate that

- The expectation that using chunk sizes which span one entire track provide the highest data throughput, since any read (or write) transfers the maximum quantity of data bytes between

[39] With the exception that the first block of a track follows the last block.

[40] Actually, the pattern is SGR, however, with only one *Requestor* process, it is also vacuously SGI.

Table 11: Read Access SDTR for Various Chunk Interleavings

Disk Chunk	Interleave Factor										
Size (bytes)	0	1	2	3	4	5	6	7	8	9	10
5632	329	381	436	294	248	376	264	254	330	241	332
8192	478	442	429	295	397	275	377	274	346	332	325
16384	483	443	425	409	522	587	557	558	528	505	482
16896	502	472	669	**728**	667	617	573	535	502	472	446

Table 12: Write Access SDTR for Various Chunk Interleavings

Disk Chunk	Interleave Factor										
Size (bytes)	0	1	2	3	4	5	6	7	8	9	10
5632	330	241	439	295	249	376	265	396	331	241	406
8192	481	305	430	296	393	275	377	372	344	415	321
16384	499	457	438	422	526	561	577	571	545	520	485
16896	498	470	623	**712**	661	611	569	531	498	469	443

each disk head seek.

- The highest data throughput is obtained with a track interleaving of 3. This is probably due to the disk's average rotational latency being minimized with this track interleaving.

Because of these results, the performance tests presented in the remainder of this chapter are performed with 16896 byte chunks and a chunk interleave factor of 3.

7.3.2.3. Sequential Access

The sequential access throughput of the TASS disk server is shown for a SGI data access pattern. The fact that only one worker process is used to issue the data accesses effectively nullifies the global/local and regular/irregular distinctions between the various sequential DAPs. Therefore, the SGI pattern used is equivalent to any of the sequential patterns mentioned in Chapter 2. Ten samples of 1000 read (or write) chunk operations are provided in Table 13. The average of those ten samples is also given, as well as the expected uncertainty of the given average. The uncertainty figure provided is determined as half the difference between the high and low samples. The largest variance observed is 21 KBytes/second between the high and low samples for write access. The consistency of the read results indicates that larger samplings would not seriously alter the averages determined from the given samples. Increased sampling for write results might serve to refine both the average and the uncertainty.

The TASS RAM disk server results exhibited less variance than those given for the TASS fixed disk server. In fact, every sample taken for both read and write access was identical. The results, for both RAM disk server prototypes, are given in Table 14. They indicate that there is no loss in throughput due to software or communication overhead when utilizing the RAM disk 2 server (without bcopy). Recall that the RAM2 disk server avoided internal memory copying by transferring data directly from the RAM disk out the serial links. Referring back to Table 10, the maximum SDTR that can be achieved between two transputers is 1243 and 1666 KBytes/second for crossbar and wire interconnects respectively. Comparing

Table 13: Disk Server SDTR Samples and Averages

Operation	Samples					Average
Read	730	731	729	729	731	
	731	730	731	731	731	730±1
Write	712	702	700	702	702	
	695	701	707	691	693	701±11

Table 14: RAM Disk SDTR in KBytes/second

Operation	Transputer Interconnect	RAM disk 1 (with bcopy)	RAM disk 2 (without bcopy)
Read	Crossbar	1102	1206
	Wire	1503	1672
Write	Crossbar	1130	1229
	Wire	1465	1652

these figures to those given in the RAM disk 2 column, indicates that the second RAM disk prototype exhibits no throughput loss due to software or communication overhead. The RAM disk 1 implementation, using bcopy, can achieve no more than 89% to 92% of the maximum I/O bandwidth. The SDTR for a third 'implementation' was also determined. In fact, this implementation used the exact same code as the RAM disk 1 version, however, the bcopy() function was never invoked[41]. The use of this third implementation verifies that it is indeed the bcopy() function that is limiting the data throughput, and that it is not the modular design of the TASS disk server. The reason why it verifies that claim is that there was no significant difference between the TASS disk server 2 results and those for the 'disabled' TASS disk server 1 prototype.

This illustrates one of the many trade-offs between modularity and performance that constantly crop up in support system software design.

7.3.2.4. Random Access

SDTR results are presented in Table 15 for random access of entries distributed throughout the entire disk address space. Each sample is generated in the same fashion as for sequential access. Ten samples of 200 operations per sample were taken. Again, the uncertainty is simply the half the difference between the high and low samples. The random access results, for both read and write operations, are very consistent, indicating that they are accurate to a very small margin of error.

[41] In other words, all processing was exactly the same as in the RAM disk 1 version, but no data was actually transferred between the RAM disk and the transfer buffers. No data was ever copied from client to RAM disk or RAM disk to client.

Table 15: Random Access SDTR over Entire Disk Address Space

Operation	Samples					Average
Read	329	329	330	330	330	
	329	330	330	330	330	330±1
Write	313	313	313	313	314	
	314	313	314	313	313	313±1

Random access SDTR results for varying length databases are also presented. The graph in Figure 18 shows SDTR for random accesses while the size of the disk region accessed is varied. Although the data points comprising the graph are not explicitly indicated, each one was determined using 1 sample of 1000 disk accesses per sample. The random number sequence used is the same for each data point. The data points in the graph are not evenly distributed, rather concentrations of points are generated in regions where the graphs change dramatically.

The figure shows the performance of the TASS disk server for both read and write accesses. The sustained throughput is shown for two cases, (*i*) where the inherent parallelism of the request/response primitives are exploited and (*ii*) where a simplistic remote procedure call (RPC) interface is forced onto the disk server to client communication. The comparison between these two cases is intended to show the limitation of the RPC style interface.

Figure 18 shows that for most random read accesses using the request/response primitives, performance of the TASS disk server ranges between 330 and 400 KBytes/second, while write accesses vary between about 310 and 360 KBytes/second. Only when the size of the disk region being accessed is very small, in the range of 50 chunks, is a significant increase in throughput (due to lower average seek times) observed. When the region accessed drops to nearly 1 chunk, performance increases to as high as about 550 KBytes/second which is as expected, approximately one half of the maximum data transfer

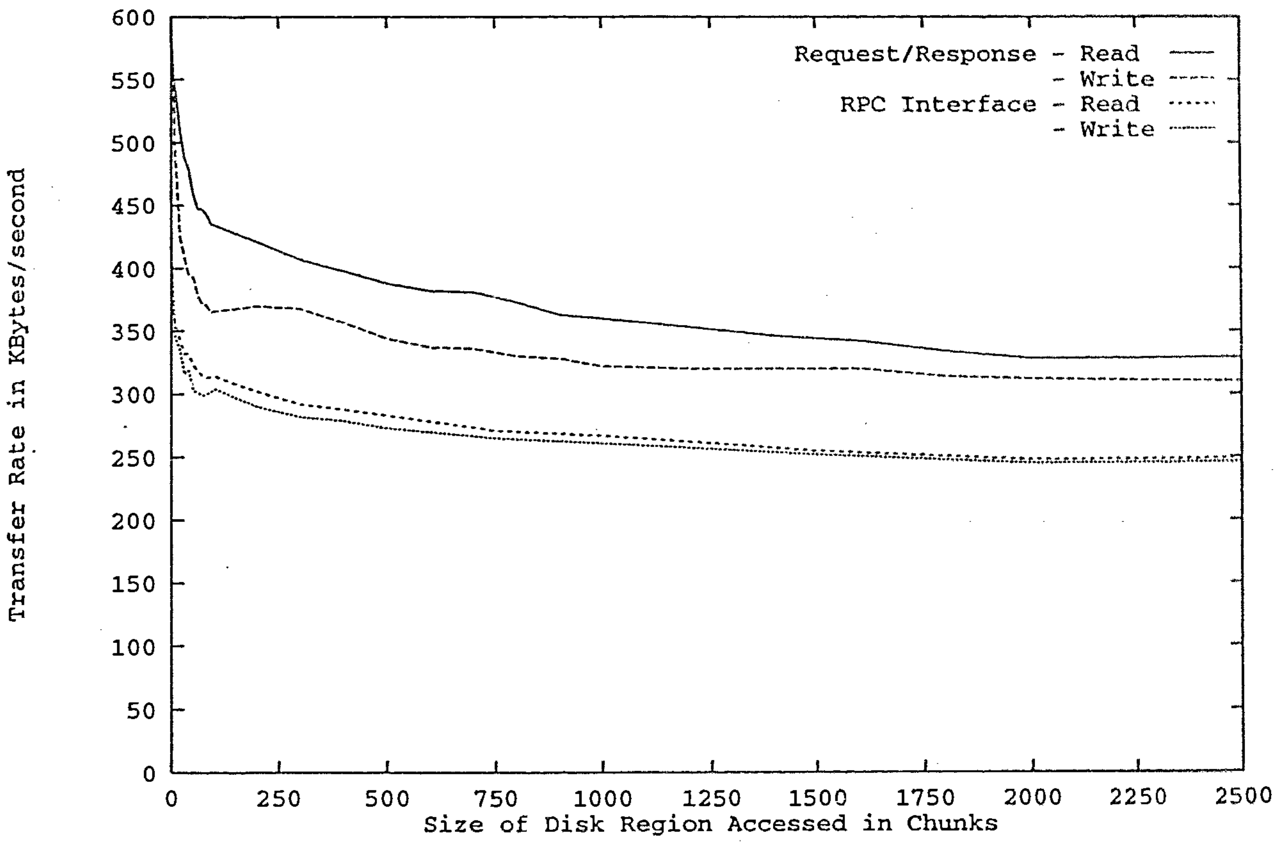


Figure 18: Random Access for Varying Sizes of Disk Region

rate for the CP-3100 disks[42]. When the RPC interface is used, it can be seen that I/O bandwidth suffers dramatically, dropping to around 80% of the bandwidth that is provided using the request/response primtives.

[42] The CP-3100 rotates approximately 60 times per second, therefore, 60 tracks could be read (assuming no seek time or rotational latency) in each second. This means that 60*16896=1,013,760 bytes could be read per second. Clearly, if we are reading a disk region of 1 chunk, half of our time will be wasted waiting for the disk to rotate around again, so one half of this maximum is the best that could be hoped for.

Memory is a random access device, so I expect that there will be no difference in the time taken to access a RAM disk sequentially or randomly. Therefore, no discussion is presented for random access to the RAM disk servers as I expect the results would be the same as that seen for sequential access (Table 14).

7.3.2.5. Processor Usage

The usage of disk server processor cycles is not calculated. The CSA Part.12 boards have too little memory to allocate additional work space for a process designed to determine free cycles. Adding such a process would reduce the disk server's available buffer space, reducing the number of buffers allocated and thereby degrading disk server performance. That performance drop would invalidate any obtained results regarding available processing power. It seems safe to assume that the limited memory space of the T222 processor, as well as the transputer serial links, are the bottlenecks to disk server performance.

7.4. Conclusions

In this chapter's preamble, a question was posed regarding what is the limiting factor to data throughput in the TASS disk server design. The results presented in this chapter show that the limiting factor of the TASS disk server is the physical disk device itself. This is a highly expected result, since it is fairly common knowledge that the limiting factor in I/O bandwidth is the physical devices, not the processor power. The RAM disk server results in Table 14 show that the disk server software can deliver data to clients at virtually the same rate as the hardware links are capable of (Table 10). Therefore, since the physical disk server software uses the same process structure as the RAM disk server, and its code is virtually identical (except for the small device dependent modules that control the actual disk devices), then the limits given for the disk server in Table 13 probably provide a fairly accurate claim as to the performance of the actual CP-3100 disk devices.

The small memory address space of the Part.12's T222 transputer does provide an effective limitation on the ability of the disk server to provide additional functionality. This is because the optimal size for the TASS chunk (with CP-3100 disks) is 16896 bytes, and to utilize buffering techniques requires a minimum of 2 buffers, and probably 3 should be used, to allow all three regions of the disk pipeline (see Figure 13) to operate concurrently. These buffers require between 33792 and 50688 bytes, effectively using all of the T222's available address space (recall from Chapter 3 that there is only about 52 KBytes of usable memory on the Part.12).

For random access, there is a slight improvement in SDTR for the disk server when the region being accessed is kept small. However, this performance boon is not that significant. From Figure 18, it can be seen that the improved rate of read access data transfers drops to about 430 KBytes/second when the region of disk being accessed reaches a span of about 50 chunks. This means that when less than 1% of the disk is being accessed, then performance can be improved to about 130% of the SDTR that would be observed if the entire disk were accessed. After that point, the SDTR drops almost linearly with the increase in the size of the accessed disk region, until the region is about 1/3 of the total disk size of 6208 chunks, at which point the SDTR graph levels off.

Also, it can be concluded that even one TASS disk server can provide more I/O bandwidth to the SFU transputer network than could be provided by the only technique previously available, which was by transferring data through the host interconnect. Under sequential access, the TASS disk server can provide 730 KBytes of data each second, whereas the host interconnect is only capable of providing the network with between 634 and 699 KBytes of data each second.

8. Cache Server Performance

This chapter details the performance of the TASS cache server software. Similar to the disk server performance discussion of Chapter 7, the analysis of cache server performance is determined according to two criterion, namely the *sustained data transfer rate* (SDTR) and *system software processor use* (SSPU). The I/O throughput that can be provided by the *Disk Support Layer* (DSL) and *Cache Layer* (CL) is shown (see Figure 11). The amount of processor cycles consumed by these layers is also discussed. No I/O throughput analysis is made of the performance of the *File Server Layer* (FSL), as that is too integrally connected with the client communication services, and as a consequence its performance is documented in Chapter 9, which covers performance of the TASS system as a whole. However, the CPU usage of the FSL will be discussed in this chapter, since it does not logically belong in Chapter 9.

8.1. Testing Environment

All tests within this chapter, unless explicitly stated otherwise, are generated using a group of identical worker processes which drive the layer being tested. The workers are responsible for issuing read (or write) data accesses to specific disk addresses, in a fashion that mimics the actual activity of the *File Server* and *Write Back* process groups (see Section 5.3.3). The addresses to use are provided to the workers through the use of a shared address generation procedure. The environment is shown in Figure 19. Since both the DSL and CL provide a procedure call interface to their clients, the operation of the worker processes is fairly simple, consisting of a procedure call to the shared address generator, followed by a procedure call to the layer being tested.

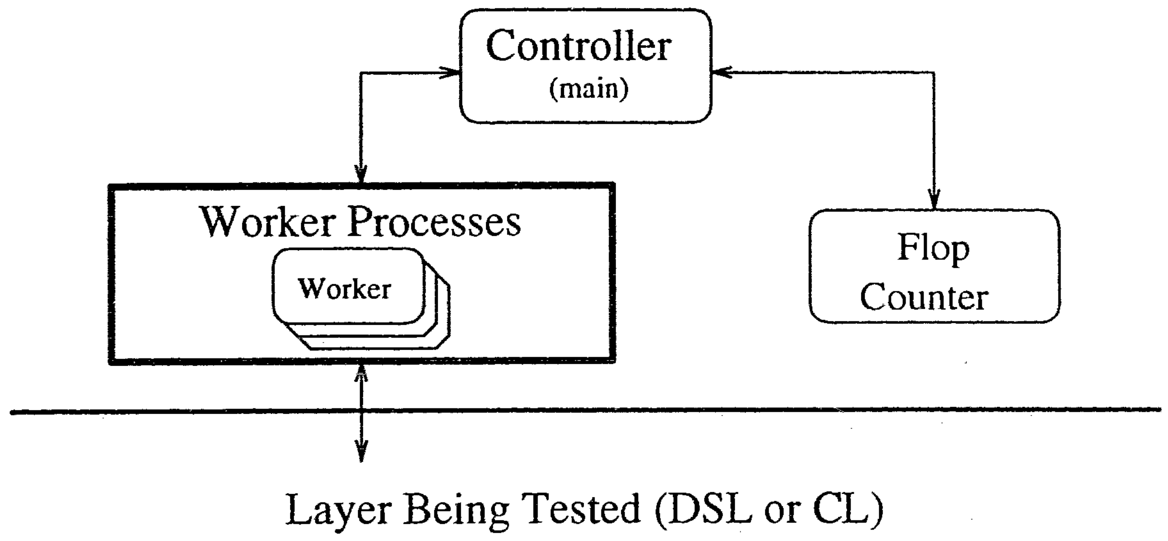


Figure 19: The Cache Server Test Environment

As each new layer of TASS software is added, the amount of processor cycles required to execute that software will be determined. The method used will be to create a low priority process, called the *flop counter*, which will execute whenever all other processes in the system are blocked. This flop counter will keep track of how long it has been running by counting the number of floating point operations that it can execute. Floating point operations will be recorded in the form of flops.

Definition: A *flop* will be defined, for the purposes of this paper, as one double precision addition accompanied by one double precision multiplication.

