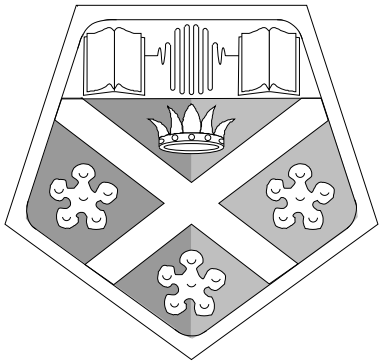
DOLPHIN:
PERSISTENT, OBJECT-ORIENTED AND NETWORKED

A REPORT
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF STRATHCLYDE
CONSISTING OF A DOCTOR OF PHILOSOPHY
THESIS DISSERTATION

By
Gordon W. Russell
February 1995



Abstract

This thesis addresses the problems associated with the design of a new generation of high-performance computer systems for use with object-oriented languages. The major contribution is a novel design for the utilization of object addressing and multiple stacks within RISC-style processors, and a proposal for object-addressed stable electronic memory capable of supporting much higher performance than contemporary designs based on the use of disk backing store. The impact of the new designs in discussed and evaluated by simulation within the context of object-oriented system design.

Publications and Acknowledgements

A number of publications have been created from this work:

- Four machine descriptions and critiques have been reproduced in [Russell and Cockshott, 1993a] from an early draft of chapter 3. This paper is also available as a technical report [Russell and Cockshott, 1993b].
- A technical report discussing DAIS' caching strategy has been produced [Russell and Shaw, 1993a]. A cut down version of chapter 5, discussing only the cache structure and its advantages, appears in [Russell *et al.*, 1995].
- DAIS' register file is based on a dynamically-resizing shift register. An initial step in the design of this register file, *Shifting Register Windows*, is available in [Russell and Shaw, 1993b]. A version of this paper is available in technical report form [Russell and Shaw, 1993c].
- The philosophy behind utilizing Flash cards to implement high-reliability persistence (discussed in chapter 8) is available as a technical report [Cockshott *et al.*, 1993a]. A patent is being prepared to cover the mechanism used in supporting Flash card reliability.
- An early implementation of MultiThreads, the portable threads library used in the DOLPHIN sub-kernel (chapter 9), is discussed in [Russell, 1993]. Note that from this, MultiThreads has enhanced considerably.

I would like to acknowledge the help of Paul Cockshott, my PhD supervisor. His knowledge and understanding has helped greatly to make this document what it is. He also produced initial critiques of two of the five machines investigated in chapter 3, although much of the criticism found therein is now my own.

I would also like to thank both Paul Shaw and Robert Lambert; colleagues and fellow PhD Students. Our discussions concerning all of our respective PhD topics has helped to both improve the breadth my of understanding and the quality of the descriptive text produced both in this and other technical documents.

Contents

Part I

An Object-Oriented Approach

Chapter 1

Introduction

This document is concerned with promoting further research into object oriented system design, which includes microprocessors, long-term storage, kernel construction, and higher level object support. One of the initial stumbling blocks for any researcher interested in this problem domain is the lack of a framework within which he or she can implement and analyse new designs.

Research into novel operating systems was given a boost with the development of Mach [Tevanian and Smith, 1989], a microkernel providing hardware independence for operating systems. The program included support for memory management, task control (both light and heavyweight threads), and inter-process communication. Now it was possible to implement applications without the overheads of a large operating system, but still with support for commonly-used primitives. Not only that, but these applications were, on the whole, highly portable between platforms. The only down-side of Mach is its complexity; in many areas it forces the programmer into certain decisions. For example, the control of its paging mechanism is not fully controllable by application programs. As Mach is targeted strongly towards virtually-addressed operating systems such as unix, it is not ideally suited as a microkernel for object-oriented systems.

An object oriented environment is proposed in this document, which allows researchers to implement algorithms targeted as specific problem areas within the object-design framework, without having to write the remaining code necessary for experimenting with their algorithm in an object-based programming system. It also allows complex object systems to be constructed without duplicating code common to many such designs. This environment, nicknamed DOLPHIN, provides a subkernel which supports routines common throughout object-oriented systems, such as process management, object creation and deletion, object swapping, and networking primitives. It is important that this subkernel does not interfere with possible object stored built on top of it, for example enforcing a particular object sharing mechanism. It is proposed that a custom-designed processor is used to provide a high degree of performance, and this is named DAIS. This processor uses object addressing as its standard addressing mode, and when combined with the subkernel should execute applications with a performance

comparable with that expected from a traditionally addressed processor programmed using C.

1.1 Overview

Chapter 2 looks at what is meant by a persistent object-oriented architecture, and how it is used by high-level programming languages to support complex data types. This chapter indicates how objects are accessed, and how this method differs from the more traditionally addressed memory system. It also describes various degrees of tagging, and demonstrates techniques whereby the types and sizes of elements contained within objects can be identified without higher-level knowledge. Conclusions are drawn as to the desirability of each technique depicted.

Before an object-based system can be designed, previous systems should be examined. Chapter 3 investigates a number of typical architectures; iAPX-432, MONADS, Rekursiv, MUTABOR, and SYSTEM/38. These are presented in overview, with special consideration made to their object-addressing schemes and performance. It has been said that RISC designs, which are at the forefront of current high-performance microprocessor designs, gain much of their advantage over non-RISCs from their cache and pipeline implementation, so this too is investigated. This chapter highlights the need for a large addressing space, simple instruction set and processor design, and a large degree of caching. This caching is necessary, since supporting object-based addressing is equivalent to adding a level of indirection on top of standard virtual addressed memory.

The DOLPHIN environment is proposed in part II, and is itself split into eight chapters. The first of these (chapter 4) is concerned with the ideas behind DOLPHIN. This examines the layering used in DOLPHIN, how its component parts fit together, and what was actually investigated during the research period covered by this document.

Chapters 5 to 7 present the DAIS architecture, which is the proposed object-addressing-based processor. The first of these demonstrates a cache design which supports object-caching for both instructions and data. The cache access rates of the design are similar to that possible in a traditionally addressed RISC. The processor is initially described using a 64 bit virtual address space, using an aliasing technique to support networked objects. This is then argued against, and a design based of 256 bit object identifiers proposed instead[1]. Next, a possible register file is proposed which avoids the vast majority of spill/refill induced stall cycles which occur in other types of file (*e.g.* SPARC windows). This register file treats each register as the head of a stack, which opens new routes for algorithm implementation and optimization techniques. Lastly, a pipeline implementation for DAIS is suggested. This proposal, through the removal of pipeline-created data dependencies, produces a 6% performance increase when compared to a SPARC-like pipeline implementation.

The support of long-term storage is investigated in chapter 8. An argument is made to use solid-state storage to support data persistence, instead of the more common rotating media. The design of a solid-state store is then proposed, providing reliability of information even under multiple device failures. The error recovery scheme is titled ERCA (Error Recovery in Card Arrays). It is suggested that this scheme could also be used to allow imperfect memory devices to be utilized, which could

[1] It should be said that the decision to use 256 bit addressing is not critical to the DOLPHIN environment as a whole, and the modifications necessary to the DOLPHIN proposal to use the 64 bit system is also discussed.

significantly reduce the overall cost of the store. Finally, the effectiveness of card-based virtual memory is investigated.

The last three chapters in this part describe three of the main modules within DOLPHIN's subkernel; MultiThreads, MultiPackets, and the memory manager. MultiThreads supports lightweight process management, using a scheme which makes it easy to port to new processors[2]. MultiPackets[3] provides a variety of network management primitives, including support for multicasting. Packet transmission is lossless, variable sized, and uses little network overhead. Lastly, the memory management facilities provide support for object and object descriptor allocation, and maintains dynamic buffer management for both short and long-term management-related data. This is based around an efficient single-chain freelist, providing automatic freespace compaction.

Part III contains some simple examples of DOLPHIN being used to support object-based networking, and future work drawn from the research into the DOLPHIN environment. The networking algorithm used is an adaptation of an virtual-memory system proposed in [Li and Hudak, 1989], modified for use with DOLPHIN-like objects. Although the scheme described has not yet been fully researched, initial results suggest that the performance of this algorithm is similar to (and in some respects better than) more mature systems (*e.g.* Orca).

Part IV contains some of the more interesting raw data collected during the analysis of DOLPHIN. It also contains an introduction to basic garbage collection techniques. This may be of interest to those readers interested in the algorithms used in local collection of the DOLPHIN environment, and also to those wishing to implement more elaborate collection schemes (*e.g.* networked collectors). A programmers reference guide for MultiThreads and MultiPackets can be found in appendix C and D respectively.

[2] MultiThreads is available for Unix-based machines, and uses no assembly language in its implementation. This allows rapid porting of MultiThreads to just about any available Unix platform.

[3] MultiPackets is also available for Unix machines, and is fully integrated with the Unix version of MultiThreads; if one thread blocks on the network, all other threads in a process can continue to execute.

Chapter 2

Background

This document is concerned with the design of orthogonally persistent systems. These systems represent a useful computing paradigm, offering significant advantages for application designers. Simply put, a system supporting persistence allows data to survive (persist) for as long as the data is required. *Orthogonal* persistence implies that all data types may be persistent, and all data can be manipulated in a uniform manner, independently from the length of time that the data has persisted.

In traditional operating systems, long-lived data is treated fundamentally differently from short-lived data. Such systems maintain short-term data under software control in virtual memory, and this memory is lost when a process terminates. Processes can terminate by choice, through external actions (*e.g.* the unix *kill* command), or by machine crashes. For data to outlive its creating process, it must be transferred to a database or file management system. This transfer necessitates code to `flatten' the data (*i.e.* translate short-term data pointers, *e.g.* addresses, to long-term pointers, *e.g.* disk-based offsets). In a persistent system, the data is maintained across process instances for as long as needed, and accessed transparently from its age. No data flattening or subsequent reconstruction is performed by the application at all.

There are three generally accepted rules (first proposed by [Atkinson *et al.*, 1983]) which must be met before a system can be described as `orthogonally persistent':

1. Persistence independence. Data creation and manipulation is independent of the data's lifespan. Transfer of data between long- and short-term storage is performed by the system, transparently from the application.

2. Type independence. All data (including programs) can persist for as long as required, independent of the data's type.

3. Data reference transparency. Identification (*i.e.* via a data pointer) and creation of data is orthogonal to the type system.

Unfortunately, many systems which describe themselves as persistent fail to achieve one or more of these requirements. Generally, all systems support the first rule by definition. Some systems allow only certain types of data to persist (*e.g.* only first order relations which do not contain pointers can persist in Pascal/R[Schmidt, 1977]). This lack of orthogonality can inhibit the implementation of certain applications. Other systems associate certain types with persistence (*e.g.* in [Liskov, 1985] and [Richardson and Carey, 1989]), which can result in dangling references and invalidated fields of structures [Morrison and Atkinson, 1990, Rosenberg, 1991].

Persistent systems exhibit three main advantages over conventional systems:

- A reduction in application complexity.
- A potential reduction in code size and execution time.
- The flexibility to implement a single model for data protection.

In conventional systems, the disk storage, tape drives, networks, and other input/output devices are all considered logically separate from main memory (figure 2.1). The system designer generally maps these subsystems onto a combination of a programming language and a database/file system. These two mappings can have complex interactions and may be somewhat incompatible. In a persistent system, all the subsystems can be manipulated directly as an area of persistent data or indirectly as part of the persistent heap's storage hierarchy.

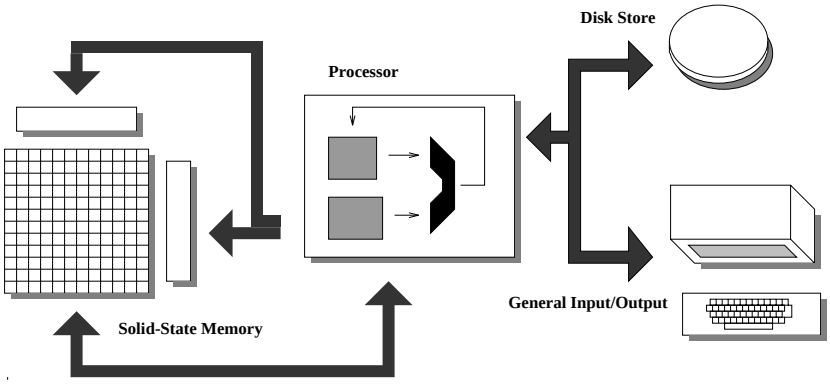


Figure 2.1: Overview of a Conventional System's Device Partitioning

With the removal of the need to flatten and reconstruct data when implementing long-term storage, the code used to perform this function can be deleted from applications. This is especially significant in database systems, and can amount to 30% of program code [Atkinson *et al.*, 1983]. With the removal of this code, execution times should also be reduced. In addition, the reliability of such applications may be increased, as reliably transferring complex data graphs between disk and memory can be error-prone. Finally, maintaining consistency between disk and memory versions of a dataset, especially where there is the need for the minimum of information loss after the loss of the memory-resident version of the data (*e.g.* after a power failure), is exceptionally difficult.

The requirement for multilevel data protection (*e.g.* file protection, memory protection, *etc.*) can be collapsed to a single unified protection scheme. This could be software- or hardware-based (or both).

Hardware supported protection has the additional advantage of providing data access from a range of different programming languages and their implementations.

The advantages of orthogonal persistence are not gained without cost; persistent systems place great demands on both hardware and supporting software. The main issues behind these demands include the requirements to:

- Provide transparent persistence of information. Failure to achieve this violates rule 1 of the requirements for an orthogonally persistent machine.
- Support data sharing over a network without compromising data consistency. Transparent sharing of data on the network simplifies program construction. There are however times when the sharing mechanism must be visible (*e.g.* to recover from machine failures where requested data cannot be accessed).
- Provide protection for data areas whose sizes are much smaller than that of a virtual page. In many systems, object sizes average much less than a virtual page, and it is essential in a secure system that when accessing one object another object on the same virtual page can never be accessed. Failure to protect against this can lead to the destruction of large amounts of objects (*i.e.* erroneously deleting an object root).
- Support concurrent data access by multiple processes. Without some sort of concurrency support, it is difficult to control updates of shared data objects.
- Maintain a object naming scheme which is valid over all machines on the network. If data is to be accessible throughout a network, then other objects pointed to by a parent object should still allow access to the child objects independently of where the parent (or the children) currently reside.

With a persistent store on a single machine, all objects which are reachable through *root* objects are considered to be persistent. Roots include the processor's registers while the computer is switched on, the stack of each process defined within the machine, and a top level object used by the operating system to hang all persistent data off when the machine is switched off. The single top level object is then accessed when power is restored to the machine.

In a networked object store, each machine contains its own roots, and when power is removed on any one machine the data stored there should be maintained. There is a question of how such networked stored scale to wide area networks; managing such a store is complex, especially as the store must remain operational even if machines in the network are switched off. Networked object stores are a current research topic, and are only discussed in this thesis as an example of the problems which can be examined by the DOLPHIN environment.

2.1 Supporting a Persistent Heap

The basic requirements in supporting a persistent object-oriented heap in the accessing, creation, and destruction of objects. Objects are accessed by their name, and this name is frequently referred to as

the object identifier, object id, or OID. In this document, PID or Persistent Object Identifier is also used when referring to an OID which is valid on a network. For the current discussion, OIDs and PIDs can be considered to be identical.

Each object in the object store is created of a given size, although many object systems allow objects to change size during their lifetimes. Data within an object can be accessed using its OID and an offset from the start of the object. Using an offset beyond the object's size or referring to an OID which does not point to an object is illegal and should be detected and signalled by the system. As most machines are at their lowest addressing level *physically addressed*, object data can only be accessed by first converting the OID and offset into a physical address. The scheme used to achieve this transformation varies from system to system, but without this translation the object-based heap is useless.

No explicit deletion of objects should be necessary. With a persistent heap, all processes running on the machine can have access to any and all of the objects stored therein. It is thus unclear when an object is no longer required, and deleting an object prematurely can result in programming errors. Instead, it is common to let the operating system handle the deletion of objects; a garbage collector running as a background process searches for objects which can no longer be reached directly or indirectly from any current or future process. All reachable objects can be found by traversing the roots of the heap; each process' stack and registers, and from the root object. The root object is an object which is never garbage collected, and as such is used to save objects which are not directly reachable from any running process (*e.g.* a user's personal database while the user is not currently running any programs). A summary of the commonly-used garbage collection techniques can be found in appendix B.

With the abolition of file-based storage, it is also useful to remove the idea of message-based network communication. The idea of a flat memory space can be expanded such that every machine on a network sees a single memory space. This requires co-operative memory management for garbage collection and object creation.

Maintaining persistence of objects can be a difficult task, especially over a network. Before an object can be accessed, it must first be loaded into main memory, either from backing store or over the network. At any time, failures can occur; the network can be interrupted, power removed from machines, operating system crashes, *etc.*. No matter what the failure, a persistent object must be preserved in a state recoverable in the future. Without this consistency of object data, the first failure which destroys a single object could result in a large number of other objects being garbage collected (*i.e.* any object which used the lost object as its only link between itself and a root).

2.2 Accessing Object-based Data

Simply providing a persistent heap for data storage may be insufficient for some languages. In object-oriented languages which permit dynamic typing, the type of information associated with an element of data in an object can be altered during execution by the program itself. Any other routines which access that data directly would automatically be able to handle the new structure. For example, consider a library routine dedicated to maintaining the current time and date. Supposing the designer of the

routine decided that it was only necessary to use an 8 bit word to store the part of the current time smaller than a single second. Initially this worked fine, but a year later the requirements for the routine changed such that this field now required a precision needing 32 bits of information. In a dynamically typed system, the change in this element from 8 to 32 bits would be automatically detected the next time this field was accessed by an application using this time routine. However, a non-dynamically typed system would require that every application which used the old structure for the current time be recompiled using the new structure. This is time-consuming at best, and may even be impossible (*e.g.* the source code for the application is unavailable, or in mission-critical systems where the application can not be stopped and restarted). Currently, for example in SELF [Chambers *et al.*, 1989], this is done through methods. However, this is only a partial solution, since in the example the method used to return the sub-seconds part of the time will now return an 32 bit integer instead of an 8 bit one.

2.2.1 Tagging

The most often used scheme for obtaining type changes in object data is via tagging. Consider the Ada definitions required to hold the board information for a game of Tick-Tack-Toe (program 2.1). This structure holds the name and running score for a player, plus a 3 by 3 array of board locations. The player's name is held in a string of 8 characters, while the score is maintained as a 32 bit integer. The board is constructed from an enumerated type which can hold either X, O, or EMPTY. Each type used in the program is mapped by the compiler onto one of the system-defined types; in this example an OID (an object id), INT32 (a 32 bit integer), FLOAT32 (a 32 bit floating point number), and INT8 (an 8 bit integer). A code is given to each of the types, which require two memory bits each. Each memory word in a tagged system has two parts, the data part and the tag field. Thus, with a 32 bit data requirement and 2 tag bits, the memory word in our example must be 34 bits wide. When the Ada structures are mapped onto our tagged object-oriented architecture, the directed graph shown in figure 2.2 is produced.

Tagging's main disadvantage is that the type of a data element can not be known before the data element is actually accessed. Thus the ALU is only notified of the type of its operands when it is actually called on to perform a calculation. This complicates the pipeline (especially if one of the types corresponds to a floating-point number) and reduces overall performance. Where a data element is referred to hundreds of times in a loop, a better solution would be to obtain the type of the element once, and then to use this knowledge in all further references to that element during the execution of the loop. Provided that the object is locked from modification by other processes during this time, any changes to the type are predictable by a compiler.

2.2.2 Object Structures and Arrays

Another method for identifying the types of object elements was proposed in [Balch *et al.*, 1989]. This scheme uses two representations for object data; arrays and structures. These two representations are bolted on top of a basic object model.

```
- - Define a Data Structure for the game of
- - Naughts and Crosses (Tic-Tac-Toe)

type NAME       is STRING(1..8);   - - Players Name
type SCORE      is NATURAL;        - - Players Score
type PIECE_TYPE is ( X,O,EMPTY ); - - Board Layout
type OXO        is array (1..3) (1..3) of PIECE_TYPE;

type STATUS is
    record
        PLAYER        : NAME;
        GAMES_WON     : SCORE;
        LAST_POSITION : OXO;
    end record;

GAME : STATUS;
```

Program 2.1: Program to Demonstrate Tagging

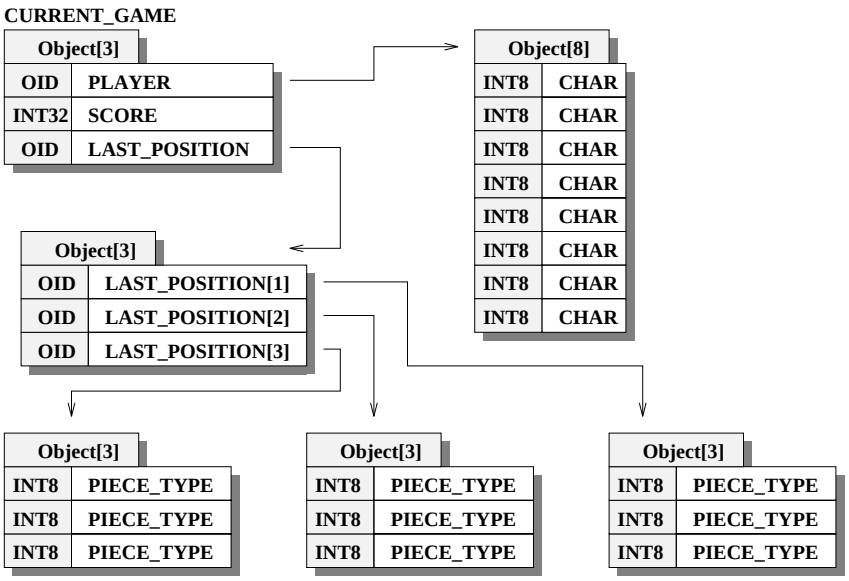


Figure 2.2: Object Representation of GAME (program 2.1)

In an array object, a header is provided which identifies the type of all elements contained with the array. This is a considerable saving for large arrays over individually tagging each array element. Also contained within the header is the code marking this header as an array header, a length in elements for the array, and the width in bytes of each array element. This structure is depicted in figure 2.3.

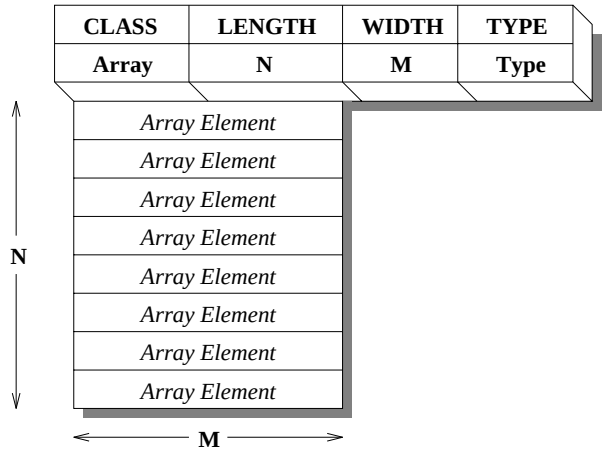


Figure 2.3: Possible Representation of an Array

With the structure object, again a header is provided identifying the object as a structure. The remainder of the header contains a length field for every element type supported in the architecture. Figure 2.4 depicts an example with four possible types; OID, FLOAT, INT32, and CHARacters. The actual data part of the object is arranged in memory size order, largest first. The example shows a structure with two object identifiers, a single 32 bit integer, and two character (8 bit) fields.

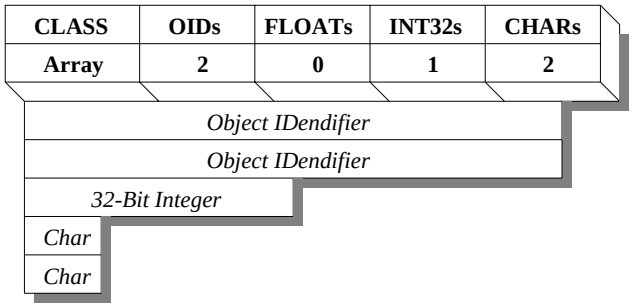


Figure 2.4: Possible Representation of a Structure

Using this scheme, data element types can be easily identified without the use of hardware-based tagging. Compilers can load the types once, and then use the loaded information for any number of subsequent data accesses. Unfortunately, this scheme does not easily support dynamic changes to the contents of objects in terms of new elements, modifying the types of elements, or even changing the size of individual elements.

Other type representations are possible, but are not discussed here as they are typically minor modifications of the above scheme. Provided OIDs can be protected from being modified and subsequently

used in an object reference (which is illegal, as only the operating system is allowed to generate OIDs), a variety of type identification schemes can be built efficiently on top of a simple non-tagged memory.

Chapter 3

Object-Oriented Systems

Research into persistent object-oriented architectures has been underway for three decades. Work on persistence is ultimately derived from the development by Kilburn *et al.* of the Manchester University Atlas machine [Kilburn *et al.*, 1962], which introduced the idea of *virtual memory*. Virtual memory was initially used as a mechanism for supporting disk caching. In an earlier machine, Kilburn [Kilburn *et al.*, 1956] combined the then new transistors with a magnetic drum for the instruction store. Previous to this, most stores were constructed from William's Tubes, mercury delay lines, magnetic cores, and moving magnetic devices. The drum-based instruction store resulted in a slow machine cycle time of 30 milliseconds, but with a store of significantly lower cost than its competitors. The virtual memory of the Atlas was used to improve the performance of drum-based systems; information was still held on the drum, but a small number of pages could be *cached* by re-mapping their virtual addresses onto magnetic cores.

At this point in history, computers were still used as processing machines utilizing off-line storage. This differs from the modern view of computers, which can interact with on-line storage and act as long-term data repositories. Even in the 1960s, with all main memory technologies based on magnetic effects and thus non-volatile, long-term data was always relegated to tapes.

The development of disk storage, which was viewed from the start as a long-term storage medium, offered interactive access to a persistent store. Initial attempts to map disks and short-term memory into a single-level store, *e.g.* Multics [Organick, 1972] and Emas [Whitfield and Wight, 1973], were not universally accepted. Here, the desire was to map each data and instruction file of the type previously stored on tape onto a range of virtual addresses. This was not without problems: disk storage requirements exceeded address-space limits, and if a range of addresses were allocated to a particular file, then file growth was difficult to perform without relocating the entire file.

As a solution to the problems introduced by the single-level store, the idea of *segmentation* was advocated by Iliffe [Blaauw and Brooks, 1964]. With a segmented architecture, memory is segmented into variable length segments, each identified by a unique segment number. In many systems, segments

are also referred to as *objects*.

In an object-oriented heap, no user-level virtually-addressed memory accesses are permitted. Instead, all data and instructions needed by applications are partitioned into objects. These objects are created with a specified length, and are identified using an *object identifier* (OID). An object can be referenced using the OID, and data elements within an object accessed using OID[index]. Using an index which is larger than the object length is illegal and should be trapped. Allowing users to generate descriptors independently should be carefully controlled, since this could allow users to find objects which they should have no access to.

Many programming languages, such as C, use linear addressing schemes as their underlying memory structure. Here, segmented architecture support would offer no significant advantages, but it can be exploited. For example, by mapping the stack into one segment, and holding heap data within various data and program segments, *etc.*, C can make use of the segmentation as a protection mechanisms. This would at the very least be an aid to debugging. However, object-based languages can make extensive use of an object-oriented heap, where the extra layer of indirection introduced between processor and memory allows for powerful system-level (as opposed to compiler-level) management of objects.

When using object-based addressing, it is important to verify that certain conditions hold when accessing objects via descriptors. Since it should not be possible for the user to fabricate descriptors (either by accidentally corrupting a descriptor or by deliberately trying to find objects created by other users), either descriptors must be immutable by user processes, or the probability of finding a valid object identifier without first being given it should be prohibitively small. If the user is able to modify descriptors, then the mapping process between descriptors and actual object addresses must be able to detect when a descriptor is illegal. For security reasons, attempting to use a descriptor which does not correspond to an object in the heap should result in the death of the current process.

Even if the descriptor is genuine, the index into the object must lie within the limits of the object length. Attempts to access an object outside its length should at least be signalled to the user. If the object heap uses multiple storage devices (*e.g.* RAM and disk), then the part of the object being accessed must be present on a memory-addressable device (*e.g.* in RAM). This can be handled in a similar way to a virtual pager.

Supporting object accesses in programs can be done in either software or hardware. Hardware implementations have many advantages over the software approach:

- Compiler writers need not be trusted. With object access verification in hardware, software failures in the program/compiler (either accidental or deliberate) cannot affect objects other than those accessible by the program.

- Software writers can write in assembly code, as the object access mechanism is contained within the processor itself.

- Compiler optimization of object accesses is unnecessary. In a software approach, equivalent speed for executing applications to a hardware-based object addressing mechanism is only possible if much of the software overhead in accessing objects is optimized away. For example, this involves data flow traces and other analysis techniques to remove bounds-checking of objects

where the analysis has shown that a boundary violation is not possible.

- With a data heap completely accessed using a system-wide object-based addressing scheme (rather than objects only being visible from the compiler up), system supported mechanisms for garbage collection and networking can be implemented transparently from the application writer.

In the remainder of this chapter, a number of object-oriented architectures are examined: the IBM SYSTEM/38, Intel's iAPX 432, MONADS, MUTABOR, and the Rekursiv. These cover most aspects of current object-oriented designs. Special emphasis is placed on investigating their object-address translation mechanisms and object data caching, since these features have a significant bearing on overall system performance. Processor speeds and memory bandwidth problems within each processor are also discussed, but only in the context of modern implementation limitations. This is to remove the effects of different fabrication technology limitations present at the time of each system's implementation. Also of interest is the range of object sizes supported. Object size characteristics are dependent on both the application and the implementation language. For a flexible system, a wide range of object sizes should be supported (from bytes to GBytes).

3.1 SYSTEM/38

The IBM SYSTEM/38 family of small business computer systems [Soltis, 1977, Soltis and Hoffman, 1987] was announced in October 1978. The previous machine, the 360 series, used 24 bit addresses, and this had been identified as a limitation by IBM engineers. The overall aim for the new machine was to provide small-scale users with facilities previously available only on large systems. The architecture was designed to promote a long life-span, and cover a range of hardware implementations. Even today, elements of the SYSTEM/38 can still be found in the newer IBM AS/400 machine. As a step towards these objectives, SYSTEM/38 utilizes 64 bit addresses.

The instruction set used supports ad-hoc polymorphism of data types and an implementation independence from hardware data-flow characteristics such as the number of physical registers [Dahlby *et al.*, 1982]. The memory is constructed as a single-level store, and all memory management is performed automatically by the machine. Memory is accessed by objects using capabilities, which in turn identify objects using OIDs. A capability is a data record, which contains the OID of the object in question, access rights to that object (*e.g.* read only, read/write, *etc.*), and other system dependent information (*e.g.* object type) [Houdek *et al.*, 1981, Farby, 1974]. Memory is configured as 40 bits wide; 32 data bits, 7 ECC (Error Correcting Code) bits, and a single tag bit.

3.1.1 Object Identifiers

The single-level store is divided up into 64 KByte segments. A segment group is formed from 256 segments. Each segment contains a header, visible only to the microcode, which defines the attributes of all object contained therein [Soltis, 1977]. New objects are created using an object template [Pinnow

et al., 1982]. Capabilities are represented using 16 bytes. Each capability is split into four fields (see figure 3.1):

- *Offset*: A 24 bit byte offset within a segment group. This allows parts of objects to be selected.
- *SGN*: A 24 bit segment group number. Each segment group is allocated a unique number.
- *SGE*: A 16 bit segment group extender. If the SGN is reused, a new SGE value is chosen. This helps to prevent accidental access of object which have been deleted (there is no automatic garbage collection).
- *Type Info*: A 64 bit field containing the type of the object being referenced, access rights, and other object attributes.

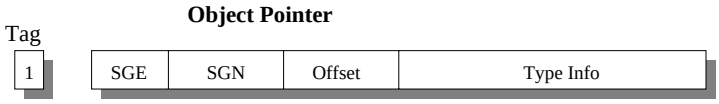


Figure 3.1: A SYSTEM/38 Object Descriptor

When an object is accessed, the SGE of the capability is compared with the current SGE linked to that object. If they are different, then the capability must have referred to an object which has been deleted, and an error is raised. This is a necessary safety check, as all object deletions are explicitly specified by the application, and not using a system-level garbage collector. With half the capability used for type information, and 16 bits for the SGE, the remaining 48 bits are used directly as a virtual address.

The memory tag bit is used to validate object capabilities. Four tag bits must be set consecutively if those memory words contain a capability. The tags are set by the processor, and cannot be altered directly by the user. Special instructions are used in handling capabilities, which validate the capabilities by checking the tag bits. Writing data which does not involve transferring a whole capability into memory clears the altered memory word's tag bits.

3.1.2 Object Mapping and Cache Layout

Virtual-to-physical address translation is supported by microcode. The 48 bits of virtual address are considered as a 39 bit page address and a 9 bit offset into a 512 byte page. This small page size is interesting; page transfers between disk and memory will use less bandwidth than a larger page size, but more pages must be managed as a consequence, requiring more CPU and memory resources to control. Small page sizes could also lead to a higher latency, as page faults may occur more often that would be the case with larger sizes.

Considering the large number (2^{39}) of entries required to fully map the address space, a conventional hierarchical page table would be impractical. A single page of memory could hold 64 page descriptors (64 bits each), needing some 2^{33} pages for the complete mapping. Of course one need not

map it fully, one only needs mapping pages for pages currently resident, but an object address space is likely to sparsely used. Each object occupies a segment group taking up 16 megabytes of address space. This alone requires 3 levels of mapping tables, so the smallest resident object would tie down 3 page frames in mapping information. In addition to the space constraints, there are speed considerations. some 6 or 7 levels of indirection would be involved in converting a logical to a physical address.

To avoid these problems the notion of the inverted page table was developed independently for the SYSTEM/38 and for MONADS [Abramson, 1981, Edwards *et al.*, 1980]. The technology was later adopted on the IBM 801 experimental RISC processor [Chang and Mergen, 1988] and the RS/6000 series [Malhotra and Munroe, 1992]. In use, the page address is hashed and used as an index into the *hash table index*. This produces a second index, which when applied to the *page directory* produces the head of a linked list. This list is then scanned until the initial page address matches that of an element in the list. By subtracting the address of this element from the address of the page directory, the frame identifier is produced. This process can be seen in figure 3.2.