In order to provide some scope as to what a flop indicates, the temporal cost to execute one flop has been determined. Testing consists of numerous samples, each determining the number of flops that can be calculated on a T800 transputer within a period of 10 seconds. The averaged results are as follows:

Time Period	10 seconds
Number of Flops Executed	2,428,111
Time Required per Flop	4.1184278 microseconds

Therefore, the T800 performs at 0.24 MFlops. Inmos states that the T800 processor operates at 2.2 MFlops at 20 MHz [Inm89]. This disparity between Inmos' and my results arises because the *flop counter* process is coded in C, so in addition to the floating point operations described, numerous accesses to main memory, to obtain data and code, are also involved in the execution of each flop. Additionally, one branch operation is required. These additional components are probably responsible for the high cost involved in calculating each flop. Furthermore, memory support for all of my flop calculations is provided entirely by main memory; no use is made of the transputer's on-chip RAM which is three times faster.

The transputer architecture has only two priority levels, high and low. Because the *flop counter* must execute at low priority, in order to work correctly, there can be no other low priority processes running within the system. The only TASS processes that normally run at low priority are the *Write Back* processes (see Figure 17). Therefore, processor usage can only be determined in an environment where there are no *Write Back* processes. Consequently, the maximum and minimum CPU usage will be analyzed using only read access DAPs. If we assume that processor cycle usage will be a constant for each unit of data transferred (ie: same processing requirements for each chunk transferred), then the maximum CPU usage, in percent of available cycles, should occur for SGI access. Similarly, the minimum CPU usage will be given for RO access. The use of read accesses instead of write accesses is a bit of a simplification, however the I/O throughput results for both will be shown to be reasonably close. So, in terms of processor usage, the results for read and write access should be similar and that helps to justify the simplification.

8.2. Disk Support Layer

This section analyzes the components of the DSL that execute within the cache server processor. The DSL is responsible for completing the abstraction of multiple disks into an array of chunks, hiding the nature in which these disks are joined, and providing a procedure call interface which allows multiple clients to concurrently access all of the disks. The actual implementation of the disk support layer utilizes many modules, however, performance is only analyzed for the hierarchically highest and lowest which are the *DiskLayer* and *DiskLink* modules, respectively. When the performance of the DSL is discussed, it is in fact the performance of the *DiskLayer* module and all of its subservient modules and subservient disk servers, that is being analyzed. The I/O throughput and CPU usage of the entire DSL is determined by analyzing the *DiskLayer* module, which in turn utilizes all of the subordinate modules. Most of this analysis is centered on the performance exhibited while driving the DSL while it is configured under the various disk joinings. The processor cycle usage of the lowest module, *DiskLink*, is given since it represents the minimum CPU requirements for communicating with the disk server and provides a base CPU usage for comparison to the CPU usage of the entire DSL.

8.2.1. I/O Bandwidth

8.2.1.1. Optimal Workloads

As mentioned earlier, the I/O bandwidth of the disk support layer will be analyzed for each of the supported disk joining techniques. Each of the disk joining techniques provides differing data throughput results, depending upon the client workload. The *Sequential Global Irregular* (SGI) DAP discussed in Chapters 2 and 7 portrays the optimal client workload for striped and interleaved disk systems. The optimal client workload for concatenated disk systems is actually the *Sequential Local Disjoint Irregular* (SLDI) DAP.

The SGI DAP is generated in the fashion described in Section 7.1.1. The SLDI DAP generation is similar but involves two groups of clients sequentially accessing two independent regions of the disk address space. This test, although seemingly contrived for a two disk system, is actually representative of a typical SLDI workload for a TASS node. For example, a two disk TASS storage node will support either one or two clients. In the event that two clients each operate upon a disk region which is disjoint from that used by the other client, then this SLDI workload is quite representative of a typical workload.

The type of DAP issued by the worker processes does not uniquely define the client workload. Additionally, the rate at which the clients issue that DAP is an important consideration. As discussed in Chapter 4, the disk is a pipeline, and each stage of that pipeline must be kept busy at all times in order to achieve maximal performance from the disk device. Therefore, a single client issuing a request through a procedure call interface cannot possibly keep three pipe stages busy, since the one request will sequentially step through these stages. Because of this, the number of active requests, hence the number of active workers must be considered in the analysis of performance.

Tables 16, 17 and 18 show the performance of the DSL for the various disk joinings during execution of a SGI DAP, SLDI DAP and a RO DAP, respectively. Each entry in these tables reflects only one sample. The SDTR for the sample was determined by timing 1000 read (or write) chunk operations. Because of the single sample, the error margin for each table entry is potentially quite high. In other words, the numbers are soft, but not likely to be wildly inaccurate. From additional samples (to verify results) and from other tests performed (and not documented herein) I would estimate that the error margin is about ± 2 KBytes/second for read access and ± 20 KBytes/second for write access. In the discussion of this section, the term *equivalent* is defined as equivalent in data throughput, within these

Table 16: Disk Layer Throughput - Sequential Global Irregular DAP

(i) Read Access

Joining Technique	Number of Disks	Number of Workers							
		1	2	3	4	5	6	7	8
Concatenation	1	423	731	731	731	731	731	731	731
	2	424	731	731	731	731	731	731	730
Interleaved	1	421	727	730	730	730	731	731	731
	2	348	844	1172	1455	1457	1458	1460	1459
	3	393	750	1263	1422	1764	2183	1978	2184
Striped	1	423	730	730	731	731	730	730	730
	2	700	1460	1461	1462	1461	1461	1461	1461

(ii) Write Access

Joining Technique	Number of Disks	Number of Workers							
		1	2	3	4	5	6	7	8
Concatenation	1	424	715	711	714	713	715	709	717
	2	423	689	685	690	686	691	679	683
Interleaved	1	422	717	713	711	703	710	703	699
	2	372	842	1143	1415	1354	1384	1393	1252
	3	404	748	1262	1594	1835	2002	2003	1977
Striped	1	424	701	699	702	705	702	706	705
	2	596	1069	1357	1377	1374	1381	1383	1371

Table 17: Disk Layer Throughput - Sequential Local Disjoint Irregular DAP

(i) Read Access

Joining Technique	Number of Disks	Number of Workers							
		1	2	3	4	5	6	7	8
Concatenation	2	374	846	1015	1458	1457	1457	1456	1455
Interleaved	2	274	500	630	748	748	749	737	750
Striped	2	474	680	680	686	684	634	605	634

(ii) Write Access

Joining Technique	Number of Disks	Number of Workers							
		1	2	3	4	5	6	7	8
Concatenation	2	374	845	1162	1401	1387	1375	1411	1306
Interleaved	2	262	483	577	692	681	678	681	747
Striped	2	425	594	594	597	594	630	592	590

Table 18: Disk Layer Throughput - Random Overlapped DAP

(i) Read Access

Joining	Number	Number of Workers							
Technique	of Disks	1	2	3	4	5	6	7	8
Concatenation	1	248	329	310	309	309	301	304	304
	2	247	411	500	543	563	570	558	537
Interleaved	1	247	331	331	330	329	322	327	326
	2	247	496	636	655	646	654	652	647
	3	250	451	599	675	735	778	788	778
Striped	1	248	329	329	330	329	319	323	323
	2	483	658	657	659	658	633	643	644

(ii) Write Access

Joining	Number	Number of Workers							
Technique	of Disks	1	2	3	4	5	6	7	8
Concatenation	1	249	318	317	318	318	309	312	313
	2	246	400	481	524	537	541	527	502
Interleaved	1	240	310	312	312	311	305	309	307
	2	248	505	615	632	629	632	628	627
	3	247	446	583	665	705	713	739	719
Striped	1	249	318	318	317	318	309	312	312
	2	411	632	635	636	636	602	612	615

approximate error limits. Since the worker threads issuing the DAP operate within the cache server transputer, the I/O bandwidth from/to the **joined** disks is not constrained by the serial link throughput of 1225 KBytes/second presented in Table 10 (see Chapter 7).

From all three tables and for all three joinings, it can be seen that as the number of worker processes increases, the I/O bandwidth provided by the DSL appears to peak, and then level off, when there are two active workers per disk in the joining. The three exceptions to this rule are

(1) in the multiple striped disk environment, every request activates all disks in the joining. Therefore, one active worker issues as many requests to physical disk devices as would be issued by **D** workers in an interleaved or concatenated environment (where **D** is the

number of joined disks). So a total of two active workers is sufficient to keep all disks busy.

(2) SGI write access for two striped disks. The low performance for two workers operating on two striped disks is due to the fact that each *Requestor* issues the **D** requests to the **D** disks sequentially, rather than in parallel (Section 5.3.1). Because a write access involves a data transfer as part of the request operation (Table 7), the actual communication of the request to the disk server takes longer for write operations than it does for read operations. The non-maximal throughput for two workers, writing to two striped disks is probably a consequence of the sequentializing of requests. Two workers issuing read requests to two striped disks observe maximal throughput, since there is no data transfer component to the request, therefore, there is less time required to issue all of the requests sequentially.

(3) A SGI DAP issued within the multiple concatenated disk environment. In this case, the SGI DAP is only ever accessing one of the physical disks, so there is no disk parallelism and as such the results are equivalent to those for 1 concatenated disk, which means that two workers should be sufficient to keep the disk busy.

The fact that a minimum of two active workers per disk is needed to drive the DSL at its maximal throughput illustrates the pipeline nature of the disk system. However, it is apparent that the time taken to access the physical disk (Figure 13, Stage (*ii*)), is long enough to enable Stages (*i*) and (*iii*) to be completed sequentially (relative to each other), but in parallel to Stage (*ii*). In other words, in the time taken for request $\mathbf{R}_i$ to be issued to disk, the postprocessing of request $\mathbf{R}_{i-1}$ can occur and be sequentially followed by the preprocessing of request $\mathbf{R}_{i+1}$[43]. This means that during the time one worker is blocked awaiting completion of a physical disk access, another worker can observe the completion of one request and issue a new one.

There are no three disk results shown for striped or concatenated disk systems. This is for a couple of reasons:

(1) The software used to determine the results shown in the Tables 16, 17 and 18 is constrained to running on the *root transputer*, $\mathbf{T}_1$. The $\mathbf{T}_1$ transputer permanently allocates one serial link (link 0), for communicating with the network's Sun host (See Figure 9). Because of this, a three disk configuration would require a disk to be connected to the root transputer on each of its remaining links, 1, 2 and 3. The three disk results shown for the interleaved disk system were made early in the development of TASS, and at that time it was possible to connect, via the crossbar switches, one of the Part.12 T222 transputers to link 3 of $\mathbf{T}_1$. However, since then the network has been reconfigured and it is no longer possible to use the crossbars to connect any of the T222 transputers to link 3 of $\mathbf{T}_1$. To make such a connection would require a physical rewiring of the network and would probably inconvenience other transputer users. I do not consider these results to be important enough to warrant that risk. No three disk striped or concatenated results were obtained prior to the network reconfiguration.

[43] Since both postprocessing and preprocessing require the CPU, they cannot actually occur in parallel. However, once Stage (*ii*) has completed communication with the SCSI Bus Interface Controller (SBIC) chip, then any worker operating in that stage will not need the CPU again until the disk access completes. Therefore disk operation can be completely parallelized with the preprocessing and postprocessing components of the pipeline. Also, it is shown in [Inm89] that the throughput of a transputer link passing data simultaneously in both directions is not as high as twice the throughput of the link when it is passing data in only one direction. Therefore, even the communication components of the pre and postprocessing cannot totally be parallelized.

(2) Although joinings of three disks are supported by TASS, their practical use is limited. A three disk TASS node could have only one client, connected by the one remaining serial link. The 1225 KBytes/second bandwidth across that link (Table 10) becomes the limiting factor in performance. Even a two disk system operating at the maximum throughput of 1460 KBytes/second will saturate one serial link, effectively nullifying the bandwidth of the third disk[44]. For non-optimal data access patterns (ie: RO) a three disk system may be useful. It is expected that when operating under an optimal DAP workload, each of the disk joinings should be able to provide the same throughput in a three disk environment as are provided by the interleaved joining.

8.2.1.2. Comparing DSL Results to Disk Server Results

As Tables 16, 17 and 18 are rather busy, I have summarized the results in Table 19. Unlike the single sample per entry technique used in those three tables, each of the entries in

Table 19: Disk Support Layer Throughput in KBytes/second by Joining and Workload

Data Access Pattern	Operation	Single Disk	Two Joined Disks			Three Interleaved Disks
			Concatenated	Striped	Interleaved	
Sequential Global Irregular	Read	731	731	1461	1456	2187
	Write	704	686	1382	1375	1965
Sequential Local Disjoint Irregular	Read	351	1457	633	748	
	Write	297	1376	627	618	
Random Overlapped	Read	327	574	637	642	765
	Write	307	540	591	597	714

[44] Even a serial link with a wire interconnect having a bandwidth of 1666 KBytes/second (Table 10) will nearly be saturated by two disks.

Table 19 are averages of 10 samples, with each sample consisting of 1000 read or write operations. Three worker processes are generated for each physical disk used in each test (ie: If the sample tests a two disk joining, then 6 worker processes are generated, if the test is for a one disk joining then only 3 workers are generated). The choice of 3 workers per disk is prompted by the fact that a minimum of two workers per disk are required to demonstrate the DSL's full I/O throughput, and in some cases three workers are necessary (eg: 2 striped disks under SGI write DAP). For sequential samplings, the initial starting address is staggered throughout the disk address space. For random samplings, the seed used to initialize the random number generator is varied.

8.2.1.2.1. Single Disk Systems

Since all disk joinings provide equivalent performance for single disk systems, only results for the 1 interleaved disk joining are shown in Table 19, data column 1. The results in that column show that the I/O throughput provided by the disk support layer (DSL) controlling a single disk, is equivalent to the I/O throughput shown for the disk server devices in Chapter 7, Tables 13, 14 and 15. The only notable difference is a very slight drop in RO write throughput from 313 KBytes/second for the disk server, down to 307 KBytes/second for the entire DSL. This difference may still be due to sample variations, however, that can not be shown from the results given in this paper[45]. It can be seen that for SLDI access, the performance of the single disk system is very comparable to RO access. This is because two

[45] The margin of error is ± 1 KByte/second for the disk server results and ± 3 Kbytes/second for the DSL results. These give a range of 312 to 314 KBytes/second for the disk server and 304 to 310 KBytes/second for the entire DSL. The range of values for the two results do not intersect, but are very close. In fact, the 1 KByte/second margin of error for the disk server RO write access is the lowest margin of error I have observed for write accesses in general.

regions of disk are being accessed sequentially and due to the single disk environment, a large disk seek must be invoked for nearly every access. The SLDI read results may be slightly higher than the RO read results since it is more likely that two consecutive disk accesses are issued to sequentially contiguous disk chunks. If enough cases occur where 2 or more accesses are sequential, then this would have a positive effect on the I/O throughput of the disk system. A similar explanation can be given as to why the SLDI write results are lower than the RO write results. Since each disk request for a write access contains a data transfer, these requests take longer to issue, and fewer pairs of sequential accesses are likely to occur. Therefore, the actual SLDI write tends to have a large seek between every disk access, which is not true even for the RO write, which should have numerous small seeks due to the random pattern of accesses.

8.2.1.2.2. Multiple Disk Systems

8.2.1.2.2.1. Sequential Access

The two joined disk results in Table 19 show that there is no substantial loss in sequential I/O bandwidth for any joining when operating under the optimal DAP for that joining. The disk server results from Chapter 7's Table 13 show that sequential read performance is limited to a maximum of 731 KBytes/second. Therefore, two disks operating together cannot possibly provide better read I/O throughput than 2*731 KBytes/second = 1462 KBytes/second. The optimal DAP for concatenated disk systems is SLDI, where the concatenated joining provides a two disk read I/O bandwidth of 1457 KBytes/second which is equivalent, within error limits, to the 1462 KBytes/second limit. Similarly, the SGI DAP is optimal for both interleaved and striped systems and the corresponding results are 1461 and 1456 KBytes/second respectively, each being equivalent to the 1462 KBytes/second limit.

The three interleaved disks results of 2187 KBytes/second agree with the maximum of 3 * 731 KBytes/second = 2193 KBytes/second.

The write results show a very small clipping of performance, with the concatenated, interleaved and striped results being 1376, 1382 and 1375 KBytes/second, respectively (for each joinings optimal DAP). These results are about 98.4% of the expected 2 * 701 KBytes/second = 1402 KBytes/second (from Chapter 7's Table 13). It is important to note that the margin of error in write results is significant *and these four figures are equivalent within those error margins*, however, since all joinings provide lower than expected results I assume that a very slight clipping of performance is occurring. This clipping is sufficiently small and the error margins are so large that I have not been able to establish a reason for any clipping, assuming that it is there at all.