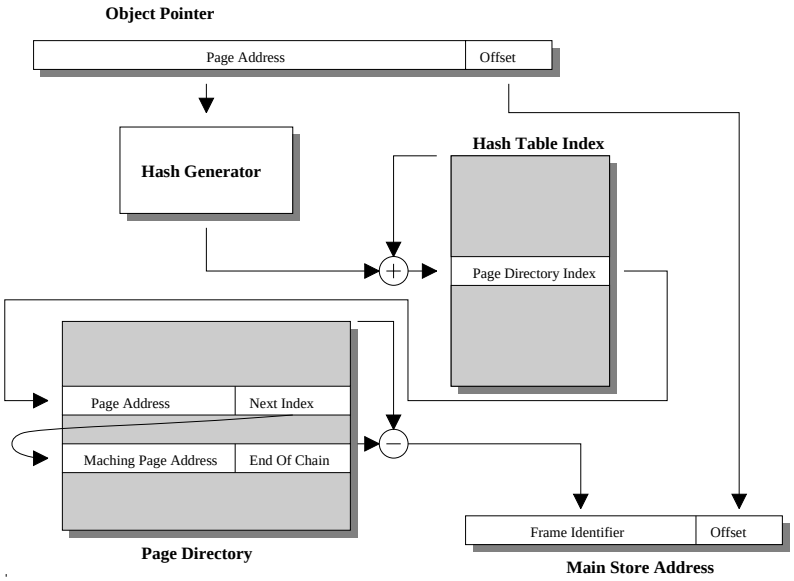


Figure 3.2: The Inverted Page Table.

The linked list contains all collisions resulting from the hashing of the page addresses, and must be scanned serially during a conversion. The entire collision chain for an address must be searched before an unmapped object can be detected as such, and this may have a detrimental effect on performance.

To further improve the translation performance, a hardware lookaside buffer is used (a TLB). This is organized as a 64 entry 2-way set associative cache. This gives a significant performance improvement [Soltis, 1977]. A direct-mapped cache would have offered more opportunities for optimistic cache accesses, at the expense of a lower hit-rate. Furthermore, bounds-checking of object limits is not a part of the virtual to address translation, but must be handled separately in microcode.

Objects can be either implicitly or explicitly locked. Simultaneous accesses by two or more processes are automatically performed either serialized (if any of the accesses involves a write) or in

parallel. There is also explicit locking available to the programmer [Berstis *et al.*, 1982]. Locking is on a per-object basis.

3.1.3 Conclusions

SYSTEM/38 exhibits a large address space (although unable to support networked virtual addressing), well thought out operating system support, and an efficient paging strategy. It was a considerable commercial success, with tens of thousands of units sold. Packaged with its own operating system and relational database providing a high-level user interface, most users were completely unaware of the underlying object-based heap.

The lack of automatic garbage collection was an interesting omission from the design. The inclusion of collection techniques would have made the object descriptor's SGE field obsolete (permitting the SGE's bits to be used in supporting a larger object space).

In RISC philosophy, it is preferable to allow only memory accesses with addresses aligned to the size of the transfer selected. For example, bytes can be stored anywhere in memory, but 32-bit transfers must occur on 32-bit boundaries (*i.e.* the bottom two bits of the address must be zero). This simplifies cache access systems, *e.g.* if 32 bits to be accessed in the cache was stored as 16 bits on one cache line and 16 bits on another, two cache accesses would be needed. Provided that the alignment of the cache lines was a multiple of the largest data element supported by the processor, and that memory addresses are always aligned to the size of the transfer required, then a transfer will never span more than a single cache line. Unaligned memory references are a rare occurrence in most programs (page E-12 of [Hennessy and Patterson, 1990]), and RISC philosophy suggests optimizing for the frequent case. Forcing alignment of SYSTEM/38 capabilities to 16 byte boundaries would have allowed future versions of the processor to use a cache more effectively.

The type information which is part of each segment and object capability would best have been held within the objects themselves. Inheritance of object types would require that all the inherited methods be copied to a new segment, with objects of the new type created from there. This hierarchical segment/object structure imposed in SYSTEM/38 is not as flexible as a single-level object store.

Subroutines within the same process may have differing object access rights, and there are options to allow inheritance of access rights between subroutines (and also with child processes). Subroutine calls and returns can therefore be CPU-intensive tasks [Hoffman and Soltis, 1982]. The effect of this depends on the period between capability changes, which in turn relates to the complexity of the types present in the application program. If the program uses simple types, then the code to support those types would be small, and thus the time between capability changes would be short.

3.2 iAPX 432

Conceived in 1975, the Intel iAPX 432 [van Rumste, 1983] represented a rapid departure from the architectural norm. The main push for the project was based on speculation by LSI specialists, who estimated that by 1980 it would be possible to integrate 100000 transistors on a single chip. The

question was how to use such devices constructively; memory chips could easily be scaled up, but processing devices were another matter. It was therefore decided to design an architecture which could make use of such high-density integration techniques.

Three areas of current computer systems were initially identified as topics to investigate:

1. Data handling and protection.
2. Parallel and concurrent process management.
3. Multiprocessor control.

Also of interest was the effects of the software crisis; software costs had dramatically increased while hardware costs declined with technology improvements. At that time as much as 80% of the total cost of a computer system can be attributed to software [Corradi and Natali, 1983]. Today, this is probably even higher. It therefore makes little sense to minimize hardware costs at the expense of software, and thus the primary objective was made to provide extensive support for the software programming environment, with the aim of reducing the software life-cycle costs. This lead the 432 to being the first architecture to implement a software-transparent multiprocessor design, and commercially the first system to attempt support for the object-oriented programming methodology [Hunter *et al.*, 1985].

The 432 was designed specially for programming in a high level language (in fact its operating system, iMAX [Kahn *et al.*, 1981], was written in terms of Ada packages), and used a complex instruction set to reduce the semantic gap between hardware and software. Each instruction was encoded so as to minimize executable size, and to maximize the benefits of this the decode logic was bit-aligned.

The choice of Ada for the 432 was not an arbitrary one [Zeigler *et al.*, 1981]. Initially, the main system language was to be Chill [Corradi and Natali, 1983], but in 1977 the United States Department of Defence (DoD) embarked on the development of Ada, a new language to solve all of its software engineering problems. This language was to be the standard development language for all the DoD's programming projects. The Intel Corporation intended to establish itself as the centre of Ada technology by using the language throughout the iAPX 432 design. However, Ada was envisaged with embedded systems in mind, whereas the iAPX 432 required dynamic system control. Thus a number of extensions to standard Ada [USDoD, 1983] were required [Intel, 1981a]. These extensions supported the following features:

- Implementation to be selected at execution time. An example of this could be a choice between two different sorting routines based on the number of items to be sorted.
- Implementation to be altered at execution time. For example, the screen output package could be dynamically replaced by a file output package.
- Implementation unknown. The user may wish to write a routine which interacts with another routine who's implementation in unknown.
- Data structures partially unknown. In a sorting routine, the sorting mechanism usually works on a single key element of each item to be sorted. It is thus possible to sort a list of structures, where only the sort key is known.

- Data structures entirely unknown. A garbage collector must be able to work with all defined structures, even if they were not defined at the collector's compilation time.

3.2.1 Physical Overview

The iAPX 432 was designed to be a multiprocessor architecture with task distribution transparency. There were in fact two different processing elements used within the design; the General Data Processor (GDP) [Budde *et al.*, 1983] and the Interface Processor (IP) [Bayliss *et al.*, 1983]. Memory storage was held in memory subsystems, which were shared between all GDPs over the multiprocessor bus. External devices, such as mass storage and communication systems were handled through the IPs, and thus result in the off-loading of I/O processing from the GDPs. An overview of an iAPX 432 example system is shown in figure 3.3.

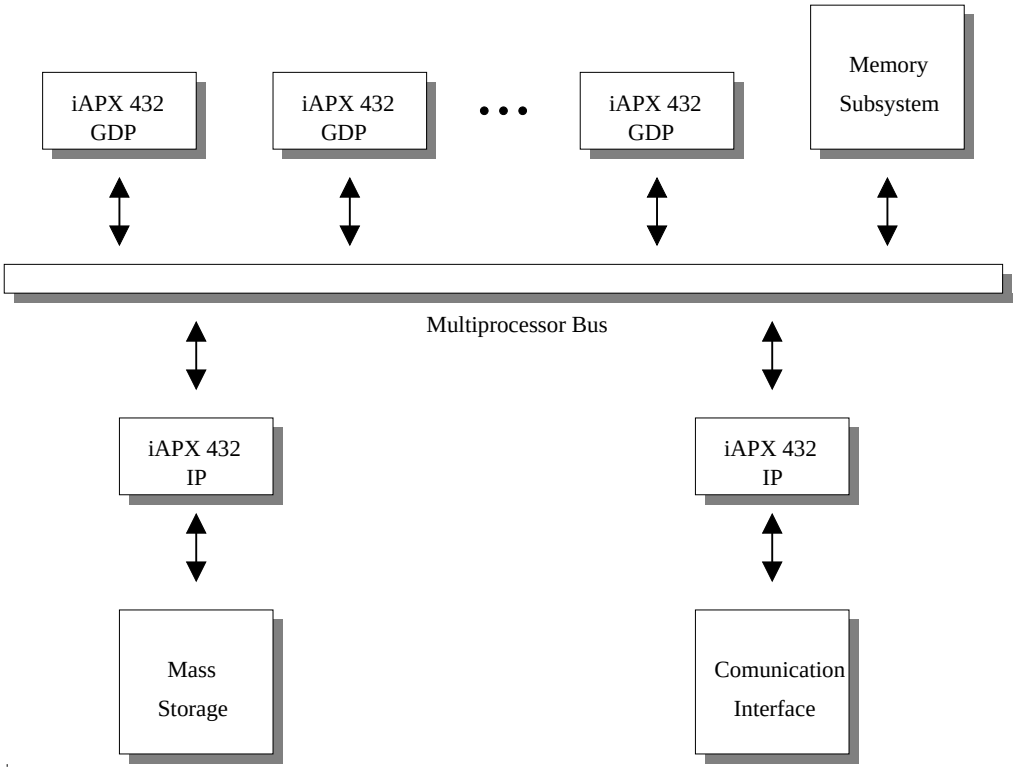


Figure 3.3: Overview of a iAPX 432 System

The GDP themselves were implemented using two VLSI devices [Intel, 1981b], with one designated as the Decoder/Microcode Sequencer (with 110000 transistors) and the other as the Execution Unit (using 49000 transistors) [Gupta and Toong, 1983b]. In comparison, the IP used only 60000 transistors.

Each device interface was a self-contained computer, containing ROM, RAM and DMA devices. The processor used in the interface typically contained an 8 or 16 bit device which has the ability to

handle interrupts (*e.g.* Intel's 8085 or 8086). Communication between the GDPs and the attached processors was accomplished via the IPs, which allowed limited access to the 432's memory subsystems. Ada was rarely used to program the attached processor, and in the case of the 8086 PL/M was often selected as the programming language. One possible representation of an interface processor is depicted in figure 3.4.

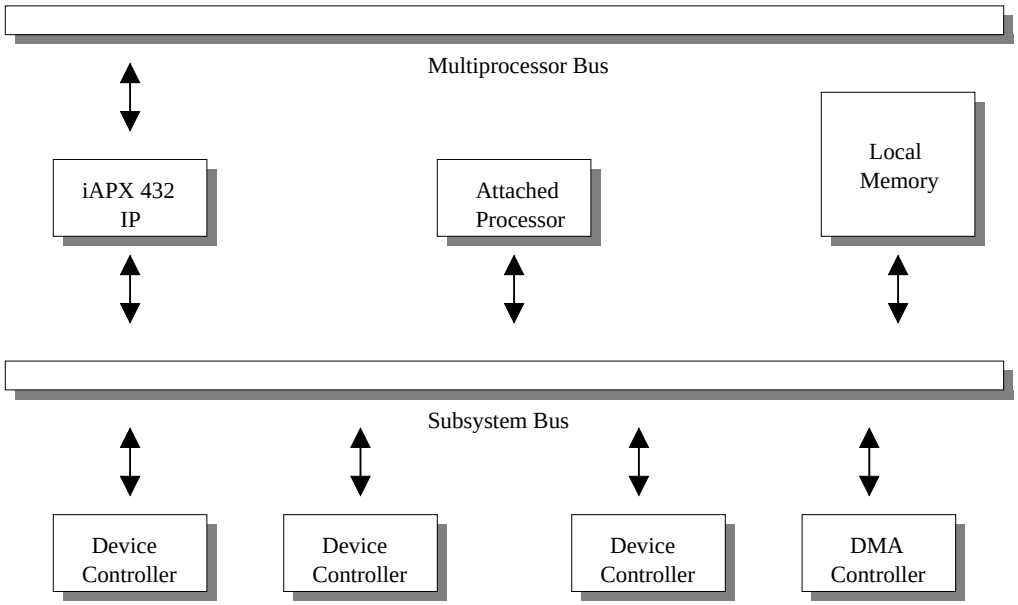


Figure 3.4: Close-up of the iAPX 432 Interface Processor

No interrupts are available within the 432 environment, and where required for controlling devices they are managed within the IPs. The Ada rendezvous is used to communicate between GDPs and IPs, and since this type of Ada communication is *blocking*, no interrupts are needed over the multiprocessor bus (*i.e.* the GDP-based task is forced to wait until the IP responds with the necessary information).

3.2.2 Object Management

The heap space in the 432 is divided into two distinct zones; *active space* and *passive space* [Pollack *et al.*, 1981]. In the passive space resides all the persistent objects, each of which can be accessed only by first moving them into the active space. Passive space objects use full object identifiers (*i.e.* 80 bit numbers) in referring to other objects. If the user initiates an access to an object contained within the passive store, then it is transferred to the active store, and the object identifiers contained within the object (UIDs) translated automatically into 24 bit active object identifiers. All objects identifiers within the passive store are active object identifiers. A record of the conversion is held in a two-way mapping table (UIDs/active object identifiers).

Of the 80 bits used for passive space object references, only 40 bits were used to locate objects. The UID was formed from four component parts; a checksum (16 bits), generation number (24 bits), POD index (24 bits), and a logical device number (16 bits). An object's POD (Passive Object Directory)

index was an offset into a table unique to each logical device, and indicated where the object resided on that device. The generation number was used to allow the POD index and logical device number to be reused, while still maintaining unique UIDs throughout the life of the system. Finally, the checksum was calculated from the other fields, and was used as an initial validation on an object reference. By making the storage device for that object part of the object's UID, flexibility in object relocation and other types of UID manipulation (such as that required for networking) was compromised.

This two space approach contrasts strongly with the one space method used by other object oriented architectures, *e.g.* the SYSTEM/38. In two space systems, the deliberate separation of the permanent store and the active objects has one main advantage; *robustness*. One of the main problems in a single space architecture is maintaining consistency between main memory and the backing store. It is the operating system which must decide when an object should be written back, and in the SYSTEM/38 this can lead to object being written out to the persistent store when it is not itself in a consistent state (*i.e.*halfway through changing a number of entries within the object).

The problem of consistency lies with potential system failures. If the system is suddenly stopped and its main memory cleared, then the system can only reload the last version of objects which were saved to the persistent store. If an object was undergoing modification at the time of the most recent save, then the contents of the object will be in an inconsistent state. This inconsistency is something to avoid. Instead, it is desirable to guarantee that objects will not be written to the persistent store unless all objects which depend on each other are all consistent, and even then all such objects must be written atomically.

For the 432, *type managers* control access to all objects. At the start of an object access, these managers indicate that an active copy of the object is needed. Once this state is achieved, then the object is altered as required until it is once again in a consistent state (*i.e.* at the completion of the type manager's modifications), at which time the object is permitted to retire into passive space. In fact, objects are not transferred to the passive space until the object is no longer accessible via any active identifier. Thus the transfer mechanism between the two spaces is only visible to the type managers (but it is visible nonetheless).

Object allocation was controlled by the compiler, using the basic Ada scope rules. Dynamic memory allocation and garbage collection [Pollack *et al.*, 1982] was maintained by the operating system, using an incremental mark-and-sweep algorithm. All OIDs written to memory were set to unswept (traditionally called the grey bit), signalling the compiler not to delete the corresponding object (and all objects below it in its reference tree) from memory. This meant that each instruction to save an OID into memory incurred an additional 9 cycle penalty in marking the grey-bit. A more standard mark-and-sweep garbage collector would have allowed OID writes in application programs to execute at a faster rate, while perhaps slowing down the garbage collection routines. As the collector runs in the background, then provided the system in not in danger of running out of memory space, the significance of slowing down the collector is negligible.

Object Mapping

The complete addressing cycle [Colwell *et al.*, 1988] used in the 432 is shown in figure 3.5. Given an active object address and object offset, the processor first finds the access rights and location of that object. The access rights are located through a two-level address translation scheme. This two-level approach is necessary since all objects are limited to only 64 KBytes in size[1], and that all system data structures (including the paging tables) are treated as objects. The active OID is split into a current context selector and an index. The selector is translated via the current context object into an object address/index pair within what amounts to a second-level page table. This refers in turn to an access descriptor, which contains the access rights for the referenced object, and the actual active OID of the object's data. The data' active OID is converted to a physical address via the same two stage process used in finding the access rights, which produces a base address for the object data. Finally, the data address is offset with the original object offset, producing the specific data item required.

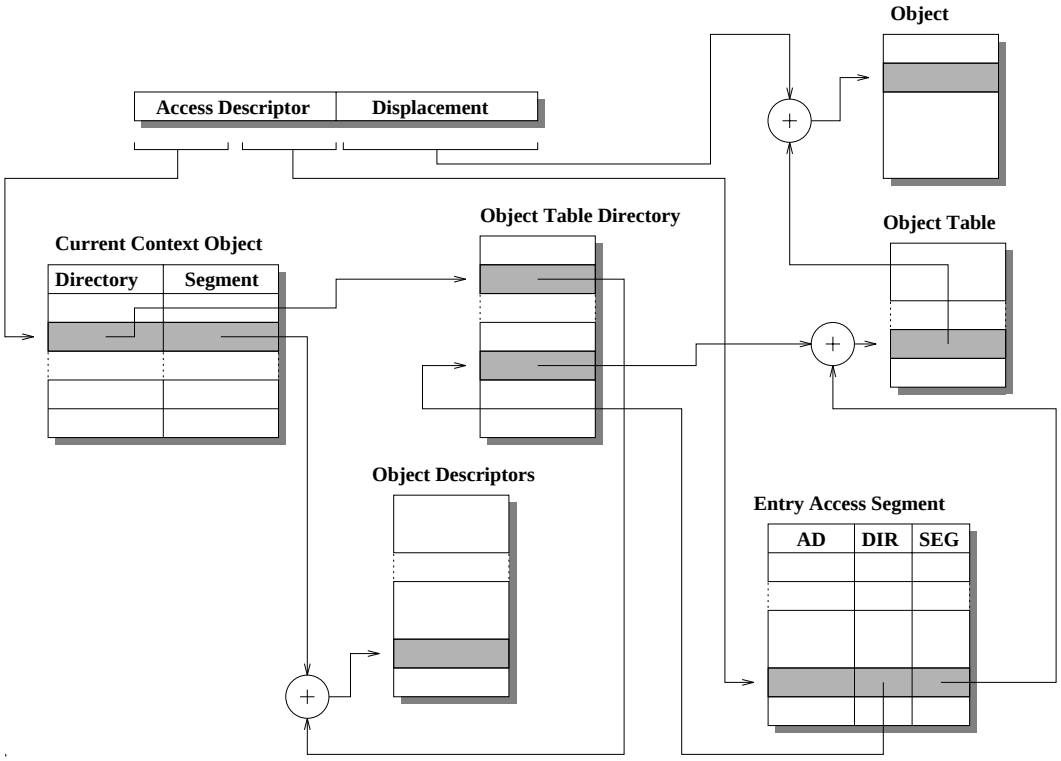


Figure 3.5: The complete addressing cycle of the iAPX-432

On-chip caching of object translation information was restricted to a 5-entry data segment cache (caching the last 5 object-to-physical address translations) and a 4-entry object-table cache (caching access descriptor to second-level address translation page). The processor also contained 14 base/length register pairs for storing links to system objects. The lack of user-accessibly registers was cited as one of the major flaws in the overall design. Note that the data-segment cache was flushed on every

[1] In fact each object is limited to 128 KBytes, but half of this must be for data and the other half for other object descriptors.

procedure call/return.

3.2.3 Performance

Procedure calls and returns are potentially quite expensive in object-based systems such as the 432. On a conventional architecture, the graph structure of a program's call patterns and data structure accesses is represented in terms of virtual addresses embedded within the executing instructions. Within the 432, the graph information is maintained in terms of access rights, object identifiers, and messages. All this information must be converted at run-time into physical addresses. The 432 requires 16 read accesses to memory and 24 write accesses for a procedure call, and consumes an amazing 982 machine cycles in the process! This gives clear advice to future object-based processor designers; if you use OIDs for procedure calls make sure that they can be translated quickly…

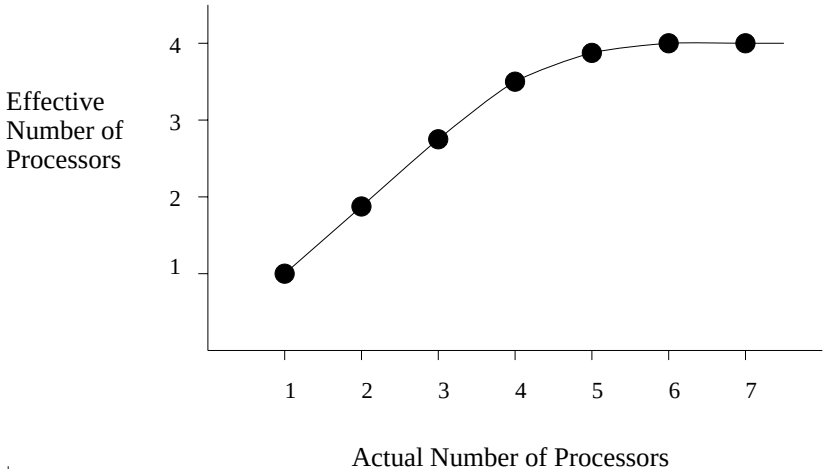


Figure 3.6: Graph of Performance vs Number of Processors

A multi-bus implementation of the 432, using bus controllers to connect onto the buses and memory controllers to process the memory requests from the multiple bus sources was initially planned by Intel. This would have undoubtably improved the performance figures shown in figure 3.6. However the required bus and memory controllers never appeared.

The instruction set allowed for bit-aligned memory references, which increased the decode logic complexity considerably without any noticeable advantages. The instruction set also excludes any literal or other embedded data handling capabilities (other than for zero and one), thus necessitating object accesses even for frequently used constants.

On the plus side, the 432 has an unparalleled performance in terms of inter-process communication [Cox *et al.*, 1983]. In a comparison between Medusa [Ousterhout *et al.*, 1980], StarOS [Jones *et al.*, 1979], RMX/86 and the 432, the 432 was always near the top of the performance ratings. Admittedly the performance measurement was taken independently from clock speed, since the sheer complexity of the 432 was such that clock-rates were significantly lower than the simpler (*i.e.* the majority) of other processors available at that time. This leads onto another important recommendation for our

new processor proposal; do not sacrifice overall performance for specific system enhancements. The 432 has excellent communication performance, but for the other 99% of the time it ran an order of magnitude slower than its competitors. It is easy to criticize with hindsight, but rather than blaming the designers for their errors it is more important to make sure that these errors do not occur in the system proposed within this thesis.

3.3 MONADS

The MONADS project was established in 1976. Its main objective was to examine and improve the methods used during the design and development phase of large-scale software systems. MONADS is still running today, and has branched out into the creation of an object-oriented environment based on hierarchical segmented virtual paged memory. Note that MONADS is only a research machine, where the design objectives are targeted towards investigating object systems, rather than being a commercial success (*e.g.* as was the case with the iAPX-432).

There have been four MONADS systems developed; MONADS-I, -II, -PC, and -MM. MONADS-I and -II were both based on Hewlett-Packard 2100A minicomputers, but in recent years the project team have been concentrating on MONADS-PC [Keedy, 1984, Rosenberg and Abramson, 1985] (a custom microcoded system) and -MM [Koch and Rosenberg, 1990] (a custom co-processor and SPARC processor combination). Both machines can be programmed in a number of languages, including a dialect of PASCAL and (to a limited extent) LEIBNIZ [Keedy and Rosenberg, 1989]. MONADS was envisioned with three major design philosophies; a module structure, an in-process model for service requests, and a uniform virtual memory. A module may contain a number of objects and executable routines, but objects within a module can not be accessed from routines external to that module. Thus, objects contained within a module can only be accessed via routines within the object's module, with data being passed by value on a stack.

The whole approach to operating system composition follows the `in-process' model [Lauer and Needham, 1979] for service requests[2]. Although Keedy [Keedy, 1980] argues that such a mechanism has many inherent advantages in a dynamic system, the domain switching and domain protection required to handle this type of design satisfactory generally requires significant hardware support.

In evaluating the MONADS design strategy, careful consideration of the object management system should be performed, including the effects of its block-type information hiding philosophy.

3.3.1 MONADS Memory Management

The fundamental user-level[3] memory addressing mechanism used in MONADS is the virtual memory address. This address is similar in design for both the machine variants, with the PC utilizing 60 bits and the MM 128. The layout of the virtual address can be seen in figure 3.7.

[2] Procedure-oriented or `in-process' techniques invoke operating system **instances** via procedure based access, as opposed to `out-of-process', which communicates with operating system **tasks** through communication channels.

[3] Although clearly the fundamental memory addressing mechanism for the underlying machine is the physical address.

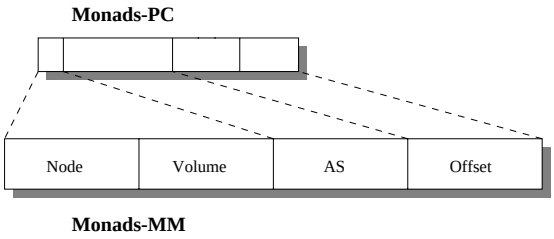


Figure 3.7: Virtual Address for MONADS

From figure 3.7, it can be seen that the virtual address is split into four component parts, *i.e.* the node, volume, AS and offset.

- The virtual memory node number indicates the machine's network address on which the desired physical memory slot is located. This is 32 bits long for the MONADS-MM machine. This may not be a sufficiently large enough range of values, especially considering that some network addressing systems now make use of 48 bit address fields (remember that many network routers rely on a hierarchical physical location description being stored within the network address, thus the resulting address description is sparse). It would be possible to alias the 32 bits to whatever was used in the global network, but then the all OIDs within an object would have to be converted whenever that object left or entered a particular machine (the mapping function would undoubtably be different on different machines).

- Disks within MONADS can be identified by one or more volume numbers. If more than one volume number refers to a single disk, then each number represents a partition within that disk. This field is 32 bits long for the MONADS-MM machine. However, this field with the node number is represented by only 6 bits in the MONADS-PC system. This is clearly insufficient for any realistic networked system. A concern arises as to how disks are replaced or updated when the OIDs contain fixed volume numbers. This is not a fundamental problem in MONADS, as a table converting these volume numbers (which would now be considered as *logical* volume numbers) to true disk volume numbers could remove this concern.

- Persistent memory is split up into a number of separate `chunks', with each of these chunks given a unique identifier. These chunks are known as *address spaces*, identified by address space numbers. Each of these spaces are divided up into a number of fixed sized pages. Note that these address space numbers are never reused, even after the memory chunk has been deleted. In MONADS-MM, these chunks can be up to 4 GBytes in size.

- In order to access data elements within each address space, an offset can be provided such that address plus offset identifies a unique memory location. MONADS-MM uses 32 bits for the offset field.

With the virtual address partitioned in such a rigid manner, the virtual addresses become fragmented (especially since virtual addresses are not reused under MONADS). Note that creating and then deleting

address spaces in a rapid manner could force the system to run out of virtual addresses. From the languages available for MONADS, objects are viewed in a form whereby they are long-lived, large, and created slowly. Reuse of virtual addresses is not an important issue with such languages. However, the use of other, more dynamic languages (such as SMALLTALK) would be hampered by this approach.

Note that the virtual addresses are not directly used by executing tasks. Instead, objects (or segments as they are known in MONADS) are accessed through object identifiers. These identifiers refer to object capabilities (similar to IBM SYSTEM/38 capabilities) [Fabry, 1974], and form the basis for object protection under MONADS.

3.3.2 MONADS Segment Capabilities

All segments used in MONADS are referenced via capabilities. Each segment is split into a number of parts (as in figure 3.8). The first part of a segment is known as the *segment control section*. This contains the actual size of the segment and details describing its contents. The actual data part of the segment is called the *information section*. It is here that information can be stored and retrieved, based on the segment control data. The final part of a segment is referred to as the *segment capability section*. Within this area references to other segments can be placed so as to permit arbitrarily complex graph structures to be represented on the system.

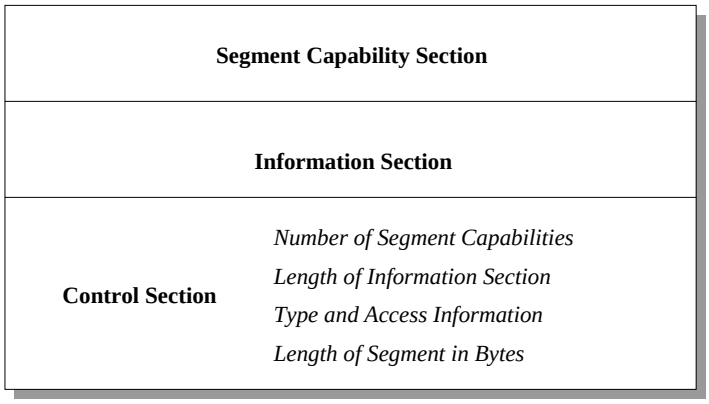


Figure 3.8: Segment Layout in MONADS

This type of fixed segment layout contains limitations which increases the implementation complexity of object oriented languages. This problem can most clearly be observed with respect to type inheritance. In typical object oriented languages, inheritance is supported by combining a current segment type with a new segment (which contains the supertype information). The combining process is often achieved by tagging the new segment at the end of the older one. This is not possible in MONADS, where it is impossible to have an arbitrary combination of data and capabilities. Inheritance may be implemented by using some kind of indirection from a `higher level' segment, which itself may be inherited by a parent in the same way, but this involves the expense of performing the indirections.

To refer to information contained within any segment, all that is required is the address of the segment (all other information concerning the segment being self contained). For efficiency, registers,

known as *capability registers*, are used when referring to segments. Reference to an element within a segment via such a register is depicted in figure 3.9.

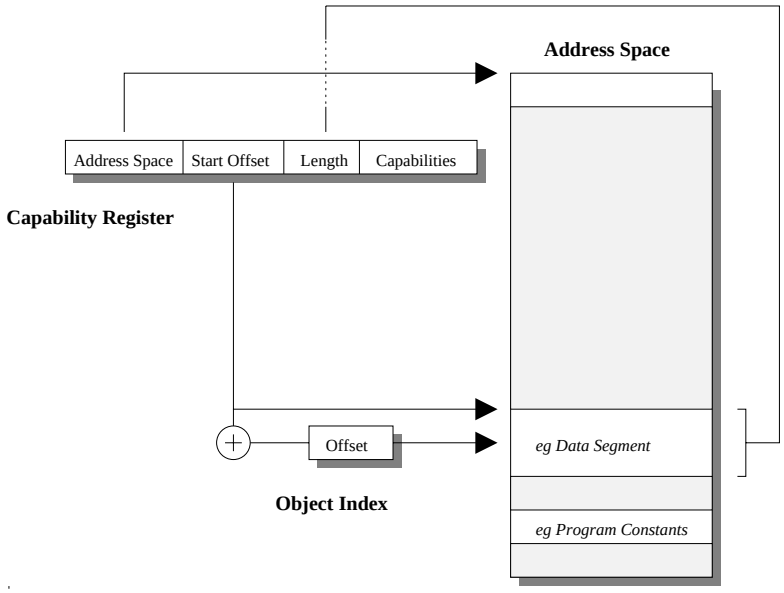


Figure 3.9: Object Mapping in MONADS

By accessing a segment, and loading a capability from that segment, all accessibly data structures can be traversed. Loading of the capabilities must be protected from malicious users, thus a special machine instruction is provided for this function. Indeed, another instruction performs the modification of the segment capabilities. The root of all addressing is found within a special segment (one per process), called the *base table*. An additional machine instruction is included to allow access to the contents of this table. Note that such instructions interfere with instruction set orthogonality and thus add to compiler complexity.

Segment capabilities are constructed such that all segment addresses are expected to reside within the current address space [Keedy, 1984]. This has the effect of saving memory space (and saves on the effort required during garbage collection), at the expense of limiting the scope of data structures. It is therefore advisable to group related objects (*e.g.* segments) together within a single address space. Segments outside of the current address space can however be referenced by using indirect capabilities, which themselves point to full virtual addresses.

3.3.3 Module Management

There are two distinct types of objects recognized by MONADS; segments and modules. Modules present a procedural interface to other external modules. Using modules, it is possible to implement such constructs as traditional `files', where the module manager can present an interface which mimics traditional file access. In fact, since MONADS describes itself as persistent, no files actually exist.

However, MONADS deviates from more traditional persistence in that capabilities referring to a segment may not be passed to other modules and freely retained.

The reason for non-persisting capabilities external to the owner module lies with a fundamental limitation with the MONADS environment. In the attempt to remove the need for a central mapping table [Keedy, 1984], compounded by the desire to allow segment slices to be passed between modules (in a controlled way), the problem of virtual memory limitations becomes a factor.

With the removal of the central mapping table, virtual memory can not be easily reused. This is especially true if capability revocation is attempted (as is possible with when capabilities are copied by modules external to the segment owner). To counteract this problem, capabilities are only permitted to be stored on the calling stack of external modules (thus are automatically deleted on a return from that module). Thus revocation of segment capabilities does not occur, and virtual memory can be safely reused.

Due to the grouping of segments within modules, the individual segments are no longer considered to be separate objects with a type associated to each, but instead as a collection of similar objects which are accessed via the procedural interface of the module. This style of data modelling is typical of architectures which use a set of capability registers to hold the roots of a relatively small number of large entities.

3.3.4 MONADS Paging

Each of the allocated memory address spaces must be wholly stored within a single volume (although a volume will typically carry more than one address space). For efficient paging, a disk page table is needed for each address space, and this is held in a protected region of the address space which it describes. A single root page exists for each volume, and points to the volume directory, which in turn refers to each of the address space paging tables.

Every MONADS node contains an Address Translation Unit (ATU), which maps the virtual addresses onto physical memory locations. The ATU is implemented as an inverted page table (as in the case of the AS/400), constructed from hardware, such that its size is proportional to the size of the physical memory map. If the page required does not exist in physical memory, a page fault is then generated. At this point two separate routes can be followed; either the faulted page is within the current node (in which case the page can be loaded off disk), or the page is stored within some other node (resulting the local node asking the networked node for the page).

On a locally resolvable address fault, the disk address of the desired page can be read from its primary page table. This table is indexed with bits 12-27 of the virtual address offset value, and thus contains 2^{16} entries of 16 bit disk addresses. It may be that this primary page table does not exist either, and therefore also generates an address fault. This results in the secondary page table for that address space being accessed, which contains 32 disk address entries (in the MONADS-PC system). Thus each entry corresponds to a single page of the primary page table ($(2^{16} \times 2)/4\text{K bytes} = 32$). The required section of the primary page table can now be loaded from disk. It is also possible that the secondary page table for the address space is not present either. The disk address of this page can be found by

accessing the root page table, which is always found at address space zero of each volume. Part of this address space paging structure can be seen in figure 3.10.

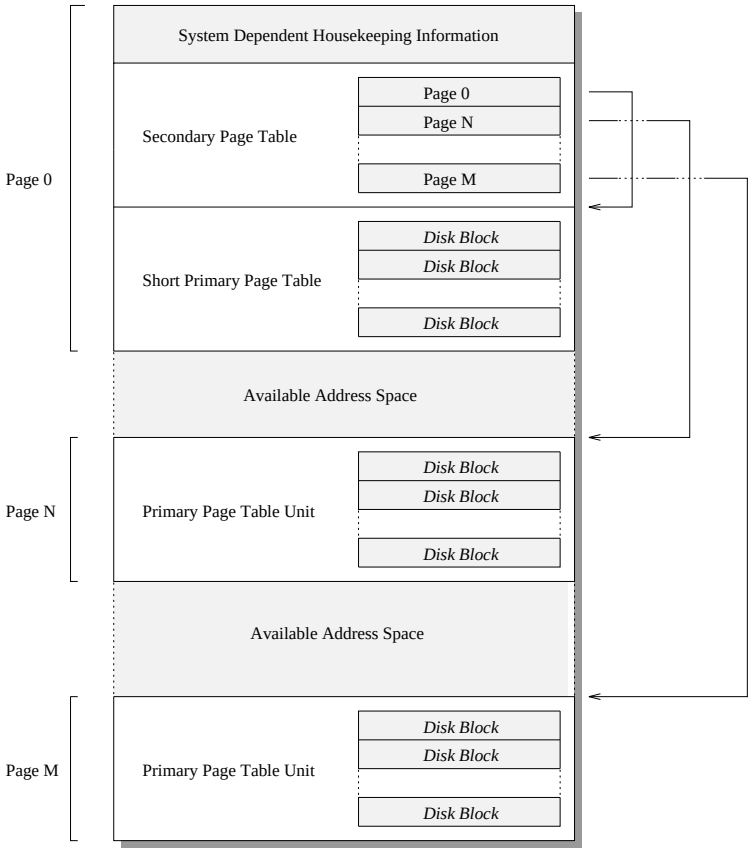


Figure 3.10: Layout of a MONADS Address Segment

The layout (at least with respect to the MONADS-PC version) of this disk address tree is such that, for address spaces smaller than 256 memory pages (with each page containing 4K bytes), the entire tree can be found within the first page of the address space. Indeed, for address spaces smaller than about 3.6K bytes the address space, fixed overhead, and primary and secondary page tables can all be found within a single page (thus resolving the page fault in a single disk access) [Rosenberg *et al.*, 1990].

3.3.5 Conclusions

The module/object hierarchy of MONADS promotes low-overhead object management, both in terms of locking and networking [Henskens *et al.*, 1990]. However, this hierarchical design makes implementing orthogonality in object accesses difficult to achieve; given any OID, it is impossible to access that object unless the access is performed from within the surrounding module. While module-based objects efficiently support block-structured languages such as PASCAL and ADA, future (and even current) object-oriented languages may have significantly poorer performance. The module interfacing

and process control adds extra complexity to the system, and this complexity is made all too clear to the programmer [Rosenberg, 1989].

There is no garbage collection system for objects, and when all objects within a module are no longer needed the module is explicitly deleted by the user. Even if garbage collection was introduced, objects cannot be deleted in isolation from their surrounding module. It may be possible for each module to contain routines which allocate and deallocate objects within the module's scope, with garbage collection within each module. Collection would have to query every object within the transient closure of the root objects. For this, collection routines would be needed within each module in the heap.

3.4 MUTABOR

The Profemo project at Gesellschaft für Mathematik und Datenverarbeitung mbH[4] (GMD), while conducting research into the design of reliable distributed architectures, created the MUTABOR design. MUTABOR [Kaiser, 1990, Kaiser and Czaja, 1990] (Mapping Unit for The Access By Object Reference) formed an object-oriented architecture supporting a secure and reliable computing base. Custom hardware was used to support both object and transaction management [Kaiser, 1988], coupled to a layered, generalized transaction-based operating system.

MUTABOR consists of a microinstruction-based coprocessor (which currently connects to the coprocessor interface of a MC68020 [Liu, 1991], extending the processor's instruction set to include MUTABOR primitives) and an address translation cache (ATC). This ATC holds physical addresses of recently accessed objects, along with other information which is used in additional logic to authenticate access rights and size constraints.

In contrast to MONADS, MUTABOR implements a more fine-grained approach to object instances. Take for example a database holding student information (*e.g.* student name, address, exam marks, *etc.*) for two hundred students. Within MONADS, the entire two hundred entries would be stored inside a single module, supporting module interfaces such as *get student record* or *assign exam result to student*. In comparison, MUTABOR would implement each student record as a separate object, and each object would have associated type information evaluated at invocation time. It is assumed that the mean object size in MUTABOR is about 300 bytes [Kaiser and Czaja, 1990], which compares favourably with other fine-grained approaches (such as 300 for iMAX [Pollack *et al.*, 1981] and 326 for Hydra [Wulf *et al.*, 1981]).

3.4.1 Object Store

The MUTABOR system implements object persistence within a long term repository known as the *passive space*. There also exists a second area, called the *active space*, which constitutes a virtual address space (mapped by the ATC) within which computations are performed[5]. If a persistent object which does not exist within the active space is referenced, then it is transferred from the passive space

[4] The German National Research Centre for Data Processing

[5] Note that *passive* and *active* are terms which are heavily overused, and as such tend not to be comparable across different systems.

by some internal manager. Thus the active store simply functions as a virtual cache for the passive store. Since the transfer mechanism is automatic and transparent, the single level object store paradigm is maintained.

Additionally, within a fine-grained system, many transient objects (such as temporary local variables) are created and then quickly abandoned. Experiments from StarOS and Hydra show that this class of object forms about 95% of all objects instantiations [Almes, 1980][page 107]. It would be a severe system bottleneck if each of these objects required a PID (*e.g.* UID or very long Unique and persistent IDentifiers as they are known within MUTABOR). To reduce this overhead, each PID located within the active space is identified by an *object short name* (OSN). These OSNs are 24 bits long (in comparison to the 48 bit PIDs), thus allowing sixteen million objects to reside simultaneously within the active space. Note that since the OSNs are smaller than 32 bits, they can be fetched and stored within a single memory cycle.

The MUTABOR system does not directly interpret the object UIDs. If an object is not present within the active space, then the *object filter* manager is informed. It is the manager's duty to perform passive UID to active OSN transfer. A second manager, the *global object manager*, is used to resolve object requests which require object transfers between MUTABOR-based nodes.

3.4.2 Memory Layout

In MUTABOR, every process is given an unique 16 bit process number. This number is used to select a context specification for each system context (thus providing every process with a different view of the object store). Each context specification consists of four entries:

- The current context. This contains the capabilities for the current context, including temporary variables.
- Current root object, which points to the user defined object on which the current context (called a TSO or *type specific operation*) is operating on.
- The current object type, which allows access to all other TSOs for the current root object.
- The parameter object which gives access to the invocation parameter. This allows data and capabilities to be passed, thus simulating either call by value or by name.

A code invocation consists of the following form:

INVOKE (target root object, TSO-index, parameter)

with the target root object specified by a capability index (*i.e.* the process local name for an object) within the current context, the current root object, the current type object, or the current parameter object. This requires that a new context specification be created for the process (including the context's own capability list), and the seven-stage mechanism involved in creating the context can be found within [Kaiser and Czaja, 1990].

In accessing an object via its capability index, entry zero of the context specification is indexed with the high order bits of the index, which selects a particular capability list. This list in turn is indexed by the remaining low order bits, which selects the capability for the object in question. Thus, each process has its own name for every capability, and this is known as the *local process object name* [Kaiser, 1990]. This mapping mechanism can be seen in figure 3.11 [Kaiser and Czaja, 1990]. The capability produced in the figure is now used to access the required object.

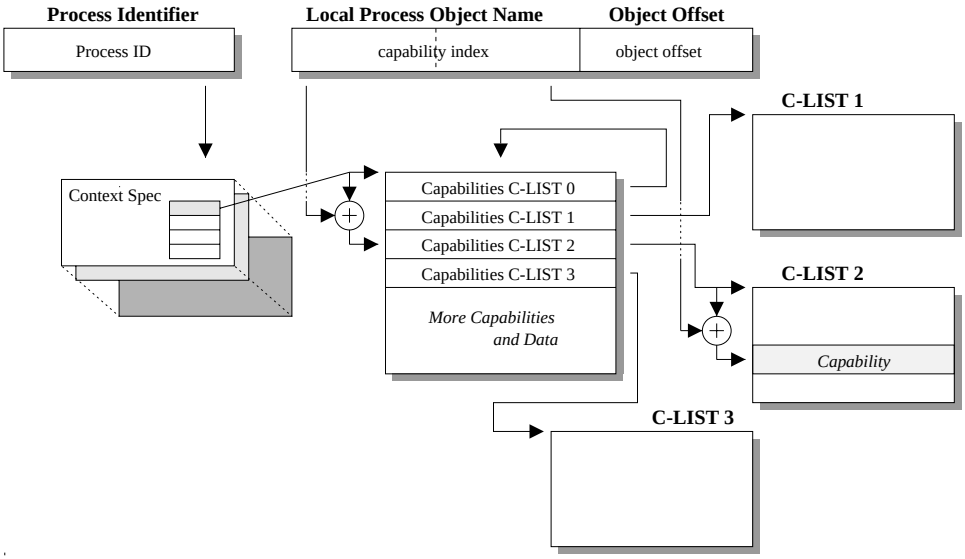


Figure 3.11: MUTABOR Context to Capability Path

The object OSN can be extracted from the selected capability: The high twenty bits of the OSN goes through a two level table to identify the page within which the object resides. Given the average object size of 300 bytes, the designers limited the mapping process to 16 objects per 4K page. Thus the last four bits of the OSN is used to select the required object header within each page. Additionally, this header contains information to check type and access rights dynamically at access time, thus allowing such facilities as object locking and shared access. This translation path is depicted in figure 3.12. Naturally, this method of object reference alleviates the need for a central mapping table (but creates a large data management task).

Also from figure 3.12, the selected object is actually split into two parts: the data part and the non-data (or capability) part. Within the capability part lies the UID for that object, a link to the TSOs for that object, some system-required red tape, and any relevant capabilities required in representing complex data structures.

The object segregation is intended to protect objects from inadvertent or malicious accesses. Such segregation can be seen in a number of other architectures, such as MONADS, and is often known as *fenced-segmentation*. For MUTABOR, the capability part of an object can only be accessed by negative address offsets, which requires special coprocessor instructions to be used. Clearly, since MUTABOR and MONADS share a similar object layout scheme, the MUTABOR also exhibits the same limitations as MONADS with respect to type inheritance.

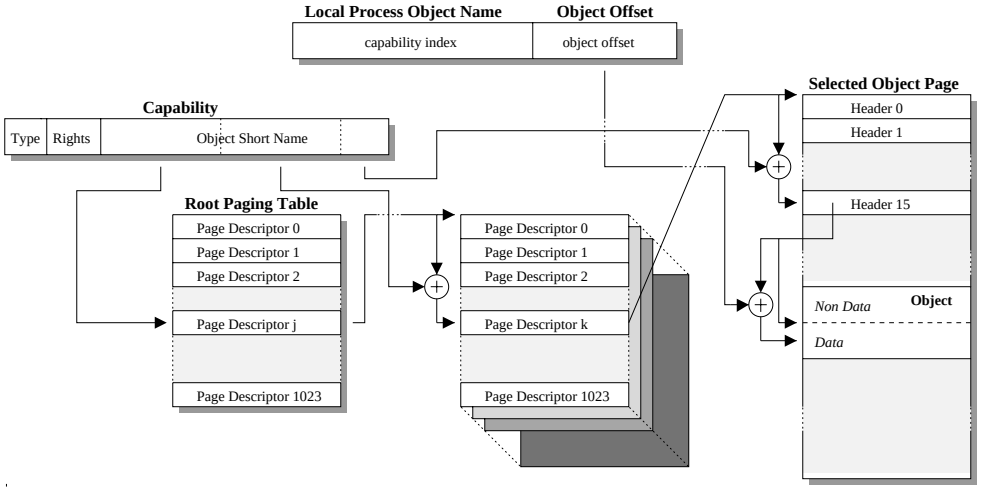


Figure 3.12: MUTABOR Capability to Address Path

The need to maintain and traverse multilevel tables is an intrinsic overhead found in all similar types of object management implementations. It is for this reason that the ATC was introduced, containing the 4096 most recent address translations. The ATC is subdivided into 16 sections by 256 entries, with the 16 last most recently executed processes allocated one section each. This gives an estimated hit-rate of greater than 95%.

3.4.3 Conclusion

Fine-grain objects mean that object-based locking is sufficient for transaction, locking, and network management. However, large objects are not well supported, since individual segments of an object cannot be locked in isolation of the whole.

The use of short object descriptors for data contained within the active space is unusual, but is a useful step in reducing the memory bandwidth and silicon overheads in handling long object descriptors. However, MUTABOR is an exceedingly complex machine (perhaps needlessly so), both in hardware design and software interface.

3.5 Rekursiv

The Rekursiv chip-set (described in [Harland, 1988]), developed by Linn (a Glasgow phonography company), and fabricated by LSI Logic, was designed for use in the construction of object-oriented architectures. The original aim for the Rekursiv was to create a programming environment to allow support of production lines, where a processing unit would be placed in close proximity to each major component of the line. From this, the Rekursiv has been used in a variety of small-scale commercial and academic ventures.

The chip-set was made from three devices; the sequencer (LOGIK), arithmetic unit (NUMERIK), and the memory manager (OBJECT). LOGIK was microcode based, and its microcode store could be

written to by the operating system, allowing instruction-based support to be implemented for whatever application (or language) was needed. An overview of the system is shown in figure 3.13.

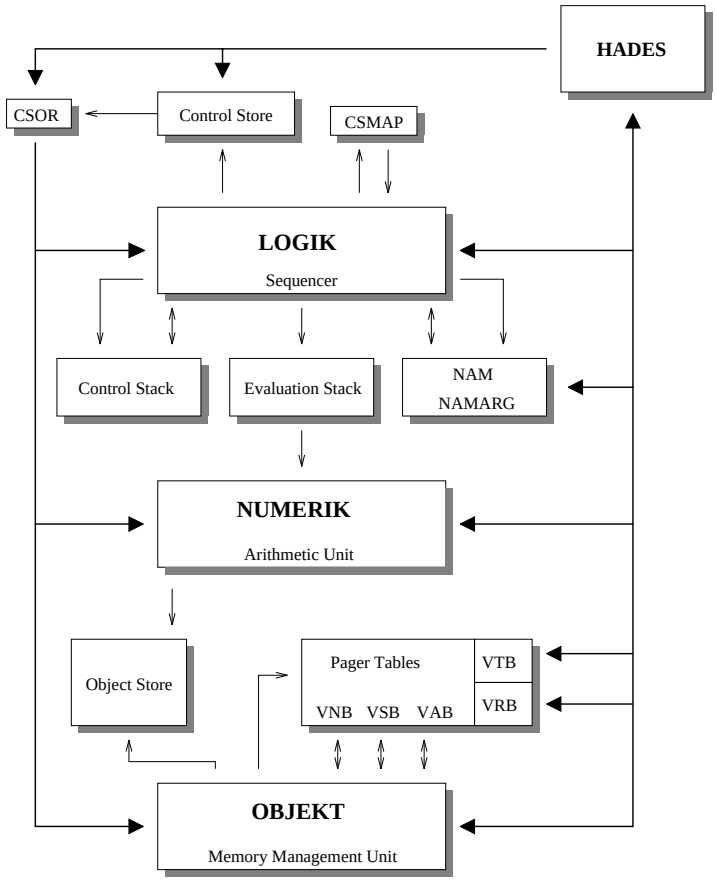


Figure 3.13: Overview of the Rekursiv

Associated with these processing units are seven distinct banks of special purpose RAM.

1. The CSMAP: a look-up table that maps opcodes to the addresses of the micro-routines that implement them.
2. The Control Store, which holds the horizontal micro-instructions.
3. The Control Stack, which holds linkage information both for micro-routines and high-level language routines.
4. The Evaluation Stack: a classic push-down store for the implementation of postfix instruction-sets and for the store of procedure-local variables.
5. The NAM or macro-instruction store.
6. The Page Tables, which map objects to physical addresses.
7. The Heap or Object Store into which currently active objects are loaded.

All but the last are implemented in high speed static memory. The motivation for providing all these distinct memories is speed. The ability to perform several memory accesses per clock cycle allows micro-code to be fast and compact. The complexity is somewhat less than it seems, since many micro-coded architectures provide the equivalents of a CSMAP, Control Store, Control Stack and Page Tables (TLBs[6]), but hide them from view. Considering only what the assembly programmer sees, however, the Rekursiv still has an unusually complex store structure. Whereas on a von Neumann architecture, the evaluation stack, the heap and the macro-instructions would all share a common memory, the Rekursiv treats them as distinct. This has the advantage of allowing simultaneous pushing of a heap operand with instruction fetching, but it is questionable whether the complexities that this introduces for the operating system are worthwhile.

3.5.1 Memory Management

The Rekursiv is a 40-bit word machine, and its object identifiers are also 40 bits wide. Objects are sub-divided into two classes; compact and extended. Compact objects are those objects which are small enough to fit entirely within a single 40 bit word, and thus the entire object can be held within an object identifier. Extended objects tend to be larger than compact objects, and are stored at the particular memory to which the extended object identifier refers.

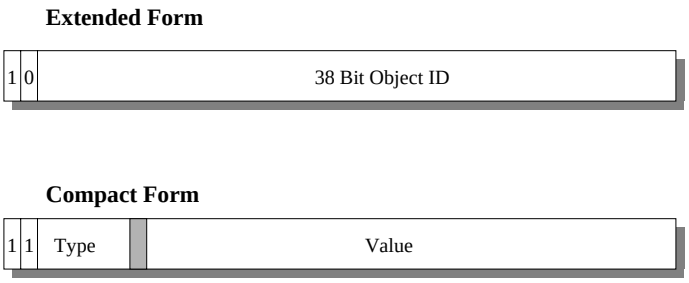


Figure 3.14: Two forms of Rekursiv Object Identifiers

The format of these two object identifiers is shown in figure 3.14. Although their physical layout is distinct, the hardware contrives to give all objects the same abstract form. All objects appear as the sequence of fields shown in figure 3.15. They all appear as a vector of 40 bit words, the first three of which describe the object, with subsequent words holding the concrete representation. Figure 3.15 also shows five of the Rekursiv's CPU registers, and these correspond to:

- *VAR*: This corresponds to the *Value Address Register*, and contains the address of the first memory-bound element of the object.
- *VRR*: The *Value Representation Buffer* holds the word pointed to by *VAR* (*i.e.* the first element of the object).
- *VTR*: The type identifier of an object is held within this register (the *Value Type Register*).

[6] Translation Look-aside Buffer

- *VSR*: The *Value Size Register* holds the size in words of the object pointed to by *VAR*.
- *VNR*: This register (*the Value Number Register*) holds all 40 bits of the object identifier.

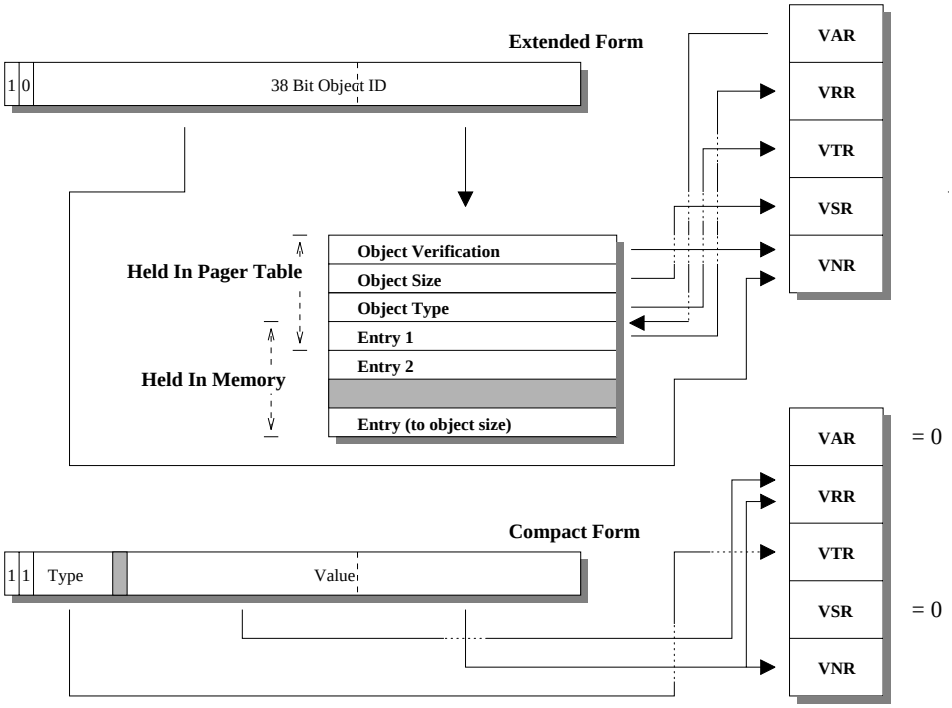


Figure 3.15: Rekursiv Object Register Fields

When an object is dereferenced, hardware in the memory management system loads this set of registers with the first four fields of the object $\{VNR, VRR, VTR, VSR\}$ and its address $\{VAR\}$. In the case of compact objects both its size and address is always zero. The object's type is extracted from bits 33 to 37 of the object identifier, and *VRR* is taken from bits 0 to 31. In the case of extended objects, much of the required information is extracted from the page tables (figure 3.16) by the following algorithm:

1. The lower 16 bits of the OIDs are used to index the page table.
2. The page tables return five fields which are loaded into the registers $\{VNR, VRR, VTR, VSR, VAR\}$.
3. The *VNR* register is compared with the OIDs and an object fault interrupt generated if they differ. This occurs whenever the hashing function selects a pager entry which corresponds to a different object. If an interrupt is generated, the current entry in the pager table is unloaded, and the correct entry is loaded in its place.
4. In parallel with step 3, the base address of the object, in the *VAR* register is added to an index register and sent to the dynamic RAM.

5. In parallel with step 4, the index register is compared with the size register (*VSR*) and a fault signalled if addressing is out of bounds.

6. Two cycles later the DRAM bank that holds the heap delivers the addressed word of the object.

The steps 1…5 can be completed in one clock cycle.

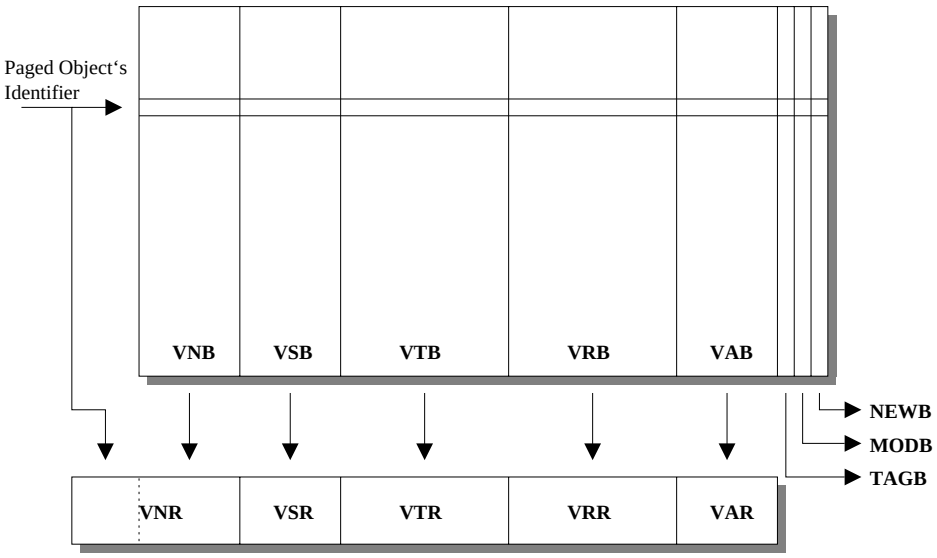


Figure 3.16: Rekursiv Pager Table Layout

This design is unusual in that it stores the type and first value field of an object in the page tables. Since these are implemented in high speed static RAM, it enables type information and the most frequently used value field to be available before the DRAM cycle has finished. Since the processor is designed to execute Smalltalk like languages which dispatch methods depending upon the object type, this could lead to considerable speed gains.

One consequence of the method used to index the page tables, is that no two objects with the same lower 16 bits of their OID can co-reside. System routines needed access to certain objects regularly. Any object which resided in the same line of the page table as a system object would quickly become persistent, and would then thrash against the system object.

3.5.2 Object Allocation

Special auxiliary hardware is provided in the memory management unit to handle the allocation of objects. A pair of hardware registers hold the top free address on the heap and the last object number allocated. Data-paths are provided to update the page tables and appropriately increment these registers in a single operation when creating a new object.

If this results in heap overflow, a garbage collection fault is signalled and a micro-coded garbage collection routine entered. Tag bits in the page tables are used for the mark phase of the garbage collection. After marking, the heap is then compacted. Since all objects are addressed indirectly through the page tables, only these have to be altered during compaction.

3.5.3 Network Addressing

An object identifier contains 38 bits available for object addressing once the tag bits have been taken into account. The preferred use of these, according to Linn, is for six bits to be used to identify on which machine on a local net the object resides. The remaining 32 bits are a re-usable object number. When object numbers are exhausted, a global garbage collection is performed and objects are reallocated object numbers lying in a dense subset of the 32-bit range.

This poses problems. The reuse of object numbers is feasible on a single machine, but on a network it becomes impracticable. Suppose that machine *A* contains an object referred to by an object on machine *B*. If the object on machine *A* is deleted and its object number reused, then accesses from machine *B* will return the wrong object.

The Rekursiv provides a large object address space compared to the SYSTEM/38, 2^{38} objects for an isolated machine, 2^{32} objects for a machine on a network as against the 2^{24} objects for the IBM machine. This means that the problem of object number reuse is less severe. But against this must be weighed the sorts of languages that are run on the two machines. Languages that require garbage collection, such as those supported on Rekursiv are much more profligate in their use of object numbers. It is the opinion of the authors that the 40-bit object numbers of the Rekursiv would prove a restriction were the machine to go into general use.

The Rekursiv's virtual memory size is physically too small for many modern applications, and the heavily restricted network-address size made networking viable in only the smallest of networks. The ability to write microcode for the processor, and the object management functions which were also microcoded, did produce a performance improvement with respect to certain system functions. However, the added complexity of supporting this was detrimental to the performance of many other processor activities [Cockshott, 1992].

3.6 Architecture Overview

Over the previous pages descriptions some of the main object oriented hardware architectures have been presented, and a number of points have been raised concerning their suitability in supporting the ideal object oriented architecture. The creators of these systems held a number of differing beliefs, and therefore it is of no great surprise to find that features considered essential within one design were ignored in another.

3.6.1 Summary

In order to summarize the systems, eight points were considered (in no particular order):

- The *Network Size* which was reachable by the system in question was classified into *LAN* (Local Area Networks), *WAN* (Wide Area Networks), and *N/A* (*i.e.* no networking).
- Whether or not the system was a *Tagged Architecture* (*i.e.*supported self-identifying data structures).

- *Paged Memory* availability is often considered to be essential in supporting access to very large object sizes.
- The *Virtual Memory Size* is also important, as it gives an estimate as to how long the system can last without reusing object identifiers.
- *Segmented Memory*, *i.e.* the support of variable sized, hardware assisted memory segmentation, is frequently used in providing object protection and management mechanisms.
- The availability of general purpose *Garbage Collection* external to the compiled language is attractive in making object oriented architectures more secure than systems using other designs.
- The *Commercial Scale* of an implementation is of interest, as it shows the amount of industrial interest and support currently vested with the particular system (as well as its availability).
- All *Available Languages* for each system have also been shown, and this helps to show the flexibility of each architecture examined.

A table showing all the systems covered (including separate entries for the two MONADS designs) against the points raised above can be found in table 3.1.

3.6.2 Conclusions

Even though it is one of the oldest of all the persistent architectures, the SYSTEM/38 shows features that subsequent efforts to produce persistent systems would do well to learn from: a large address space, a well thought out support for operating systems and an efficient paging mechanism. It has been a considerable commercial success with tens of thousands of sites running the machines. Since it comes packaged with its own operating system and relational database which provide a high-level user interface, most users remain unaware of the novelty of its underlying architecture.

The iAPX-432 was a landmark machine in many respects. As a profit-making exercise it should be considered a failure, but from a research point of view it was an excellent project. RISC designers frequently use the 432 as a target for pot-shots, but its futuristic memory strategy was only really limited by available silicon. Its addressing scheme could, however, be simplified. For example, bitwise addressing is useful only to a very select few, and it therefore unnecessary unless there is going to be no performance degradation in implementing it (which is not true in this case).

The more modern MONADS-PC and MM architectures represent an interesting divergence from the traditional view of persistent object oriented systems, by using a block structure approach to the design. This does support the current block-like languages of today, but may limit the development of future (and perhaps even current) object oriented languages. Its module grouping structure, while simplifying many elements of program construction, reduces the overall flexibility of the system.

Additionally, extra complexity is added to the system by way of module interfacing and process control, and this complexity is made only too clear to the programmer [Rosenberg, 1989]. However, the object mapping mechanisms used in MONADS represent many of the features which have been demonstrated by current research as highly desirable.

Table 3.1: Summary of Object Oriented Architectures

	Machine Architectures					
	iAPX 432	MONADS-PC	MONADS-MM	MUTABOR	REKURSIV	SYSTEM/38
First Reference	1975	1976	1990	1988	1984	1978
Network Size	N/A	LAN	WAN	N/A	LAN	N/A
Tagged Architecture	No	No	No	No	Yes	Yes
Paged Memory	No	Yes	Yes	Yes	No	Yes
VM Size (bytes)	2^{40}	2^{60}	2^{128}	2^{32}	2^{38}	2^{48}
Segmented Memory	Yes	Yes	Yes	Yes	Yes	Yes
Garbage Collection	Yes	No	No	No	Local Only	No
Commercial Scale	Small Scale	Experimental	Experimental	Experimental	Small Scale	Large Scale
Available Languages	Ada Chill	Pascal LEIBNIZ	Pascal LEIBNIZ	*Library Access*	C Lingo PS-Algol Prolog Smalltalk	RPG

MUTABOR, the research project from GMD, demonstrates the effectiveness of large translation buffers within object oriented systems. Its fine-grained approach to object management is also attractive, in comparison to the more coarse-grained SYSTEM/38 and MONADS designs. In addition, the project also highlights the need for short object identifiers for active space objects, which is desirable in any limited data bus system (especially in systems build around non-custom parts, where the overhead of long object identifiers can be severe). On the negative side, MUTABOR is clearly a complicated architecture (perhaps needlessly so). The amount of table traversal required to locate an object which is not present within the object buffer can be high, and therefore the probability of memory faulting during the traversal is also high. Such a design may not be desirable for real-time architectures.

Finally, the Rekursiv was examined. Its design was such that many of the common object management functions were implemented in microcode, and although this had the effect of improving certain system functions, many other parts of the design were adversely effected by the processor complexity (partially created by the three-chip implementation). The limited virtual memory space was also a problem, especially when used on a network. The poor performance of the design, the difficulties in porting languages so as to best use the writable microcode store, and the networking and virtual memory limitations all assisted in preventing the Rekursiv from becoming desirable to the public.

3.7 Directions

Throughout this chapter a number of different object-based architectures have been examined. These should in no way be considered an exhaustive survey, but the methodologies described above do cover the majority of available systems. In the remainder of this document, a new approach to object-oriented system design is proposed, under the broad heading of the DOLPHIN environment. From the survey given in this chapter, it is important to make various recommendations on the design of DOLPHIN so that past mistakes are not repeated in this design.

The main objective must be to create a system which has an overall performance on par with non object-oriented architectures implemented in the same technology. No one is going to buy a design which runs their applications slower than running them on their current systems. This undoubtably means that the translation of OIDs to physical addresses, which is the main bottleneck in any object system, much be performed quickly. There are basically two solutions to this; convert OIDs to physical addresses when the object data is loaded (bit swizzling) or use hardware to perform the translation on every reference. Bit swizzling, although allowing the use of standard off-the-shelf processors, is rejected due to the complexity in maintaining consistency between the swizzled object data and the version of the data containing real OIDs (for use in the long term store, on the network, *etc.*), and also in maintaining object protection. Without the level of indirection provided by hardware-based translation, any type or object protection in a system using swizzling must either be based on virtual memory management, which is of memory-page granularity, or software-based (which was rejected earlier in this chapter).

In a system using OIDs at the instruction level, translation must be fast. This could most likely be achieved by a combination of pipelining, segmented memory, and intelligent caching. Since RISC

architectures are amongst the fastest processors around today, it is recommended that a RISC-like instruction set be used. Tagging of memory data to show data types (*e.g.* integer, float, characters, *etc.*) is generally unnecessary, since a compiler can be written to identify the type of a data element by other means (*e.g.* a lookup table), and maintain this knowledge in a user register on-chip. This allows for compiler optimization, whereas a tag-based strategy requires that the type be interpreted on every reference to that data element. Tagging of OIDs may be useful in protecting OIDs from malicious or accidental modifications, but protection could be offered through sparse OID allocation (such as in the iAPX-432's checksum field). In either case, a large virtual address space would be useful.

Since the DOLPHIN environment is to offer a testbed for further research into object systems, it is important to provide as flexible a system as possible. The design should support Wide Area Networks (WAN), with a network addressing scheme larger than the already overused internet strategy. Since the system language is not yet known, no decisions should be made to alienate the design from any particular language. It is also important to realize that many programs which have been developed recently have been written in C, and thus the system should allow C programs to execute with little or no modifications.

In the area of networked garbage collection, there is still much research to be performed. As such, the DOLPHIN system should provide the necessary interface to implement a variety of garbage collectors. Local garbage collection, while still a research topic, should be provided by default, although tabs should be provided to replace this collector to both match current collector technology and collection requirements introduced by dependencies created by the network-based collector.