8.2.1.2.2.2. Random Access

The results for random access are not as impressive. A pronounced loss of I/O bandwidth is observed. For example, two disk servers can provide 2 * 330 KBytes/second = 660 KBytes/second RO read throughput (see Table 15). The performance demonstrated by the DSL is 574, 637 and 642 KBytes/second for the concatenated, interleaved and striped joinings, respectively. The interleaved and striped joinings achieve approximately 97% of the maximum 660 KBytes/second. However the concatenated results are significantly lower at 87% of this limit. Again, the interleaved and striped results might agree within error margins, but the fact that they are slightly lower tends to convince me that a loss of bandwidth is occurring. I cannot explain the dramatic loss of bandwidth for the concatenated disk systems, but this loss seems to occur at all levels of TASS. For example, the RO write results also show that the interleaved and striped joinings provide 95% of the maximum 2 * 313 KBytes/second = 626 KBytes/second (see Table 15). However, the concatenated results are about 86% of the 626 KBytes/second maximum. My only guess is that the random address generator used is somehow consistently distributing more accesses to either the lower or higher portion of the joined disk address space. This non-random distribution would have less effect on interleaved or striped disk systems since the concentration on one half of the disk address space would affect both disks equally. However, the effect on concatenated disk systems would tend to concentrate accesses on one disk, underutilizing the other. I believe that there is no reason why concatenated systems should behave worse for random access, if in fact that access is truly random. Therefore, I believe the only solution to the problem is the fact that a pseudo-random number generator is used. Coupling pseudo-random numbers with the truncation nature of the transputer's integer arithmetic could be responsible for such a consistent non-random sequence of 'random' addresses.

Three disk servers can provide 3 * 330 KBytes/second = 990 KBytes/second of read bandwidth and 939 KBytes/second of write bandwidth (from Table 15). However, the results for the three interleaved disks show distinctly lower I/O throughput of 765 and 714 KBytes/second for RO read and RO write, respectively. The explanation for this is a flaw in the testing environment and requires a detailed explanation. Figure 17 shows the process hierarchy of the CS2 prototype. There is only one pool of *Requestor* processes, with each *Requestor* accessing any disk, depending upon the requirements of the request it is servicing. While implementing the DSL software I ran some initial tests to determine what the best size of the *Requestor* process pool was. As I expected, since issuing requests does not take that long, these tests showed that one *Requestor* per disk should be more than sufficient to provide maximal I/O throughput. I believe that these low results for the three disk system illustrate that my initial expectations and tests were incorrect. Even though there are 9 workers issuing the DAP, there are only 3 *Requestors* available to relay those accesses to disk. What I assume is happening is that a sequence of three requests to one disk, which is all too possible in a RO DAP, tie up all three *Requestors*, leaving no *Requestors* available to access the other two disks. In fact, in a RO DAP it is not very likely that at all times one *Requestor* will be issuing a request to each disk. It is far more likely that two *Requestors* will be vying for the control of one disk, while one of the other disks remains idle. No size of requestor process group could guarantee that this sort of problem did not occur, but larger sized requestor process groups could offset the problem to a greater degree. Alternatively, a pool of requestors could be dedicated to each disk, however, implementing this environment would have been more difficult and was not attempted. For reasons mentioned earlier in Section 8.2.1.1, verifying these claims by running new tests on a three disk system with a larger requestor process pool is not appropriate.

8.2.2. Processor Cycle Usage

8.2.2.1. DiskLink Module

The *DiskLink* module is the lowest layer of the DSL. It provides a set of request and response procedure stubs that are used to both (*i*) issue actual disk service requests and (*ii*) accept disk server responses. The layer hides the nature of the communication protocol used, providing services of the **RequestReadChunk**, **RequestWriteChunk** and **AcceptResponse** variety. The reason why this specific module is documented here is that it was used during the performance tests shown in Chapter 7. It was not discussed within Chapter 7 since it is actually a part of the cache server code. In my opinion, the module represents the smallest (or very nearly so) set of primitives that can be used to intelligently converse with the disk server. Therefore, its I/O throughput performance is intrinsically part of the disk server performance. The throughput results for the *DiskLink* module **are** the throughput results for the disk server. However, the amount of cache server processor cycles used by the *DiskLink* module are not, and probably should not be, documented as part of the disk server performance analysis. Therefore, the CPU usage is documented here.

The processor power required by the DiskLink module is considered to be the minimum needed to converse with the disk server. It is provided in addition to the analysis of the disk layer's processor cycle usage (given in Section 8.2.2.2). The conversion from the request/response primitives provided by DiskLink, to the procedure call abstraction provided by the disk layer requires the use of additional process groups and data structures. The cost of providing this abstraction will be documented by a comparison between the results of the DiskLink and disk layer analyses.

The processor cycles required by the DiskLink module are calculated by averaging the results of ten samples for each device and data access pattern indicated in Table 20. The SGI read and SGI write samples use initial addresses which are distributed throughout the disk address space. The RO samples distribute individual accesses throughout the entire disk address space. Each sample performed 1000 operations, either reading or writing a chunk each operation. The time required to execute the operations, plus the number of flops calculated during each test are provided. From those results, the time required to calculate the flops is estimated, based on the figure of 4.118 microseconds per flop execution (section

Table 20: Processor Cycle Usage for DiskLink Module

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	23.1	5.4	1.0	4.5±1.6
	SGI Write	23.7	5.5	1.1	4.8±2.1
	RO Read	51.0	12.2	0.6	1.3±0.2
	RO Write	53.9	12.9	0.7	1.3±0.1
RAM1	Any	15.1	3.5	0.8	5.2±0.4
RAM2	Any	13.8	3.2	0.7	5.4±0.3

6.3.5). The time required by the DiskLink software is determined using the total time and the estimate of flop calculation time. From that result, the percentage of total time required to execute the DiskLink software can be determined.

The third data column, *Seconds Used*, shows how much processor time is required to issue and accept the 1000 requests and responses. For all devices and all DAPs (except the CP-3100 SGI DAPs), this time is constant at approximately 0.7 seconds, which translates to about 700 microseconds of CPU use for each request/response pair. The reason why more time is used during a SGI DAP on the CP-3100 disks than is used for the RO DAP is not known and is under investigation. The high margin of error in the SGI samplings makes such an investigation quite difficult. There is no obvious reason why the SGI DAPs should require more processing power than do the RO DAPs, since the only difference in the tests is whether the current address is set by a call to rand() for RO DAPs, or incremented by 1 for SGI DAPs. Neither operation should require a great amount of processing power, and in fact, the call to rand() should be the most expensive of the two. In the rest of this chapter I make the assumption that the SGI DAP figures given in Table 20 are too high.

The actual percentage of available processor cycles used ranges from 1.3% for RO accesses to the CP-3100 disk server up to 5.4% for accesses to the RAM1 and RAM2 disk servers. It is important to note that these figures are ballpark figures, since the error margin ranges from 0.1% for the RO write results up to 2.1% for the CP-3100 SGI write results. The reason for these high error margins is the high variance in the sampled values for the number of MFlops calculated. Since the number of seconds used is calculated directly from the MFlops figures, the error margin in the flop count extends to the seconds used figures. This might explain why the number of seconds used during the SGI DAP tests is higher than for the other tests. If we use the more likely value of 0.8 seconds for the SGI DAP results, instead of the 1.0 and 1.1 seconds shown in Table 20, then the percentage overhead would be

3.5% for read access and 3.4% for write access.

CPU usage is shown for both read and write accesses, contrary to what was said in Section 8.1, in order to provide some justification for the claim made in that section that the CPU usage for SGI read will be equivalent to that for SGI write and that RO read usage will be similar to RO write. No further CPU usage results for write access will be given in upcoming sections.

8.2.2.2. DiskLayer Module

Table 21 shows the processor cycle usage for the disk support layer. This table is broken down into four portions, namely (*i*) one disk, (*ii*) two concatenated disks, (*iii*) two interleaved disks and (*iv*) two striped disks. This table has few surprises since in all cases the percentage overhead for the DSL is higher than for the *DiskLink* module (see Table 20). There are no substantial increases in the CPU usage, which is expected since the DSL was implemented without the use of busy waiting or any processor intensive operations. The fact that the total time, number of flops and seconds used results given for striped disks are approximately twice as large as the corresponding results for interleaved and concatenated disks is not an unexpected result. Recall that each chunk access in the two striped disks environment requires communication with both disk servers whereas each access in the interleaved or concatenated environments requires communication with only one disk server. Therefore, the greater CPU usage in the two striped disk environment is simply indicative of the fact that twice as many disk requests are prepared and twice as many disk responses are accepted than would be prepared and accepted in the other joinings. This is verified since the percentage overhead for striped disks is nearly equivalent to that of the other joinings.

Table 21: Processor Cycle Usage for the Disk Support Layer

(i) One Disk - Any Joining

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	23.4	5.4	1.1	4.6±1.8
	RO Read	51.7	12.2	1.6	3.1±0.1
RAM2	Any	14.1	1.6	1.0	7.5±0.1

(ii) Two Concatenated Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SLDI Read	16.0	3.7	1.0	6.4±0.3
	RO Read	29.0	6.8	1.3	4.2±0.1
RAM2	Any	8.2	1.8	0.9	11.3±0.7

(iii) Two Interleaved Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	16.8	3.7	1.0	6.2±0.3
	RO Read	26.0	6.0	1.2	4.5±0.1
RAM2	Any	7.4	1.6	0.9	11.9±0.7

(iv) Two Striped Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	32.5	7.5	1.8	5.5±0.2
	RO Read	54.4	12.7	2.2	4.0±0.1
RAM2	Any	14.8	3.2	1.6	10.9±0.3

8.3. Cache Layer

At this level of the TASS analysis, I have decided to not include results for concatenated disk systems. I believe that consistently showing SGI DAP results for striped and interleaved disk systems, as well as SLDI DAP results for the concatenated system is confusing to the reader and leads to a poor flow within the discussion. Presenting SGI DAP results for the concatenated system is not interesting, since it represents a non-maximal, non-minimal DAP for that architecture. Some tests which I have made (but do not include in this paper) show that the two concatenated disk system performs equivalently to the striped and interleaved systems, when each is operating under the optimal DAP for that joining. The only difference is that the consistently poor RO DAP performance of the concatenated systems, which was discussed in Section 8.2.1.1, still manifests itself at this level of the analysis.

Additionally, results for the RAM disks will also be discontinued. At this level of the analysis, the size of the cache, at about 119 chunks (for the interleaved joining) is nearly the size of the RAM disk at about 123 chunks. Therefore, using a RO DAP would result in a high percentage of cache hits, consequently invalidating the results. Using a SGI DAP would require the workers to wrap around from the last disk chunk back to the first disk chunk. This would probably produce accurate results if the LRU replacement managed to flush out each chunk just before it was required again. Due to the internal workings of the TASS system, this could not be completely guaranteed for a single RAM disk system, so it was not attempted.

8.3.1. I/O Bandwidth

8.3.1.1. Optimal Workload

When issuing a read oriented SGI DAP, there is no need to consider the effect of the actual cache on SDTR results. Regardless of whether the cache is empty or full (assuming no cache hits and no dirty entries), then each address request in the SGI DAP will miss in the cache and a free clean buffer will be available for reading the data into. So, using an empty cache is the same as using a full cache. For write oriented DAPs, we cannot start with an empty cache, since we are trying to show **sustained** data transfer rates. Therefore, we don't want the first **C** chunks (where **C** is the size of the cache) to be written to the cache without any corresponding disk access. Under a sustained SGI write, the cache would always be filled with dirty chunks, so we must pre-fill the cache with dirty chunks, prior to beginning the timing tests. Additionally, the dirty chunks selected for pre-fill must be chosen so that no write access will hit within the cache on one of those pre-filled addresses.

The performance of TASS for the interleaved and striped joinings, while the cache layer is being driven by a varying number of worker processes, is shown in Table 22. As was seen in the disk support layer results, there is a minimum requirement of two active workers, per component disk, to achieve maximal throughput for the interleaved joining. For the striped joining, two active workers can drive the system at its maximal throughput during read access. However, for write access to a two striped disk environment, three workers are still required. This was discussed in Section 8.2.1.1. It is important to note that the addition of

Table 22: Cache Layer Throughput - Sequential Global Irregular DAP

(i) Read Access

Joining	Number	Number of Workers							
Technique	of Disks	1	2	3	4	5	6	7	8
Concatenation	1	407	720	723	717	717	722	722	722
	2	408	722	723	726	725	725	724	726
Interleaved	1	418	730	729	729	730	730	730	730
	2	354	833	983	1454	1320	1455	1440	1456
Striped	1	423	730	730	730	730	730	730	730
	2	707	1453	1455	1452	1453	1458	1454	1450

(ii) Write Access

Joining	Number	Number of Workers							
Technique	of Disks	1	2	3	4	5	6	7	8
Concatenation	1	418	585	583	585	587	586	588	589
	2	418	586	586	584	589	586	588	587
Interleaved	1	418	583	588	588	586	586	589	587
	2	373	831	1116	1155	1126	1140	1151	1133
Striped	1	417	584	582	581	583	586	586	581
	2	595	1043	1159	1164	1168	1160	1168	1164

the new software layers (the DSL and CL) increase the length of the pre-processing and post-processing stages of the disk access pipeline. Although, only SGI write to two striped disks is currently affected, this increase in pipe length cannot continue without ramifications. This problem is only just starting to show, since it only affects one joining and one type of access. The addition of the the DSL, using the *Requestor* and *Acceptor* process groups, adds an internal software-based communication to the length of the pre-processing and post-processing components of the disk access pipeline. These communications are necessary to relay worker requests to the *Requestor* which passes them to disk and also to relay responses from disk to the worker, via the *Acceptors*. Recall that the reason two workers are sufficient to drive the disk server and the DSL is that one worker (and other processes working for it) can completely execute both the post-processing and pre-processing pipe segments (on two separate requests) during the same time another worker is in the disk access segment (see Chapter 4). From the results in Section 8.2.1.1, the increase in pipe length due to the DSL is not enough to prevent one worker from executing the two pipe segments. However, adding the cache layer requires additional pre-processing and post-processing communication with the *CacheManager*. The reason why SGI write access to two striped disks is the first DAP to be affected is that (*i*) SGI access has the shortest disk access component of the pipeline, (*ii*) write access transfers data between cache server and disk server in the pre-processing segment of the pipe and (*iii*) the fact that the striped disk implementation sequentializes the issuing of requests to the component disks (see Section 5.3.1) means that the pre-processing pipe segment is significantly longer for that particular DAP than it would be for any other DAP. Read results are not affected since the data transfers from the disk server to the cache server are more parallelized due to the use of multiple acceptor pools (one pool per link) which allow all data transfers to occur simultaneously (depending upon how good the transputers DMA hardware is).

The results for the RO DAP are shown in Table 23. There are no suprising results in these tables. Again, two active workers per disk are enough to keep the system flowing at its maximal RO bandwidth. In the RO DAP, the time taken for a request to complete the disk access portion of the pipeline is very large, by comparison to an access under the SGI DAP. Therefore, the fact that the cache layer has increased the length of the pre-processing and post-processing segments of the pipe is not significant.

As was done for the disk support layer, I have summarized the results for the cache layer in Table 24. Each entry in this table is an average of ten samples, each sample involving 1000 data accesses. For write access the cache was prefilled and for read access the cache was initially empty. The results shown in this table are discussed in the next two sections, 8.3.1.2 and 8.3.1.3.

Table 23: Cache Layer Throughput - Random Overlapped DAP

(i) Read Access

Joining	Number	Number of Workers							
Technique	of Disks	1	2	3	4	5	6	7	8
Interleaved	1	246	330	329	329	330	329	322	322
	2	244	488	629	647	653	646	654	651
Striped	1	247	331	331	330	331	330	323	324
	2	462	655	656	657	658	656	638	642

(ii) Write Access

Joining	Number	Number of Workers							
Technique	of Disks	1	2	3	4	5	6	7	8
Interleaved	1	246	317	317	317	317	310	312	315
	2	242	473	577	602	609	610	603	586
Striped	1	247	318	317	317	317	308	311	314
	2	399	618	618	618	619	601	595	599

Table 24: Cache Layer Throughput in KBytes/second by Joining and Workload

Data Access Pattern	Operation	Single Disk	Two Joined Disks	
			Interleaved	Striped
Sequential Global Irregular	Read	721	1445	1445
	Write	584	1124	1162
Random Overlapped	Read	330	650	656
	Write	317	616	610

8.3.1.2. Sequential Access

Table 24 shows a noticeable drop in SDTR for SGI access patterns. For example, the DSL provides SGI read access from a single disk at 730 KBytes/second, however the cache layer supports only 98.8% of that figure, namely 721 KBytes/second. Single disk SGI write access has dropped to 584 KBytes/second, which is 83% of the DSL's 704 KBytes/second. I/O bandwidth reduction also occurs in the two disk environment, with read access dropping to 1445 KBytes/second (both joinings) as compared to the DSL's figures of 1456 and 1461 KBytes/second for interleaved and striped disks respectively. This constitutes a 1% reduction in SDTR. The cache layer's two disk write access SDTR is 82% and 84% of the DSL's figures, for interleaved and striped systems.

The reason for this significant loss in SDTR is intrinsically involved with the operation of the cache server software. Most of the processes in the TASS cache server are event driven, where the processes block on some communication channel and await input[46]. The *File Servers* and *Acceptors* are driven by events that are signalled through the hardware serial

[46] The *Write Back* process group is not event driven. Rather it decides when to act and effectively causes an event for the *Cache Manager*.

links. In addition, since the *File Server* and *Acceptor* process groups are divided into pools, each allocated to a particular hardware link, the individual processes know where their next communication will come from (namely the hardware link they are dedicated to). The *CacheManager* and *Requestor* processes are also event driven, but their events are signalled by the *File Servers* and *Write Back* process groups, via internal communication channels, often called soft links. Neither the *CacheManager* or *Requestors* know, in advance, what other process will signal them, in indication of an event to be serviced. Therefore, they must be able to block on a large number of soft links, returning to the active state only when one of those links becomes active. Fortunately, Logical Systems C provides the ProcAltList() function that performs just this task. Therefore, extensive use was made of ProcAltList() in the TASS cache server design. Unfortunately, ProcAltList() has a few drawbacks. The order of soft links within the list given to ProcAltList() defines an implicit prioritization of requests. If two soft links in the list go active before the process calling ProcAltList() can awake, then the soft link appearing earliest in the list will **always** be serviced first. This means that the client thread (either *File Server* or *Write Back*) that is earliest in the list will has a higher effective priority than all the other threads in those groups.

Although the DSL interface is part of the problem, the problem did not noticeably affect the DSL results for the following reason. Each worker using the DSL blocks until its outstanding disk access completes and the response from that disk access is returned to the worker. Immediately after that the worker issues a new request with little to no lag due to computation (the worker process is a very simple one involving a loop containing the two operations **GetAddress** and **ReadChunk**). Because of this, the actual requests issued to the DSL are temporally spaced by about the time needed for a single disk access. Therefore, there is plenty of time for a *Requestor* to pick up that request and issue it to disk before any new requests can be issued by a different worker. This leaves little chance that the

ProcAltList() prioritization will affect the *first-come, first-served* (FCFS) request order. The worker processes in the cache layer tests do a great deal more work (considering the communication with the *CacheManager* et al.) and so the lag between obtaining the DSL response and issuing a new DSL request is larger. Therefore, when including the cache layer in the software, there is a greater chance that the DSL will aid in the shuffling of the SGI requests.

Now back to the question as to why the SDTR for the write and read accesses has dropped within the cache layer. What is shown by the results is that the implicit prioritization of ProcAltList() is shuffling the first-come, first-served ordering of worker requests, both at the junction between the *CacheManager* and at the *Requestors*, so that by the time the requests are issued to disk they are no longer in the optimal sequential ordering. Since no disk scheduler exists in the TASS system (see Section 4.1.1), this problem cannot be rectified and sub-maximal I/O bandwidth is the consequence. The reason why the loss in throughput is more pronounced for write access, dropping 18% from 1375 to 1124 KBytes/second as opposed to the read access loss of 1% from 1456 to 1445 KBytes/second (for interleaved joining), is that the worker threads are issuing the read requests to the addresses indicated by the SGI DAP, and as discussed earlier, there is a slight temporal spacing between individual requests, due to the physical disk access. Therefore, the read accesses do not get shuffled that readily. However, during write accesses, the actual requests issued to disk are in fact write backs of earlier write requests and are selected by the *CacheManager* based on the LRU ordering of the cache list itself. Since writing an actual request to cache does not take that long, and is first preceeded by an entire disk access including post-processing, there is a much larger chance that the order in which the write requests eventually get executed to the cache will not be the order indicated by the SGI DAP. Then, when those entries are returned to disk, as a delayed write-back, that shuffling hidden within the LRU order will degrade I/O throughput.

8.3.1.3. Random Access

The I/O bandwidth through the cache layer, when a RO DAP is being executed, is equivalent to the performance provided by the DSL, as shown in Table 19. In fact, a very slight increase in throughput is shown for every configuration and every joining. For example, the two interleaved disk results for the cache layer show a bandwidth of 650 KBytes/second for read access. The results for the same configuration through the DSL is only 642 KBytes/second. This is an increase in bandwidth of about 1%. However, the CL and DSL throughput figures are equivalent within the error margins of the samples. Due to the high error margins in the samples (around ± 15 KBytes/second), I am hesistant to claim that the 1% increase in throughput is an indication that some disk accesses are hitting within the cache. However, considering that two interleaved disks have 12416 chunks and the size of the cache is 119 chunks, then each disk access has about a 1 in 100 chance of being a hit. So on 1000 disk accesses we should score about 10 hits, which should give about a 1% improvement in throughput.

It should also be noted that the problem of shuffling within the disk request order does not affect the RO DAP, since there is no way to randomize a random ordering.

8.3.2. Processor Cycle Usage

The CPU usage figures for the cache layer are shown in Table 25. The results show a significant increase in CPU usage, over the corresponding DSL figures, of between 4% and 6% for SGI DAP access. The RO results also show about a 3% increase in CPU use, except for the two interleaved disk joining which seems to have dropped slightly. I don't believe that the CPU usage has gone down in this case, rather it is probably an erroneous result. As I

Table 25: Processor Cycle Usage for the Cache Layer

(i) One Disk - Any Joining

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	23.6	5.2	2.1	9.0±0.1
	RO Read	51.0	11.6	3.2	6.2±0.1

(ii) Two Interleaved Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	12.2	2.6	1.2	10.2±0.6
	RO Read	26.1	6.1	1.1	4.2±0.1

(iii) Two Striped Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	23.6	5.1	2.7	11.4±0.7
	RO Read	51.5	11.6	3.8	7.3±0.1

have tried to stress, these CPU usage results are very soft and should be taken with a grain of salt[47].

8.4. Server and File System Layers

The I/O bandwidth performance of these two layers will be documented in Chapter 9 as part of the TASS storage node performance analysis. The operation of these layers is too inter-related with the client communication to provide a bandwidth analysis from 'inside' the

[47] In fact, I am convinced that the actual margin of error for these CPU usage results is probably higher than the ones I have listed in the tables. However, it has been a long time since I used complex statistical oriented margin of error calculations and because of that I have used more simplistic methods that probably do not capture the variance in figures as accurately.

cache server processor. However, the processor cycle usage is documented in this section, since it logically belongs in the cache server analysis.

8.4.1. Processor Cycle Usage

The processor cycle usage for the server and file system layers is obtained using a different technique than was described in Section 8.1. Here external client processors are used to issue the DAP. The final results from the flop counter are passed to one of these clients and printed by that client. The total time required to execute the DAP was determined on one of the client processors. Because of the fact that the two vital pieces of information, namely the total time and the flop count, were determined on separate processors leads me to believe that the error margin of these samples should be much higher than what is indicated in Table 26. As expected, the processor cycle usage figures for the server and file system layers, shown in Table 26, are either equivalent or slightly higher than the corresponding results for the cache layer. The only exception being RO read access to the two striped disk configuration. As stated earlier, I believe that either this result is a little too low, or the corresponding striped disks RO result from the cache layer is a little too high.

8.5. Conclusions

The disk support layer component of the cache server software effectively relays data between the subservient disk server(s) and the supervisory cache layer. The layer hides the request/response nature of the disk server/cache server communication protocol, abstracting it into a more programmer friendly procedure call interface. No appreciable loss in I/O bandwidth occurs due to the addition of this software layer. The results for two striped disk SGI write access hint that the 'length' of the disk access pipeline of Figure 13 is increasing, since the number of active requests required to achieve maximal throughput increases from 2

Table 26: Processor Cycle Usage for the File System and Server Layers

(i) One Disk - Any Joining

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	23.4	5.1	2.4	10.3±0.1
	RO Read	51.2	11.9	2.1	4.0±0.1

(ii) Two Interleaved Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	14.9	3.1	2.2	15.0±0.1
	RO Read	46.3	10.8	1.9	4.1±0.1

(iii) Two Striped Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	28.5	6.1	3.2	11.2±0.1
	RO Read	53.4	12.3	2.9	5.4±0.1

to 3 within the DSL. However, the growth in the pipe length does not affect the I/O bandwidth for any other DAPs or disk join techniques. It is likely that the problem (which affects two striped disk SGI write) is due to the sequential way in which component disk requests are issued to the subordinate disk servers in the striped joining[48] This problem could be rectified if a technique, with low communication and computation requirements, could be encoded to allow parallelization of the individual disk request communications within the striped environment.

[48] In other words, each disk access issued to the DSL will require a disk access being issued to each of the component disks in a striped disk joining. In the current TASS implementation, these component requests are issued sequentially, rather than in parallel.

All three joining techniques, concatenating, interleaving and striping, provide equivalent I/O bandwidth regardless of the number of subservient disks. This assumes that an optimal DAP, for that configuration, is being issued. The only exception is two concatenated disks, when operating under a RO DAP. In this case, I/O bandwidth is significantly lower, but since there is no obvious reason for this I believe it to be an unlucky side-effect of the pseudo-random number sequence used.

The I/O bandwidth of the TASS system starts to degrade with the introduction of the cache layer. This loss of throughput is localized to the SGI DAPs and is due to scrambling of the disk request sequence. This randomizing of the SGI sequence is caused by the implicit prioritization imposed by the ProcAltList() construct of the Logical C compiler. Throughput degrades because any change in the optimal sequential request sequence effectively increases the average disk seek time for the entire DAP, which increases the average time required to execute each request, therefore reducing I/O bandwidth.

Even though additional inter-process communications are used to coordinate the activities of the *File Server* threads and the *Cache Manager* thread, the additional time required to perform this communication does not yet seem to be seriously affecting the performance of the TASS system. However, as for the DSL, the requirement for three active requests (rather than two) in order to obtain maximal write bandwidth for two striped disks is still apparent in the results given in Table 22.

There is some indication that the cache layer has a positive effect on I/O bandwidth under a RO DAP. RO throughput at the cache layer appears to have increased by about 1% over the DSL's RO bandwidth. However, the large margin of error in the results prevents verification of this fact. Considering that the size of the cache is 1% of the entire disk address space in a two disk joining and 2% of the disk address space for a one disk joining, the expected increase in bandwidth should be about 1% or 2%.

When CP-3100 physical disks provide the system's data store, the DSL uses no more than 7% of the T800 cache server's available processing power. If higher bandwidth disks are used, the results obtained using RAM disk servers, indicate that the CPU utilization will increase to an upper limit of about 12% of the available processing power.

The TASS cache server software, as an entire unit, utilizes a fairly small portion of the T800's available processing power. The largest use of CPU cycles is about 15% of the available processing power and that occurs with two interleaved disks, operating under a SGI DAP (see Table 26). This leaves, in the worst case, about 85% of the available processing power free for execution of more complex code segments, such as file systems, write back opportunity determination, access pattern predictions, etc. This allows ample opportunity for enhancement of the existing TASS system with little worry regarding loss of performance due to overworking the cache server processor. Each new primary level of the TASS cache server software uses between 1% and 5% of the available processing power, depending upon the number of subservient disks and nature of the DAP. As expected, the CPU use analysis of the entire cache server software (Table 26), shows that the two striped disk environment requires less processing power than does the two interleaved disk environment[49] since each client request transfers twice as much data with the same amount of work. However, within the DSL the two striped disk system does require twice the work, per request, so the CPU usage for a two striped disk system is slightly higher than is required by the one disk environment.

[49] Although not shown here, the CPU usage of the concatenated disk system is comparable to the interleaved environment, when a SLDI DAP is substituted for the SGI DAP.

9. TASS Storage Node Performance

This chapter discusses the performance of the TASS storage node as a unit. The tests in this chapter portray the SDTR that the node can provide to client processors, as well as the percentage of CPU cycles required to provide that I/O bandwidth. Performance results for TASS storage nodes are based upon transfers between either disk or cache, and main memory of one or more client processors. The presentation of results in this chapter will be a slight departure from what has been used in the previous chapters. The primary partitioning of results will still be by I/O bandwidth and CPU usage. However, rather than dividing the I/O bandwidth sections first by DAP, then by type of data access (read or write), this chapter will section them first by type of access, then by DAP. Hopefully, this break in technique will not be confusing. The method used in the previous chapters would have produced quite a few more ugly tables than arise using this new breakdown.

9.1. Testing Environment

Tests of the entire TASS storage node are accomplished using the processor configuration shown in Figure 20. Note that the cache server can only communicate with four other processors, due to the number of serial links provided to each transputer. Therefore, the use of three clients and three disk servers in the diagram is meant to show the maximum number of processors in each category. The actual configuration would use some combination of clients and disk servers adding up to no more than four. The root transputer is responsible for organizing the DAP to be issued. Basically, the root transputer, $\mathbf{T}_1$ gives each client processor any start-up parameters that they require, for example, whether the test is for read access or write access, etc. Once the clients have performed any relevant initialization, including testing whether the storage node is functional or not, they inform $\mathbf{T}_1$ that they are

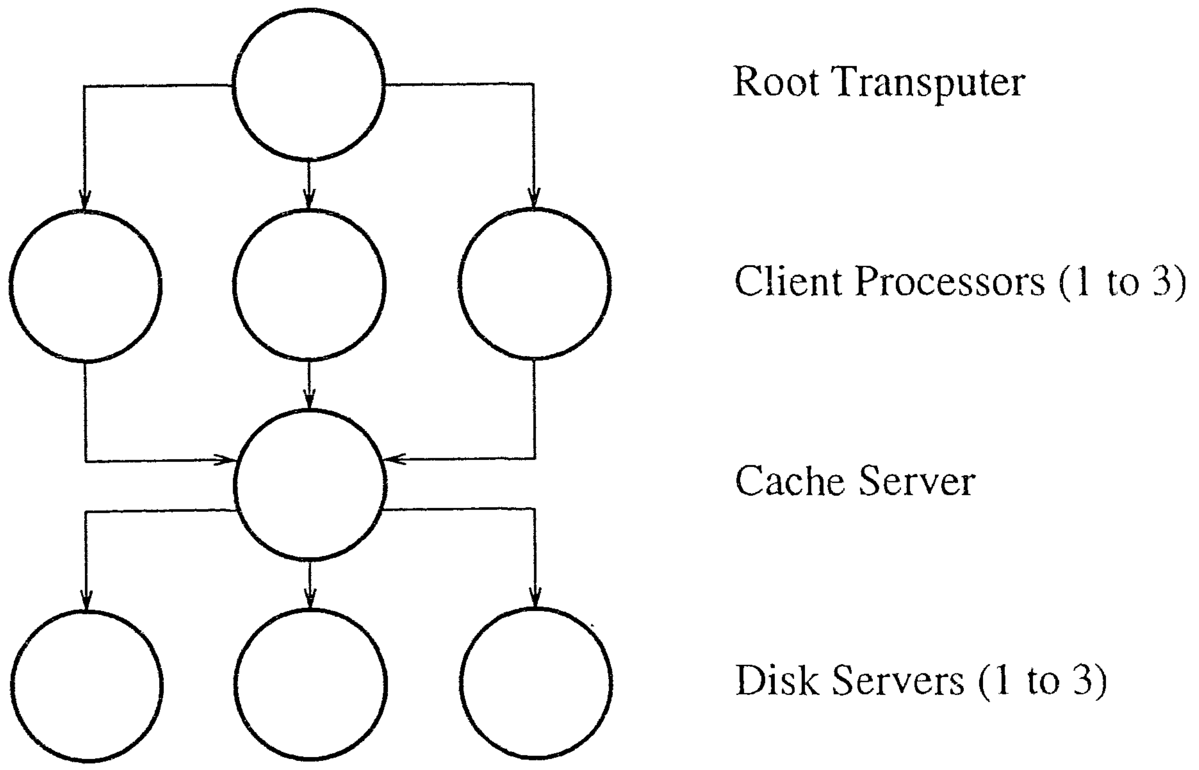


Figure 20: The TASS Storage Node Testing Environment

ready for the test to begin. Once all clients are ready, $\mathbf{T}_1$ issues addresses, as appropriate to the selected DAP, to each of the client processors. When the first address is received, each client makes an initial timestamp and then issues the appropriate disk request. This continues, with each client requesting an address from $\mathbf{T}_1$ and issuing that request, until an invalid address is issued by $\mathbf{T}_1$ to each client. That invalid address signals the end of the test and causes the clients take a final timestamp. From the two timestamps, and the number of actual disk accesses made between these timestamps, the clients can determine the SDTR that the TASS storage node provided to them. The final SDTR of the TASS storage node is determined to be the sum of the rates observed by each of the client processors. In other words, if client **C** observes a SDTR of 730 KBytes/second, and client $\mathbf{C}_1$ observes a SDTR of 711 KBytes/second, then the SDTR provided by the TASS storage node is considered to be 730 +

711 = 1441 KBytes/second. Although not reproduced in this thesis, I have performed tests that show that the client communication with $\mathbf{T}_1$ between every TASS storage node access does **not** noticeably affect the SDTR results obtained. This is because the time required for the client to communicate with $\mathbf{T}_1$ is considerably smaller than the time that the client is blocked, waiting for the disk access to complete.

For the same reasons discussed in Section 8.3, the results for concatenated disk systems have been excluded from this chapter.