It is interesting to note that all the systems examined above still use rotating media to maintain persistent information. Any data held in main memory must be transferred to the backing store before it can be considered persistent. This can be a severe bottleneck on performance, especially in database applications. It also complicates heap management, requiring tricks such as shadow paging to maintain heap/backing store consistency over power failures. In DOLPHIN the upcoming technology of Flash Memory is utilized. Although Flash is described as the sole persistent store for DOLPHIN it could in reality be replaced by the more traditional rotating media if required (naturally in the process sacrificing the huge performance gain realized using Flash in exchange for a cost saving).

In part II, the DOLPHIN environment is described, including the design philosophy behind its approach. It also contains suggestions on a number of variations which can be made in the basic design, which can be adopted depending on circumstances (*e.g.* there are two versions of the processor proposed, dependent on available silicon technology).

Part II

The DOLPHIN Environment

Chapter 4

The DOLPHIN Environment

Program reuse and data abstraction are key features of modern software engineering. The desire for compiler support of these features has lead developers to investigate object-oriented languages. These languages use objects at the data storage level, where objects encase both program data and a functional interface to that data. This information abstraction helps to promote object reuse and software maintenance.

In an object-based language, the location of data in memory is no longer of concern to the programmer. In fact, objects can be moved about in both short- and long-term memory transparently from the application. If the system automatically moves objects to long-term storage when an application exits, it is said to support a *persistence object-oriented heap*. Objects saved in this way can then be restored when the application restarts, removing the burden of long-term data management from the programmer.

One problem with pure object-oriented languages running on standard architectures is performance; many object languages are interpreted rather than compiled (*e.g.* Smalltalk), and thus run between one and two orders of magnitude less quickly than optimized C. Although advances have been made in compiler design for object-oriented languages, true object-based addressing in hardware should offer both safer and more flexible implementations. Several hardware designs have been developed, and these were investigated in chapter 3, with a number of failings identified. These included a lack of network support, poor overall performance, too small a virtual address space, and insufficient thought concerning the support of object modifications during program execution.

From these failings, this document examines the support required to implement a networked object-oriented environment, such that its overall performance could be similar to that of a virtually-addressed system. We term this the DOLPHIN environment. The task is broken down into three sections: processor-level support, local object management, and network management.

4.1 DOLPHIN

DOLPHIN is proposed not as a complete object-oriented computer, but as a vehicle for further research into object-oriented systems. As such, it does not provide high-level entities such as editors or programming shells. It does provide ties for a high-level operating system, by providing routines for object creation, deletion, garbage collection, lightweight threads, networking, and memory management. This should give researchers a base to implement high-level ideas, without having to delve into the low-level code required to support them.

This chapter gives an overview of the approach used in the design of DOLPHIN. As you will see, it is based on a layered approach, which simplifies the design somewhat, while also allowing for experimentation in the implementations of various levels. In addition, the layered design gives some degree of hardware independence in the higher levels of a full object-oriented computer.

4.1.1 Design Partitioning

The DOLPHIN environment is constructed with a hierarchical design, where a full computer system utilizing DOLPHIN would consist of four (or more) separate levels. Figure 4.1 shows this hierarchy. The levels are:

1. DAIS: This is the object-oriented processor which lies at the heart of the DOLPHIN design. Its user instructions all operate on object addressing, as opposed to virtual addressing schemes typical of today's high-performance processors.

2. SubKernel: This is similar to the traditional idea of a microkernel, and contains the basic routines common to most object-oriented operating systems. It helps to isolate researchers from the low-level systems required in implementing object-based computer systems.

3. Operating System: This uses the SubKernel to implement things useful to the users, such as editors, directories, user accounts, and programming shells.

4. Applications: This layer contains the applications executed by users. Possible applications could be word processors, spreadsheets, *etc.*.

This document only considers the bottom two levels of DOLPHIN, defining the DAIS processor and the SubKernel. These should provide a powerful platform on which to investigate, and experiment with, future object-oriented design strategies.

4.2 Overview

An object-oriented processor, DAIS, is proposed in chapter 5. This utilizes object-based addressing at the instruction level. It also contains a number of other features: multiple high-performance hardware stacks in place of flat registers, processor instruction and data caching based on objects and offsets, and a pipeline free of data-dependency stalling.

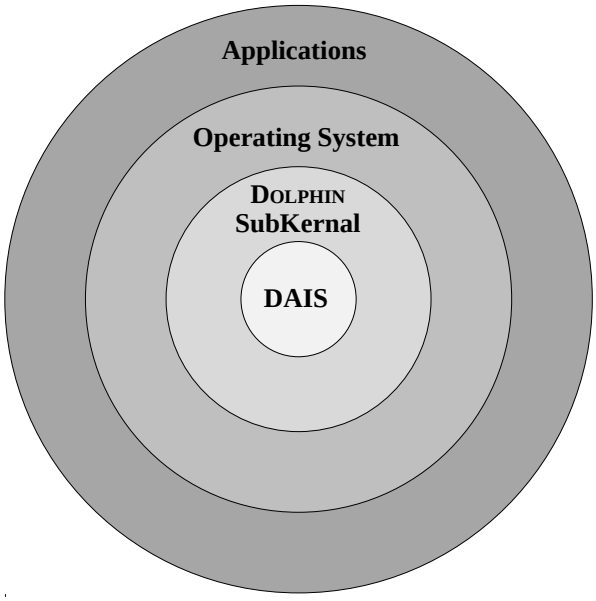


Figure 4.1: The DOLPHIN Environment

Surrounding the processor is a SubKernel, supporting object persistence and multitasking. Local object management provides for object creation, garbage collection, and object memory management. This SubKernel also contains a network management system for objects, which promotes low network-bandwidth requirements. The networking scheme uses a custom transmission protocol which supports multicasting and point-to-point transmission of reliable virtual-sized packets. These are packets which can be of any size (up to 4 GBytes), independent of the underlying network-protocol limitations.

The SubKernel also contains memory management routines, designed for flexibility of use. These provided two memory areas; one swapped automatically between memory and the backing store, and the other held only in main memory. Allocation of the swap area could be for read-only pages (with respect to user threads) or read-write, with primitives to convert between the two representations (*e.g.* to allow separate threads to have different object access rights). The backing store is proposed to be constructed from solid-state memory components, and a design is presented using PCMCIA-based Flash cards.

By combining together all of the algorithms and hardware outlines described within this document, a design proposal is produced whose performance can compete with the best of current object-based systems, while providing the flexibility required for further research into object paradigms.

4.2.1 DAIS

The initial proposal for DAIS, the object-based processor, is described based on a system using 64-bit OIDs (plus one tag bit); DAIS-64. These OIDs can then be aliased to larger identifiers for worldwide object references. This approach allows the use of local identifiers which can be much smaller than those used to identify objects on the network, which in turn reduces memory bandwidth requirements

for loading descriptors. This suggests a reduced memory storage requirement. Additionally, world-wide and local object identification methods are separated, allowing future modifications to descriptor layouts.

Also presented is DAIS-256, which operates on 256-bit object identifiers (no tag). This approach does away with the complexity of aliasing worldwide object identifiers to local equivalents, and is considered as the processor used in the remaining discussions on DOLPHIN. However, a system which makes use of this size of OID may suffer performance degradation in comparison to one based on smaller OIDs. It is proposed as a possible implementation to:

- Give an example of DAIS without tagging.
- Remove the complexity of the conversion between networked identifiers and OIDs from the other discussions contained within this thesis.
- Simplify the software simulation of the DOLPHIN simulator used in chapter 12.

The caching structure of the DAIS architecture incorporates features allowing the architecture to run at around the same speed as non object-based processors. DAIS achieves efficiency by providing only the minimum of support for objects. In RISC architectures, on which DAIS' memory interface is based, only load and store instructions need be concerned with objects, which greatly simplifies processor design.

A cache structure based on objects and offsets rather than on virtual addresses allows object data to be accessed without the need for an address translation. Such a scheme also allows bounds checking to be achieved simply by examination of validity bits, removing the need for arithmetic comparison. A bit in the status word of an object allows for locking at the object level, which is useful for small objects. For large objects, page-based locking is more desirable, and is provided through the virtual memory manager. DAIS combines both these types and holds the read/write information in the data cache, providing an efficient mechanism for deciding on the validity of writes.

Analysis is undertaken on the effects of choosing different cache sizes. This is done by dynamically tracing binaries for four different benchmarks in a non object-oriented architecture. Such an analysis is sufficient to provide performance evaluations for instruction and data caches, but not for the secondary object cache. Therefore, the sole conservative assumption is made that a 128 entry object cache will have a 90% hit rate. However we believe that the real figure may even be higher that this. The results of the analysis shows that a non-superscaler version of DAIS-64 with external cache takes 1.34 cycles to execute each instruction. This result is for a system with 8K of instruction cache, 4K of data cache and 2K of object cache. The object cache justifies its existence since a system with 8K of both instruction and data cache but no object cache has poorer performance[1].

The DAIS-256 processor is taken to be the processor used for subsequent discussions of the DOLPHIN environment. However, a smaller OID-sized processor, combined with aliasing of the OIDs to

[1] Removing the object cache from the design frees up 2K of cache, but in the comparison 4K of cache is added to the data cache. This is due to the cost (in simulation time) that using a cache size which was not a power of two would have had in the analysis. This would suggest that the trade-off in having an object cache is even stronger than the analysis suggests.

larger network identifiers, would be significantly more efficient. This decision is made purely to simplify later discussions.

Register File

DAIS's proposed register file, documented in chapter 6, uses a number of independent stacks, one for each register, to hold user information on-chip. Analysis is performed which shows that this approach, if implemented on a SPARC processor as a replacement for its own register file, reduces the memory bandwidth caused by register window stalls to under 5%, given approximately the same amount of silicon. Multi-stack performance can also be improved by increasing the amount of silicon available without degrading register access time, since the stack cache is not on the register read/write busses.

With registers remapped as stack heads, programmers are offered a new register interface, which may result in interesting compiler optimization techniques. This also means that interrupts can be handled without a change of context, with local parameters simply mapped ahead of the current application's variables. Macro-instructions can also be simpler to implement, where chunks of assembly code can be inserted into a program without significant knowledge of the surrounding register allocation scheme.

An interesting modification to the multiple-stack scheme could be to remove the register at the head of each stack, and instead access the stack buffers directly on every register access. The result of this would mean that stack transfers could be required on any register access, rather than just on register pushes and pops as at present. While this would increase the complexity of the register file, it is suggested that this could further decrease the stall cycles for the same amount of silicon. This should be investigated in future work into this area.

DAIS Pipeline

In OID/offset based addressing, instructions using displaced address calculations are illegal. In a virtually-addressed processor, displaced address instructions add an offset onto an address. In DAIS, addresses are OIDs, and adding a number to an OID is illegal. What is allowed is to specify an OID/offset pair in a load or store instruction, where the OID and the offset are held in separate registers. An offset from an offset within an object (*e.g.* accessing an array[x+y]) must be done in two instructions; first adding x to y, and then performing the load or store.

The only exception to this is stack accesses, since the current point reached through stack allocation (the frame pointer) is defined by both an OID and an offset (the USP register). The USP register is the only one in DAIS which contains both an OID and an offset (the frame pointer). DAIS is proposed with only a four-stage pipeline, incurring a stall when the executing program performs a displaced address calculation on stack data. The reasoning behind this shortened pipeline is given in chapter 7, where analysis shows that this stall occurs less than 1% of the time. The benefits of a shorter pipeline is that DAIS contains no pipeline-induced data dependencies. In comparison to a SPARC, eliminating the dependencies produces a 6% performance increase.

DAIS' branch prediction logic is on-par with many advanced RISC designs available today. Although not truly novel, it is still a requirement to avoid unnecessary pipeline stalls, therefore maintaining DAIS' competitiveness with other non object-oriented processors.

The Complete DAIS-256

With the pipelining strategy coupled to the caching and register stack design, DAIS should be able to execute object-addressed programs just as fast as the traditionally addressed variety. Note that the novel register stack design of DAIS is not a crucial feature of the processor, and thus could be removed from any implementation in silicon of this design. However, its advantages of flexibility and resulting performance makes it an attractive addition.

A compiler for DAIS would preferably try to make use of the register stacks for performing local data scoping, as well as for parameter passing. For example, if an outer loop count is not needed in the inner loop, then the inner loop could could use the same stack, obscuring that register for the duration of the inner loop before being `popped' off that stack. This should promote efficient register management techniques. The design of a compiler for DAIS should be considered as future work.

4.2.2 Solid-State Backing Store

Solid-state storage, although currently expensive when compared to disks, is over time becoming more attractive for general-purpose applications. The performance gains to be had are significant, both in terms of reduced data-access times and with new algorithms feasible only with a direct-mapped store (*e.g.* database compression).

In an object-store, the loss of any single object can result in many more objects being made inaccessible. To minimize information loss, some form of redundancy is required. RAID techniques, although useful in block-based devices such as disks, does not offer the ability to automatically recover data without loading in the entire block surrounding the failure. ERCA proposed in chapter 8, describes a methodology by which automatic recovery schemes can be created, such that a single card failure in the store can be both detected, corrected, and the requested data returned in a single read action on the store. Using this scheme, a number of initial ERCA-based models are constructed, including ERCA 4 which also allows for double card failures to be detected.

It is suggested that the requirement to use 100% perfect Flash chips in the store are one reason for the store's resulting cost. One way to avoid the need for such perfection is to use ERCA to automatically repair single card failures, and thus a store with only a maximum of one failure in each array row can be accessed as if no errors exist. Under ERCA 4, the facilities also exist to detect a second error occurring on a row, and for the data contained therein to be transferred to a more reliable area of the store. Through a combination of Flash management and ERCA, it should be possible to maintain a highly-reliable store even with the use of Flash components with less than perfect construction.

In comparison with a disk-based store, Flash stores result in significant speedups. With a disk system, it is normally considered essential to allow as much of an application to exist in main memory as possible, in order to avoid `thrashing'. The analysis in chapter 8 of a number of benchmarks indicates

that applications with run-times greater than a few seconds incur serious performance degradation once the available memory is decreased beyond 40-60% of that required. In a system using Flash, the critical requirement for main memory is significantly lower. During a simulation of TEX, the program runs at half speed with only 5% of the needed memory available. With disk, the same speed is not possible below about 45%, while at 5% the program takes many orders of magnitude more time to run.

Provided that the cost of Flash storage is reduced as predicted by economic investigation (also present in chapter 8), perhaps made even cheaper through the use of non-perfect devices, then this technology could easily spell the end of the disk-based storage medium. In relation to DOLPHIN, it offers an excellent basis for providing object persistence.

4.2.3 Lightweight Threads

The process manager used in DOLPHIN (chapter 9) is similar in construction to lightweight thread models available in other operating systems. For example, in UNIX lightweight threads all share the same process space, and similarly in DOLPHIN each thread can also access any data currently in the store. To promote portability in the SubKernel, and to allow the threads package to be used in systems other than DOLPHIN, the thread library is written entirely in C, using only four system-dependent routines (which must be written in assembler for DAIS, but which are standard routines in UNIX). The analysis of this package shows that writing the routines in this manner will not significantly affect the resulting efficiency. This chapter demonstrates that a threads library written entirely in C need not be less efficient than an assemble-language version. The library supports all the standard thread requirements of critical code protection, scheduling, thread creation, sleeping, and signal handling.

With respect to DOLPHIN, MultiThreads offers a fully-functional process handling scheme to both operating system and application program designers. It contains all the routines needed to construct thread-based code within the DOLPHIN environment.

4.2.4 Network Packet Transmission

The network interface provided in DOLPHIN uses a set of primitives collectively known as *MultiPackets*. This supports a lossless approach to packets, with generous maximum packet sizes and efficient piggybacking of handshaking information. MultiPackets allows the researchers involved with object-based networking on DOLPHIN to experiment with a variety of network coherency schemes. Its performance, while slower than that of datagrams, outperforms sockets by a measurable degree (between 5% and 10%). More important than just performance, MultiPackets is constructed with knowledge of MultiThreads, and thus even under the Unix version it allows multiple threads in the current process to use the network without causing other threads to block.

The MultiPackets supports both single node and multicast transmissions in its standard protocol. Chapter 10, which contains a discussion of MultiPackets, also describes a way whereby the basic TCP/IP routers could be adapted to increase the overall performance of multicasts. This would certainly be a useful achievement, since many network coherency protocols can use multicasts to good effect. In any case, by supporting multicasts by default, their actual implementation can be isolated from all

other routines involved with the network. This permits further research to be performed in this area without disturbing other levels of DOLPHIN.

Overall, MultiPackets offers a faster and more flexible network interface than that possible using standard TCP/IP sockets. It also provides a higher-level interface than that available with standard datagrams.

4.2.5 Memory Management

Chapter 11, the memory space used in DOLPHIN is described. Virtual memory is mapped onto RAM and the Flash store via the Flash management system discussed in chapter 8. Above this, the SubKernel controls the allocation of virtual memory space. This allocation scheme is built on top of a fast single-freelist algorithm, which also supports compression of free blocks. When analysed this method demonstrates better performance (in terms of allocation/deallocation speed) than the other allocators examined (typically by approximately 20%). Where it does not out-perform an algorithm in speed, it runs a close second, adding the advantage of memory compaction and the resistance to fragmentation.

This chapter also discusses how objects can be allocated and deallocated, without compromising object security. Overall, the memory management facilities, coupled to the processor design of DAIS, help to provide a fast and flexible object-oriented architecture. When this is joined with the other parts of the SubKernel (*e.g.*MultiThreads and MultiPacket), the environment created meets the design objectives of DOLPHIN; to provide a strong basis for further research into the area of object systems. In the next part of this document. This environment is used to realize a simple exercise in object network management. This part also presents work identified for future research in this area.

4.2.6 Networked Object Management

As a way to demonstrate the use of the DOLPHIN environment, an object network manager is implemented in chapter 12 using the facilities available in the SubKernel. The algorithm is based on a modified Li and Hudak model originally used to support shared virtual memory pages in a small cluster of machines. It is rewritten to support networking of both objects and their descriptor structures, and with the ability to remove references to data once that data is no longer reachable locally.

Two benchmarks are executed on this networked system; a travelling salesman problem and a matrix multiplication. The benchmarks are used primarily to demonstrate that the networked environment actually works, but also to allow a comparison to be made with another object system for which the results of running these benchmarks had been obtained. The comparison shows that DOLPHIN performed similarly to that of the other system (Orca implemented on a modified Amoeba kernel). This is an excellent result, considering that the other system was designed with efficient networked object support in mind. The Orca implementation broadcasts writes over the network to perform its information sharing, which although allowing fast writes to shared objects, inhibits the effectiveness of multiple network environments.

4.3 Realization of the DOLPHIN Environment

Although this document talks of the complete DOLPHIN environment, the work performed so far only indicates a profitable direction which the environment, if fully implemented, could take. The complete development of such an environment (a processor, surrounding hardware, operating system, and development environment) was financially and chronologically impossible within the research period described here. Instead, the research concentrates only on the key issues of the design;

- the effects of object-based addressing on the processor,
- how changes in processor register implementation could be used to reduce processor-memory traffic,
- how persistence could be implemented so that making objects persist did not become a major bottleneck of the system,
- how a SubKernel could be written for DOLPHIN without the need for extensive processor-dependent machine code in hard-to-debug code segments (*e.g.* the process-management routines),
- the benefits of a custom network protocol in object-network management, integrated fully with the SubKernel process manager,
- and how object and kernel memory can be managed, supporting both read-only and read/write objects, with a memory allocation system designed especially for highly dynamic systems.

In the processor-based research, all analysis is made either algebraically or by simulations based on implementing the part of DAIS under investigation within a SPARC processor. This was possible through the use of library routines, supplied by Sun, which allowed single-stepping of a program at the machine code level, under control of a second program. The second program could therefore monitor the instruction set usage and register and memory activity of the application under investigation. This information could then be used in a simulation of the element of DAIS being analysed. This avoided the need to write a full DAIS simulator and optimizing object-oriented compiler, and also avoided the problems of `how good the optimizer actually was', as well as the effects of a specific instruction set design (as opposed to the object cache, register stacks, and pipeline). It is also suggested that the instruction set and compiler cannot be sensibly designed independently for best performance. A compiler and DAIS emulator are planned for the future.

Object persistence using Flash cards is once again analysed with the single-step routines used in the DAIS analysis. This allows memory traffic to be monitored during the execution of real-world applications. It may produce results which reduce the apparent effectiveness of the Flash-management system, since object-based implementations of each application would avoid code necessary to load and save data files. Considering the disparity in the analysis results between the disk- and Flash-based systems, the effects of whether an application is object or virtual-memory based should be insignificant.

The SubKernel routines discussed in this document were completely implemented in C code. Their portable nature allowed them to be executed in a Unix process, within which they were debugged,

tested, and analysed. These routines were used in the two demonstration applications discussed in chapter 12.

Chapter 12 used a software-based object interface to access object data. As stated, this used the SubKernel networking and process management routines. The interface itself allowed for object creation, and object reads and writes given an OID, index, a data pointer (for holding the data), and a size (for the transfer). These were written specifically for the examples, and are not part of the DOLPHIN system. They are not documented within this report.

In the future it is hoped to implement a full version of the DOLPHIN environment. This will likely use a high-performance virtual memory based processor, which either interprets DAIS code or executes native code binary translated from a DAIS-compiled version. This would allow development of a DAIS compiler, while retaining the flexibility to alter parts of the DAIS design.

4.4 Summary

In conclusion, the DOLPHIN environment proposed here should provide the object-oriented research community with a useful starting point for their own investigations. I myself will be using this system, and advancing its status from a proposal to a full implementation (see the chapter on future work). However, this system is not just a rehash of previous work, or just another programming aid. It covers over three years of work on making object systems efficient in executing programs, and on providing the support required to perform other leading-edge research into this area, including support for networked objects.

Chapter 5

DAIS: An Object-Based Processor Cache

The DAIS architecture is an attempt to support object-based addressing at the instruction level. Object addressing has been made the responsibility of hardware both for security (obligatory for world-wide object addressing) and efficiency. In providing this hardware support, lessons in traditional processor design have been borne in mind; the triumph of RISC over CISC has told us that simplicity and provision of the minimum general support result in flexibility and speed. For this reason, we have examined the simplest methods of object access first, and moved only to more complex ones when we deemed it necessary. The dictionary definition of *dais* is a low platform, and we believe that the DAIS architecture is the lowest (simplest) hardware platform which will support secure world-wide object addressing.

DAIS makes use of a cache structure based directly on object descriptors and offsets, rather than the more common physically or virtually addressed cache. This cache structure has been designed to allow object bounds-checking to be performed by a simple bit-ANDing, rather than arithmetic comparison. A problem with cache misses on a object-addressed processor is that when the primary cache misses, not only does the new object data need to be loaded, but also information on the object itself, such as length and access rights. These extra accesses can significantly increase the miss penalty. To counter this, a secondary object cache is also used which caches such information, making miss penalties more in line with traditional caches. We present analysis on performance with and without this object cache.

Currently DAIS is only a proposed architecture, and that no hardware version of the processor exists. This chapter deals only with the caching strategy, with details of other aspects (*i.e.* the register file and pipeline) of the processor's design presented in subsequent chapters.

5.1 Addressing models

In RISC processors all memory accesses occur in load and store instructions. An example of such a load might be

```
LOAD r1,[r2]
```

The effect of which would be to perform the operation $r_1 \leftarrow memory[r_2]$. On a simple RISC micro-controler equipped with fast memory, this can be performed in a single clock cycle since the contents of r_2 can be simply placed on the address bus with the data returning on the next cycle, giving the sequence.

$$abus \leftarrow r_2, r_1 \leftarrow dbus \tag{5.1}$$

where operations on one line are assumed to take place in a single cycle. As memory sizes go up, the speed tends to go down as slower cheaper components are used, but RISC designers try to maintain single cycle load operation by the use of caches.

5.1.1 Cached RAM access

The sequence of operations involved in a load from address r_2 into r_1 now is:

if r_2 in cache then $r_1 \leftarrow dcache$ else

1. $abus \leftarrow r_2$
2. *wait 2 cycles*
3. $r_1 \leftarrow dbus$

5.1.2 Virtual memory access

When virtual memory is introduced the logical sequence of operations becomes longer. In principle we have to do the following:

1. Use the top 20 bits of r_2 to index the page table.
2. If page not present cause an interrupt, else $t \leftarrow PageTab[r_2(31:12)]$
3. $r_1 \leftarrow Memory[t + r_2(11:0)]$

This now involves two memory accesses where we had one before, each of which could take several clock cycles. Virtual memory only becomes viable to the extent that caches could be used to short circuit this process. First, address translation caches were used to allow the virtual to physical address translation to be done in one cycle. Then processor designers introduced caches indexed by virtual rather than physical addresses. The end result is that for perhaps 95% of loads, the **Virtual memory access** is reduced to a **Cached RAM access** that runs in one cycle. Virtually-addressed caches are not frequently used in data caches, due to context-switch overheads (changing virtual to physical mappings usually means that a virtually-addressed cache must be flushed).

Let us now look at the logical sequence of operations involved in object addressing.

5.1.3 Object Mapping

In DAIS, OIDs are used for all user-based memory references, including stack accesses. This makes memory referencing uniform over the whole instruction set. All object references are specified in the form *OID* plus *offset*, where a legal offset must lie between zero and the length of the object minus one inclusive.

One of the forms that a load instruction can take is:

```
LOAD r1,r2[r3]
```

As before, r_1 is the destination register. The address is now provided by two registers; r_2 specifies an OID and r_3 an offset into the object. Other variants of this instruction include r_3 being replaced by a constant offset.

To access main memory, the OID/offset pair must be translated into a physical RAM address. RISC philosophy suggests that all aspects of a processor should be kept as simple as possible, and the simplest possible translation between OIDs and physical addresses is for the OID to be its object's **physical RAM** address. This approach, which we shall term case 1, incurs no translation overhead in accessing objects, but is not without its difficulties.

Additional instructions have to be planted on each object access to test if the OID currently refers to disk or RAM. When an object is returned to disk, any pointer fields must be checked to ensure that they point at disk addresses. This is undesirable because it implies that the swapper must know about the data structures used by higher level software.

In traditionally-addressed processors, virtual memory is often used to enable main memory to act as a direct-mapped window onto an I/O mapped backing store. This approach unifies backing store and main memory addressing. If our object processor used the virtual address of the object data as the OID, then the address renaming required to support the backing store in case 1 is no longer required. We shall term this approach case 2.

This has its own difficulties. In particular it makes it difficult to compact the virtual address space as objects are deleted. This leads to a fragmented virtual address space in which excessive paging can occur during object allocation or traversal. Although algorithms do exist to allow heap compaction on a single processor, their use on a distributed shared virtual memory is much more problematic.

Additionally, since *every* object access should be bounds checked against the length of the object, as well as other checks which may be required (*e.g.* read/write bits), direct use of virtual addresses provides no protection of this sort. It also becomes hard to provide access control at a granularity of less than a page with only virtual memory based protection.

We propose that an OID be the virtual address of the object descriptor, which in turn contains (amongst other things) the virtual address of the object's data part. The virtual memory itself is split up into a number of partitions, each handled by independent memory management software. The object descriptors lie in their own partition, while object data lie within another. Object addressing is now a level of abstraction on top of virtual addressing.

The logical sequence of steps to load data into r_1 from an object (OID in r_2) with an offset held in r_3 is:

1. The OID is used as the virtual address of the object descriptor as in section 5.1.2.
2. From this, the virtual address of the object data is obtained.

   $\mathrm{oa} \leftarrow VMem[r_2 + \mathrm{addrfield}]$
3. The object offset is then added to the object data address,

   $\mathrm{fa} \leftarrow \mathrm{oa} + r_3$

   to obtain a field address.
4. The bounds of the object are now found from the descriptor.

   $\mathrm{br} \leftarrow VMem[r_2 + \mathrm{boundsfield}]$
5. The bounds are compared with the field offset for access validity.

   Interrupt if $r_3 > \mathrm{br}$
6. The field address is used as the virtual address of the object data to be accessed.

   $r_1 \leftarrow VMem[\mathrm{fa}]$

   This is again a virtual memory access.

This process is clearly more costly than a simple memory access. It involves three memory accesses and four arithmetic operations. Even assuming that the full paraphanelia of virtual addressed caches is available, this is still at least a five cycle operation. One of our key aims has been to show that this can be achieved in a single cycle.

5.1.4 Comparison to other Systems

The proposal given above is not the only possible method for successfully implementing object-based addressing. Other object systems have approached the problem in different manners. For comparison purposes, let us consider five other architectures; the Rekursiv, MUTABOR, MONADS, the iAPX-432, and SYSTEM/38.

The Rekursiv [Harland, 1988] holds the object descriptor and data together in a single unit. No virtual memory management is used, and instead OID to physical address translation is performed in a 64 K-entry direct-mapped hashing table. When an object is needed, this table is checked. If the OID is not present in the table, then the OID which is currently held there is removed, and its corresponding object descriptor and data copied in its entirety onto the backing store. The desired object is then copied from the store, and the table updated with the new OID. The direct-mapped strategy used in the table could cause object thrashing between main memory and the store, and the lack of virtual memory makes supporting large objects less efficient than smaller ones.

MUTABOR [Kaiser, 1990, Kaiser and Czaja, 1990] uses a *two-space* model, as opposed to a flat object space. The processor accesses objects within the *active space*, where short object names are used as the virtual address of their respective objects. When an object is needed but not present in the active space, it is copied from the *passive space*. The passive OIDs are larger than those used in

the active space, and are based on backing-store (rather than virtual) addresses. Transfers between active and passive space require that all OIDs be converted to the appropriate active/passive form. This conversion is similar to that discussed for case 1 above. The need to perform conversion is fundamental in this two-space approach. For simplicity, a single object naming scheme for all objects in the heap would have been preferable.

Each object in MUTABOR must be contained within a single 4 KByte page. Up to 16 objects can share a single page. Objects larger than a page are not supported. Capabilities (*i.e.* methods) for an object are also stored within the same page. Inheritance of methods requires that the methods from one page be copied to the new method's page. This is not as flexible as a true single-level store, where only a pointer to the old methods would have to be duplicated.

MONADS [Keedy, 1984, Rosenberg and Abramson, 1985] uses a hierarchical construction of modules and objects. A module contains executable code and objects, and generally represents a source-code module. Object data contained within a module can not be read outwith the module, and instead has to be passed via procedure calls with the executable parts of the module. This layered approach is similar to that used in MUTABOR. OIDs are formed from the virtual address of the module, plus type, offset from the start of the module, and length information for the desired object. These OIDs are called capabilities. Since the capabilities carry their own length information, changing the length of an object would require that all capabilities for that object be tracked down and changed, so this action is not supported by MONADS.

The iAPX-432 [Hunter *et al.*, 1985, van Rumste, 1983] holds object permissions, data, and other object information in a number of separate objects. Yet more details on objects are held in type managers (*i.e.* methods). This allows support for flexible inheritance and object length modification, at the expense of being overly complex. The 432's OID translation mechanism passes through many levels of indirection, before pinpointing the required object data. This makes OID translation a costly process. In a similar approach to MUTABOR, the 432 makes use of a two-space memory model, except that transfers between the spaces is under control of the type managers (unlike the automatic scheme of MUTABOR). When necessary, the type managers ask for an object to be transferred into the active space. There, the manager will make all necessary changes to the object, before finally asking the system to return the object into the passive space.

Finally, SYSTEM/38 [Soltis, 1977, Soltis and Hoffman, 1987] uses an OID structure which contains the virtual address of the object, a checkcode, and type information (access rights, length, *etc.*). The checkcode is required to avoid using an OID to reference a previously deleted object, which is possible in SYSTEM/38 since it uses explicit object deletion and *not* automatic garbage collection. Holding the type information in the OID complicates object size changes (not done in SYSTEM/38). This system could only supported a maximum of 2^{24} objects, although it was possible to have OIDs to select areas within objects (although the need for this is questionable).

5.2 DAIS

OID to physical translation in DAIS is based on the object-descriptor/object-data model proposed earlier. In this approach, memory addressing is performed using OID/offset pairs. We call the object-based view of memory *object space*. Each OID is the virtual address of the respective object descriptor. The descriptor contains the virtual address, object status, and the overall length (in bytes) of the object data (see figure 5.1). We say that all virtual memory accesses lie in *virtual space*. Using virtual-to-physical translation, virtual addresses are converted to physical ones, which all lie in *physical space*. The connections between object, virtual, and physical space are shown graphically in figure 5.2. In object space, each unit of offset from an OID is equivalent to 8 bits.

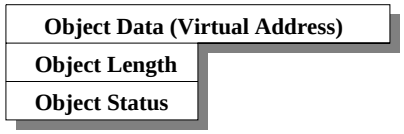


Figure 5.1: DAIS Object Descriptor

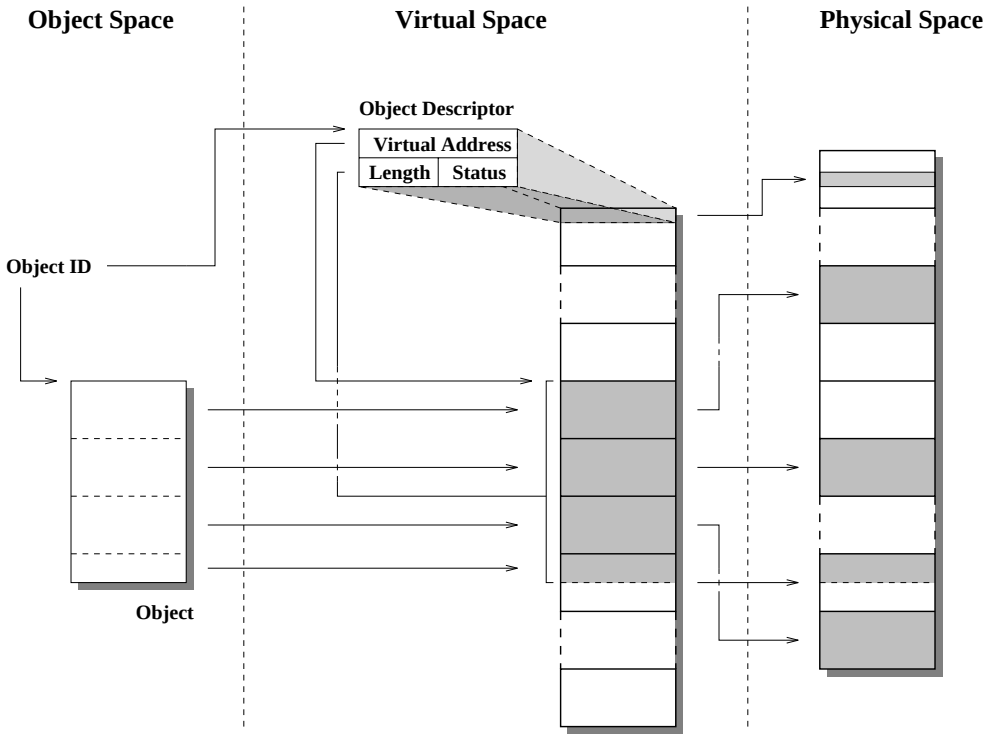


Figure 5.2: Mapping of Objects onto Physical Space

The OID size used in the DAIS proposal is largely independent of the underlying design of DAIS. The OID size used in DOLPHIN is not discussed until section 5.9. It is however assumed that an OID can be communicated between the processor and the external memory interface in a single transfer. Without this assumption all transfer rates concerning internal cache spills and refills will need to be

adjusted with respect to the time to load OIDs and their relative frequency of occurrence in cache activity.

5.3 Primary Processor Cache

RISC designs frequently make use of on-chip caches to hold instructions and data. To reduce the complexity of the initial DAIS design, we use separate direct mapped data and instruction caches. Together, these caches are collectively named the primary processor caches. Other cache organizations could be investigated in the future.

The caches are mapped using a hash of the OID and offset. This allows different parts of one object to be spread over several cache lines. The use of OIDs and offsets to cache object data is not a technique found in other object-based processors, where caching of the conversion from OID to virtual address (with caching of object data occurring as a second step using the resulting virtual address) is generally accepted as the norm. This allows us to bypass the OID to virtual address translation step of object accesses when the relevant object data is in the primary caches. The structure of the D-cache (data cache) is shown in figure 5.3. The I-cache (instruction cache) is similar to the D-cache, except that the cache access - write block, along with the dirty and RO parts of the cache, are not implemented (the I-cache is read-only).

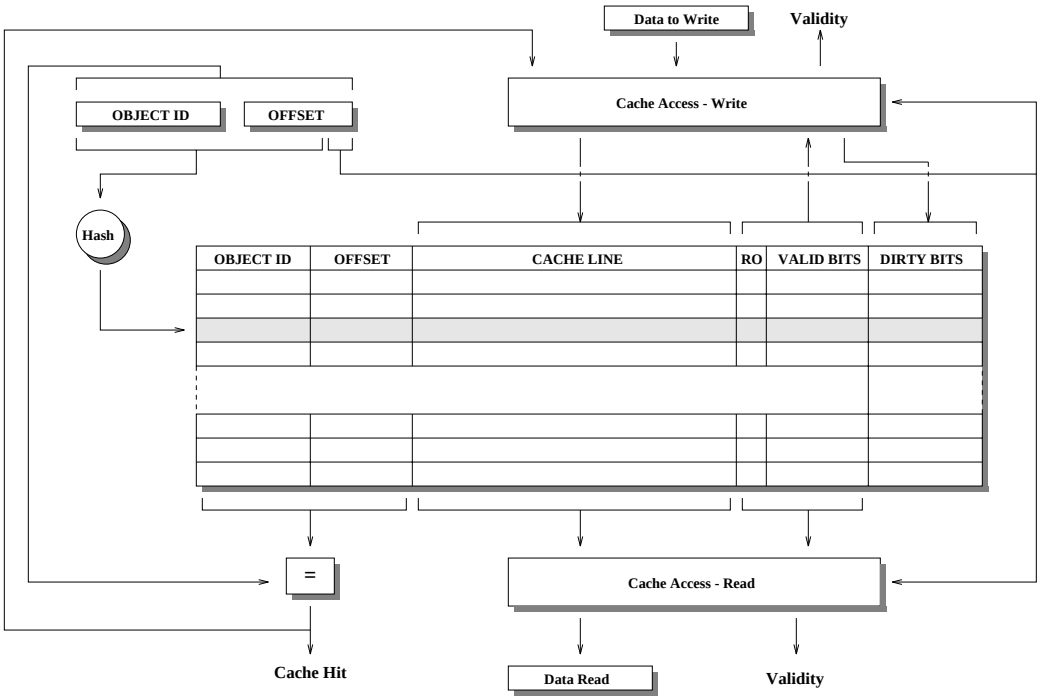


Figure 5.3: Structure of Data and Instruction Caches.

In the D-cache, a write to any word of the cache line sets the dirty bit for that word. This is used when the cache is being flushed (*e.g.* if the line is being replaced by another object entry). Only words

in the cache marked as dirty need to be written back to memory. In this way, the D-cache acts as a *write-back* cache. If the D-cache contains information which is read-only, the RO bit of that cache line is set, and any attempt to write to that line is trapped by the processor.

To save performing an arithmetic-based bounds-check on each object access, valid bits are used instead. Each valid bit corresponds to a single byte (the smallest element which can be accessed using a load/store instruction) of the cache line. If the bit is clear, then that byte of the cache is unaccessible. The valid bits are set for bytes within the bounds of an object, and cleared for all other data. Attempting to access data whose valid bit is clear is interpreted as accessing the object beyond its actual size, and is trapped.

If objects were close enough together, then a cache line could contain data from more than one object. Even so, only the object hashed to that line is accessible, with data from the others being unaccessible (their corresponding valid bits would be clear).

The use of the primary caches frequently eliminate the need to convert OIDs to virtual addresses. Range checking of object accesses is also eliminated in the traditional sense, requiring only a bit test involving the valid bits of the relevant cache line. These two features give DAIS a similar memory access rate as those obtainable from traditional (*i.e.* virtual-addressed) processors. This is for the same external bus size (plus an additional bit for the tag) and speed. Many modern processors (*e.g.* the latest series of Alpha processors) use 64 bit virtual addresses, which will take up the same space as that of an OID (again ignoring the tag overhead).

5.4 Primary-Cache Misses

On a primary cache miss, the desired object's descriptor must be accessed. This contains the virtual address of the object data, along with the object's length and status information. The data can then be transferred from main memory to the cache. Once the transfer has completed, all the dirty bits (for the D-cache) are cleared, and the valid bits are initialized. The valid bits are set only for the data bytes which lie within the object's boundaries. This is calculated from the object's length, and the offset (from the start of the object data) used in loading the cache line. Cache line data is aligned to line-length boundaries, which simplifies the cache-load hardware.

For the D-cache, before a cache line can be loaded the current contents of the line must be checked. If the line contains dirty words, then these words must first be flushed to memory. This requires that the line's relevant object descriptor be loaded, allowing the virtual address of the dirty words to be calculated.

The RO (read-only) bit of the D-cache is calculated from the virtual memory protection present on the object data being loaded. If the protection is read-write, then the RO bit is cleared, otherwise it is set. There is also a FRO (force read-only) bit in the status information held in each object descriptor. If that bit is set, then so is the RO bit for that object in the primary cache. This access protection could be used in an operating system to implement object locking.

With the need to access an object descriptor at least once on each primary cache refill, the idea of caching the object descriptors themselves appears attractive, and therefore this descriptor cache has

been included in the DAIS proposal. We have termed this cache the *secondary* or *O-cache*. Technically, the caching of descriptors could be performed by the D-cache, but it was felt that the higher access rate of non-descriptor information in comparison to descriptor accesses, coupled with the direct-mapped design of the D-cache, would cause the cache entries used for descriptors to be quickly replaced by object data. This would have significantly reduced the effectiveness of descriptor caching. Separate caches also helps to avoid the problem of having both the descriptor and its respective object data mapped onto the same cache line; if the line currently holds dirty data, then the process of loading the descriptor information in order to flush the cache line to memory would in turn destroy the data to be flushed. Without the D-cache, the simplest solution to this problem is to access descriptors directly in memory, marking them as non-cacheable. The structure and information pathways of the O-cache are shown in figure 5.4. The benefits of using this cache are investigated in the analysis section.

The interconnections between the CPU core, the D-, I-, and O-caches, cache miss management, virtual memory management, and physical memory is shown in figure 5.5. Although the figure shows the CPU only operating with object-based addressing, it can also use physical addressing. Physical addressing is restricted to supervisor mode, and allows the object management routines to bypass both the fault- and virtual memory-manager.

With so little information held within each cache entry, cache access rates should be similar to standard data caches. Additionally, since an object's virtual address and overall length changes infrequently, the primary caches are rarely flushed.

Consider a DAIS implementation whose data and instruction cache both hold 256 lines of information. From the analysis presented later in this chapter (section 5.8), which assumes an OID size of 64 bits[1], it is suggested that an instruction cache size of 8 KBytes is used, and this corresponds to a line length of 256 bits. Again from the analysis, the data cache line length suggested is 128 bits (4 KBytes of storage). This equates to a data cache consisting of 7584 bytes of storage, and the instruction cache to 12032 bytes. The object cache contains 128 entries, requiring 2464 bytes of storage. The total space for all three caches is approximately 22 KBytes, which is similar in size to many modern processor designs. These byte sizes all include the tag and other overheads associated with the respective caches, as well as the actual data part.

5.5 Benefits of the Object Cache

The layered approach to DAIS' caching structure has been put forward as a benefit over other object-oriented architectures. However, what exact benefit does the secondary (*i.e.* object) cache give? Here, a theoretical analysis of secondary cache performance is presented. The symbols used in this analysis

[1] As OID size increases, then the effective size of the data cache will be reduced, since a larger proportion of each data cache line is needed to hold OIDs (in comparison to holding non-OIDs whose size does not change with different OID-size implementations).

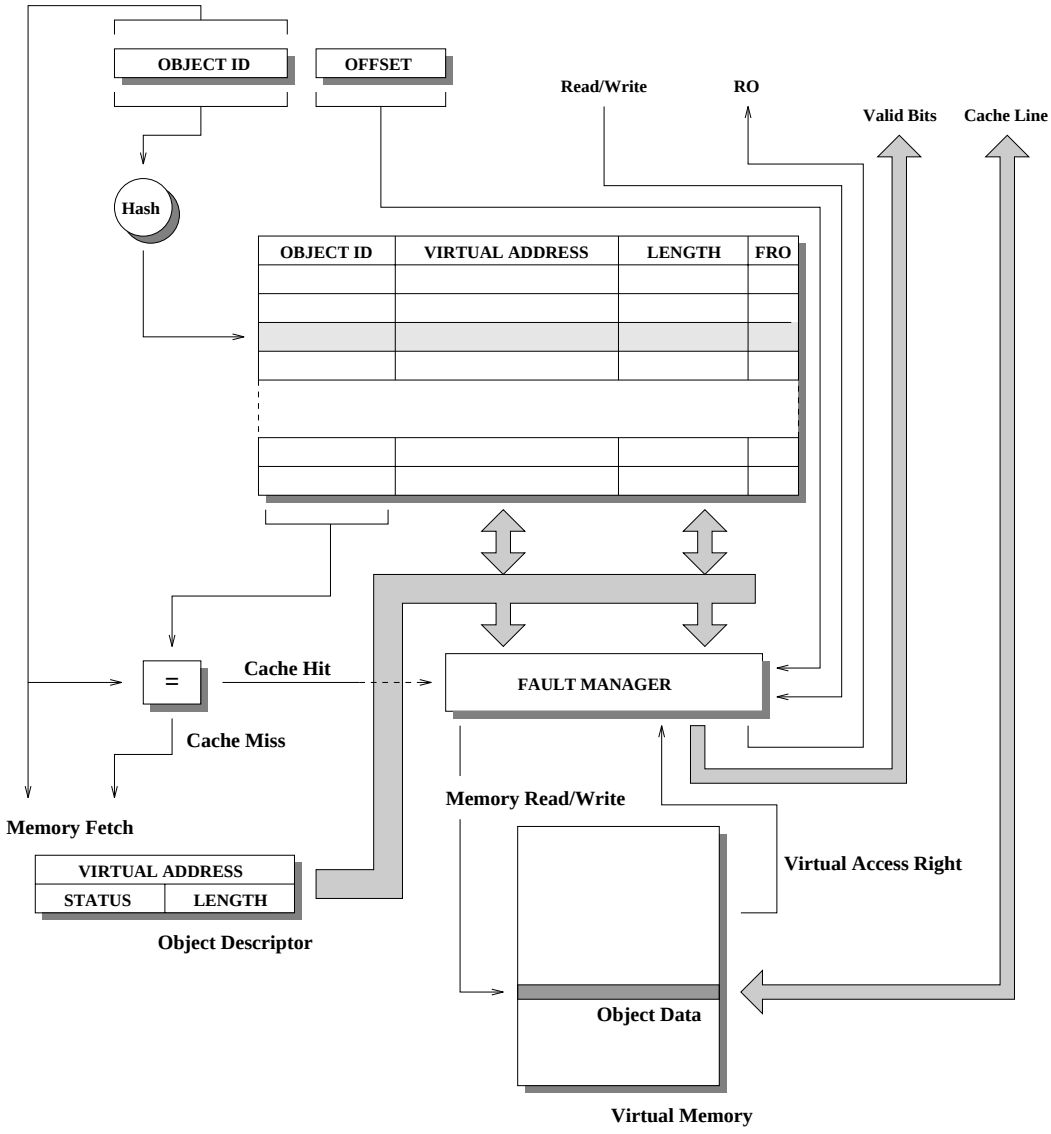


Figure 5.4: Structure of Secondary Object Cache

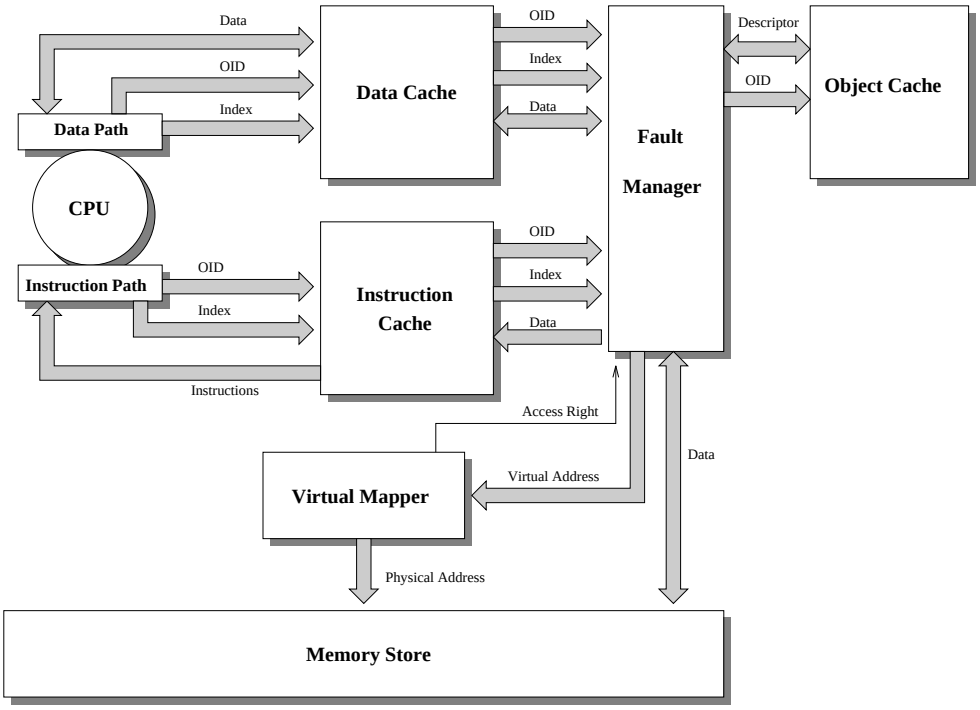


Figure 5.5: Overview of DAIS' CPU to Memory Pathways.

are shown below:

M_I, M_D, M_O	=	I-, D- and O-cache miss rates
P_I, P_D, P_O	=	I-, D- and O-cache miss penalty
T_I	=	Time to read one I-cache line
T_D	=	Time to read/write one D-cache line
m	=	Fraction of instructions referencing D-cache
d	=	Probability that a particular memory word in D-cache is dirty
n	=	Number of memory words per D-cache line

To analyse the cycle-per-instruction (CPI) benefits of the O-cache, the analysis shall begin in general terms. To allow fair comparison with other RISC-based architectures, DAIS is considered to contain a pipeline which (ignoring pipeline stalls) allows single-cycle instruction execution. The overhead per instruction for I-cache misses is $M_I P_I$ cycles. Similarly, the overhead per instruction owing to D-cache misses is $mM_D P_D$. The total CPI is thus:

$$CPI = 1 + M_I P_I + mM_D P_D \tag{5.2}$$

Here, both miss penalties include the time taken to fetch the object descriptor and then fetch the cache line of object data. To discover performance with and without O-cache, it is necessary to derive formulae for P_I and P_D under both conditions.

5.6 CPI without O-cache

If the I-cache misses, then the miss penalty P_I must be the time P_O to load the object descriptor plus the time T_I to load in the missing cache line. On a D-cache miss, each dirty memory word held in the cache line must first be written to memory. Given the probability d of a cached memory word being dirty, then P_I and P_D are defined as:

$$P_I = P_O + T_I$$

$$P_D = (2 - (1 - d)^n)P_O + (1 + d)T_D$$

In section 5.4, it was argued that, in systems without an O-cache, descriptors should not be allowed to be loaded into the D-cache. If this was allowed, the variable P_O would have to be changed to take this into account. In addition, some way would have to be found to get around the problems discussed in that section.

Note that M_D, the data cache miss rate, will be different if the data cache size is changed. A similar argument holds for M_I and M_O for changes to the size of the instruction and object caches respectively.

5.7 CPI with the O-cache

When an object cache is included, the I- and D-cache miss penalties P_I and P_D must be updated to reflect O-cache hits and misses. On an I- or D-cache miss, the object cache is scanned. The object descriptor is returned in one cycle if contained within the object cache. If not, the object descriptor must be fetched from memory. In addition to scanning the object cache, the appropriate line of the cache must be read in.

P_D is again similar to P_I, except that is must also include the probability that the data line to be replaced contains dirty data. Saving dirty data requires access to the O-cache, which may in turn cause an O-cache miss. Thus:

$$P_I = 1 + M_O P_O + T_I$$

$$P_D = 2 + M_O P_O [2 - (1 - d)^n] - (1 - d)^n + (1 + d)T_D$$

5.8 Some Example Figures

Here, some figures will be substituted into both CPI equations to give the reader some idea of the increase in efficiency the O-cache gives.

TEX, DeTEX, Zoo, and Fig2dev were simulated. These programs were chosen for their general availability, coupled with their cpu-bound and non-interactive nature. If we consider all heap and stack accesses in these applications would be converted to object accesses in DAIS, then the analysis performed should not be unreasonable. The miss rates for direct mapped caches of 4K and 8K bytes with line lengths of 16 and 32 bytes are shown in tables 5.1 and 5.2. These were derived using SHADOW[SUN, 1992] on a SPARC. For the case of the D-cache, the miss rates are *per instruction*,

Table 5.1: Data Cache Miss Rates (Per Instruction)

Benchmark	4K cache		8K cache	
	16 byte lines	32 byte lines	16 byte lines	32 byte lines
TEX	1.79%	1.88%	1.24%	1.23%
DeTEX	1.15%	1.93%	0.56%	1.32%
Zoo	2.46%	2.26%	1.84%	1.53%
Fig2dev	1.04%	0.99%	0.53%	0.47%
Average	1.61%	1.76%	1.04%	1.14%
Weighted Av	1.76%	1.86%	1.21%	1.22%

Table 5.2: Instruction Cache Miss Rates

Benchmark	4K cache		8K cache	
	16 byte lines	32 byte lines	16 byte lines	32 byte lines
TEX	5.99%	3.74%	3.74%	2.35%
DeTEX	7.31%	5.08%	3.73%	2.72%
Zoo	1.12%	0.78%	0.35%	0.23%
Fig2dev	9.95%	6.70%	5.74%	3.98%
Average	6.09%	4.08%	3.39%	2.32%
Weighted Av	5.94%	3.81%	3.74%	2.37%

and not per D-cache access. This means that m can be omitted from equation 5.2 when estimating performance. The averages shown in these tables are both `straight', and weighted on the number of instructions executed by each of the simulations. Other results from these simulations (pertaining to write-backs of cache lines) where used to estimate d at 0.2.

The size of OIDs was assumed to be 64 bits. In the applications examined, the proportion of address pointers to non-address information was small, and so the effect of increasing address size of 32 bits (as used in the SPARC) to 64 bit OIDs was ignored. In other applications whose data consists of a larger proportion of OIDs to data, then the effect of using 64 bit addressing could become significant.

Unfortunately, the miss rate of the O-cache cannot be predicted exactly from such a cache trace. Analysis shows that over half the O-cache accesses are caused by I-cache misses. Since temporal locality of instruction objects is high, it is anticipated that nearly all O-cache misses will result from D-cache misses. For analysis purposes, the O-cache miss rate was assumed to be 10%. Interestingly, doubling this value only has a small effect on the performance. In a system without an external cache, the performance degradation is under 4%. With external cache, the degradation is only 2%.

Four scenarios have been staged; all combinations of processors either with or without O-cache and an external cache. When the processor has no O-cache, each object descriptor must be fetched from main store. The assumption is made that without an external cache each main store access requires 5 processor cycles. For a system with an external cache, access is assumed to require and average of 2

cycles. Together with the assumption of a word-sized (64 bit plus tag) external data bus, these times determine P_O, T_I, and T_D. The miss rates used for the D- and I-caches where the *weighted averages* shown in tables 5.1 and 5.2. For all four cases, the resultant CPI (disregarding other processor stalls) is shown for different combinations of I- and D-cache sizes (4 and 8 KBytes) and line lengths (16 and 32 bytes) in figures 5.3–5.6. The minimum CPI from each cache size combination is highlighted.

Table 5.3: CPI with no O- or E-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	2.64	**2.59**	2.20	**2.16**
	4K/32	2.93	2.89	2.49	2.45
	8K/16	2.50	**2.45**	2.06	**2.02**
	8K/32	2.67	2.63	2.23	2.20

Table 5.4: CPI with O- but no E-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	**1.97**	2.10	**1.71**	1.78
	4K/32	2.22	2.34	1.95	2.03
	8K/16	**1.89**	2.02	**1.63**	1.70
	8K/32	2.04	2.17	1.78	1.85

Table 5.5: CPI with E- but no O-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	1.66	**1.64**	1.48	**1.46**
	4K/32	1.75	1.74	1.58	1.56
	8K/16	1.60	**1.58**	1.42	**1.41**
	8K/32	1.66	1.64	1.48	1.47

Table 5.6: CPI with both O- and E-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	**1.44**	1.48	**1.32**	1.34
	4K/32	1.54	1.58	1.42	1.44
	8K/16	**1.40**	1.44	**1.28**	1.30
	8K/32	1.47	1.51	1.35	1.37

From these four tables, the trade-offs in choosing D- and I-cache sizes and line lengths can be investigated. The benefits of including an object or external cache are also part of these tables. In all cases adding an O-cache resulted in better performance than doubling the size of a 4 KByte D-cache. For the cache setup detailed in section 5.3, the CPI are:

O-cache	External Cache	CPI
No	No	2.16
No	Yes	1.46
Yes	No	1.78
Yes	Yes	1.34

Hence, the O-cache gives a 21% speed improvement when there is no external cache present, and a 9% improvement if there is.

5.9 A Networked Object Store

The ideal object store should support the abstraction of an infinite size. In practice, only a finite object store is possible. However, a finite store can appear infinite provided that the virtual address space is much larger than can be used during the lifetime of the system. Consider a super-processor implementation of an object system. Give it 10 ns main memory, 256 bit data bus, and a life-span of 100 years. In its lifetime, the processor could write and read (with no stall cycles) every location of a $2^{62.13}$ byte memory once. This would suggest that 2^{64} bytes of virtual memory for a single machine would be sufficient.

It is assumed that the average object size (including management overhead) for the object system is 512 bytes. Average object sizes (ignoring management overheads) given for MUTABOR (300 bytes) [Kaiser and Czaja, 1990], iAPX-432 (300 bytes) [Pollack *et al.*, 1981], and Hydra (326 bytes) [Wulf *et al.*, 1981] suggest that this is a reasonable value. With a virtual memory size of 2^{64} bytes, this would allow 2^{55} objects to exist simultaneously upon a node.

For a networked machine, node addresses based on internet gives only 2^{32} possible combinations. This is slightly less than the number of people currently alive in the world, and is too restrictive if the addresses are geographically hierarchical. Even ethernet-based addresses may be too restrictive, especially if in the future computers are composed of many hundreds (or thousands) of individually addressed processors. To allow for future expansion, we will use 64 bits.

On the network, objects must be fully identified, and therefore require at least 128 bits (network address combined with local virtual memory size) for identification. Networked object descriptors must be impossible (or at least prohibitively difficult) to forge, and some researchers (*e.g.* in [Cockshott, 1990] and [Wallace and Pose, 1990]) have suggested adding a random number to networked descriptors, which acts as a descriptor *checkcode*. These codes make the descriptors appear to be sparsely allocated, making valid descriptors difficult to guess. Considering the possible performance of future machines, with respect to the number of guesses they could make in a second, we prefer a checkcode size of at least 72 bits. This suggests an descriptor size of 200 bits. For memory alignment purposes, the descriptor size should be rounded up to 256 bits.

5.10 DAIS-64

Although the above discussion suggests that a descriptor should be 256 bits, this size of OID could be expensive to implement with today's VLSI technology. For performance reasons, the OID should be transferable between on-chip caches and registers in a single cache access. The registers in DAIS would also have to be capable of holding an entire OID. Immediately suggesting a processor with a virtual address size of 256 bits could cause DAIS to be immediately rejected as a viable system, even though this size of address could be effectively used within the next few years. Instead, we first propose DAIS-64, an object processor which uses 64 bit virtual addresses.

Since most objects used by the local machine were created locally, the use of network-addresses and random-number bits is redundant for many descriptors. 64 bit virtual addresses could be used for addressing data contained in the local address space, which can be aliased to 256 bit network object descriptors whenever a network object is being referenced. This has the benefit of reducing the processor bandwidth requirements in handling descriptors, reducing the amount of space needed to hold a descriptor, and reducing the amount of processor transistors needed in handling descriptors. The network descriptor (named PIDs or *persistent identifiers*) to local descriptor (named OIDs or *object identifiers*) aliasing function can be controlled by an object management system. However, the complexity of using aliases will increase the management overheads when dealing with object networking.

DAIS-64 uses a word size of 64 bits (the width of the data bus). Each word also has a corresponding tag bit. OIDs are protected from modification using this tag. If the tag is set, then the word is thought

of as an OID. The tag bit can only be set by the operating system (or by copying the whole of another OID in a single action), although it can be cleared by writing data to any part of the word.

It could be argued that the short OIDs of DAIS-64 are cheaper and simpler to implement than a design based on the proposed 256 bit network PIDs. The short OIDs are aliased to network PIDs, with networked PIDs used for network-based communications and the short version used for all object references local to the processor. This small OID has little redundant information (in comparison to the networked variety), resulting in memory savings and increased cache efficiency. However, this argument fails to examine the problems surrounding the translation between PIDs and OIDs.

When an object is sent onto the network, the data part of the object must be scanned for OIDs. This is an order n action. All identifiers found must be translated into PIDs. Translation could be performed by looking up each OIDs object descriptor page, which contains the PID. If the processor itself performed this translation, then a significant part of the processor loading could be consumed by network traffic.

When an object is read from the network, all network-wide descriptors must be immediately identified. These must be converted to local OIDs before the object data can be read. Object descriptors which have not been seen before in the local machine must have an object descriptor record created for them. The translation process would probably be most efficiently performed by first hashing the PID, and then using that as an index into a table of linked lists to object descriptors. This process would undoubtably be done by the processor, since the translation may require memory map modifications, memory allocation, hash-table changes, *etc*.. This overhead, coupled with that incurred with outbound data traffic, makes the scheme using OID/PID translation decidedly processor intensive, not to mention complex to implement.

Security is another issue which is an important factor in object networking. If someone is monitoring the network, then it is possible to read valid PIDs straight out of data packets, and then to use the PIDs fraudulently in order to either gain access to information, or to commit malicious acts. Both of these are undesirable considering the fragile nature of an object heap, where a single modification could cause gigabytes of information to be lost.

One final factor which should be considered is the emulation of DAIS. Currently there are no plans to implement a DAIS processor in silicon. There is an initiative to implement an efficient binary-compatible processor based on off-the-shelf components. The use of tag to protect OIDs may complicate this approach, since the tag bits would have to be stored separately from each memory word if the underlying processor did not itself support tagging (which is likely considering the lack of tag-based processors).

5.11 DAIS-256

An alternative to tag protection for short OIDs is to use the whole 256 bit PID for local object references. Under this scheme the OIDs and PIDs are identical. Provided that the OIDs are allocated such that, given any valid OID, the probability of making a change to the OID such that the OID references a different object is so small as to be insignificant. For example, consider that the 72 bit checkcode for

a OID, proposed in section 5.9, is calculated using a good random number generator. If all the other bits contained within the OID are fully allocated (unlikely), then the probability of gaining access to an object without first being its checkcode is equivalent to finding the correct checkcode. Even if your were able to try a million variations per second, then it would still take an average of 4×10^{15} years to identify. If the operating system is designed such that any locally executing program which uses an invalid OID is immediately killed, and that any networked machine which requests an invalid OID is ignored for, say, 5 seconds, then even a million machines could not (probabilistically) succeed to break a checkcode before the the death of our planet. This level of security is undoubtably suitable for all perceivable needs.

It is attractive to include a level of protection against network snooping. In DAIS, this could be achieved in the following manner:

1. A networked node requests access to a particular object by sending the OID of the object in question to the node which possesses the required data. However, the checkcode part of the OID is not sent, instead being blanked out.

2. The node possessing the required object data sends the data to the requesting node. The data is encrypted before transmission, using the locally known OID's checkcode field as a key.

3. The encrypted data is received and then decrypted by using local version of the OIDs checkcode field. This data also contains some redundant information, allowing the requesting node to confirm that the data has been decrypted correctly.

In this way, OID checkcodes would never be transmitted un-encrypted over the network. This would protect data against unauthorized reading and (providing that the algorithm to give write access to an object relies on correct decryption of the object data) writing.

This last provision can be met while solving another security issue; consider a network snooper which detects an object request. The request does not contain the checkcode, and thus it is improbable that the snooper could request the object itself and then decode the information. However, the snooper, in requesting the data, could disrupt other genuine users of the data, by preventing them from accessing the information. In a network management system where objects requested can have read or read/write locking, the snooper could read/write an object and therefore prevent all other users from ever accessing the object's data.

A simple solution to the malicious locking problem is, when the object is requested, the request message contains the OID (minus the checkcode), the requesting node, and a set of fields relevant to the request (such as the type of lock required, offset into the object, *etc.*), including a duplicate of the requesting node's address. This set of fields is encrypted using the checkcode of the requested object before transmission. If the node receiving the request finds that the requesting node address in the unencrypted part of the request does not match the encrypted version after decrypting with the object's checkcode, then that request is ignored (and all valid messages from that node are delayed for a time, say 5 seconds, for the same reason as the 5 second delay used if an invalid OID is used). This makes

malicious requests for objects, to which the checkcode is not known locally, highly unlikely to succeed within the lifetime of our proposed implementation.

5.11.1 Virtual Mapping of Objects

Since each OID is a virtual address to its object descriptor, checking whether any given OID is valid involves using the OID to look up the descriptor. If the virtual address of the descriptor was used in its entirety, then each virtual page would probabilistically hold only one descriptor (due to the addition of the checkcode in the virtual address). Instead, the checkcode field of the OID is ignored. The remainder of the OID is used as the virtual address of the descriptor. The descriptor contains the full OID (including the checkcode) for the object, and this is checked against the OID specified. A match indicates that the OID was valid for use with that object descriptor. If there was no match, or the virtual address used to find the descriptor was not mapped in memory, then the specified OID is considered invalid.

One problem with this approach is that both the descriptors and the object data itself lie in the same virtual memory. Thus if large objects are created containing an abundance of zeros, then successfully guessing the virtual address of part of the object data could result in access to the whole of memory. This virtual address could be stored as an OID, with the checkcode field set to zero. When the OID is dereferenced, the virtual address will select an area of the large zero-filled object as the OIDs descriptor. By manipulating the zero-filled object, the OID's descriptor could be used to access any part of virtual memory. This would pose a severe security problem as so must be prevented.

It was indicated earlier that a benefit of having separate object descriptors and object data parts was that, by holding these two entities in different memory areas, fragmentation of object descriptors was avoided. If these two memory areas were defined such that an object reference could only involve the area containing the object descriptors (and not the data), then the security hole indicated above could be closed. In this way, the only part of virtual memory which can be successfully selected as an object descriptor are object descriptors.

Each page in the descriptor area of memory can contain a number of descriptors. If a descriptor is in use the OID contained within it will have a valid OID. Descriptors not in use have an OID which selects an area of memory not used for object descriptors, and thus is impossible to access by an OID dereference. We call these memory areas *partitions*. For expansion purposes, we have space for 256 partitions in the OIDs, but only two are currently used (descriptors and data). Figure 5.6 shows the layout of the 256 bit OID.

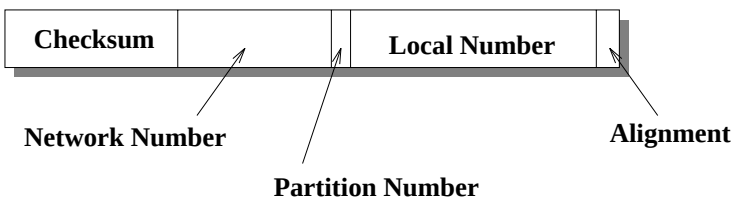


Figure 5.6: Field Description of Long OIDs

5.11.2 Performance Effects of Long OIDs

The use of DAIS-256, with its 256 bit OIDs, suggests a performance degradation in comparison to DAIS-64. However, it is not really suggested that DAIS-256 should ever be implemented. What was intended was to describe a processor which not only avoided the use of tag bits, but also avoided the complexity of using two (or more, if the object network was constructed hierarchically) different object identification schemes in later discussions on the use of DAIS in this thesis. In addition, chapter 12 uses a DOLPHIN software simulator, which included a simulation of the DAIS processor. It was felt that the complexity of using aliased OIDs in that system would unnecessarily complicate the design (*e.g.* where do you store the tag bit of DAIS-64 in a simulator running on a machine with a 32 bit address bus). For this reason, DAIS-256 is used as the processor in DOLPHIN for the purposes of discussion, although a processor utilizing an aliased OID scheme would be desirable for a final version of DOLPHIN.

5.12 Summary

In this section, two object-based processors were presented; DAIS-64 and DAIS-256. DAIS-64, using a single tag bit to protect OIDs, needs OIDs of only 64 bits in size. This makes OIDs cheap to use in comparison to the 256 bit OIDs suggested for networking. It also complicates object management, since a mapping must be maintained between the 64- and 256-bit OIDs to allow objects to be shared on the network. Tag bits are traditionally frowned on by the computer architecture community, and as a result few bus standards support them.