9.2. I/O Bandwidth

9.2.1. Read Access

The read access performance results are provided for both SGI and RO DAPs. The I/O bandwidth provided by the TASS storage node is shown in Table 27. Results are provided for most of the possible TASS storage node configurations. Each configuration is described by the number of clients served, the number of disks used and the disk joining technique

Table 27: Read Access Sustained Data Transfer in KBytes/second

Configuration			Only Hits	Chunks Prefetched Ahead							Random Overlapped
Clients	Disks	Joining		0	1	2	3	4	5	6	
1	1	All	1186	298	298	536	730	730	730	730	206
2	1	All	2344	536	536	536	730	730	730	730	333
3	1	All	3453	730	730	730	730	730	730	730	333
1	2	Interleaved	1196	280	277	595	819	1054	1142	1142	205
2	2	Interleaved	2321	596	596	596	826	1070	1304	1450	368
1	3	Interleaved	1200	281	281	545	869	1036	1151	1155	205
1	2	Striped	1209	450	448	846	1180	1182	1182	1182	341
2	2	Striped	2392	847	847	847	1279	1448	1445	1446	636
1	3	Striped	1221	558	558	1079	1204	1204	1204	1204	446

applied. Following the configuration description are the data columns. Data column 1 shows TASS throughput when all client accesses are cache hits. Data columns 2 through 8 present the throughput when the indicated number of chunks have been prefetched prior to their access by the client. In many of these cases the average delay due to each cache miss is often reduced, resulting in significantly higher data throughput. The ninth data column provides results for random accesses to the TASS storage node.

Each entry in Table 27 is determined by averaging the sustained transfer rates obtained during ten samples. Each of these samples involved 1000 chunk read operations.

9.2.1.1. Only Cache Hits

The results in data column 1 of Table 27 show that **each** TASS client, continuously hitting within the cache, can read between 3453/3=1151 and 1221 KBytes/second, for the 3 client/1 disk and 1 client/3 striped disk configurations, respectively. This indicates that for cache hits, the sustained TASS read throughput is at least 94% of the maximum link throughput of 1225 KBytes/second.

9.2.1.2. Sequential Global Access with Prefetching

The second through eighth data columns show the I/O bandwidth of the TASS storage node when operating under a SGI DAP. The individual columns display the throughput for varying degrees of prefetch. As the degree of prefetch increases, the throughput also increases until the maximal disk throughput is reached. This is because each prefetch request activates a *File Server* thread within the TASS cache server. These threads directly access the disk support software, bringing the requested disk chunk into the cache. Any subsequent read request from a client also activates a server thread, however this thread need not access the disk support layer because it will hit within the cache, since the previously issued prefetch will have brought, or be in the process of bringing, that chunk into the cache. Therefore, the number of server threads actively using the disk support layer is equal to the degree of prefetch (the number of chunks looked ahead). When a thread handling a read request queries the *CacheManager* regarding address A, which is currently being prefetched, then that thread will be blocked until the prefetch completes. As the degree of prefetch increases the time between issuing the prefetch to address A and issuing the read to address A also increases. Therefore, the higher the prefetch degree, the less time server threads, attempting to read data, spend blocked at the *CacheManager* and consequently, the less time the disk access appears to take, from the client's point of view. Columns 2 through 8 of Table 27 show that for all storage node configurations and for all DAPs, throughput increases as prefetch degree increases, until the maximal throughput (for that configuration) is reached. The only cases where increased prefetch degree does not improve throughput are (*i*) when the limiting bandwidth has been reached and (*ii*) when the prefetch degree is increased from 0 to 1. When the prefetch degree is 0, no prefetch operations are issued by the clients, so when the read request is received by a server thread, that thread must directly access the DSL (via the cache layer). With prefetch degree of 1, the threads handling the prefetch operations access the DSL and because the read requests are issued almost immediately after the prefetch requests, the threads handling those read requests immediately block at the *CacheManager* interface and wait for the prefetch to complete. Because of this, the results for the 0 and 1 prefetch columns are equivalent, since in both cases, the same number of file server threads (one per client processor) are directly accessing the DSL.

As discussed in Sections 8.2.1.1 and 8.3.1.1, the maximal SDTR for sequential access patterns cannot be achieved unless there are between two and three server threads simultaneously accessing each disk. Table 27 shows that in the 1 disk configurations, maximal SDTR is not achieved until prefetching reaches degree 3. The 1 disk/3 client configuration is a special case, since the 3 clients each activate one server thread to service their read requests. As shown by comparison to the 0 prefetch column for this configuration, these three threads, regardless of prefetch degree, are sufficient to provide maximal throughput. Comparing the single disk configurations to the results for the disk support layer and cache layer (Sections 8.2.1.1 and 8.3.1.1), shows a discernible increase in the number of active servers required to achieve maximal throughput. The increase to the current requirement of three active servers, as opposed to the cache layer and disk support layer requirement of two active servers is an indication that the disk access pipeline (see Chapter 4) has (once again) increased in length. Effectively, the communication beteen the client processors and the cache server processor have added two new components to the pipeline, one prior to the pre-processing stage and the other after the post-processing stage. Since this communication is performed as an RPC call, these new stages have no inherent parallelism with each other. Furthermore, these new stages increase the time required for each request to proceed through the entire pipeline, therefore effectively increasing the number of active requests that are required to keep the disks busy.

The multiple disk configurations display more variety. All single client configurations (both 2 disk and 3 disk systems) exhibit a clipping of read throughput at between 1142 and 1200 KBytes/second. This is a consequence of the Part.7 crossbar bottleneck discussed in section 7.2.2. The throughput for two client configurations peaks at 1450 KBytes/second, with 6 and 4 prefetches, for interleaved and striped disks respectively. Therefore, TASS throughput for a 2 disk system is equivalent (within error margins) to the cache layer's throughput of 1445 KBytes/second, shown in Table 22. For interleaved systems, the performance peak is for six prefetches since (due to the pipeline increase for the client interface communications) three active *File Server* threads are required to keep each disk busy. As discussed in section 8.2.1.1, striped systems require only two active server threads in order to provide maximum disk support layer throughput, regardless of the number of disks joined. The increased pipe length, due to the client interface communications, was expected to increase this requirement to three active server threads, rather than the four indicated in Table 27. I believe this is a consequence of the fact that the striped disk system has a slightly longer pre-processing pipe stage than does the interleaved system, since requests to the striped disk servers are sequentialized, rather than parallelized. Although this longer pre-processing pipe did not affect striped disk bandwidth for read access in the Chapter 8 results, I believe that by coupling it with the client interface communications, it is starting to have an impact.

9.2.1.3. Random Access

The last column of Table 27 shows the maximum throughput that a TASS storage node can provide to clients using random data access patterns. Single disk systems achieve SDTR equivalent to that provided by the disk support layer and cache layer (see Tables 19 and 24), only if multiple clients are accessing the TASS node. For multiple disks, only the two client, two striped disks configuration achieves SDTR equivalent to the results presented in Tables 19 and 24. The poor performance displayed by the other configurations is due to the remote procedure call interface used by TASS clients. Each client can only issue one random access request at a time, therefore, there is never more than one active server thread per client. Two active server threads are required per disk, to provide maximum SDTR for a random data access pattern. Therefore, single client systems can keep only one server thread active, which prevents maximum SDTR from being achieved. Similarly, two client systems with two interleaved disks can keep only one server thread active per disk, on average. Striped disk systems require only two threads to keep all disks active, therefore a two striped disk system achieves maximum throughput when controlled by two clients.

The additional communication required by the client interface does not increase the required number of workers from two to three, as it did for the SGI access, since the physical disk component of the pipe, for RO DAPs, is still too large.

Figure 21 shows the I/O throughput of the TASS storage node when the *TASS File* being accessed varies in length[50]. Four graphs are shown in the figure, these show the RO DAP throughput for (*i*) two striped disks, (*ii*) two interleaved disks, as well as any single disk system, (*iii*) one disk server, controlled by request/response primitives and tested with a constant stream of incoming requests. This graph has been doubled to indicate the maximum throughput that could be provided by two such disk servers, controlled independently of TASS and (*iv*) one disk server connected to its client by an RPC interface. Graphs (*iii*) and (*iv*) are reproduced from Chapter 7's Figure 18. Graph (*iii*) is simply a doubling of Figure 18's read graph, while graph (*iv*) is the same graph as Figure 18's RPC read graph. The reason why graphs (*iii*) and (*iv*) are included in Figure 27 is to show how the random read performance compares to direct disk access. McVoy & Kleiman state that some users "get rid of the file system altogether by using the raw disk" [McV91]. Users resorting to this desperate technique are usually those running database applications, which often have a predominance of random access activity. If the TASS system does not provide as good, or better performance than could be achieved using direct access of the disk servers then such users may still have to resort to such techniques.

It can be seen in the figure that as the region of disk being accessed increases, the I/O throughput of the TASS storage node, using a RO DAP, drops from approximately 1200

[50] Recall that the term *TASS File* was defined for referencing contiguous regions of disk chunks (see Section 3.2.2.2). In other words, a *TASS File* of 4000 chunks, starting at chunk address 50, would include all chunks in the contiguous range of addresses 50 through to 4049.

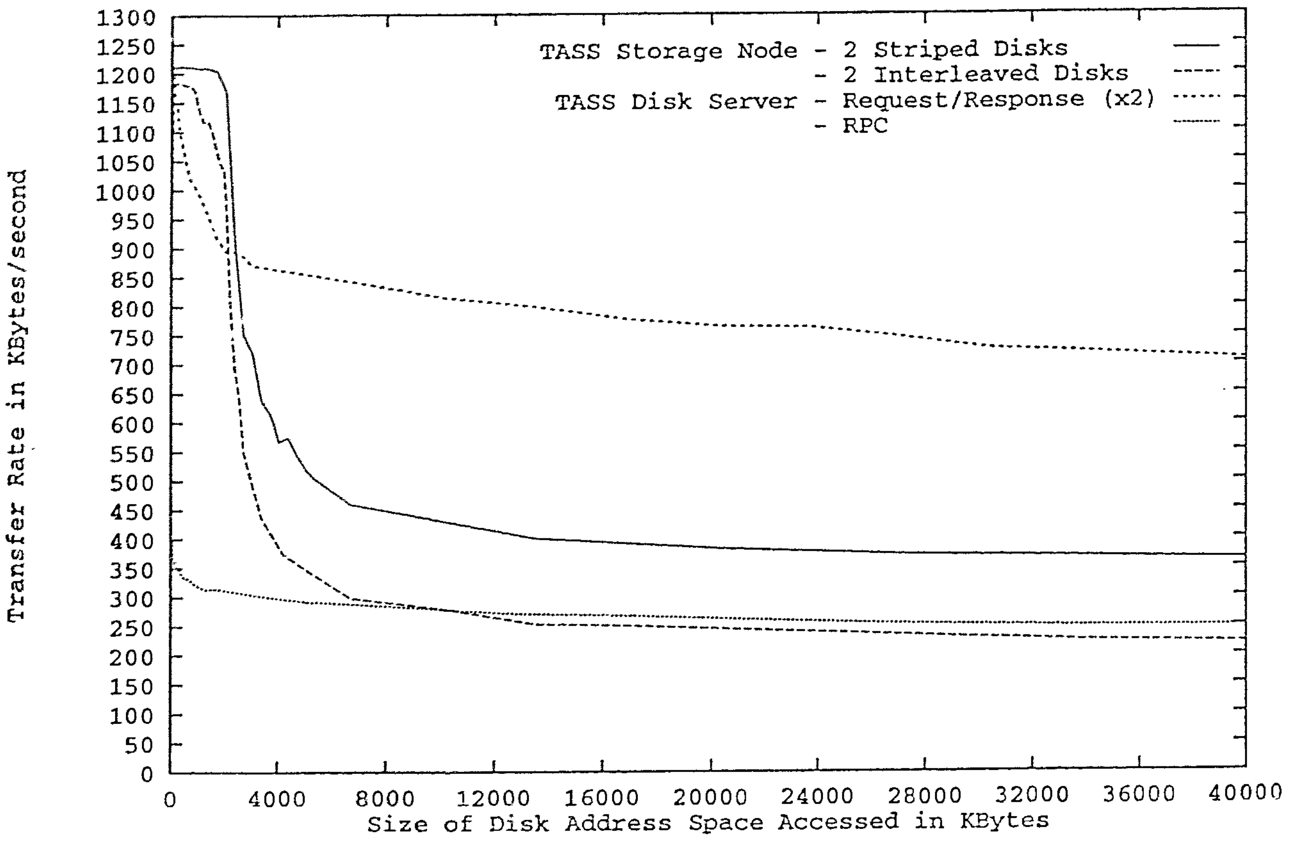


Figure 21: Random Read Access Performance While Varying File Size

KBytes/second down to about 360 KBytes/second (for striped disks) and 225 KBytes/second (for interleaved disks). These results are equivalent to the SDTR figures in Table 27 for the case when all accesses are cache hits, and for the RO DAP results (also in Table 27) for each joining. The cache size is approximately 2000 KBytes (2 MByte main memory for the T800 cache server), and as soon as the cache fills, performance of the TASS storage node makes a sharp plunge. When the file size is about 4 MBytes, which is twice the cache size, performance for both striped and interleaved access has dropped to about 150% of the sustained RO DAP results in Table 27. By the point where the region of disk being accessed is approximately 6 MBytes (3 times the cache size), performance is virtually equivalent to the sustained

RO DAP results. Therefore, except for very small files, the cache has virtually no advantageous effects on disk throughput.

Note that the two interleaved disk graph is used to represent the single disk TASS storage node environment as well. In the RO DAP tests on the TASS storage node, requests are issued one at a time, with no prefetching. Therefore, in the interleaved disk system, only one disk is accessed per request and there is no disk parallelism. Consequently, the two interleaved disk graph is virtually the same as the single interleaved (or striped) disk graphs. By comparison to the graphs for the disk server, it is clear that a user could get better performance by using **one** disk server than can be obtained by using a TASS storage node with **two** interleaved disks. This is because in the extra communication required to pass the data from the cache server to the client processor (as opposed to the direct disk server to client processor link used in graph (*iv*)), causes an additional lag in the time between the request being issued by the client and the response being received. Therefore, the resulting interleaved throughput is actually lower than can be obtained by direct disk server access. The two striped disk environment appears to be better than the interleaved environment, however, this is not necessarily the case. Recall that the striped disk environment moves twice as much data per request as does the interleaved environment. Therefore, the striped disk results should be compared to **two** RPC disk servers since two disk servers are being simultaneously accessed. A mental doubling of the RPC disk server graph should suffice to show that the striped disk environment also fails to improve on direct disk access.

The graph for the two request/response disk servers is a bit like *"comparing apples to oranges"*. The results in that graph show what can be provided by two disk servers when two or more requests (per disk) are simultaneously active, at all times. It is included in Figure 21 to illustrate what sort of performance should be achievable by the TASS storage node if the client processors are not restricted to the limitations imposed by the RPC nature of the TASS client interface[51].

9.2.1.4. Comments

The apparent advantages of striped systems, namely fewer prefetches required to achieve maximum sequential throughput, and better random performance, are offset by the fact that the chunk size for striped disks is larger. For example, sequential access with degree three prefetch to a two striped disk system, will actually look-ahead the same number of bytes as a degree six prefetch on an interleaved system. Furthermore, for a random system, prefetching the next chunk to be used, prior to issuing the read request for the current chunk, would provide sufficient active threads for most interleaved configurations. This would fetch the same amount of data from disk as would be read by any a single random access using two striped disks. It would also have the advantage of allowing the user to fetch that data from two different places in the database, rather than just one larger place, as is done using the striped disks.

9.2.2. Write Access

The SDTR for TASS write access is given in Table 28. Results are given for the same storage node configurations as were used in Table 27. Prefetching is not applicable to write access[52], so the breakdown of the data columns differs from that used in Table 27. The first

[51] In other words, it shows what should probably be achievable if a better client interface were implemented and if that interface allowed multiple client processes, within one client processor, to be involved in the issuing of the RO DAP.

[52] This assumes that the entire chunk is being overwritten. For partial chunk writes, prefetching the chunk into the cache well in advance of the actual write operation would help significantly.

Table 28: Write Access Sustained Data Transfer in KBytes/second

Configuration			Only Hits	Sequential Global Irregular		Random Overlapped
Clients	Disks	Joining		Without Write Back	With Write Back	
1	1	All	1157	298	702	324
2	1	All	2287	513	633	323
3	1	All	3413	525	579	323
1	2	Interleaved	1163	273	1145	572
2	2	Interleaved	2302	596	1219	599
1	3	Interleaved	1170	295	1156	691
1	2	Striped	1198	374	1200	616
2	2	Striped	2390	757	1124	615
1	3	Striped	1212	430	1197	933

data column of Table 28 shows the write performance when every access hits within the cache. Columns 2 and 3 show the performance of the TASS storage node when operating as a delayed write back cache. Column 2 indicates the I/O throughput when no write back process group is operating within the cache server and column 3 indicates the throughput when a 4 threaded write back process group is active. Finally, data column 4 shows the random access performance of the TASS storage node, also with a 4 threaded write back process group active.