In comparison, DAIS-256 has no overheads in converting object data from local to networked identifiers, and this processor uses the full network OID as the format for local memory object references. However, the cost of implementing such a processor is significant with today's silicon fabrication technologies. In this thesis, DAIS-256 is used in the software simulation of the DOLPHIN environment, where silicon cost is irrelevant. In a final version of DOLPHIN, a processor with a more conservative OID size would be desirable (*e.g.* DAIS-64 with PID/OID aliasing). In reality, the remainder of the DOLPHIN environment is largely independent of the OID size (excluding any necessary alias management).

Independent of whether DAIS-64 or -256 is used, its caching structure incorporates features allowing the processor to run at around the same speed as other non object-based processors. DAIS achieves efficiency by providing only the minimum of support for objects. Provided that DAIS possesses only load and store instruction for accessing objects (in a similar way to RISC processors restricting memory access to only load and store instructions) then only these instructions need be concerned with object references, with the vast majority of other instructions unaffected by the use of object-based addressing.

For those interested in the overhead per object of using DAIS object descriptors, this can be calculated by adding the OID size, virtual address size, and 8 bytes (4 for the object length and 4 for assorted object status flags). For addressing purposes, it would probably be best to round this up to the nearest power of two, making the overhead in DAIS-64 to be 32 bytes per object. Later in this thesis, the method of managing free space in the store is presented, and this adds an additional 8 bytes for

objects with a size less than one virtual page, or nothing for objects greater than a page.

A cache structure based on objects and offsets rather than on virtual addresses allows object data to be accessed without the need for an address translation. Such a scheme also allows bounds checking to be achieved simply by examination of validity bits, removing the need for arithmetic comparison. A bit in the status word of an object allows for locking at the object level. This is useful for small objects. For large objects, page-based locking is more desirable, provided through the virtual memory manager. DAIS combines both these types and holds the read/write information in the data cache, providing an efficient mechanism for deciding on the validity of writes.

Analysis has been undertaken on the effects of choosing different cache sizes. This was done by dynamically tracing binaries for four different benchmarks on a non object-oriented architecture. Such an analysis is sufficient to provide performance evaluations for instruction and data caches, but not for the secondary object cache. Therefore, the sole conservative assumption was made that a 128 entry object cache have a 90% hit rate. However we believe that the real figure will be higher that this. The results of the analysis showed that a non-superscaler version of DAIS-64 with external cache takes 1.34 cycles to execute each instruction. This result is for a system with 8K of instruction cache, 4K of data cache and 2K of object cache. The object cache justifies its existence since a system with 8K of both instruction and data cache but no object cache has poorer performance.

Chapter 6

DAIS: A Multiple-Stack Register File

The use of registers has grown considerably since the accumulators of von Neumann's 1945 machine. Index registers appeared in 1951 [Kilburn, 1989] as part of the Manchester University Digital Computing Machine, followed in 1956 by general-purpose registers (within the Pegasus computer from Ferranti) [Seiwiorek *et al.*, 1982][page 10]. General-purpose registers are used to hold commonly accessed data, such as local variables, pointers, parameters, and return values. They cannot however, hold heap-based variables or other aliased data [Hennessy and Patterson, 1990][page 116].

Register usage plays an important part in the DAIS processor. In the previous chapter it was argued that, in a load/store architecture, only the load and store operations need understand the requirements of object-based addressing. In this way, a standard RISC core could be used to perform the majority of remaining CPU functions (add, subtract, multiply, *etc.*). In load/store designs, it is important to have enough user-accessible general-purpose registers to avoid unnecessary thrashing of data between registers and memory.

One problem with the use of general-purpose registers is in the overhead incurred over subroutine calls, where register contents must be saved to memory and restored on return. In [Hennessy and Patterson, 1990][page 450], Hennessy and Patterson show that this overhead equates to between 5% and 40% of all data memory references. The common solution is to use many on-chip registers.

Large register files are managed either by software or hardware. In architectures where all general-purpose registers are viewed as a single register file, software techniques [Richardson and Ganapathi, 1989, Wall, 1987] use global program knowledge to attempt to maintain values in registers over subroutine calls. To gain this global knowledge, register allocation is carried out at link time.

Hardware management strategies centre around *register windows*. Here, the register file is split into several banks, with a bank allocated on each call, and deallocated on return. The on-chip banks are constructed as a circular buffer: when a bank is requested that would result in a previously allocated bank being overwritten, the information contained therein is saved to memory (*window overflow*). On returning to a previously saved register bank, that bank is loaded from memory (*window underflow*).

Software techniques for maintaining values in registers have the advantage that the hardware is kept simple. However:

- Linking for a windowed register file is faster, and dynamic linking is easier to support.
- In the software solution, more directly addressable registers means more instruction bits are required to identify operands.
- Adding registers in a windowed architecture is transparent to the instruction set (and the user), while adding to a non-windowed system is not.

It should be stated that register windows cannot readily replace *all* processor registers, since globally accessible registers will still be required (*e.g.* program counter, user stack pointer, window overflow stack pointer, *etc.*). Although windowing floating-point registers is possible, current architectures typically leave these registers global.

6.1 Common Register Windowing Schemes

Register windowing can be split into two general sub-classes: fixed- and variable-sized. In a fixed-sized register windowing scheme, the number of registers per bank is defined by the hardware designer, whereas in a variable-sized scheme, the bank's size is specified by software at allocation time. These two systems are discussed in the following sections.

6.1.1 Fixed-Sized Register Windows

Fixed-sized register windows are used by both the SPARC chip-set [Glass, 1991] and the RISC II [Patterson and Séquin, 1982]. For these processors, the active window (*i.e.* currently accessible block of registers) is split into three parts: *in*, *local*, and *out*, with each holding eight registers. The *local* part contains registers accessible only while that window is active, *out* holds parameters to be passed to subroutines, and *in* holds the current subroutine's parameters as supplied by the parent. Whenever a new window is created, the *out* registers of the previous window become the *in* registers of the new window. This mapping is undone when the new window is deallocated.

Three fixed-sized windows are shown in figure 6.1. Each column represents the parts accessible from any one window. Parts lying on the same row are directly mapped onto one another. For example, the *out* part of window 2 is mapped directly onto the *in* part of window 3. The underlying register file is shown on the right.

Increasing the register file size increases internal bus capacitance. Provided that number of windows is kept small, this does not appear to affect processor cycle time [Hennessy and Patterson, 1990][page 454]. However, with many on-chip banks, cycle time will certainly be affected, suggesting an upper limit on design scalability. Fixed-sized windows offer no flexibility in the number of parameters passed or locals declared. If the number of parameters exceed the size of the *in* register part, then the remaining parameters must be held in memory. Alternatively, if some registers within a bank are unused within a subroutine, then window overflow/underflow will involve redundant memory transfers.

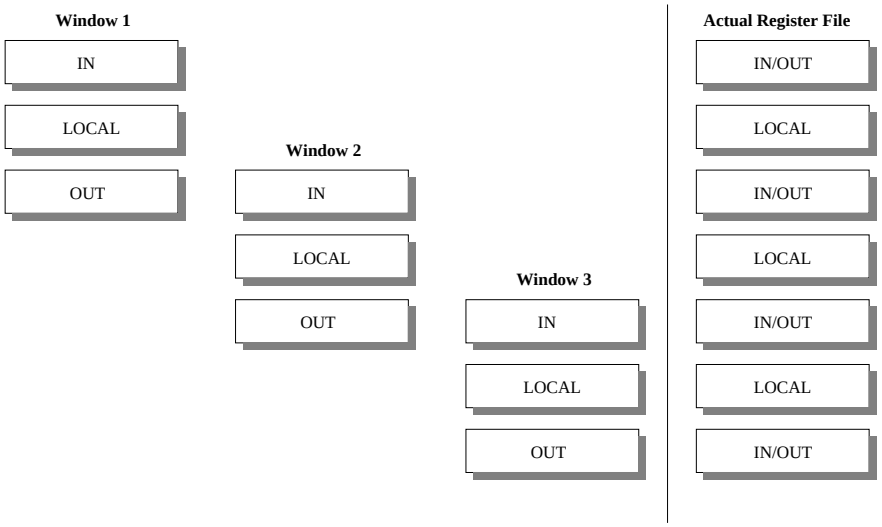


Figure 6.1: Three Fixed-Sized Register Window Banks

6.1.2 Variable Register Windows

An organization supporting variable-sized register banks is shown in figure 6.2. Here, a global register stores the current window position. Its value is added to every register reference, and then passed to a decoder, selecting the desired register.

The only instruction used in controlling the windows is a *shift*. On a subroutine call, the parent shifts the current window position to select the first parameter to be passed (*i.e.* a position after the parent's local variables). A negative shift undoes this on return from the subroutine. Once called, a subroutine can access registers from the current window pointer onwards. The Am29000 [Adv, 1987] provides support for variable-sized windows in a similar way to that depicted.

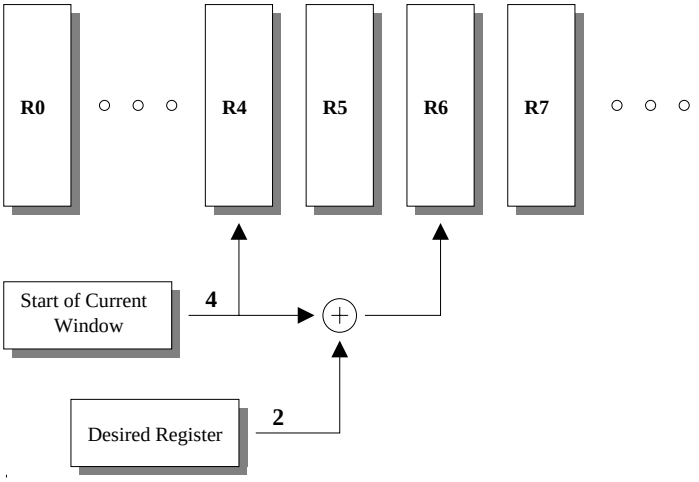


Figure 6.2: Organization of Variable-Sized Register Windows

Although this scheme supports variable-sized windows, there is an overhead of an addition on

each register access. The problem of scalability identified with fixed-sized windows remains unsolved, but the number of memory accesses due to window underflow/overflow is reduced when compared to fixed-sized register windows.

6.1.3 Design Goals for a Better Windowing Mechanism

Each of the existing register management systems described demonstrates advantages over the other. For example, variable-sized windows offer flexibility at the cost of performance, while the fixed scheme provides better performance at the expense of flexibility.

DAIS contains a new windowing model, *multi-stacks*. Its aim is to improve over existing register management schemes, such that:

- The new design contains the flexibility to allocate the exact number of registers required.
- Currently accessible registers (those contained within the active window) should be the registers closest to the ALU, minimizing signal propagation times.
- The design should be without register-access overhead, such as that incurred by the addition in variable-sized windows.
- The return address for a subroutine should be stored on-chip to support fast call-return cycles.
- The design should be scalable, such that adding more on-chip space for register storage does not adversely affect access times or logical complexity.
- The memory accesses required to manage register overflow and underflow should be controlled so as to minimize the number of processor stalls incurred by this activity. The initial aim was to reduce such stalling to less than 5% of that incurred by the SPARC.

An initial register model, *Shifting Register Windows*, was developed to reach these aims [Russell and Shaw, 1993b] [Russell and Shaw, 1993c]. This was based on mapping the entire register file onto a single variable-length stack. Allocating a register involved shifting all the elements within the stack to the right, while the new register was pushed on from the left. Deallocation was achieved by popping from the left, and shifting the remaining elements leftwards. Unfortunately, the propagation time for data to move from one stack element to another proved to be the limiting factor in register allocation/deallocation rates. Multi-stacks overcomes this by using multiple stacks, effectively parallelizing allocation requests over a number of stacks.

In DAIS, the general-purpose registers are mapped onto independent stacks, such that each stack head appears as a register. Instructions operating on these registers have the option of `popping' off the top element of a source stack, and to either `push' on data to the destination stack or replace the data contained in the destination stack's top element. The program counter can also also be implemented on a stack, supporting fast subroutine calls and returns. Using such an approach, a number of advantages are realized:

- Register allocation is performed simply by writing data to a stack, removing the overhead of register-allocation instructions.
- Most registers allocated could be deallocated for free (without instruction overhead), by making the last simple read of a register a `pop'.
- Each register can be viewed as both a single storage element and as the head of a stack. This allows the use both traditional register-based and stack-based programming methodologies to be used interchangeably as and when required.
- Register `dribbling' (*i.e.* the ability to transfer data between the register buffers and memory during idle memory-bus cycles) can be achieved simply with a stack-based buffer.

6.2 DAIS' Register Stacks

A simplified [1] block diagram of multi-stacks is shown in figure 6.3. This comprises a set of stacks, the heads being the visible register file. At the bottom of the stacks is a spill/refill arbiter handling requests from the stacks to transfer data elements between the stacks and memory.

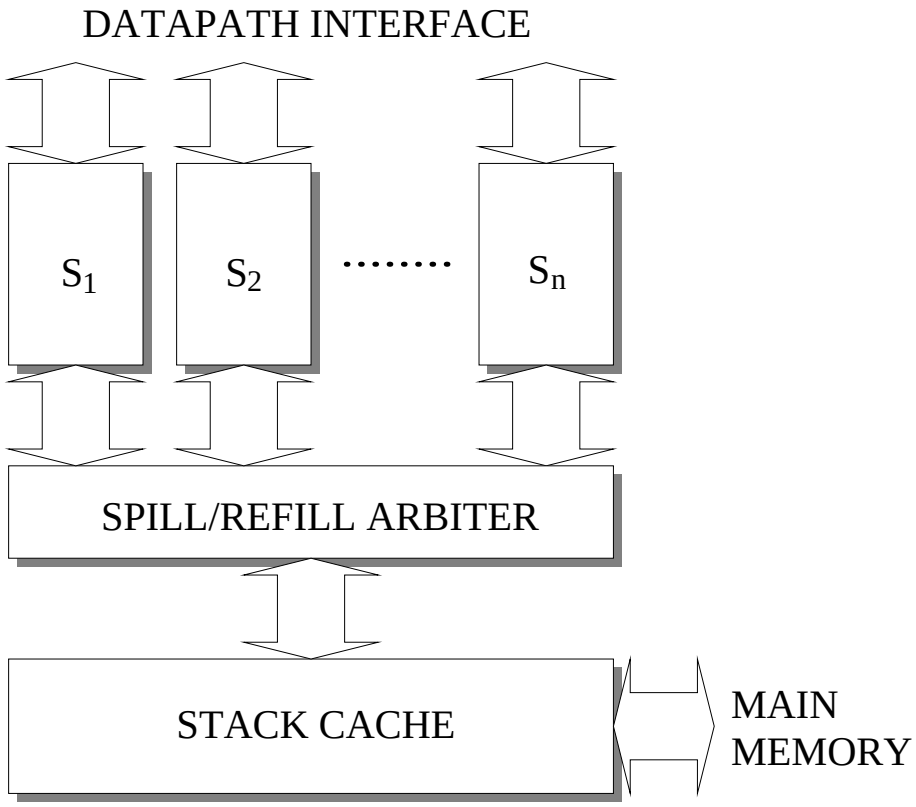


Figure 6.3: Block Diagram of the Stack Organization

[1] The datapaths shown in figure 6.3 are all shown as unique point-to-point connections. In reality, many of these can be shared between registers, and this is demonstrated later (see figure 6.4).

To reduce the latency of stack spills and refills, a cache is present between the arbiter and main memory. This stack cache significantly reduces the number of accesses to main memory, increasing performance. Moreover, since cache is naturally denser than the equivalent amount of storage in registers, it is efficient in area.

6.2.1 Stack Implementation

Each register stack is held mostly in memory, with only the topmost elements of each held on-chip for fast access. Consider the possible workings of push and pop operations within this organization. If a data element is pushed on, it must reside in the hardware array, and some other data element is moved out to memory. Similarly, on a pop, a data element is moved in from memory. This type of operation would result in the memory being heavily used even during minor variations in stack depth.

A better solution is achieved if the movement of data between memory and stacks is decoupled from the push and pop operations. Assume each hardware stack can hold between 0 and n elements.[2] Also assume spill and refill limits s and r, where $s \geq r$. When the buffer contains more than s elements, it would be desirable to spill an element to memory. Likewise, when it contains less than r elements, some should be drawn from memory. A `dead area' of $s - r + 1$ elements exists where no spilling or refilling is required. This reduces memory traffic generated by thrashing. The processor stalls when push operations are attempted while the selected register buffer is full. The same situation occurs when attempting to pop from a stack with an empty buffer. When such a stall occurs, the appropriate memory transfers are performed to either make space in an on-chip stack or to pull data from memory. The push/pop causing the stall can then go ahead, and the processor continues as normal. A full description of the hardware for the stack buffers used is given in [Russell and Shaw, 1993b].

6.2.2 Cache Design

The stack cache lying between the register stack arbiter and main memory is a fully associative, write back cache, with a random replacement policy and block size of one word. Pieces of information stored in each cache block are data, tag, dirty bits, and the in-use bit. To begin, all cache blocks are marked unused. A cache miss results when no in-use blocks have a tag matching the incoming address. When this occurs, a cache block is chosen at random. If the block contains any words which are dirty, then those words are written into main memory. The new data is then either read from memory into the free block (on a read), or written into the free block (on a write). The dirty bits are then cleared, and the in-use bit set. On a cache hit, the data is produced. The dirty bit for that word is cleared on a read and set on a write. This behaviour on a read may seem odd. However, because of the nature of stacks, a memory location will never be read twice without a write in between. A cached memory word can thus be marked clean after being read.

[2] These figures do *not* include the visible register at the head, only the remainder of the stack (the body).

6.2.3 Memory Management for Multiple Stacks

With more than one stack in memory, managing the stacks of each process is more complex that the single stack found in more traditional processor designs. However, such management is required only on a context switch and during register growth.

Context switching involves simply exchanging the stack pointers for a set belonging to the next scheduled process. Additionally, if the stack cache is virtually addressed, it may have to be flushed. However, this is not be the case if there is a virtual to physical translation step inserted between the stack arbiter and the stack cache.

Stack growth can be dealt with via two different approaches. The simplest approach is for stack spill addresses to be set to word indices $a \ldots a + n - 1$ for n stacks. Then, when any stack is spilled or refilled, the spill address is moved by n memory words. This has the disadvantage that in the worst case, only $1/n$ of the stack memory is being used. This may not be a problem for small numbers of stacks.

The other approach is more complex and relies on support from the virtual memory manager. Each stack is allocated their own p pages for growth, which should be ample except in exceptional cases. For example, 256 KBytes would be an excessive amount. Below this area, a page is unmapped. A stack may grow downwards (one word at a time) so much that it attempts to write to the unmapped page. The memory management unit reports the fault, and the operating system then maps the page in, making sure the page below is unmapped. Whenever the stack grows onto a new page, and the page below is mapped, then the stack is moved to a new area. This is done by remapping all of its pages to a new position in memory, changing the stack spill/refill pointer to point to the new position, and flushing the stack cache. Since there can be no references to data on the stacks, this move operation is safe.

6.2.4 Stack Layout

The organization of the register stacks is shown in figure 6.4.[3] To support fast reading and writing of stacks, the heads of all stacks are located together, with the bodies in another area. Since read and write busses need only span the length of the heads, access times can be kept low. The push/pop buses are connected to the first element of each stack body. Owing to the layout, their length is significantly greater than that of read/write buses. This implies that registers can accessed quickly because of the extremely small bus length, with a larger times being devoted to pushing or popping. This equates to one half-cycle and two half-cycles respectively for the pipeline described in chapter 7.

Any general-purpose registers not requiring a stack body (for instance a traditional user stack pointer) can be placed with the heads of the stacks, but with no associated stack body.

Examining figure 6.4, it can be seen that there are two unidirectional pop buses and one bidirectional push/pop bus. Normally, the push/pop bus is used as a push bus when the destination stack is being pushed. However, in offset addressed store instructions, there is no destination stack, but three sources (*e.g.* STORE R0,[R1+R2]). In this instance, all three buses are available to pop the source stacks.

[3] Although in the diagram the push/pop buses are bent (for reasons of space), in reality this would not be the case.

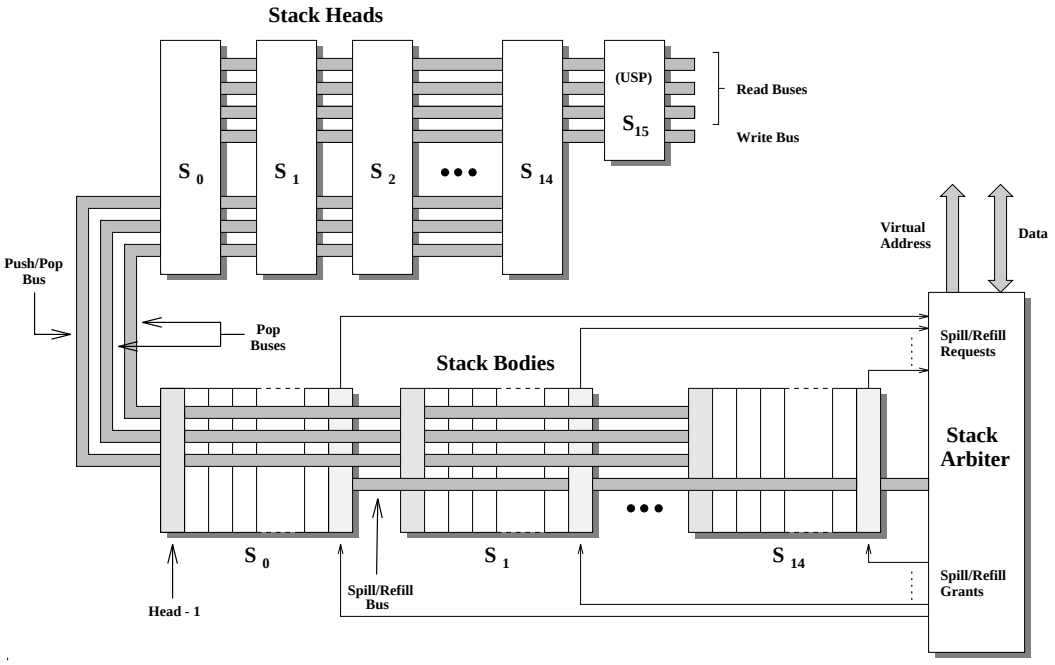


Figure 6.4: Layout of the Register Stacks.

A spill/refill bus connects to the last element of every stack body. This bus can carry a data element to spill to memory or data coming from memory to refill a stack. Depending on their internal states (how full they are), the stack bodies will compete to either spill or refill data. Their requests are handled by an arbiter that instructs at most one stack to spill or refill. Internal to the arbiter are virtual addresses (one for each stack) that dictate where data will be placed when a stack spills. The virtual addresses grow downwards when stacks are spilled, and upwards when refilled. Special management instructions are used to read and write these virtual address registers.

The program counter is based on a stack similar to the arrangement shown in the figure, but requires only one push/pop bus, a read bus, and a write bus.

6.3 Analysis

In Shifting Register Windows, much of the performance gain demonstrated originated from the use of asynchronous memory transfers. Here, stack elements could be transferred between the stack and main memory, while the processor continued to execute instructions from within the bounds of its cache(s). Provided that a transfer was completed before the processor needed access to main memory, the transfer can be described as *free* (*i.e.* not causing a processor stall). If the processor attempted to access memory during an asynchronous transfer, then the processor was forced to wait.

In DAIS, the multiple stacks can transfer information to and from the stack cache independently from other processor activities. However, if the transfer would involve forcing the cache to make a main-memory access, two implementation possibilities exist. Either the cache-memory transfers

occur using the same asynchronous transfer mechanism as that used in Shifting Register Windows (*i.e.* the transfer is started as soon as the memory bus becomes idle), or the transfer is avoided until demanded by a stack request. In the former, which is termed "Asynchronous Transfers Enabled", there is the possibility of interfering with instruction and data cache transfers, which would cause processor stalls. In the latter case, termed "Asynchronous Transfers Disabled", hysteresis-initiated transfers between the cache and main memory are ignored, but stack-actions which demand that the data transfer be performed cause the cache to transfer the information immediately. This means that the cache is ignoring opportunities to use the main memory bus for free, and that all transfers made cause a processor stall, but it also means that no processor stalls can be caused by the use of hysteresis.

It is of interest to see whether allowing the stack cache to asynchronously access main memory in the same way as Shifting Register Windows will offer advantages. It is also interesting to investigate the trade-offs between stack depth, stack cache size, block size, and stack hysteresis values. To this end, a register file simulator was constructed. This was built around SHADOW[SUN, 1992], a SPARC-based binary analysis library. This library allowed our simulator to monitor the instruction stream of an executing application. The simulator converts SPARC register management instructions to fit the DAIS' register model. Four benchmarks were used in the analysis; TEX, DeTEX, fig2dev, and zoo.

Each stack was constructed so that it produced a virtual address of where it could find the stack element last saved by that stack. When a stack decided to transfer data from itself into memory, the virtual address was incremented to point to a free virtual memory location, and the stack element saved to that address. Loading from that address caused the virtual address to decrement to the stack element saved just prior (chronologically) to the element loaded. The stack cache intercepted the stack/memory transfers, in an attempt to reduce external memory traffic. The stack cache used the virtual address of where to store the stack elements as a tag for identifying the data, using a fully associative mapping scheme. The stack used a random replacement policy. A direct mapping scheme for stack management was used initially (as it is cheaper in hardware terms), but its performance was significantly inferior to that found in Shifting Register Windows. A fully-associative, least-recently-used replacement algorithm results in only slight performance improvement over random replacement, but incurs a significant hardware overhead in its implementation.

In all simulations, the time to transfer data between a stack and the stack cache was defined to be 1 clock cycle, and transfers between the stack cache and external memory was defined to be 2 clock cycles (*i.e.* an external cache hit). The simulator used code and physical characteristics (*e.g.* external memory access rates) extracted from a software simulation system for SPARC machines, which was supplied by Sun Microsystems.

6.3.1 The Benefit of Asynchronous Transfers

The DAIS simulator was set up to model multiple stacks for both stack depth 2 and 3. These depths exclude the register at the head of each stack. For each depth, the simulation was performed for both asynchronous transfers enabled and disabled. Stack depth 2 has the low and high hysteresis value set to 1 (start trying to perform transfers when there are either less or more than 1 element in the stack).

Hysteresis for stack depth 3 was set to 1 (low) and 2 (high), thus spilling was attempted when the stack contained 3 elements, and refilling attempted when the stack was empty.

For TEX (figure 6.5), even at maximum stack depth and cache size, memory transfers were always required. Asynchronous memory transfers produced inferior performance to simulations without such transfers. The difference between stack depth of 2 and 3 was small, and was generally less than the gain achieved by using the resources required to increase the stack depth within the stack cache instead.

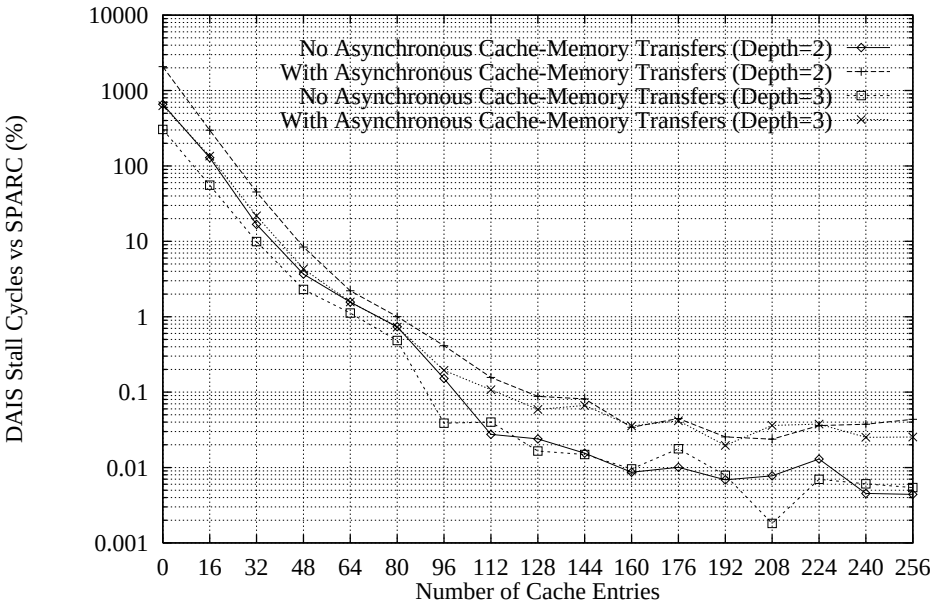


Figure 6.5: Analysis of depth 2 and 3 in TEX

The remaining three benchmarks all exhibited similar performance curves (DeTEX is shown in figure 6.6), with stall cycles becoming constant (at between 1-10% of SPARC stall cycles) for the cases with asynchronous transfers activated. Without asynchronous transfers, stall cycles reached zero between cache sizes of 48-64 entries. Notice that 0% is plotted at $-\infty$ on the logarithmic scale, and so any line drawn to 0% appears to drop vertically.

The graphs suggest that there is no justification in allowing asynchronous transfers between main memory and the stack cache. Not only is there no performance improvement in using asynchronous cycles, management of such cycles increases design complexity. Even with a cache size of zero (spills and refills access memory directly), asynchronous transfers were the poorest performers. The results also suggest that stack depth is not a critical parameter.

6.3.2 Stack Depth

The analysis of asynchronous transfers resulted in the suggestion that stack depth was not crucial for performance. Reducing the depth has the advantage of reducing the amount of silicon area needed by

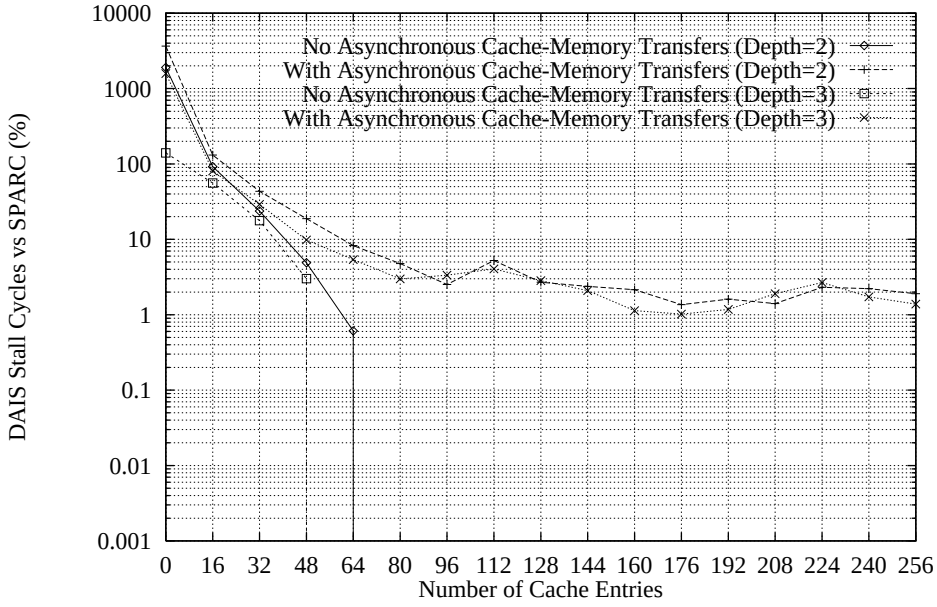


Figure 6.6: Analysis of depth 2 and 3 in DeTEX

the register file. It was therefore decided to simulate the register file using a stack depth of 1 element.

One difficulty with a stack depth of 1 is that there is no clear-cut hysteresis value to choose. The value should be selected to allow early transfers of stack elements before the stack receives a request which it cannot handle (*i.e.* a push on a completely full stack, or a pull on a completely empty one). This is not possible with a stack depth of 1, which can only have hysteresis values of $\{0,0\}$, $\{0,1\}$, and $\{1,1\}$. With $\{0,0\}$, the stack will attempt to spill if it contains 1 element, but does nothing when empty. Values $\{0,1\}$ mean that the stack never spills or refills unless forced to do so by a push or pop request. Finally, values $\{1,1\}$ result in the stack attempting to always remain full.

Figure 6.7 contains the results of a simulation of TEX, with the stack depth set to 1 and a hysteresis of $\{0,1\}$,$\{0,0\}$, and $\{1,1\}$. Fig2dev, zoo, and DeTEX were also simulated, with all three producing similar results (the DeTEX data is containing in figure 6.8). In all simulations, the performance of the stacks using the values $\{0,1\}$ and $\{1,1\}$ were almost identical. However, the $\{0,0\}$ simulation performed measurably better. In the TEX case, the performances are equivalent until around a cache size of 96 elements, where $\{0,0\}$ begins to perform around an order of magnitude better. For the other benchmarks, the performances remain similar until approximately size 48, where $\{0,0\}$ almost immediately becomes zero (while the other traces stabilize at approximately 1%). This is largely caused by the random replacement algorithm; early spilling helps to maintain the cache in an near-optimum state, where little replacement is needed. Note that, even for cache sizes smaller than 96 elements, the performance measured using a stack depth of 1 is not radically different from the results produced using depths of 2 and 3.