All results in Table 28 are determined using 10 samples, each sample consisting of 1000 chunk writes. The cache was pre-filled with dirty chunk entries, prior to each sample commencing.

9.2.2.1. Only Cache Hits

A cache hit, for the purposes of the *Only Hits* column of Table 28, occurs when a write access to address A finds address A within the cache. In this case, the same (possibly dirty) cache entry is overwritten, with no disk access required to reinstate any dirty chunks into permanent storage.

Table 28, column 1 shows that the maximum write SDTR provided by TASS occurs when all accesses hit within the cache. The observed SDTR is within 94% of the crossbar throughput limit of 1225 KBytes/second. Throughput for cache hits to a **n** disk striped system is slightly better than for **n** disk interleaved systems (**n** $\geq$ 2). Recall that the size of chunks in the striped system are larger by a factor of **n** than the chunks within the interleaved system. Therefore the quantity of data transferred per operation is also larger by a factor of **n**. Regardless of disk joining, the overhead per operation will be equivalent, discounting the actual data movement over the serial link, therefore **n** times the data is transferred per unit of overhead.

9.2.2.2. Sequential Global Access

The results for write access using a SGI DAP are divided into two columns. The columns display the I/O bandwidth of the TASS storage node for this access pattern when (*i*) no write back process group is operating within the cache server and (*ii*) when a four threaded write back process group is enabled.

9.2.2.2.1. Single Disk Systems

As for most of the results in this paper, all three joining techniques are bandwidth equivalent for the single disk environment. With just one disk, and no write back threads, I/O bandwidth of the TASS storage node is significantly lower than the throughput provided by the cache layer (Table 22). The TASS storage node can provide a single client with only 298 KBytes/second, which is 71% of the cache layer throughput (with just one worker) of 418 KBytes/second. With two or more clients, the 1 disk TASS node performs slightly better, providing approximately 520 KBytes/second which is 89% of the 583 KBytes/second provided by the cache layer. The loss in bandwidth is again due to the RPC nature of the client interface. Since each client issues an RPC call, it can only activate one *File Server* thread within the cache server processor, and this thread accesses the cache layer and subsequently the disk support layer as an agent for the client. The bandwidth observed by the *File Server* threads can be no greater than what is indicated in the cache layer results (Table 22)[53]. The fact that the client first has to issue the write request to a file server thread, and then receive the associated response, adds additional communication time to each write request, reducing the effective bandwidth.

With the inclusion of a write back process group, the performance for the single disk system improves dramatically. A single client observes I/O bandwidths equivalent to that provided by the raw disk server (Tabie 13). Since one client is issuing all requests, those requests get executed to cache in the pure SGI order. If we assume an LRU replacement algorithm, then the order in which cache entries are written back to disk will also be the pure SGI order, consequently, no shuffling of the SGI order can occur during communication with the *Cache Manager*. Because a write back process group is active, it will be the *Write Back* threads that return most of those entries to disk. Furthermore, since the individual *Write Back* threads are put in the same order within the ProcAltList() calls of both the *Cache Manager* and the *Requestors*, they have the same implicit priority, with respect to all other *Write Back* threads, at both of these inter-process communication points. Because of this, the order in which write backs are issued to *Write Back* threads at the cache manager interface will usually be the same order in which those write requests are accepted by the *Requestors*. Therefore, there is little chance that any shuffling of the SGI order will occur at the DSL and

[53] Assuming the same 'shuffling' of requests is taking place (Section 8.3.1.2).

therefore, the SGI order of the requests should be maintained through to the physical disk.

The results in Table 28 show that when two or more clients are issuing the SGI DAP to a storage node equipped with a write back process group, the throughput is significantly better than when no write back group is active. However, the results fall short of the 702 KBytes/second limit of the disk server. Once again, this is the SGI order shuffling problem rearing its ugly head. Even though two clients may issue their individual requests to the TASS storage node in SGI order[54], the *File Server* thread that accepts the write request for address A_i from client 0, may have a lower implicit priority (within the *Cache Manager's* ProcAltList() call) than does the *File Server* thread that accepts the write request for address A_{i+1} from client 1. Because of this, if the two requests arrive at the *Cache Manager* simultaneously, the write request for address A_{i+1} will be processed first and the SGI pattern will be shuffled.

9.2.2.2.2. Multiple Disk Systems

In the multiple disk systems, there is a very dramatic improvement in effective bandwidth when a write back process group is utilized. Without the write back group, a single client using a TASS node can utilize less than 300 KBytes/second of bandwidth for interleaved disks and only 374 or 430 KBytes/second of bandwidth for 2 or 3 striped disks, respectively. However, with the use of the write back process group, performance for all

[54] This in itself is not assured. In the testing environment described in Section 9.1, each client processor obtains the next address to issue from the root transputer. There is no guarantee that that client processor will actually issue the write request for that address to the TASS node before another client processor manages to obtain the subsequent address in the DAP and issue it to the TASS node. However, since each processor is a T800, each is clocked at the same speed, and each executes the exact same code between the acceptance of the address from the root transputer and the issuance of that write request to the TASS node, it is highly likely that the SGI order will be maintained.

configurations exceeds 1124 KBytes/second. A single client accessing a two disk TASS storage node can utilize all of the 1124 (interleaved) or 1162 (striped) KBytes/second bandwidth that is available at the cache layer interface (see Table 24). With a single client and three disks, the bandwidth available to the client is clipped by the 1243 KBytes/second limitation of the CSA Part.7 crossbar switch (see Table 10). Two client, two disk configurations also provide performance equivalent (within error margin) to the cache layer results given in Table 24.

9.2.2.3. Random Access

Random access write results are equivalent to the results for the disk support layer. Threads within the write back process group use lulls in client activity to flush dirty cache entries to disk. Since the access is random, any re-ordering of the disk access sequence, due to the prioritization mentioned earlier, should have little effect on disk throughput. Additionally, if the size of the write back process group is large enough, there should be two threads accessing the disk at all times[55]. The above tests were run with a four threaded write back group. This small sized group may explain why the performance for a 3 disk interleaved system is much worse than that for a 3 disk striped system. The interleaved system would need approximately six active threads to keep all three disks busy, whereas the striped system would still only require two threads.

Figure 22 shows the I/O bandwidth of the TASS storage node for RO write access to a varying length database. The TASS bandwidth for 2 disk storage nodes is shown for both

[55] Recall that only two active threads per disk are required to provide maximum random access throughput to disk.

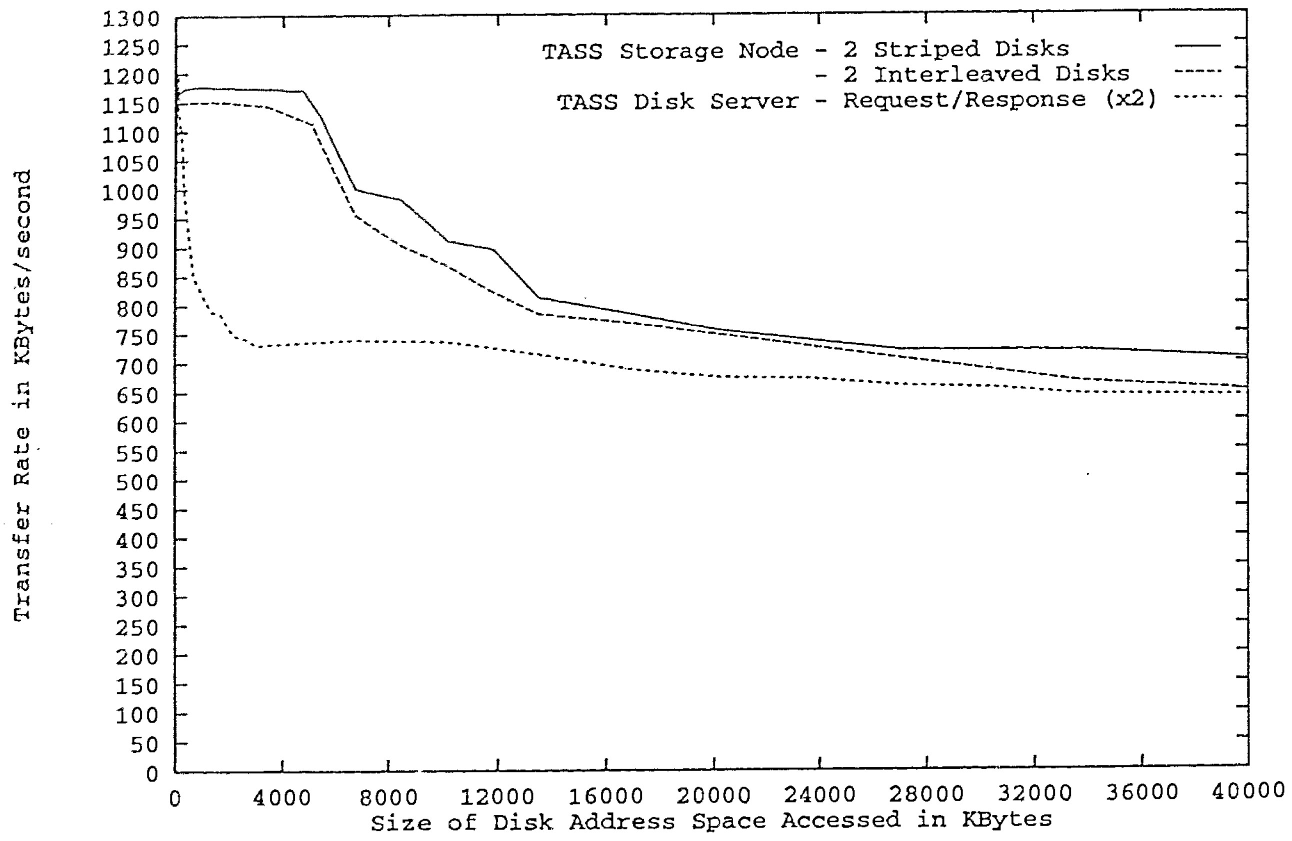


Figure 22: Random Write Access Performance While Varying File Size

striped and interleaved joinings. These graphs are contrasted with the maximum throughput that could be obtained by directly accessing two disk servers using the **request/response** interface. Clearly, the use of the cache enhances throughput for both joinings, when accessing databases of up to 40 MBytes in length. As the databases grow larger, up to the maximum of 200 MBytes (each CP-3100 stores about 100 MBytes), TASS node throughput drops to the levels described in Table 28, which are equivalent to that provided by 2 disk servers (see Table 10)[56]. The reason that the performance of the TASS storage node for RO read access is poor is that only one active *File Server* thread is operating within the cache server at any time. During the RO write operation, only one thread is ever issuing an actual write request, however, four write back threads will be actively attempting to keep the cache clean, hence four or five threads will be accessing the disk support layer, which provides the required workload in order to obtain maximal disk throughput. Therefore, unlike the RO read results for the TASS storage node, the write results in Figure 22 indicate that the cache enhances the performance of the disks.

9.2.2.4. Start-up Performance

Here I vary from previous sections in that the issue under discussion is not sustained data transfer, rather the section analyzes the effect of the cache on I/O throughput during short bursts of write activity. I have chosen to call the sub-section *Start-up Performance* since it deals with write access to an empty cache, which is what happens during the start of a write oriented DAP. Write accesses can be entirely handled within the cache, without the need for a corresponding disk access, if there is an available non-dirty cache entry to overwrite. Furthermore, that entry must be the one selected for replacement by the cache replacement algorithm. When a write DAP begins, (assuming no previous write accesses ever occurred), no entry within the cache is dirty. Therefore, at least the first **C** write chunk operations can be handled by the cache without any required disk accesses (where **C** is the size of the cache in chunks). In other words, the first **C** operations will be performed at a bandwidth indicative of every access being a cache hit. However, once the cache fills, then

[56] Actually, the results indicated in Table 28 for the interleaved joining are a little less than could be obtained using two disk servers, however, the throughput values are equivalent within the sampled error margins.

dirty entries must be written back to disk prior to the data involved in the current write request being copied into the cache. Eventually, this will cause throughput to degrade to the sustained transfer levels indicated in Table 28.

The graphs of Figure 23 show the start-up performance for a one disk TASS storage node, serving a single client which is issuing a SGI write DAP. Each of the three graphs shows the I/O bandwidth of the system, as observed by the client processor, as a single SGI write DAP proceeds from the first operation through to the one-thousandth operation. To achieve this, an initial time stamp is taken before the first operation is issued. Then, after the

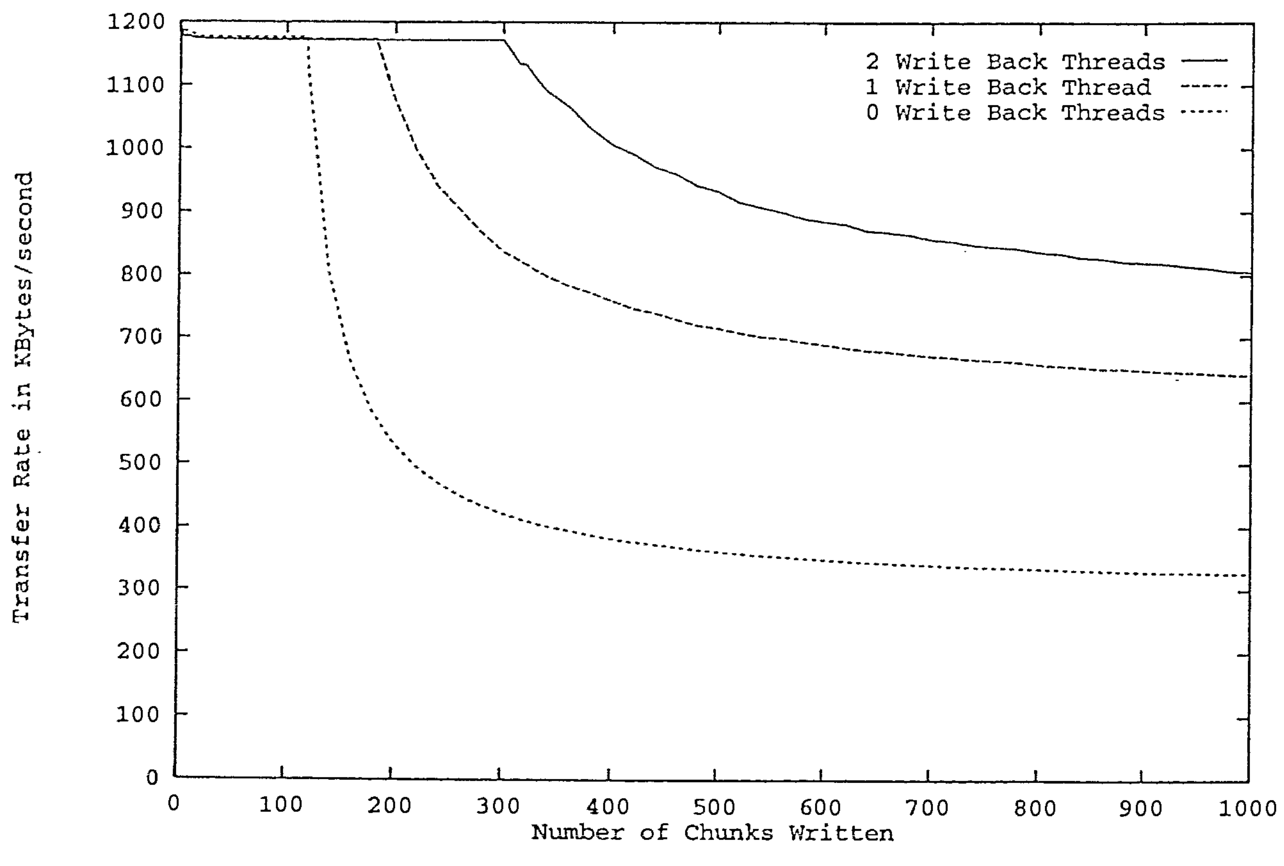


Figure 23: Start-up Performance Under a SGI write DAP

completion of each individual operation, another time stamp is taken. This produces a sequence of time stamps that can be used to determine the effective bandwidth after the first *n* operations, as *n* proceeds from 1 through to 1000. There are three graphs, each showing the start-up performance of the system with either 0, 1 or 2 *Write Back* processes operating within the cache server. Any tests with 3 or more *Write Back* processes are not shown as their graphs only marginally differ from that of the 2 threaded *Write Back* process group. The graphs illustrate the effectiveness of the write back process group at transparently cleaning the cache even during periods of high client (but low disk) activity. Not only is the SDTR under a SGI write DAP better with the write back process group included (see Table 28), but the size of the cache is effectively increased for the initial burst of write activity. With 0 *Write Back* threads operating in the cache server, the bandwidth plunges dramatically after about 120 accesses, which not coincidentally, is also the size of the cache. In other words, the first 120 accesses go to the cache and then delayed write backs must begin and the bandwidth falls off. However, as the number of active *Write Back* threads increases, the number of accesses that can be processed at the *'cache hit'* bandwidth of 1200 KBytes/second also increases until with 2 threads, the first 300 accesses proceed at the *'cache hit'* rate. From the vantage point of the client processor, the cache appears to be 300 chunks in size, rather than the actual 120 chunks.

As a corollary, if a client issues bursts of write activity where each burst is less than 300 operations, then assuming the gap between bursts to be of sufficient length for the *Write Back* processes to completely cleanse the cache, then the client will always appear to be accessing the TASS storage node at the *'cache hit'* bandwidth.

No similar analysis was made for read performance since each read access must go to disk, regardless of the state of the cache, effectively making the start-up performance equivalent to the sustained performance.

9.3. Processor Cycle Usage

One of the primary considerations in the TASS design is to provide disk services to clients, without using a substantial portion of the processing power available to those clients. The results in Table 29 show the CPU usage requirements for a single client processor, accessing a TASS storage node. These results illustrates that the TASS client interface achieves the low processing power utilization goal. For the high bandwidth SGI access patterns, CPU usage ranges from 1.3% for single disk systems up to 2.1% for two disk systems. For the lower bandwidth RO DAP, CPU usage ranges from 0.3% to 0.5%. Since both the client interface and the test program used to determine the flop usage are very simple code segments, the low error margin indicated in Table 29 is probably justified. Almost all samples agreed in percentage overhead to within 0.02%, however, since the figures are rounded

Table 29: Processor Cycle Usage for the TASS Client Interface

(i) One Disk - Any Joining

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	23.5	5.6	0.3	1.3±0.1
	RO Read	83.2	20.1	0.2	0.3±0.1

(ii) Two Interleaved Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	15.0	3.6	0.3	2.1±0.1
	RO Read	83.0	20.1	0.05	0.3±0.1

(iii) Two Striped Disks

Device	Data Access Pattern	Total Time in Seconds	Number MFlops Calculated	Seconds Used by DiskLink Software	Percent Overhead
CP-3100	SGI Read	28.6	6.8	0.6	2.0±0.1
	RO Read	101.1	24.4	0.5	0.5±0.1

off to one decimal place, the error margin must also be rounded up to 0.1%. This is nice to know after some of the high margins of error observed in earlier CPU usage calculations.

9.4. Conclusions

The TASS client interface achieves most of its goal of providing high bandwidth disk services between client processors and disk servers. Furthermore, there is very little processing power consumed within the client processor during the execution of disk access functions.

For SGI read access, the disk bandwidth available to client processors is equivalent to what could be obtained through direct access to the disk server. When two or more disks are provided by the TASS storage node, a single client cannot utilize the entire bandwidth of those disks, since the crossbar switches used for interprocessor communication clip that bandwidth at approximately 1150 to 1200 KBytes/second. To provide the highest possible SGI bandwidth, the client must issue prefetch instructions or utilize the automated prefetching facilities provided by the TASS interface. Fortunately, SGI DAPs are highly predictable and the accuracy of prefetches within that DAP is, as a consequence, quite high. In a single disk system, 3-chunk look-ahead (alternatively I have used the term degree 3 prefetch), is sufficient to achieve the maximal disk bandwidth. For two interleaved disks, 6-chunk look-ahead is required. This indicates that three active *File Server* threads must exist, within the TASS node, for each disk within the joining. For two striped disks, 4-chunk look-ahead is required to keep all subservient disks busy. The degree of prefetch required to keep any three disk joining busy cannot be determined from the results in Table 27, since the throughput is clipped by the crossbar switches well before maximal SDTR is achieved. The additional communication required between client processors and the cache server has increased the length of the disk access pipeline. As a consequence, the time required for each request to proceed through the pipeline has increased and because of that, the number of active workers (*File Server* threads accessing the DSL and CL) required to provide maximal throughput has increased by at least one, over the results indicated for the cache layer.

For read access using a RO DAP, TASS performance is quite poor. This is due to the simplistic RPC interface, which prevents more than one request from being active, within the cache server, for each client using the TASS storage node. In fact, better RO throughput could be obtained by clients if they used an RPC interface to the physical disk itself. Only when the region of disk being accessed in a RO fashion is very small, in the neighbourhood of 2 MBytes, does the effect of the cache provide any improvement over the disk server's throughput. Alternatively, if the clients know in advance the address which will be issued next in the RO sequence (sometimes this is the case in random access), they could observe better sustained throughput by prefetching that address explicitly, prior to issuing the current RO read access.

For SGI write access, the performance of the TASS storage node is comparable with the performance of the cache layer. In some cases (eg: one client/one disk) throughput has even increased over the cache layer levels. This is due to the transparent activity of the write back process group.

The write back process group is an effective tool for transparently cleaning the cache, while also providing a surplus of active server threads, as compared to the number of threads that can be kept active by clients alone. The increase in active threads serve to keep the disk support layer busy, thereby increasing the effective bandwidth observed by client processors. Also, for bursts of write activity, as well as for the initial phases of a prolonged write DAP, the write back processes can provide the client with the appearance of having access to a larger cache than that which is actually provided.

Finally, the TASS client interface requires only a very small portion of the available processing power within each client processor. Therefore, programmers should not (usually) have to worry about using the TASS system within their code, since they will still have almost all of the processing power of the T800 transputer, with the additional bonus of an attached mass storage system.

10. Summary

The Transputer Auxiliary Storage System meets nearly all of the goals set for it. It provides a consistent local file system for use within the transputer network. Because TASS is an LFS, it is not capable of serving all nodes within a given network, however, it does adequately achieve the goal of providing pockets of mass storage support to subsets of processors within the network. In this role it has been effectively used by a number of parallel sort applications, which are documented in [Atk92]. Prior to the implementation of TASS, users of the SFU transputer network were forced to pass all data into (and out of) the network using the host interconnect. This limited the rate at which data could be provided to an application program to approximately 700 KBytes/second. Each disk operating within a TASS storage node can provide between 250 and 730 KBytes/second of I/O bandwidth to the transputer network, providing an I/O bandwidth increase of 35% to 105% of that of the host interconnect. With TASS, some (if not all) application data can be maintained within the network, and provided to application programs from TASS storage nodes. The rate at which this data can be provided to application programs is dictated only by the number and throughput of the TASS node's subservient disk devices, as well as the number of transputer links available for connecting client processors to those TASS nodes.

The TASS system is designed according to a rigidly defined hierarchy of processing elements. These, from top down are the client processors, the cache server and the disk servers. This physical composition of the storage node allows programmers to easily configure the TASS storage node, varying both the number of subservient disks as well as the number of clients to be served by the node. This allows storage architecture to be tailored to the task at hand. Again, this functionality has been utilized in [Atk92].

A hierarchy of software layers is distributed across these processing elements. The four primary layers are defined as the client interface layer, the file system layer, the cache layer and the disk support layer. Each level defines a procedure call interface that provides the layer's services to those of higher levels in a simple and concise fashion. Communication issues and protocols are abstracted within the boundaries of these layers. The highest level, the client interface, provides an efficient (although admittedly primitive) set of primitives that are easily used, yet provide effective access to the mass storage devices (disks). The simplicity of the client interface is a result of TASS design consistently stressing performance over functionality.

The software design is highly influenced by limitations of the physical hardware. Some of the primary examples of hardware functionality which direct the software design are:

- The 64 KByte address space of the T222 transputer prevents its use as a cache server. This dictates that a T800 transputer be allocated as the cache server in the TASS storage node.
- The use of a separate T800 transputer as a cache server allows for the parallelizing of multiple disk servers beneath each cache server. Had a single processor been used as both cache server and disk server, parallelizing of disks would have to be implemented using the SCSI bus protocol, thereby complicating the device dependent software. In this environment, the I/O throughput bottleneck for multiple disks is the SCSI bus itself.
- The bandwidth of the transputer serial links limits the throughput between each TASS node and any given client to between 1240 KBytes/second and 1666 KBytes/second, depending upon whether a crossbar or wire interconnect is utilized.
- Since the transputer processor only provides two priority levels, there is little flexibility for varying the priorities of competitive and/or cooperative process groups.

For sequential read access, which constitutes the majority of disk accesses [Nel87][Flo86][57], the I/O bandwidth of TASS is virtually equivalent to the maximum bandwidth that can be provided using either the CP-3100 disk devices or the TASS RAM disk devices. Therefore, little bandwidth loss occurs during the execution of the TASS software. However, random read performance does not fare as well. Due to the simplistic RPC nature of the TASS client interface, it is usually impossible for TASS clients to provide a sufficient workload to achieve maximal throughput from the disk devices. The performance of TASS for sequential write access also fails to meet the expectations set out in the TASS requirements. A single client, independently accessing a TASS node can utilize the maximum I/O bandwidth of the subservient disk devices. When two or more clients attempt to write to a shared storage node, bandwidth suffers due to an unfortunate side-effect imposed by the compiler's ProcAltList() library procedure. On the plus side though, random write access to the TASS node can proceed with a **higher** bandwidth than could be achieved using direct disk access.

Two software prototypes are implemented for each of the TASS cache server processsor and the TASS disk server processor. In each case, the initial prototype was abandoned in favour of the second one. In the case of the disk servers, the first version was abandoned due to poor performance, whereas the first cache server prototype's perceived failing was in the complexity of its central process, the *Cache Manager*.

The first disk server prototype (DS1) utilizes four process groups, and as a consequence, requires significant inter-process communication to control and sequence the operation of

[57] [Nel87] states that only 1/3 of all disk accesses are write accesses. [Flo86] analyzes the nature of both write and read accesses and determines that the majority of data accesses are sequential in nature.

these process groups. Due to this, performance of DS1 is sluggish. This is because systems utilizing similarly complex process group structures often suffer performance oriented problems due to the context switching and communication overhead imposed by those process hierarchies. This has been discussed in [Cla85] and [Atk88] and although not quantitatively verified here, some qualitative issues are addressed that reinforce these claims. The second disk server prototype integrates the activities of all process groups into a single server group. The resulting disk server software is not only more efficient, but also more understandable.

The first cache server prototype (CS1) utilizes (or would utilize had it been maintained) the same process groups as does the second prototype (CS2). Almost all of the work required during the execution of each client request was handled by the *Cache Manager*. All other process groups virtually operated as dummy message relays between other processors and the monstrous *Cache Manager*. Because of this, the code for the *Cache Manager* is vast and cumbersome. The CS2 prototype reduced the complexity of the cache manager by offloading the responsibility for much of its functionality onto the *File Server* threads. This provides for more understandable code although it may increase communication requirements to some degree. The results in Chapter 8 indicate that shuffling of the disk access sequence occurs at the communication interface between the *Cache Manager* and the *File Servers* and this has caused sequential write throughput to degrade significantly. Further vertical integration of the cache server design may improve I/O bandwidth by reducing the amount of inter-process communication and thereby eliminating the dependence of the software on the ProcAltList() procedure. For example, I am considering a technique whereby the operation of the *File Server* process group and the *Requestor* process group could be integrated. This would eliminate one of the primary communication interfaces within the cache server.

11. Future Work

TASS provides a suitable test-bed for further research. Some of the possible avenues for such future work are documented in this chapter.

11.1. New Hardware

The T9000 transputer, the newest family member, is pending release from Inmos. This transputer will provide significantly more processing power than is available using the current transputer workhorse, the T800. Additionally, Inmos plans to incorporate into the T9000 a hardware based message router that will provide point-to-point communication between all T9000 transputers within a network. Furthermore, the bandwidth of the serial links will be expanded to 20 MBytes/second. Increasing the bandwidth of the transputer serial links would eliminate the clipping of throughput observed for the SGI Read results (Table 28) and make the three disk storage node practical. Using the T9000 or TI C40 (see next paragraph) with their significantly higher link bandwidths would allow for effective use to be made of three disk TASS nodes. Because processor use is not a bottleneck in the current TASS implementation, further increasing processing power over the T800 levels probably has few ramifications for the TASS system.

Texas Instruments has released the TMS320C40 microprocessor, which is usually referred to as the C40 [Tex91]. Although this is not a transputer device, each of these processors is equipped with six transputer compatible serial links. As with the T9000, these links are capable of transmitting 20 MBytes/second. Due to the six serial links, using a C40 as a cache server would increase the number of possible TASS storage node configurations. Between 1 and 5 subservient disks could be joined within a TASS storage node, providing a higher degree of disk parallelism. Similarly, between 1 and 5 clients could be serviced by each TASS node, which could serve to keep more server threads active, achieving maximal disk throughput using lower degrees of prefetching.

Improved transputer-based SCSI interface boards are now becoming available from various vendors. For example, Paralogic Inc., of Bethlehem PA, has released a board that uses a T800 transputer as its control processor. This provides a processing platform capable of supporting both the cache server and disk server components of a TASS storage node. Multiple disks can be connected to the board's T800 using the SCSI bus interface. Since this is a more complicated SCSI environment than is used within the current TASS implementation, a more complex SCSI driver would be required (however, since SCSI drivers are fairly common software demons (sic), a good template for a driver design could be easily acquired).

11.2. Increased Cache Capacity

The results presented in this thesis are obtained using T800 cache servers. Each of these T800s is equipped with only 2 MBytes of RAM. Using a T800 (or other 32 bit transputer) with 8 MBytes of memory would allow the TASS node to provide approximately four times the cache space than is provided by the current version. Enlarging the cache space would serve to increase the average hit ratio, which would enhance I/O throughput of the system as a whole.

11.3. Expanding Disk Server Functionality

The request and response primitives provided by the TASS disk server do not provide all of the functionality that may be desired by potential users of such disk servers. Some suggestions for expanding the set of disk access primitives are included in this section. I have not implemented these suggested primitives as they are not required for my research, however, they would probably be necessary in a fully functional system.

To allow a client processor (like the cache server) to know that all disk operations have been completed, a **Flush Disk Queue** primitive could be included in the disk server interface. Receipt of this request at the disk server would prevent any new disk requests from being accepted until all outstanding disk requests have completed.

Although a **Format Disk** primitive has been implemented as part of the device dependent layer of the disk server software, access to that primitive was not extended to client processors. Adding it to the disk server interface would probably be required in a more comprehensive disk system.

Clients of the disk server cannot reset the disk if they perceive a problem in its operation. Therefore, a **Reset** primitive would also be required in a complete disk system.

Finally, a **Chunk Test** operation might be required to allow clients to test whether a disk chunk is storing data correctly. This would be required to allow a file system to maintain a *bad blocks* list.

11.4. Additional Disk Joinings

The concatenated, striped and interleaved disk joinings are not all of the possible joinings of multiple disks that can be provided. For example, Section 2.5.1 discussed the RAID concept which proposes a hierarchy of techniques for providing highly reliable disk storage from inexpensive disk devices. The simplest technique defined in the RAID hierarchy is *mirroring*, where two disks can be joined together to provide data redundancy. In other words, a two disk system can be implemented such that each chunk is written to both disks. Then, if an access to one of the disk returns an error, or perhaps a cyclic redundancy check (if provided by the hardware) indicates an error in the data read, then it is highly likely that the other 'mirrored' disk will still accurately store the data and can pass it on to the client.

Adding this joining technique to TASS would not be a trivial matter. Given two disk servers $\mathbf{D}_1$ and $\mathbf{D}_2$ and the disk support layer process environment described in Chapter 5, the following issues arise. An *Acceptor* process that receives a disk response from $\mathbf{D}_1$ which indicates an error in disk operation has occurred would have to guarantee that an accurate copy of the data (which has been obtained in parallel from $\mathbf{D}_2$) is returned to $\mathbf{D}_1$. Therefore, that *Acceptor* must issue a new request for a write operation to $\mathbf{D}_1$. This requires a communication to the *Requestor* process group which will issue the request. This situation may violate the no-circular wait criterion that has been used in TASS for deadlock prevention.

11.5. Removing the Implicit Prioritization of ProcAltList()

Many of the I/O bandwidth losses that occur within TASS are due to shuffling in the order of disk requests, between when they are accepted from clients and when they are issued to disk. Most of this shuffling occurs as a side-effect of the implicit prioritization imposed by the Logical C ProcAltList() function. Alternative techniques for providing the functionality of ProcAltList() without this hidden penalty, could be explored.

11.6. Non-RPC Client Interface

Another source of bandwidth loss in the TASS system is the simplistic RPC interface used between client processors and cache server processors. This interface could be modified to allow multiple processes to share the interconnecting communication link, thereby allowing a higher degree of parallelism in client access to TASS.

11.7. More Functional Local File Systems

As stated frequently in this thesis, the TASS chunk I/O interface sacrifices functionality in the pursuit of efficiency. However, CPU utilization results (see Chapter 8) have shown that plenty of processing power still remains within the cache server processors. Therefore, more functional file systems could probably be implemented without reducing data transfer bandwidth.