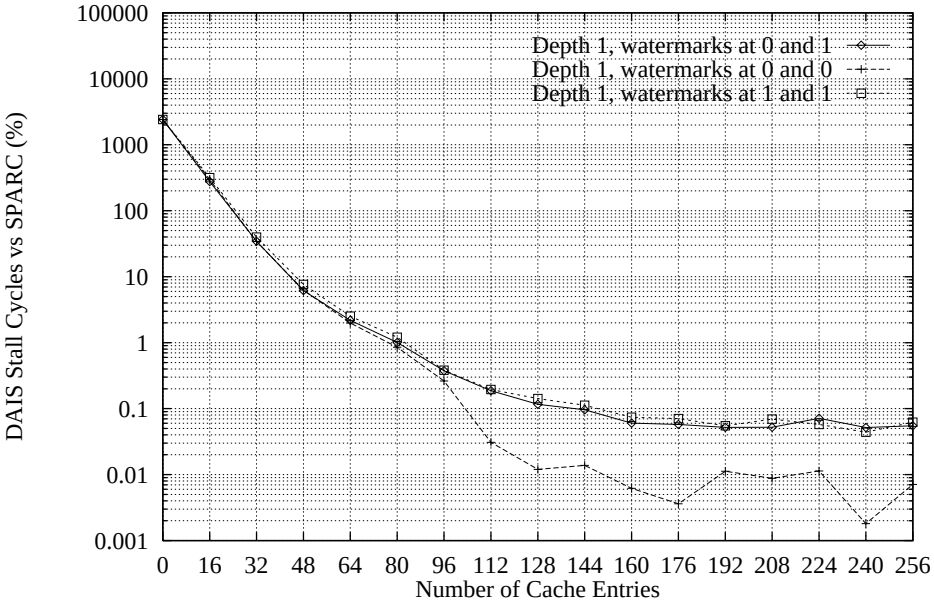


Figure 6.7: Stack Depth 1 in TEX (no asynchronous transfers)

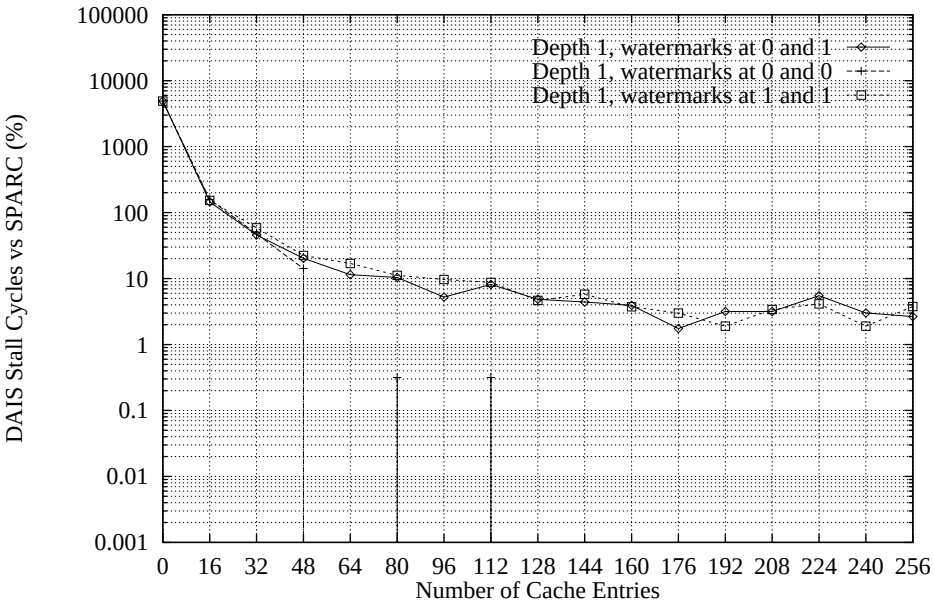


Figure 6.8: Stack Depth 1 in DeTEX (no asynchronous transfers)

6.3.3 Stack-Cache Construction

The construction of the stack-cache is a trade-off between silicon area, cache capacity, and cache efficiency. The cache contains not only the stack elements written to the cache, but also the virtual addresses of the stack elements in question. If only one element was stored in each cache line, then the cache capacity would be split approximately 50/50 between stack element storage and cache tags. Increasing the width of each cache line (the block size) has the effect of reducing the percentage area dedicated to tag information. However, the effect on cache hit rates in doing this is unclear.

The simulations were re-run using both a stack depth of 1 (hysteresis {0,0}) and 2 (hysteresis {1,1}), using stack cache block sizes of 1, 2, and 4 stack elements. The TEX simulation was executed first, and the results are shown in figures 6.9 (depth 1) and 6.10 (depth 2). These indicate that there is negligible performance difference between using a block size of 1 or 2 elements. A Block size of 4 results in an increase in stall cycles of between 2 and 5 times. For a depth of 2, an annoying anomaly occurs at cache size 160, where the percentage is equal to zero. This is an effect of the interaction between the register usage of TEX, and the random replacement algorithm used in the cache. Note that for depth 1, only the trace for block size 1 remains non-zero once the cache size grows beyond 112 elements.

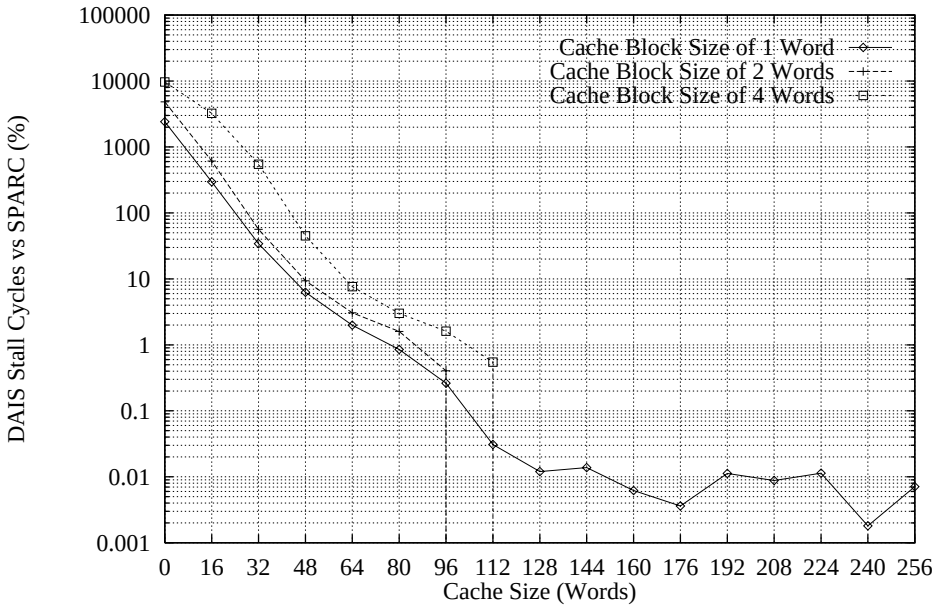


Figure 6.9: Stack Depth 1 With Varying Stack Cache Line Length TEX

The remaining benchmarks all exhibit a similar performance trend (the DeTEX results are shown in figure 6.11 and 6.12). Here, all results have reached 0% by a cache size of 64 elements. Up to this point, the block sizes of 1 and 2 words remain almost indistinguishable, with block size 4 returning results averaging at an order of magnitude worse. However, block size 4 is almost always the first trace

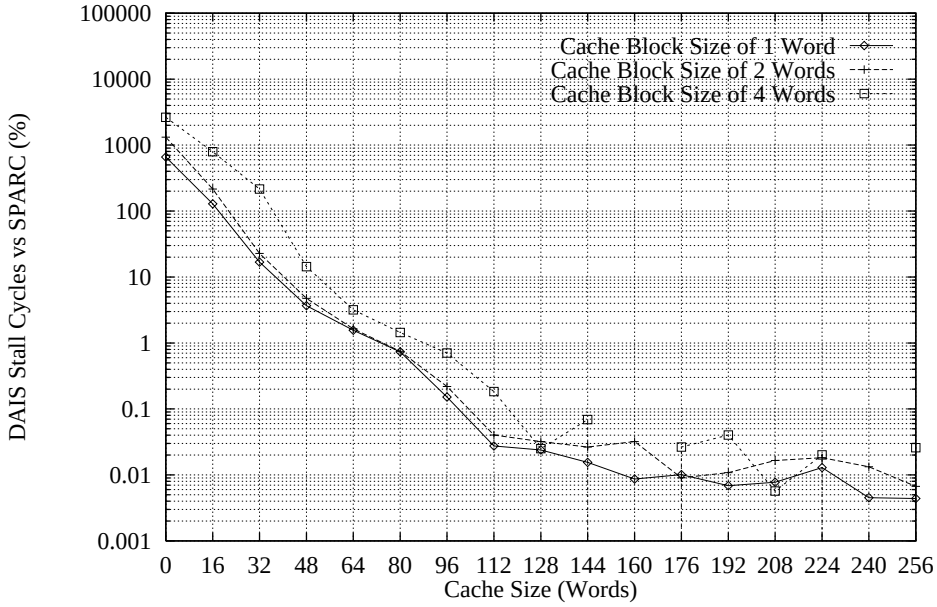


Figure 6.10: Stack Depth 2 With Varying Stack Cache Line Length (TEX)

to reach 0%.

6.3.4 Recommendations

From the simulation results presented above, some conclusions can be drawn with respect to the multiple-stack register file implementation. All the simulations indicate that a stack cache is needed to avoid excessive processor stall cycles. Data transfers between this cache and main memory should be performed only when both a stack is forced to spill (or refill) via the cache, and the cache cannot handle the request without memory transfers. Asynchronous memory cycles degrade overall performance (as well as increase complexity) and should be avoided.

The stack depth used in a multiple-stack system should either be set to 1 or two elements (depending on space considerations). Depth 1 generally performs as well as depth 2. Depths beyond 2 do not produce a significant performance change. Increasing the cache capacity beyond approximately 128 elements also has little affect on performance. Cache block size of either 1 or 2 has similar performance, and a cache width of 4 creates around 2-5 times the number of stall cycles.

To reach the initial aims of producing a register file causing less than 5% of the stall cycles produced by the SPARC`s windowing strategy can be realized using a stack depth of 1, a cache size of 64, and a block size of 2 elements. Interestingly, such an arrangement uses a similar amount of silicon to the SPARC implementation, while allowing the possibility of faster register accesses on the register file, due to the decreased bus capacitance.

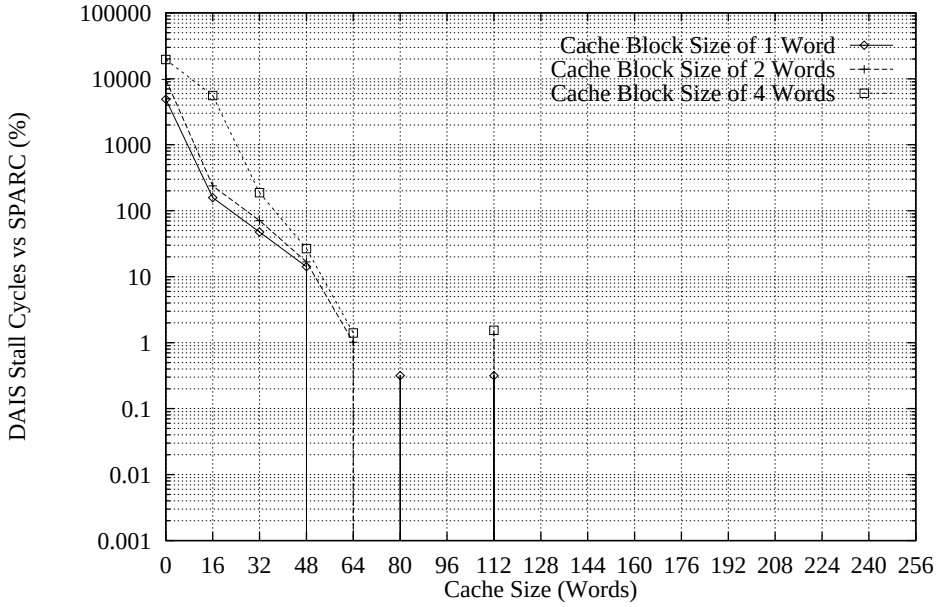


Figure 6.11: Stack Depth 1 With Varying Stack Cache Line Length (DeTEX)

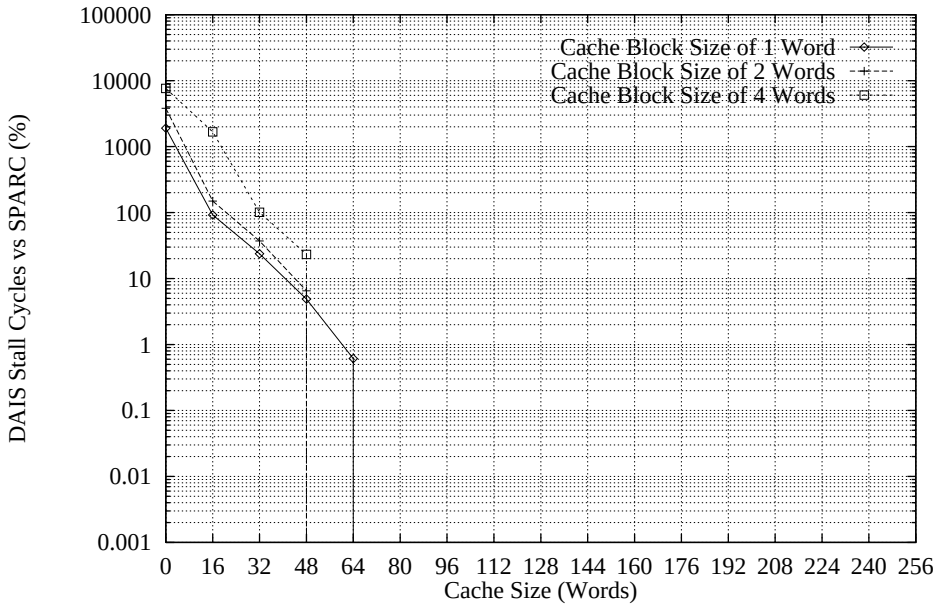


Figure 6.12: Stack Depth 2 With Varying Stack Cache Line Length (DeTEX)

6.4 Conclusions

DAIS's register file incurs under 5% of the SPARC window stalls, given approximately the same amount of silicon. Multi-stack performance can also be improved without degrading register access time, since the stack cache is not on the register read/write busses.

With registers remapped as stack heads, programmers are offered a new register interface, which results in interesting compiler optimization techniques. It also means that interrupts can be handled without a change of context, with local parameters simply mapped ahead of the current application's variables. Macro-instructions are also simpler to implement, where chunks of assembly code can be inserted into a program without significant knowledge of the surrounding register allocation scheme.

An interesting modification to the multiple-stack scheme could be to remove the register at the head of each stack, and instead access the stack buffers directly on every register access. The result of this would mean that stack transfers could be required on any register access, rather than just on register pushes and pops as at present. While this would increase the complexity of the register file, it is suggested that this could further decrease the stall cycles for the same amount of silicon. This should be investigated in future work into this area.

Note that the performance gain calculated in chapter 5 is independent the particular register strategy used. It is thus possible to implement an object-based processor without the register stacks (perhaps using a single register window) while retaining excellent object access performance, just as it is possible to implement a virtually-addressed processor with the multiple stack register file.

Chapter 7

The DAIS Instruction Pipeline

In this chapter the design of the DAIS pipeline is examined. This is to demonstrate that the effect of object caches and multiple-stack register files does not necessarily incur increased CPI (cycles per instruction) due to pipeline dependencies. It is also demonstrated that object-based addressing may actually reduce such dependencies. For simplicity, only single pipeline processor is considered, although many standard superscaler techniques could be used to provide support for superscaler implementations. A possible instruction set for DAIS is also discussed, suggesting how the functionality of the multiple register stacks could be utilized at the programming level.

The RISC philosophy in instruction set design produced a significant performance increase over CISC with respect to instruction fetches. In CISC, with its variable-sized instructions, the prefetch unit reads in a memory word; if the whole instruction was not contained within that word, further fetches were performed. The uncertainty in instruction length meant that pipelines were complex to implement. RISC, with fixed-sized instructions, required only simple pre-fetch units.

In DAIS, the ability to call an object as a subroutine, using absolute addressing, was identified as being a useful feature. By absolute addressing, it is implied that the OID of the subroutine is embedded within the instruction stream. However, the size of OIDs were deemed too large to allow this. Instead, subroutine handling must be handled slightly differently. Branches and subroutines which lie within the current object being executed are reached using a **sbra** or **scall** instruction, which specifies an offset relative to the current PC. Branches to other objects first involve loading the necessary OID into a register, and then using the register as a parameter to long-distance branch or subroutine instructions (*e.g.* **bra** or **call**).

DAIS' instructions are 32 bits long, and two instructions are stored in each instruction word. The instruction prefetch unit treats the memory as being 64 bits wide, thus it is independent of whether the DAIS-64 or -256 is being used.

The RISC-like policy used when designing DAIS has resulted in a load/store based architecture. This decision meant that support for object addressing affected only the load and store instructions,

leaving most other (*i.e.* register to register) instructions unmodified. Loads and stores of 1, 2, 4, and 8 bytes are supported. Memory accesses must be aligned in multiples of the indicated data size, *i.e.* load/stores of 1 byte can occur at any index value, while transfers involving 4 bytes (32 bits) can only occur when index is in the set $\{0, 4, 8, 12 \ldots\}$. Attempting to access data using an incorrect alignment causes a processor exception.

Programmers are free to modify OIDs in any way they wish. Naturally, DAIS-64 notes that an OID, whether the OID was in a register or in memory, has changed by clearing the tag bit corresponding to that OID. It is impossible to use an OID with a cleared tag bit to reference object data. DAIS-256, without the benefit of a tag bit, allows any data to be used in an object reference, relying instead on the sparseness of the OID allocation process and their corresponding checksums. In both systems, detected use of an illegal OID results in a processor exception. Loading or storing using an index which lies outside the length of an object will also cause an exception. Range infractions can be detected using the primary data cache's valid bits.

7.1 Register Organization

DAIS has 17 important registers. There are 15 general-purpose (GP) registers, a program counter (PC), and a user stack pointer (USP). Each GP register is arranged as a stack; the current value of the register is delivered by reading the top of the stack. The PC is also arranged as a stack; the current PC is the top of stack while return addresses are stored underneath. The USP is a single register managed in the normal way (by subtraction and addition), to store and restore data. In the following sections, the structure of the GP, PC, and USP registers will be described.

7.1.1 General-Purpose Register Stacks

Each GP register stack is 65 bits wide for DAIS-64 (including a tag bit), and 256 bits wide in DAIS-256. Each register can contain data or an OID.

7.1.2 Program Counter

The PC is made up of an OID pointing to the object currently being executed, and an index. The object being referred to could be a method, procedure, module, or an entire program. The index represents the current program position within the object. Calls and branches can cause transfer of control to a different object, or a new part of the current one. Range checking of code objects is performed automatically with the valid bits in the instruction cache.

During program execution, an instruction word is fetched from the instruction cache to a *prefetch buffer*. The prefetch buffer therefore contains two instructions. If, in DAIS-64, an instruction word is fetched which is not tag-bit clear, an exception is raised. The pipeline fetch extracts instructions individually in a serial fashion from the prefetch buffer, provided that none of the two instructions transfer control flow. For each extraction, the PC's index is incremented by 4. When the prefetch buffer is exhausted, a new instruction word is fetched to refill it. An exception occurs if the PC's index is not a multiple of 4.

7.1.3 User Stack Pointer

The USP, like the PC, is made up of an OID and an index. The OID is set to identify the stack object, and the index to a large number. The value of this number indicates how many bytes of stack space are available (an exception is raised if a USP load or store is issued when this index is negative).

The index reduces (the stack grows downwards) as data are allocated. When the USP is read by ALU instructions, only the index is visible. Allocating 40 bytes on the stack is achieved by subtracting 40 from the USP (and therefore subtracting 40 from the USP's index value). This can be deallocated by adding 40 to the USP.

When a load or store instruction involves the USP (*e.g.* LD USP[8],S2), the index given (here 8) is added to the USP's index to give an absolute stack index. The USP OID and absolute index are then used to access the object data in the normal way. Load and store instructions may only be supplied with positive indices. Hence, loading and storing stack memory can only be done upwards of the current position. Therefore, the USP must always point to the lowest indexed piece of valid data. The DAIS instruction set supports both constant- and register-based (*e.g.* LD USP[S1],S2) indexing of objects.

7.1.4 Special Instructions

Due to the extra width of the USP and PC registers (32 bits wider than the user registers), and the fact that the PC is not connected to the same bus as the user registers, special instructions are required to access them.

The PC is read with the GETPCO (get PC OID) and GETPCI (get PC index) instructions which move the PC's OID or index to a specified stack. Writing to the PC is done by branch and call instructions. These can change either the PC's OID, index, or both. In the latter case two source stacks are specified. The USP's index is read and written by normal instructions, however its OID is accessed by GETUSP and PUTUSP. These instructions move between the USP's OID and a specified stack.

7.2 Register Extraction

The instruction set promotes regular instruction fields (known as *fixed-field decoding* [Hennessy and Patterson, 1990, page 202]), allowing fast prediction of the registers needed to be read by a particular instruction, even while it is still being decoded. This helps to simplify the pipeline by reducing register fetch latency.

Register prediction is based on the information shown in figure 7.1. The prediction system always extracts three register fields from the instructions, regardless of whether they are actually registers or not. Incorrect prefetch of registers has no side effects, since the correct registers will be known at the beginning of the execute cycle.

For each register-stack identified in each instruction, there exists a pop bit. This bit indicates whether or not that stack is to be popped during the execution of the instruction. If a source stack has its pop bit set, then the stack is popped (and the top value lost) after its value is read. Normally, the result of an instruction is pushed onto the destination register-stack. However, if the destination stack's

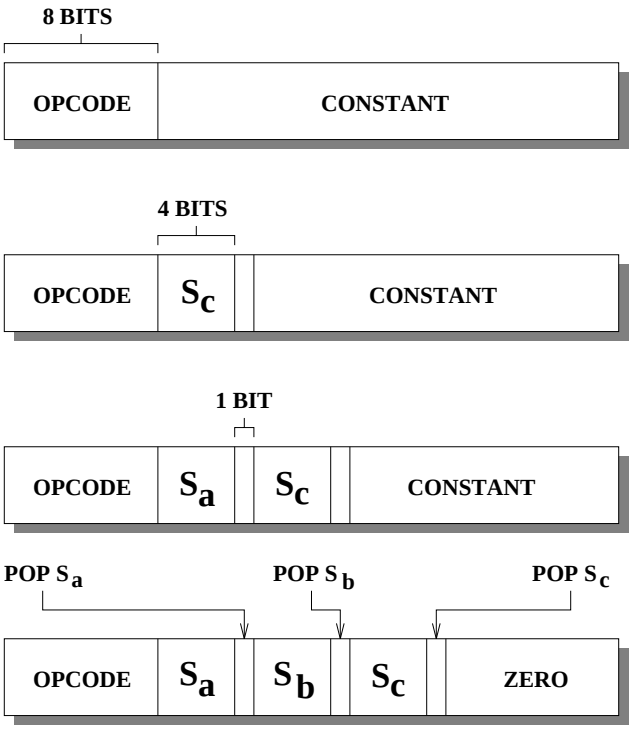


Figure 7.1: Instruction Operand Layout.

associated pop bit is set, an optimization is available; no popping or pushing of the stack is necessary, and the result is simply written over the top element. An exception is raised if the USP (which is not on a stack) is a source and its pop bit is set, or the USP is the destination and its pop bit is clear. Note that the state of the pop bits must be known for certain, unlike the optimistic fetching of the register contents. Thus in the decode stage the first half of the clock phase must be spent extracting the correct registers for subsequent popping (if any), which begins in the second clock phase in the decode stage. This is discussed in section 7.4.

The only non-orthogonal part of the instruction set is with instructions which have two source operands which both point to the same stack. If both operands are not marked to pop, then there is no inconsistency. However, if either are marked to pop, then it is unclear as to which elements of the stack the operands refer. To avoid uncertainty, only the second operand in this scenario can be popped (both operands are effectively set to the save value). Attempting to set the first operand's pop bit results in a processor exception.

By making the pop bits available over the entire instruction set, many opportunities can be found to perform deallocation of registers which are no longer needed. Where such deallocation is not possible, a POP instruction is also provided, which contains a bit-mask of the stacks to be `popped'. This bit-mask can also pop from the PC stack, thus supporting register deallocation and `return from subroutine' in a single instruction. There are no explicit register allocation instructions, since an instruction destination stack which is not popped first has its resultant data pushed onto the stack. Dynamic analysis of SPARC binaries shows that 1–2% of instructions executed were SAVE or RESTORE instructions (*i.e.*

the instructions which allocate and deallocate register windows).

7.3 Branching

Instruction branching is handled in a similar way to many RISC designs, using a TLB (*translation look-aside buffer*) keyed on current PC to obtain the next PC (known as a *branch-target buffer* [Hennessy and Patterson, 1990, pages 310-311]). Only PCs which hold a branch instruction are stored in the buffer. This helps to maintain single-cycle execution through branch prediction. However, the first time a particular branch is reached (or if the information calculated when the instruction was last executed has been overwritten in the TLB) it is impossible to predict that it is a branch instruction until it has been decoded. In a normal implementation of a PC TLB this would cause a pipeline stall. DAIS allows the user to avoid many unconditional branch stalls.

Since DAIS fetches are based on instruction words, a fetch of a instruction-word aligned instruction means that the next instruction has also been loaded. If the second instruction is an unconditional branch, this is detected within the first instruction's decode cycle, with the destination PC for the branch being calculated in parallel with the detection. The calculated PC is then used by the fetch stage in the next cycle, making the branch appear to take only a single cycle with no pipeline stall. This type of branch is termed a *fast branch* operation.

If an instruction word is loaded whose PC is found in the PC TLB, then the fast branch operation can be completely optimized out. For example, if an instruction is loaded, and the prefetch detects the current PC in the TLB, then the TLB is used to obtain the next normal instruction after the branch (missing the unconditional branch instruction completely). This optimization can only be done with a fast branch construct, and to allow the TLB to cache other subroutine calls and conditional branches (*i.e.* instructions which cannot be optimized out as they depend on current state), a bit in the TLB is used to signify whether the second normal instruction can be optimized away. The TLB can also be used to cache subroutine returns.

Branch instructions found in the first half of an instruction word and unconditional branches are not optimized away in this scheme, but must be handled using a standard branch prediction technique. In addition, this scheme could be expanded to always fetch both the current and the next instruction. This would allow detection of an unconditional branch instruction during the decode stage of the previous instruction. This is largely equivalent to having a separate unconditional branch pipeline, as found in some superpipelined processors, but without many of the overheads.

7.4 Pipeline

The DAIS pipeline, adapted from the five-stage pipelines of the MIPS chip-set [Kane, 1987] and DLX [Hennessy and Patterson, 1990], is shown in figure 7.2. This four-stage pipeline combines ALU and memory operations into a single stage, thus decreasing data dependencies.

Each pipeline *stage* is broken into two equal *phases*. In the first stage, both phases are dedicated

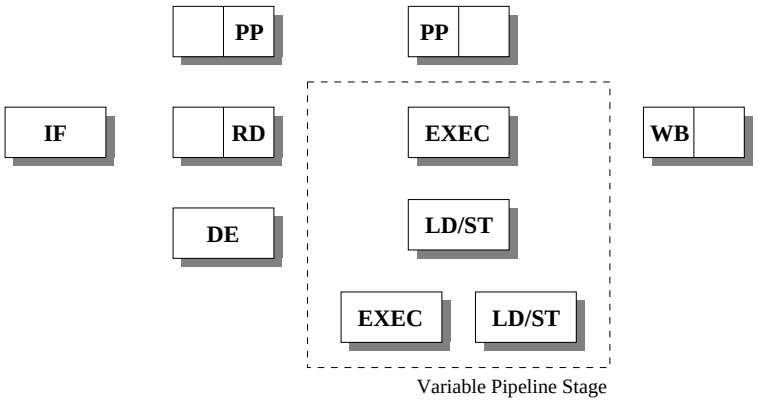


Figure 7.2: Pipeline used in DAIS.

to instruction fetch. Stage two consists of decoding instructions and reading source register stacks. Decode may occupy the whole cycle, but register-stack reads are restricted to the second phase. The fourth stage is dedicated to register-stack write-back. This must complete by the end of phase one.

Stage three is the most complex of all stages. Within this, a push/pop (PP) process and a variable pipeline stage operate in parallel. PP actions are identified in phase one of stage two, and are completed by the end of phase one of stage three. This gives PP an entire cycle with which to move information between stack heads and bodies. This is useful, given the long lengths of PP buses in comparison to the read and write buses (see section 6.2.4).

The variable pipeline stage can consist of either an ALU action, a load/store (LS) operation, or both in sequence. The ALU action occurs for instructions such as *add*, *sub* and *bra*, while the LS action occurs for load or store operations. When a memory operation involves the user stack, an ALU cycle is inserted before the memory cycle, incurring a one cycle overhead.

In five-stage pipelines, the ALU cycle immediately precedes the LS cycle. This means that effective address calculations for instructions such as LD [R0+R1],R2 will not stall the pipeline. In DAIS, displaced memory operations of this type are illegal, since adding to an OID is meaningless. Displaced addressing is notionally replaced by object-indexed addressing of the type LD R0[R1],R2, where R0 is an OID and R1 an integer index. Where an index plus index offset is required, an intermediate instruction is needed to add the two indices together, with the result being used as the offset in the load operation.

Few instructions require *both* the ALU and the LS step since object offsets do not imply pointer additions in the traditional sense; this work is done by the cache. This has one exception; when a memory operation involves the user stack, an ALU cycle must be inserted before the LS cycle. This extra cycle adds the offset specified to the USP's offset. This new offset is then used in the memory operation.

7.4.1 Pipeline Efficiency

Using a forwarding mechanism from the output of stage three of the pipeline to the beginning of the same stage allows DAIS to run without data dependency stalls. Unfortunately, where there is a load or store operation using the USP, there is an associated stall. How often does this occur?

Our analysis has been directed towards a register windows machine, since it matches our stack-based architecture well. We chose the SPARC, since it has good range of software support. User stack accesses were analysed using SHADOW [SUN, 1992], a tool to trace SPARC binaries. Results showed that under 5% of data memory references were to the user stack and that an extra ALU cycle would have to be inserted in DAIS for less than 1% of instructions. The performance degradation is therefore under 1%.

7.5 Programming Examples

As a demonstration of the use of pop bits within the instruction set, and of the fast-branch construct, consider Ackermann's function and resulting assembly code shown in program 7.1. The code was produced by hand using standard compilation methods, and optimized using tail recursion elimination. Where an instruction operand is popped, that parameter is marked with a circumflex accent, *e.g.* $\widehat{\text{S1}}$. The **ret** instruction is really **pop** PC. Parameter passing is performed here using a `caller pop' strategy, where the caller both allocates and deallocates the parameters. The return parameter must be pushed on by the callee.

One benefit of the stack-based instruction set is the ease of writing macro instructions. Macros can be written which use the stacks for local variables, and provided all local variables are popped before the macro completes, the locals can be allocated independently from the surrounding instructions.

The DAIS code produced in this example contains 14 instructions, which compares favourably with the 19 instructions produced by the standard SPARC compiler (level 4 optimization). A non-superscaler implementation of the SPARC requires 6 cycles in *ack*, 8 cycles in *case2*, and 12 in *case3*. These figures are the best possible (*e.g.* the SPARC predicts every branch correctly). For DAIS, *ack* takes between 3 and 4 cycles, *case2* between 4 and 7, and *case3* between 8 and 12. The variations in the figures reflects the optimistic and pessimistic views of branch prediction. Note that even with the largest cycle times (*i.e.* all branches miss-predicted), DAIS times are no worse than the best possible from the SPARC. The cycle times all ignore the effects of cache misses.

7.6 LOAD-Induced Data Dependencies

Owing to DAIS' four stage pipeline, which brings together ALU and LOAD/STORE cycles into a variable pipeline stage, it never has to stall due to data dependencies. When the LOAD/STORE stage follows the ALU stage, data loaded by one instruction is not immediately available to the following instruction. The processor must stall for a cycle until data can be fetched.

```
int ack(m,n) int m,n;
{
  if (m == 0)
        return(n+1);
  else if (n == 0)
        return(ack(m-1,1));
  else
        return(ack(m-1,ack(m,n-1)));
}
```

```
S1 = m, S2 = n, S0 = return value

ack:
        sbra    S1 ¡¿ 0,case2
        add     S2,1,S0
        ret
case2:
        sbra    S2 ¡¿ 0,case3
        ldc     1,S2^
        sub     1,S1^,S1
        sbra    ack
case3:
        cp      S1,S1
        sub     1,S2,S2
        call    ack
        pop     S1,S2
        cp      S0^,S2^
        sub     1,S1^,S1
        sbra    ack
```

Program 7.1: Example of Assembly Code Generated for DAIS

Optimized SPARC assembler outputs were statically analysed to produce the results given in table 7.1. Averages and instruction-based weighted averages are given. These results indicated that DAIS achieves a 6% speedup by eliminating these load dependency stalls.

Table 7.1: LOAD-Induced Stalls on the SPARC Architecture

Benchmarks	Instructions	Load Instructions		% Instructions causing stall
		Number	No. Stalling	
TEX	60542	8364	3115	5.15%
DeTEX	3750	686	455	12.13%
Zoo	10391	1287	544	5.24%
Fig2dev	29080	8257	2587	8.90%
Average				7.86%
Weighted Average				6.46%

7.6.1 C Language Support

In order to make the DAIS processor more attractive to users, it may be useful to also allow programs written in C to be executed. A peculiarity of the C programming language is that of *pointer arithmetic*.

Since DAIS has no notion of pointers, only objects, the implementation of the C language has an associated overhead. A variety of schemes are possible, including mapping the entire C program (code, static data, constants, and heap space) into a large single object. This approach does not allow C programs to interface to objects created with object-oriented languages. Instead, a second scheme is suggested.

C pointers could implemented by using an OID/index pair as in the USP. Pointer addition is realized by adding the appropriate offset to the the index, and pointer difference by subtracting indices. Note that in subtraction, the OIDs of both OID/index pairs must be equal. The compiler should plant appropriate code to perform this check. Since pointer subtraction is uncommon in C, this should be a relatively small overhead.

If a C compiler is written to detect pointers which are not used in pointer arithmetic (for example, an array declared with function scope and used only via array/offset access), then such pointers can be handled without the overhead of OID/index pairs. With variants of C, such as C++, pointer arithmetic is frowned upon. It is the intention of the author to modify a public-domain version of C++ so that pointer arithmetic is made illegal, and to use this compiler and the resulting language as the standard form for developing software for the DAIS processor. The development of this processor should be considered as future work.

7.7 Conclusions

In OID/offset based addressing, instructions using index plus index address calculations are not available, since adding a number to an OID does not make any sense in an object-oriented heap. The only exception to this is stack accesses, since the current point reached through stack allocation (the frame pointer) is defined by both an OID and an offset (the USP register). Therefore, DAIS uses only a four-stage pipeline, and incurs a stall when the executing program performs an displaced address calculation on stack data. Analysis shows that this stall will occur less than 1% of the time. The benefits of a shorter pipeline is that DAIS contains no pipeline-induced data dependencies. In comparison to a SPARC, eliminating the dependencies produces a 6% performance increase.

DAIS' branch prediction logic is on-par with many advanced RISC designs available today. Although not truly novel, it is still a requirement to avoid unnecessary pipeline stalls, therefore maintaining DAIS' competitiveness with other non object-oriented processors.

With the pipeline described in this chapter coupled to the caching strategy and stack design described earlier, DAIS should be able to execute object-addressed programs just as fast as the traditionally addressed variety. Note again that the novel register stack design of DAIS is not a crucial feature of the processor, and thus could be removed from any implementation in silicon of this design. However, its advantages of flexibility and resulting performance makes it an attractive addition.

Chapter 8

Persistence without Disks

In all the systems examined in chapter 3, reliable data persistence was provided by disk-based storage. Even in most non object-based computers, disk storage is viewed as the norm in long-term data retention. Solid-state storage (such as RAM) is judged suitable for holding only volatile or ephemeral information. RAM has a significantly higher access rate (nanoseconds compared to milliseconds), but is more expensive per MByte, than disk storage. Thus most systems hold only the `authoritative' version of data on disk, and move copies of the data into RAM for manipulation. Maintaining consistency between RAM and disk (*e.g.* through check-pointing and shadow-paging) is an important management issue, the handling of which can cause further performance penalties than that incurred by the RAM/disk access rate discontinuity. This penalty is particularly noticeable in disk-based database systems, where record transaction rates can generally be measured at between 100 and 1000 per second.