11.8. Distributed File Systems for the Transputer Multiprocessor

As mentioned in Chapter 1, the goal of TASS is to provide a local file system which can be used as a building block within a distributed file system, which is to be implemented later. Obviously, no future work chapter would be complete without some reference to this DFS. During the period of this research, a few ideas have occurred to me as to what form that DFS might take. I will briefly describe two of these ideas in hope that they might prompt an undertaking of the DFS design.

One of the primary components in DFS design is communication. Every processor controlling some group of disks must be able to communicate with any client processor, as well as all the other disk control processors in order to provide coherent operation within the DFS. I propose two DFS architectures, one for a transputer network that has point-to-point communication[58] while the other is based on the message passing model currently used in the

[58] Any processor can send a message directly to any other processor. Routing of messages between two indirectly connected processors (via a chain of processors) is performed either in hardware or using some generic software package, like the Trollius Operating System.

SFU transputer network[59].

11.8.1. DFS using Point-to-Point Communication

This model is the simplest of the two because many DFS have been built within similar environments, for example Andrew [How87], Sprite [Ous88],[Nel87] and PIFS [Dib90], and as such a surplus of research and ideas for such a system are available. Basically, the TASS storage nodes are configured as part of a generic network. The point-to-point communication allows any processor to communicate with the TASS node, and vice versa. Additionally, the environment required for this model may become a reality whenever the T9000 series transputer is realeased.

An important issue in DFS design is *location transparency* [How87]. This is a form of information hiding whereby the clients of the DFS need not know the physical structure of the file system (ie: how many disks, where are they in the network, how is each file distributed over those disks). Provision of a location transparent DFS, built upon TASS nodes, requires the modication of the TASS client interface. Any client request to read some region of disk, either directly as in the TASS chunk I/O interface, or as part of a file within a more advanced file system, must be routed to all of the appropriate TASS nodes in order to be satisfied[60]. This knowledge of where the disk region is must be maintained at some level of the DFS and be easily available to the client. Some of the alternatives for providing this abstraction are:

[59] Only transputers that are directly connected to each other are able to pass messages to each other. Routing of messages between indirectly connected processors has to be handled by the DFS software.

[60] A client request to read 500 bytes from a file may actually require access to more than one TASS node if that file is distributed across many disks.

(1) Let the individual storage nodes maintain all the information regarding the file system. Then, if a request for data arrives at a storage node, but cannot be satisfied by that node, then the storage node will relay the request to whatever node(s) are able to satisfy the request.

(2) Dibble [Dib90] utilizes a *Bridge* server to maintain all file control structures. Clients issue requests to the Bridge server which informs the client as to where the required data can be found.

(3) The directory structures and file access tables can be maintained within each client processor. This requires a coherency algorithm be provided to guarantee that every client is using the current state of the directory structures and file access tables.

Similarly, client side caching of data can, and as illustrated by [How87],[Nel87], probably should be implemented. Again, this requires coherency algorithms to guarantee that all clients are accessing current copies of the data.

The interface between the TASS storage node and its respective clients has to be modified to allow TASS *File Servers* to keep track of what process on which processor issued the disk access request. Then, when the file operation is complete, the *File Server* can use the point-to-point communication environment to pass the response back to the appropriate client.

These are just some of the issues to be pursued in such a DFS environment.

11.8.2. DFS for SFU Style T800-based Network

Given that no point-to-point communication exists in the target network, an alternative method for providing the required communication between client processors and component

TASS nodes must be established. In keeping with my penchant for allocating hardware to solve irritating problems, I propose the architecture shown in Figure 24. In this environment, the set of TASS nodes are placed beneath a chain of DFS nodes. Each DFS node allocates two communication links to the DFS chain, one communication link to a client processor and one link to either a TASS storage node, or another client processor. This requires, in the worst case, that each client processor is matched with one DFS processor[61]. This chain of DFS processors can then be used to relay client requests (and subsequent responses) to (and from) the appropriate TASS storage node.

Additional levels of caching could be provided within either the processors comprising the DFS chain, the client processors, or both. Any cache coherency algorithms required to sustain these multiple caches could utilize the DFS chain as a common communication medium.

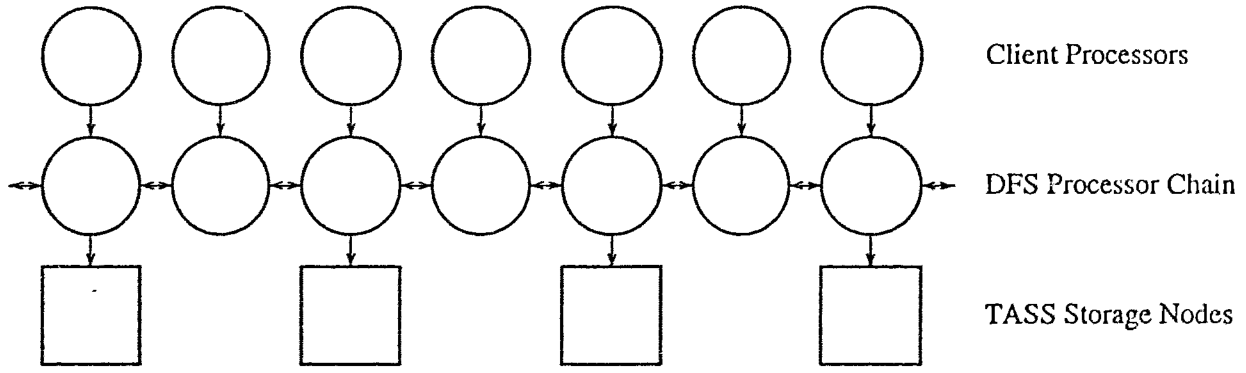


Figure 24: A Distributed File System Environment

[61] In other words, discounting the processors already allocated to TASS storage nodes approximately one half of the network's remaining processors are used for the distributed file system. However, these nodes need not be the most expensive flavour of transputer, they could be T222 or T414 varieties, which are cheaper and less functional.

This technique is not very scalable, since as the network size grows, so does the DFS chain. The longer the DFS chain, the longer it will take a request to proceed from the issuing client to the target TASS node. Similarly, the response time will increase and as a consequence, throughput will suffer.

Appendix A: Glossary of Acronyms

C	A wonderful programming language.
CCL	Client communication layer. Two software layers, one utilized in the cache server processor and the other utilized in the disk server processor code. This layer is the software module that implements the CCI.
CCI	Client communication interface. The client side component of the client interface layer.
Chunk	The unit of data transfer used in TASS.
CL	Cache Layer. One of the primary software layers in TASS.
CRA	Cache Replacement Algorithm. Method for determining what information should remain in a full cache, and what information should be replaced with more pertinent information. The most common form of CRA is least recently used (LRU).
CSA	Computer Systems Architects. A transputer board manufacturer in Provo, Utah.
DAP	A data access pattern. See section 2.4.3 for a discussion of the various DAPs.
DDL	Device Dependent Layer. The layer of software within the disk server that actually controls the physical disk devices. This layer is tailored for dealing with specific hardware components, abstracting hardware realities into device independent services.
DFS	Distributed file system.
DIL	Device Independent Layer. The top most software layer of the disk server processor code.

DMA	Direct memory access. The transputer utilizes DMA circuitry to transfer data between communication links and internal memory. This allows communication and computation to occur in parallel.
DSL	Disk Support Layer. The lowest of the primary layers in the TASS design. This acronym is additionally used to indicate the highest of the secondary or component layers within the disk support layer.
DS1	Notation used to describe the first TASS disk server prototype.
DS2	Notation used to describe the second TASS disk server prototype.
FIFO	First In, First Out. A cache replacement policy. Also, a term sometimes used to describe queue operation.
Flop	A unit of measurement for processor cycles. Within this paper, one flop is defined to be one double precision floating point multiplication and one double precision floating point addition.
FSL	File System Layer. One of the primary software layers in the TASS design.
I/O	A contraction for "input and output".
KByte	One kilobyte, which is 1,000 bytes.
LFS	Local file system.
LRU	The least recently used cache replacement algorithm. See also CRA.
MByte	One megabyte, which is 1,000,000 bytes.
MFlop	One million flops.
MID	Multiple Inexpensive Disks. A simple acronym to describe using multiple inexpensive disks to substitute for more expensive disks (SLEDs).
Modula-2	A programming language.
ms	Millisecond.

MSS	Mass Storage Support.
MTTDL	Mean Time To Data Loss. The average time between disk failures for all instances of the same model of disk device.
MTTF	Mean Time To Failure. The same as MTTDL.
ns	Nanosecond.
PET	Positron Emission Tomography. A medical process for providing 3D images of human anatomy.
PVI	Positron Volume Imaging. One of the technique used in PET.
RAID	Redundant Arrays of Inexpensive Disks. The concept involved in using multiple inexpensive disks to provide the same I/O bandwidth, storage capacity and reliability that would normally be provided by large expensive disk packs.
RAM	Random Access Memory.
RAM1	A RAM disk driver. This particular driver is implemented in conformance to the device and services modularity used for all TASS disk drivers. The driver utilizes bcopy() for transfers between the *disk* (memory) and internal buffers.
RAM2	A RAM disk driver. This driver is implemented independently of other TASS disk drivers. The driver does not conform to the device and services modularity used by other TASS disk drivers. Due to the speciality of its implementation, there was no need to use bcopy() within the driver software.
RISC	Reduced Instruction Set Computer. This is a style of processor design which provides a very small, very efficient instruction set.
RO	Random Overlapped. A type of data access pattern.
SBIC	SCSI Bus Interface Controller. A chip designed to provide memory-mapped SCSI bus control to hardware devices.

SCSI	Small Computer Systems Interface. A standard proposed by ANSI for providing device support to small computer systems.
SDTR	Sustained Data Transfer Rate. The rate, in KBytes/second at which data can be transferred between two endpoints, usually disk and main memory for a client processor.
SGI	Sequential Global Irregular. A type of data access pattern.
SGR	Sequential Global Regular. A type of data access pattern.
SLDI	Sequential Local Disjoint Irregular. A type of data access pattern.
SLED	Single Large Expensive Disk. An acronym for the traditional large disk devices used in large scale computing systems.
SMI	Server message interface. The server side component of the client interface layer.
SML	Server message layer. Two software layers, one utilized in the cache server code and the other a component of the client processor libraries.
T222	A 16 bit transputer device.
T800	A 32 bit transputer device.
TASS	Transputer Auxiliary Storage System. The software testbed used for research within this thesis. Also, a Soviet information service known for fast data absorption and inconsistent release.
WORM	Write one, read many cache locking technique.

Bibliography

[AMD81] Advanced Micro Devices, Inc., *Bipolar Microprocessor Logic and Interface Data Book*, 901 Thompson Place, P.O. Box 453, Sunnyvale, California, 94086 U.S.A., 1981.

[ANS86] American National Standards Institute, *Small Computer System Interface (SCSI)*, New York, 1986.

[Atk88] M. Stella Atkins, *Experiments in SR with Different Upcall Program Structures*, ACM Transactions on Computer Systems, Volume 6, Number 4, November 1988, pages 365-392.

[Atk91] M. Stella Atkins, and Yan Chen, *Performance of SUN-Transputer Interfaces: some surprises*, Proceedings of the Transputing '91 Conference, IOS Press, pages 124-138, April 1991.

[Atk92] M. Stella Atkins and Mark Mezofenyes, *Sorting Large Files on a Transputer Network*, Transputer Research and Applications 5, The Proceedings of the 5th North American Transputer Users Group, IOS Press, 1992.

[Bac86] Maurice J. Bach, *The Design of the Unix Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, U.S.A., 1986.

[Baw91] Hugh Bawtree, *Restructuring the Run-Time Support of a Distributed Language*, MSc. Thesis, School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada, November 1991.

[Ber86] Eric J. Berglund, *An Introduction to the V-System*, IEEE Micro, August 1986, pages 35-51.

[Che84] David R. Cheriton, *The V Kernel: A Software Base for Distributed Systems*, IEEE Software, April 1984, pages 19-42.

[Che88] David R. Cheriton, *The V Distributed System*, Communications of the ACM, Volume 31, Number 3, March 1988.

[Cla85] D. D. Clark, *The Structuring of Systems Using Upcalls*, Proceedings of the 10th Symposium on Operating Systems Principles, ACM, New York, 1985, pages 171-180.

[Com84] Douglas Comer, *Operating System Design, The XINU Approach*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, U.S.A., 1986.

[CSA89] Computer System Architects, *Part.6, Part.7, Part.8, and Part.12 Users' Manuals*, Computer System Architects, 950 North University Avenue, Provo, Utah, 84604, U.S.A., June 1989.

[CSA90] Computer System Architects, *Logical Systems C for the Transputer: Version 89.1 User Manual*, Computer System Architects, 950 North University Avenue, Provo, Utah, 84604, U.S.A., June 1989.

[CSD79] W. S. Avis, P. D. Drysdale, R. J. Gregg, M. H. Scargill, *Canadian Senior Dictionary*, Gage Publishing Ltd., Vancouver, British Columbia, Canada 1979.

[Dib90] Peter C. Dibble, *A Parallel Interleaved File System*, PhD. Thesis, Dept. of Computer Science, University of Rochester, March 1990.

[Dud92] Timothy J. Dudra, M. Stella Atkins, *An Auxiliary Storage System for Transputer-based Multicomputers*, Transputer Research and Applications 5, The Proceedings of the 5th North American Transputer Users Group, IOS Press, 1992.

[Flo86] Rick Floyd, *Short-Term File Reference Patterns in a UNIX Environment*, Report TR 177, Computer Science Department, University of Rochester, March 1986

[Gib89] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz and David A. Patterson, *Failure Correction Techniques for Large Disk Arrays*, Proceedings of the 1989 ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Pages 123-132, April 1989.

[Hil88] Mark D. Hill, *A Case for Direct-Mapped Caches*, IEEE Computer, Pages 25-40, December 1988.

[How87] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West, *Scale and Performance in a Distributed File System*, Proceedings of the 11th Symposium on Operating Systems Principles, ACM, New York, 1987.

[Hwa84] Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, Toronto, Ontario, 1984.

[Inm89] Inmos Ltd., *Transputer Data Book, 2nd Edition*, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, U.K., 1989.

[Kot91] David Kotz, *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*, PhD. Thesis, Dept. of Computer Science, Duke University, 1991.

[McV91] L. W. McVoy and S. R. Kleiman, *Extent-like Performance from a UNIX File System*, Proceedings of the USENIX Conference, Winter 1991.

[Mor88] Joseph P. Moran, *SunOS Virtual Memory Implementation*, Proceedings of the European UNIX User's Group, April 1988.

[Nel87] Michael N. Nelson, Brent B. Welch, John K. Ousterhout, *Caching in the Sprite Network File System*, Proceedings of the 11th Symposium on Operating Systems Principles, ACM, New York, 1987.

[Ous88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, *The Sprite Network Operating System*, IEEE Computer, February 1988.

[Pat88] David A. Patterson, Garth Gibson and Randy H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Proceedings of the 1988 ACM Conference of the Special Interest Group on Management Of Data (SIGMOD), ACM Press, Pages 109-116, June 1988.

[Pet85] James L. Peterson and Abraham Silberschatz, *Operating Systems Concepts, 2nd Edition*, Addison-Wesley, Don Mills, Ontario, Canada, 1985.

[San85] Russell Sandberg, et al., *Design and Implementation of the Sun Network Filesystem*, Usenix, June 1985.

[Sch88] Martin E. Schulze, *Considerations in the Design of a RAID Prototype*, Report Number UCB/CSD 88/448, Computer Science Division, University of California, Berkely, California, 94720, U.S.A., August, 1988.

[Sin88] Jaswinder Pal Singh, Harold S. Stone, Dominique F. Thiebaut, *An Analytical Model for Fully Associative Cache Memories*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1988.

[Smi78] Alan J. Smith, *Sequential Program Prefetching in Memory Heirarchies*, IEEE Computer, pages 7-21, December 1978.

[Smi87] Alan J. Smith, *Cache Memory Design: An Evolving Art*, IEEE Spectrum, pages 40-44, December 1987.

[Sta88] William Stallings, *Data and Computer Communications, Second Edition*, Macmillan Publishing Company, 866 Third Avenue, New York, New York, 10022, U.S.A, 1988.

[Sto89] Harold S. Stone and John Turek, *Optimal Partitioning of Cache Memory*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, 1989.

[Tex91] Texas Instruments, *TMS320C40 User's Guide*, 2564090-9721 revision A, May 1991.

[Thi89] Dominique F. Thiebaut and Harold S. Stone, *Improving Disk Cache Hit-Ratios through Cache Partitioning*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, 1989.

[Wil89] John Wilkes, *DataMesh - Scope and Objectives: a Commentary*, Hewlett-Packard Computer Systems Center, Palo Alto, California, July 1989.

[Wil91] John Wilkes, *The DataMesh Research Project*, Proceedings of the Transputing '91 Conference, IOS Press, April 1991.

[Yan92] Cui-Qing Yang and YaoShuang Qu, *Supporting Communications in a Transputer Distributed Environment*, Transputer Research and Applications 5, The Proceedings of the 5th North American Transputer Users Group, IOS Press, 1992.