It is suggested that this traditional reliance on disks to maintain persistent data is now far from optimal, and that a radical re-design may lead to much higher performance with lower costs. The factors underlying this argument are that:

- The absolute costs of both RAM and other VLSI components is in steep decline.
- While the cost differential between disk and solid-state storage is narrowing, the miss-match between the data-access requirements of high-speed processors and backing-store performance is continuing to widen.
- Database access rates are a limiting factor in database design, since not only are databases growing in size and complexity, but their contents are being used for a wider range of services.

A good solution to holding persistent data reliably while maintaining a high bandwidth path to the data would be to hold all data in a persistent, solid-state, directly addressable heap. This would result in a viewpoint where the main store (*e.g.* the processor's memory) held the authoritative version of data, without the need for RAM/disk consistency management. Such a viewpoint promotes new

program methodologies, which would previously have been inefficient to implement on a RAM/disk arrangement. An example of this is the work done on dictionary-based encoding of relational databases [Cockshott *et al.*, 1993b]. This scheme replaces data within database records with a key. The key can be used to regenerate the data via a dictionary. This offers advantages in both speed of database searching and in database compression. One attractive technology for implementing a solid-state store is Flash memory; it is persistent, available in PCMCIA packages (allowing it to be maintained easily if a fault occurs in one of the memories which make up the store), and is byte-addressible.

8.1 Flash and Operating Systems

On the first machines to use Flash memory, the new devices were configured by software to look like a disk drive. Whilst this produced some performance gains, through the elimination of seek times, it fails to take full advantage of the new technology. In preserving the old file metaphor for non-volatile store it imposes software overheads. Although the hardware is randomly readable at the byte level, the file system imposes a block level access. Associated with this are the overheads of copying the data from Flash store into the operating system's `disk' buffers. In all probability, this will be followed by a second copy from the system buffers into the user address space whenever an application program performs a read.

Although an emulation of conventional file systems is essential for porting existing software, new software could take advantage of a more direct access to the store. For efficiency, one wants something analogous to simple memory addresses, allowing persistent data to be accessed as arrays, operated on and passed by reference to procedures like volatile data structures. In particular, one should avoid loading a program from Flash memory into RAM, preferring instead to execute the program in-situ.

Flash memory has read access times comparable to DRAM, but its write cycle is lower. Random byte reads are possible, and so in principle are random byte writes, but this is vitiated by an inability to write a byte more than once. Between writes the locations must be erased, and erasure is a block operation applying to a page at a time. These characteristics should be hidden from the user.

It should in principle be possible to do this using existing virtual memory technology. Logical pages resident in Flash store could be marked as read only in the page tables. Attempts to perform writes to them would cause a page fault that would be trapped by a driver routine. This would cause the faulting page to be copied and re-mapped to a battery backed SRAM before the write is allowed to proceed.

As a background task, the page can be copied to a new location in Flash memory, after which the original page can be erased. Failure of the controlling computer system could be catered for by transferring data between SRAM and Flash using a checkpointing algorithm. For maximum protection, a transaction model would also secure the system from SRAM failures.

8.2 Flash-Based Storage

A Flash-based storage device could be constructed from a single board which contains all the Flash memory components. However, such an implementation is not without its problems:

- If the Flash store is constructed from a small number of boards, it is difficult to utilize 3D space. This can lead to designs which cover a large surface area. Even if implemented using multiple boards, the minimum spacing between boards will be a limiting factor in the design.
- Users cannot replace chips on a board. Only in a modular design where the user is electrically isolated from the store would user maintenance be desirable. This is especially true for stores which allow for `live' or `hot' replacement of failed memories.
- Support for improvements in fabrication densities are hard to provide without holding the Flash chips on modular boards whose interface is independent of its capacity.

A better approach to the storage device would be modular memory cards. These cards would preferably be ready packaged in a small, sealed unit, with edge connection allowing cards to be arranged to match available space. Their interface should be designed such that changes in card capacities could be supported without redesigning the hardware. In fact, such cards already exist, based on the PCMCIA standard.

PCMCIA-based memory cards are available in a variety of storage technologies, including battery-backed SRAM and block-erasable Flash memory. The standard currently supports up to 64 MBytes of addressable memory (although there are unused connections which could be used to increase this). Memory itself can be accessed in either 8 or 16 bit words (user definable).

For the purposes of argument, consider that Flash cards were available in 80 MByte packages. Disk capacity is commonly measured in gigabytes, which is significantly greater than a Flash card. The store would therefore have to be constructed from multiple cards addressed sequentially. It was also suggested that Flash-based storage could be used in constructing single-level stores, with the card contents directly accessible by the processor (rather than indirectly, such as on disk); clearly, with Flash cards having a maximum data-word size of 16 bits, efficient processor interfacing (especially with the 64 bit data buses of newer processors) would require multiple cards addressed in parallel. Some processors may require extra data bits to hold tag information or CRC checks.

For a persistent store based on an array of Flash cards, reliability is an important factor. Data lost in an object stores cannot easily be selectively restored from a backup archive, due to object interdependencies. From the range of literature on Flash cards and disk drives, the expected life of a disk drive is significantly less than that of Flash storage. Not only that, but Flash memory is resistent to significantly more physical abuse than disks (expecially when in operation). Although single Flash card is much more reliable that a disk drive, a device constructed from multiple cards may not be; if card failures are independent of each other, and occur at random intervals, then the probability of all cards continuing to work decreases as the number of cards increases. The probability that a system is operating at any particular moment is known as the *system availability*.

The availability a_i of card i is calculated from the card's MTTF (mean time to failure) and its MTTR (mean time to repair):

$$a_i = \frac{\text{MTTF}_i}{\text{MTTF}_i + \text{MTTR}_i} \tag{8.1}$$

It is presumed that all cards used have the same availability value, $a = a_1 = a_2 = \ldots = a_n$. Intel report that their current-generation Flash cards have a MTTF of 10^6 hours [Intel, ND]. Taking into account the time required to find a working Flash card and exchange it with a faulty card, and the time to write 1280 card blocks (*i.e.* initialize an 80 MByte card), the MTTR was estimated at 2 hours. This is for a well-maintained system, with a technicial on standby to repair faults (*i.e.* replace the damaged card(s)) within half an hour. The system availability A is given by:

$$A = \sum_{j=r}^{n} \left(\begin{array}{c} n \\ j \end{array} \right) a^j (1-a)^{n-j} \tag{8.2}$$

Here, n signifies the number of cards in the system, and r represents the number of cards which must be functioning for the system to remain operational.

In order to support a 64 bit-wide data store it is necessary to combine 4 (if the interface used is 16 bit) or 8 (if the interface used is 8 bit) cards together into an *array line*. Failure of any one card on a line would cause the whole line to become unusable. For the 8 card version, the availability for a row is $1.0 - 1.6 \times 10^{-5}$, giving a MTTF of approximately 14.27 years. With a 4 card row, the availability rises to $1.0 - 8 \times 10^{-6}$, equating to a MTTF of approximately 28.54 years. Since a single card failure would cause permanent loss of data, these availabilities may unacceptably low for some applications (*e.g.* banking databases, central file servers, *etc.*);

Supposing that redundant information was also stored in the array row, such that the loss of a single card would not cause any loss of information. Although this would reduce the storage efficiency of the array, the resulting availability will be dramatically improved. For example, if single card failure recovery could be achieved using a combination of six cards, then the availability rises to $1.0 - 6 \times 10^{-11}$, or a MTTF of just over 3.8 million years of continuous use! This level of resilience should be more than adequate for all conceivable storage requirements.

Possible failure scenarios which could occur within a Flash-card array include:

- Failure of one or more bits within a single card (bits inverted, incorrect address decode).
- Failure of an entire card (*e.g.* card removed).
- Failure of part of the array's logic circuits
- Failure of the entire array.

Since both of the first two types of failure require the replacement of the faulty card (the PCMCIA cards are supplied in a sealed package), any errors detected are considered to be failures of the entire card. Note that the PCMCIA cards are fulls enclosed, and include addressing and decode logic. Failure of logic within the cards themselves is considered a failure of the entire card (or at least as much as the logic failure affects). The last two types of failures are best done by either shutting the system down

until the fault can be traced, or, in the case of mission-critical systems, through the use of redundant hardware.

ERCA, Error Recovery in Card Arrays, is a collection of algorithms developed with a minimum specification of providing support for error recovery of any single Flash card failures. Four classes (or levels) of ERCA are currently defined:

ERCA 1: Automatic detection of a single card failure, but manual identification of the failed card is required. The previous read action on the array must be undone.

ERCA 2: Automatic detection and recovery from single card failures. Additional failures after the first go undetected.

ERCA 3: Single card failure detection, automatic recovery. Lower redundancy than ERCA 2. ERCA 3 is not implemented at its maximum possible efficiency for the number of cards it uses, but instead is used as a discussion point to introduce some of the techniques used to implement ERCA 4.

ERCA 4: Single card failure is transparently recovered. Double card failure is automatically detected, but manually identification by the operating system is required before the ERCA controller can transparently recover. Redundancy is lower than ERCA 3, while ERCA 4 has the highest availability of all other ERCA classes.

All ERCA classes are defined with 64 bit data bus, and some of the levels have additional bits for tagging or other user-defined purposes. This in no way prevents systems with a smaller data bus from using ERCA, since it is a simple change to make the store appear to have a smaller bus (in a similar way to standard memory-interleaving). The definition of the four classes are really only to act as a was of describing the methodology behind an ERCA-maintained solid-state store, as the precise characteristics of each store will vary according to the surrounding hardware (*e.g.* the processor's data bus width). ERCA 4 was designed with DAIS-64 in mind.

Basically, two forms of recovery are used in ERCA:

1. **ADAR**, *Auto Detect Auto Recovery*. In this scheme the ERCA controller can automatically detect a card failure, and can also transparently recover from it. The user need never be informed of the failure (although the system manager should be informed so that a replacement card can be inserted).

2. **ADOR**, *Auto Detect OS-Assisted Recovery*. The ERCA controller can detect when a card failure has occurred, but it cannot identify which card has failed. The operating system is informed immediately of the error. The operating system then needs to read the cards in that array row, comparing each card's data to the relevant checksum (data on a card is checksummed in a similar way to a disk-block or network packet checksum). Once the failed card has been identified, it's identity is passed back to the ERCA controller. The ERCA controller can then perform transparent recovery from the failure.

Errors in the array store cannot be detected until that row of the store is accessed by the processor. Preferably, the processor will be able to respond to a memory cycle abort (*e.g.* BERR on MC68020 processors [Motorola, 1985][pages 5-34 to 5-39]), allowing the reading process to be suspended while an exception handler (initiated by an ERCA-generated interrupt) surveys the card failure. If the processors cannot be interrupted during a memory cycle, data *must* be returned to the processor before an interrupt request can be processed. Should the ERCA level implemented on the store not correctly recover from the failure (*i.e.* the failure-recovery for the problem is class ADOR), this data read must be somehow undone (*e.g.* by unwinding the read or killing the reading process), since the data returned is incorrect. Undoing read actions is a complex and clumsy proposition, and so the ERCA level should be chosen such that this undo action occurs very infrequently (*i.e.* approximately zero probability of occurrence within the lifetime of the store).

Once a card failure has been correctly identified, the Flash store can still be accessed and will return correct information as if the failed card was still operating. ERCA 4 will even allow access to the store with two simultaneous card failures. On the first occurrence of an error, the operating system is always informed that a card failure has just occurred. The computer user should then be informed which card has failed. Once a replacement card has been inserted, the card information is automatically reconstructed. Card removal and replacement can be performed while the machine is powered up, and even when the store is being accessed. If a card has failed, it will not be accessed again. This prevents sporadic or intermittent faults degrading the performance of the system. Only physical replacement of a card from the store will allow the port to which the failed card was connected to be accessed again.

8.2.1 ERCA 1 - RAID

The problem of data resilience between multiple interdependent storage units is not just restricted to card arrays; arrays of hard drives also exist in high-reliability (and cost effective) disk storage. RAID [Leg, Spring 1991] was developed to manage arrays of disk drives, in an attempt to improve disk I/O performance by utilizing multiple disks in parallel.

With the increased number of disk devices, storage reliability is reduced. To counteract the drop in reliability, RAID uses XOR-based check-data similar to the scheme depicted in figure 8.1. Here, bytes stored on disks 1 to m ($m = 3$ in this example) are XORed together, forming a check value which is held on the check disk. Each disk also has a block checksum (*e.g.* the sum of all bytes stored in that block).

Whenever a data disk fails, the data which it contained can be calculated by XORing the remaining data disks and the check disk together. Failure of the check disk requires no recover operation, since the original data is still directly available on disks 1 to m. For example, data a on failed data disk i can be calculated from:

$$a_i = \left(\bigoplus_{j=1}^{m} a_j (1 - \delta_{i,j}) \right) \oplus a_{\text{check disk}} \tag{8.3}$$

Here $\delta_{i,j}$ is the Kronecker delta, which evaluates to 1 if $i = j$, and 0 otherwise. Since disks are accessed using block transfers, disk failure can be detected either by non-response of the disk to commands

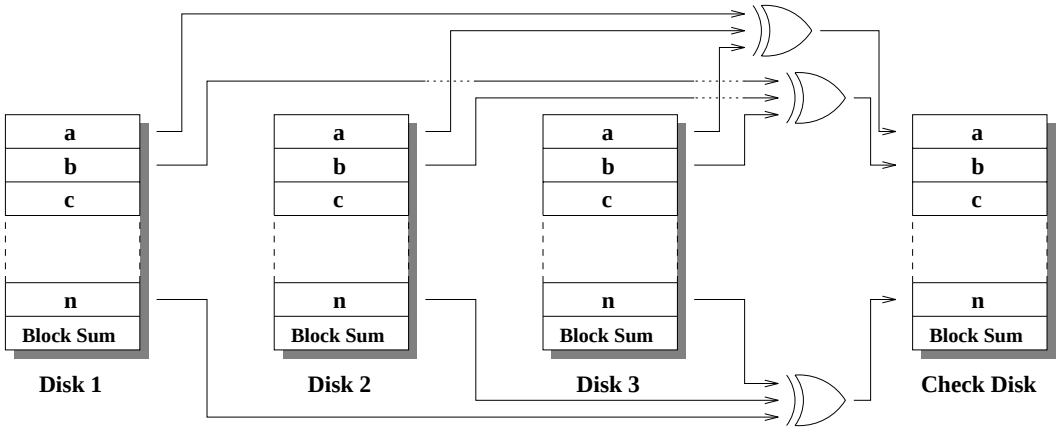


Figure 8.1: XOR Mechanism used in RAID Disk Storage

(electrical failure), or block checksum discrepancies.

For a Flash card storage unit based on RAID, identification of failed cards poses a significant problem; since the read action which was in progress caused the error to be detected, it must be completed before the processor can begin the process of locating the failed card. The fact that the store always returns something means that the read will almost always have to be undone (only on a check-card failure will the read not have to be undone).

Unwinding from a read is possible in a kernel routine, where the unwind routine can have knowledge of kernel-based store-access mechanisms and internal structures. If the read occurred within a user program, then immediate program termination is the only safe way of dealing with the problem (before the erroneous data can corrupt any other data items). Unwinding a user program read would require instruction flow analysis, and even then may be impossible in some situations (*e.g.* processors with a delayed branch, imprecise interrupts, superscaler implementations, *etc.*).

For a card store based on ERCA 1, the highest possible availability is constructed using four cards for data retention (in 16 bit mode), and a single card for the check information (also in 16 bit mode).

During a single card failure, the check card will always report an error. If a second failure occurred while the first was still under repair, then the second failure would go unnoticed. This is undesirable for a high-reliability system.

8.2.2 ERCA 2

In ERCA 2 (and beyond), multiple XORs are generated for each row in the array, as opposed to RAID's single XOR calculation. The XORs, when combined with the user data used in constructing them, form hamming codes, which in turn help to pinpoint failed devices. Once the erroneous devices have been identified, the missing data is recovered from the XORs in a similar way to RAID.

ERCA 2 provides error recovery using three redundant *check cards* $\{\omega_1, \omega_2, \omega_3\}$ and four data cards $\{\alpha_1 \ldots \alpha_4\}$. Each card is used in 16 bit mode. The data cards hold information as written by the user (mimicking a 64 bit wide data store), while the check cards are used solely for error detection and

recovery.

Table 8.1 shows the relationship between data cards and check cards. Where • appears at the intersection of data and check card, then the data card is XORed with the other terms for that check card. The combination of XOR terms for each check card can also be seen on the right of this table, written as boolean logic. When data is written into the store, ω_i is calculated from the relevant XOR terms: for example, bit z of ω_1 is calculated by XORing bit z of α_1, α_2, and α_3 together.

Table 8.1: Exclusive OR Terms Used in ERCA 2.

	Data Cards				
Check Cards	α_1	α_2	α_3	α_4	Resultant Equation
ω_1	•	•	•		$\alpha_1 \oplus \alpha_2 \oplus \alpha_3$
ω_2		•	•	•	$\alpha_2 \oplus \alpha_3 \oplus \alpha_4$
ω_3	•		•	•	$\alpha_1 \oplus \alpha_3 \oplus \alpha_4$

Whenever data is read from the store, the XOR values are recalculated and compared with those stored on the check cards. If the calculated value differs from the value stored in the check card then that card signals FAULT, otherwise it signals OK. If any FAULT signals are generated, then the faulty card can be identified from table 8.2.

Table 8.2: Error Codes Against Card Faults (ERCA 2).

Check-Card Status			
ω_1	ω_2	ω_3	Card Fault Identified
FAULT	OK	FAULT	α_1
FAULT	FAULT	OK	α_2
FAULT	FAULT	FAULT	α_3
OK	FAULT	FAULT	α_4
FAULT	OK	OK	ω_1
OK	FAULT	OK	ω_2
OK	OK	FAULT	ω_3

Once the faulty card is located, the fault will either be ignored (if it is a check card) or, if it is a data card, the missing data will be recovered. Recovery is achieved by XORing a check card which contains the missing card as a term, with all the terms (except the failed one) used to construct that check card. For example, failure of α_1 could be recovered by either $\omega_1 \oplus \alpha_2 \oplus \alpha_3$ or $\omega_3 \oplus \alpha_3 \oplus \alpha_4$.

In ERCA 2, the redundancy required is 75%. That is, 75% more storage is required to implement the error recovery. This in effect means that 42.8% of the Flash store is committed to recovery information. In ERCA 3 and 4, the design target was to achieve under 50% redundancy. The support of a tag field, in addition to the 64 bit data field, was also examined[1].

[1] It was specified that ERCA 3 and 4 should allow support for a tag field of at least 2 bits, effectively increasing the data storage requirement to 68 bits per array row.

8.2.3 ERCA 3

Error codes used in the Flash array must have at least two FAULT signals in order to support automatic identification of a failed card, since a single FAULT signal could mean that a check card had failed instead. This gives the number of possible data cards d which can be supported by c check cards as:

$$d = 2^c - c - 1 \tag{8.4}$$

In ERCA 2, c was equal to three, allowing four data cards to be supported. However, this equation shows that as the number of check cards increases linearly, then the number of data cards which can be supported increases exponentially. Thus, if each Flash card could be divided such that there appears to be double the number of cards (each half the original size), then proportionally less of these half-sized cards would be required than in the full-sized case. This reasoning was used in the design of both ERCA 3 and ERCA 4.

ERCA 3 is uses thirteen cards in 8 bit mode, with four assigned to recovery and the remaining nine used to store data (64 bits plus an 8 bit tag). The redundant cards, $\{\omega_1 \ldots \omega_4\}$, are calculated from the data cards $\{\alpha_1 \ldots \alpha_9\}$ in a similar way as ERCA 2. The XOR terms are shown in table 8.3. Note that the table contains a capacity for 11 data cards, but only 9 are used by the store.

Table 8.3: XOR Terms Used in ERCA 3.

	Data Card										
Check Card	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8	α_9	α_{10}	α_{11}
ω_1	•	•	•				•		•	•	•
ω_2	•			•	•		•	•	•		•
ω_3		•			•	•	•	•		•	•
ω_4			•	•		•		•	•	•	•

Single data-card failures occur whenever two or more check-cards signal FAULT. If only one check-card signals a FAULT, then it is that check-card which contains the error.

ERCA 3 requires thirteen cards operating in 8 bit mode. Even with such a large number of cards, the availability of the store is $1.0 - 3.12 \times 10^{-10}$, giving a mean time to failure of 0.73 million years, and a redundancy of 44.4%. ERCA 3 has been made largely redundant by the next ERCA level, but serves as a useful introduction to the design strategy of ERCA 4.

8.2.4 ERCA 4

ERCA 4 expands on ERCA 3, providing ADAR for single and ADOR for double card failures. Cards are used in 8 bit mode. Eight cards hold data, three hold only redundant information, and one card holds half data and half redundancy information. This equates to a redundancy of 41.2%.

Each 8 bit card is split into two sub-cards of 4 bits each. User data is held on sub-cards d_0 to d_{15}. User tag information is held in the tag sub-card (4 bit tag). The XOR values are held in check

sub-cards, which are in turn split into primary $\eta^p_{\{1\ldots4\}}$ and secondary $\eta^s_{\{1\ldots3\}}$ check sub-cards. These are mapped onto cards in accordance with table 8.4.

Table 8.4: Card Layout in ERCA 4.

	Sub-Part	
Card	1	2
1	η^s_1	η^s_2
2	η^s_3	tag
3	η^p_1	η^p_2
4	η^p_3	η^p_4
5	d_{15}	d_{14}
6	d_{13}	d_{12}
7	d_{11}	d_{10}
8	d_9	d_8
9	d_7	d_6
10	d_5	d_4
11	d_3	d_2
12	d_1	d_0

The XOR values are calculated in the same way as ERCA 2 and ERCA 3, with each XOR evaluated from a different set of sub-cards. The XOR combinations are shown in table 8.5. Note that cards 1, 3, and 4 (plus sub-part 1 of card 2) are not data cards, but hold the check information. Their contents are not used in the right-hand-side of any XOR calculation, as errors in these cards have no effect on the user data stored on that line of the store. The primary check terms are distributed so that, when taken in conjunction with the secondary terms, unique error codes are always generated under single card failures.

When the store is re-read, the XOR values are recalculated and compared with the values held within the check sub-cards; any differences causes the differing check sub-cards to signal FAULT.

Given that k is the set of primary check sub-cards and l is the set of secondary check sub-cards, then the set of check sub-cards which use sub-part y of card x is given by the set $a_{x,y} \subseteq k$ for primary check sub-cards, and $b_{x,y} \subseteq l$ for secondary check sub-cards. Differences in check values can be evaluated using DIFF(d), $d \in k \cup l$, where DIFF$(d) = 0$ if there is no difference in d between the value stored during the data write and the value calculated during the data read, and DIFF$(d) = 1$ otherwise (*i.e.* if there was a difference). Two functions used in evaluating error conditions are SOR(c) (the ORing together of all elements in the set c) and SAND(c) (the ANDing together of all elements of set c), defined as:

$$\text{SOR}(c) = \begin{cases} 0 & \text{if } c = \{\} \text{ or } \sum_{k \in c} \text{DIFF}(k) = 0 \\ 1 & \text{otherwise} \end{cases} \tag{8.5}$$

$$\text{SAND}(c) = \begin{cases} 0 & \text{if } c = \{\} \\ \prod_{k \in c} \text{DIFF}(k) & \text{otherwise} \end{cases} \tag{8.6}$$

SOR(c), SAND$(c) \in \{0, 1\}$. An error on card x will result in one or more of $a_{x,y}$ producing a DIFF

Table 8.5: XOR Terms for ERCA 4.

		Check Sub-Cards						
Card	Sub-Part	η_1^p	η_2^p	η_3^p	η_4^p	η_1^s	η_2^s	η_3^s
1	1							
	2							
2	1							
	2	•	•	•				
3	1							
	2							
4	1							
	2							
5	1	•				•		
	2		•				•	
6	1	•						•
	2		•			•		
7	1	•					•	
	2		•					•
8	1			•		•		
	2				•		•	
9	1			•				•
	2				•	•		
10	1			•			•	
	2				•			•
11	1	•		•		•		
	2		•		•		•	
12	1	•		•			•	
	2		•		•			•

of 1. An error on card x cannot affect any check sub-card which is not a member of $a_{x,1}$, $a_{x,2}$, $b_{x,1}$, or $b_{x,2}$. Thus an indication v of card x failing can be given by the boolean expression:

$$v_i = (\text{SOR}(a_{x,1}) + \text{SOR}(a_{x,2}))\overline{(\text{SOR}(k - (a_{x,1} \cup a_{x,2})) + \text{SOR}(k - (b_{x,1} \cup b_{x,2})))} \tag{8.7}$$

As well as v_x being true, card x can only contain an error if the check values occur in the correct combinations. For instance, table 8.5 shows for card 5, both check sub-parts η_1^p and η_1^s, and/or both η_2^p and η_2^s, must signal FAULT for card 5 to be in error. The combinational check w for card x is defined in boolean terms as:

$$w_x = \overline{(\text{SOR}(a_{x,1} \cup b_{x,1})\overline{\text{SAND}(a_{x,1} \cup b_{x,1}))}} + \overline{(\text{SOR}(a_{x,2} \cup b_{x,2})\overline{\text{SAND}(a_{x,2} \cup b_{x,2}))}} \tag{8.8}$$

Thus, the error status s of card x is given in boolean terms by:

$$s_x = v_x w_x \tag{8.9}$$

Once an error has been linked to a single card failure, the missing information can be regained by XORing check values with data from functional data cards, in the same way as previous ERCA systems. Errors which cannot be linked are taken to be failures of check values.

On a change of error codes (*e.g.* going from no error to a particular error code, or performing a read action which generates a new error code) the processor is immediately notified with a high priority interrupt. The array cards can then be investigated, with the array controller being notified of the failed cards. ERCA 4 always identifies single card failures correctly[2], but may be fooled into making an error if more than one card fails. Therefore, on detection of a multiple-card failure, the last read on the store must be undone.

Once the ERCA controller has been informed of the two failed cards by the operating system (using card-based checksums), the controller can automatically recalculate the missing data. Table 8.6 demonstrates the recovery procedure needed when recovering from a double card failure. All double card failures are listed, with recovery split into two *cycles*. In the table, • marks a failed card.

The naming convention using for data recovery in table 8.6 follows the format $r_s(t)$, where $1 \leq r \leq 12$, $s \in \{1, 2\}$, and $t \in (k \cup l)$. From this, the recovery of sub-part s of card r can be obtained using check-value t XORed with all terms used in calculating t except the term found in the card to be recovered. Where two or more recovery actions occur within a single cycle (separated by commas), these may be calculated independently in parallel. Once all cycle 1 calculations have been completed, the check values used within the second cycle must be updated to reflect the recovered data before cycle 2 recovery actions can commence.

All the calculations shown in table 8.6 should be easily implemented within a single VLSI device, with plenty of space left over for the array controller itself. Once the processor has informed the controller of the failed cards, recovery of the missing data can once again proceed automatically. The two cycle recovery process of double card failures is little more than an additional gate delay, so performance under two card failure should remain identical to normal array store accesses.

[2] Verified through simulation of all possible single card failures. A formal methodology for producing and confirming ERCA recovery schemes is being investigated.

Table 8.6: Recovery in ERCA 4 from Double Card Failures

Failure Identified on Two Cards												Order of Data Recovery	
1	2	3	4	5	6	7	8	9	10	11	12	Cycle 1	Cycle 2
•	•											$2_2(\eta_3^p)$	
•		•											
•			•										
•				•								$5_1(\eta_1^p)$, $5_2(\eta_2^p)$	
•					•							$6_1(\eta_1^p)$, $6_2(\eta_2^p)$	
•						•						$7_1(\eta_1^p)$, $7_2(\eta_2^p)$	
•							•					$8_1(\eta_3^p)$, $8_2(\eta_4^p)$	
•								•				$9_1(\eta_3^p)$, $9_2(\eta_4^p)$	
•									•			$10_1(\eta_3^p)$, $10_2(\eta_4^p)$	
•										•		$11_1(\eta_1^p)$, $11_2(\eta_2^p)$	
•											•	$12_1(\eta_1^p)$, $12_2(\eta_2^p)$	
	•	•										$2_2(\eta_3^p)$	
	•		•										
	•			•								$5_1(\eta_1^p)$, $5_2(\eta_2^p)$	
	•				•							$6_1(\eta_1^p)$, $6_2(\eta_2^p)$	
	•					•						$7_1(\eta_1^p)$, $7_2(\eta_2^p)$	
	•						•					$8_1(\eta_3^p)$, $8_2(\eta_4^p)$	
	•							•				$9_1(\eta_3^p)$, $9_2(\eta_4^p)$	
	•								•			$10_1(\eta_3^p)$, $10_2(\eta_4^p)$	
	•									•		$11_1(\eta_1^p)$, $11_1(\eta_2^p)$	
	•										•	$12_1(\eta_1^p)$, $12_2(\eta_2^p)$	
		•	•										
		•		•								$5_1(\eta_1^s)$, $5_2(\eta_2^s)$	
		•			•							$6_1(\eta_3^s)$, $6_2(\eta_1^s)$	

Continued (1 of 3)...

Table 8.6 (cont): Recovery in ERCA 4 from Double Card Failures

Failure Identified on Two Cards												Order of Data Recovery	
1	2	3	4	5	6	7	8	9	10	11	12	Cycle 1	Cycle 2
		•				•						$7_1(\eta_2^s)$, $7_2(\eta_3^s)$	
		•					•					$8_1(\eta_3^p)$, $8_2(\eta_4^p)$	
		•						•				$9_1(\eta_3^p)$, $9_2(\eta_4^p)$	
		•							•			$10_1(\eta_3^p)$, $10_2(\eta_4^p)$	
		•								•		$11_1(\eta_1^s)$, $11_2(\eta_2^s)$	
		•									•	$12_1(\eta_2^s)$, $12_2(\eta_3^s)$	
			•	•								$5_1(\eta_1^p)$, $5_2(\eta_2^p)$	
			•		•							$6_1(\eta_1^p)$, $6_2(\eta_2^p)$	
			•			•						$7_1(\eta_1^p)$, $7_2(\eta_2^p)$	
			•				•					$8_1(\eta_1^s)$, $8_2(\eta_2^s)$	
			•					•				$9_1(\eta_3^s)$, $9_2(\eta_1^s)$	
			•						•			$10_1(\eta_2^s)$, $10_2(\eta_3^s)$	
			•							•		$11_1(\eta_1^s)$, $11_2(\eta_2^s)$	
			•								•	$12_1(\eta_2^s)$, $12_2(\eta_3^s)$	
				•	•							$5_2(\eta_2^s)$, $6_1(\eta_3^s)$	$6_2(\eta_2^p)$, $5_1(\eta_1^s)$
				•		•						$5_1(\eta_1^s)$, $7_2(\eta_3^s)$	$5_2(\eta_2^p)$, $7_1(\eta_1^p)$
				•			•					$5_1(\eta_1^p)$, $5_2(\eta_2^p)$, $8_1(\eta_3^p)$, $8_2(\eta_4^p)$	
				•				•				$5_1(\eta_1^p)$, $5_2(\eta_2^p)$, $9_1(\eta_3^p)$, $9_2(\eta_4^p)$	
				•					•			$5_1(\eta_1^p)$, $5_2(\eta_2^p)$, $10_1(\eta_3^p)$, $10_2(\eta_4^p)$	
				•						•		$11_1(\eta_3^p)$, $11_2(\eta_4^p)$	$5_1(\eta_1^s)$, $5_2(\eta_2^s)$
				•							•	$12_1(\eta_3^p)$, $12_2(\eta_4^p)$, $5_1(\eta_1^s)$	$5_2(\eta_2^s)$
					•	•						$6_2(\eta_1^s)$, $7_1(\eta_2^s)$	$6_1(\eta_1^p)$, $7_2(\eta_2^p)$
					•		•					$6_1(\eta_1^p)$, $6_2(\eta_2^p)$, $8_1(\eta_3^p)$, $8_2(\eta_4^p)$	
					•			•				$6_1(\eta_1^p)$, $6_2(\eta_2^p)$, $9_1(\eta_3^p)$, $9_2(\eta_4^p)$	

Continued (2 of 3)...

Table 8.6 (cont): Recovery in ERCA 4 from Double Card Failures

Failure Identified on Two Cards												Order of Data Recovery	
1	2	3	4	5	6	7	8	9	10	11	12	Cycle 1	Cycle 2
					•				•			$6_1(\eta_1^p)$, $6_2(\eta_2^p)$, $10_1(\eta_3^p)$, $10_2(\eta_4^p)$ $6_1(\eta_3^s)$, $11_1(\eta_3^p)$, $11_2(\eta_4^p)$ $6_2(\eta_1^s)$, $12_1(\eta_3^p)$, $12_2(\eta_4^p)$ $7_1(\eta_1^p)$, $7_2(\eta_2^p)$, $8_1(\eta_3^p)$, $8_2(\eta_4^p)$ $7_1(\eta_1^p)$, $7_2(\eta_2^p)$, $9_1(\eta_3^p)$, $9_2(\eta_4^p)$ $7_1(\eta_1^p)$, $7_2(\eta_2^p)$, $10_1(\eta_3^p)$, $10_2(\eta_4^p)$	$6_2(\eta_1^s)$ $6_1(\eta_3^s)$
					•					•			
					•						•		
						•	•					$7_2(\eta_3^s)$, $11_1(\eta_3^p)$, $11_2(\eta_4^p)$	$7_1(\eta_2^s)$
						•				•		$12_1(\eta_3^p)$, $12_2(\eta_4^p)$	$7_1(\eta_2^s)$, $7_2(\eta_3^s)$
							•	•				$8_2(\eta_2^s)$, $9_1(\eta_3^s)$	$8_1(\eta_3^p)$, $9_2(\eta_4^p)$
							•		•			$8_1(\eta_1^s)$, $10_2(\eta_3^s)$	$8_2(\eta_4^p)$, $10_1(\eta_3^p)$
							•			•		$11_1(\eta_1^p)$, $11_2(\eta_2^p)$	$8_1(\eta_1^s)$, $8_2(\eta_2^s)$
							•				•	$12_1(\eta_1^p)$, $12_2(\eta_2^p)$, $8_1(\eta_1^s)$	$8_2(\eta_2^s)$
								•	•			$9_2(\eta_1^s)$, $10_1(\eta_2^s)$	$9_1(\eta_3^p)$, $10_2(\eta_4^p)$
								•		•		$9_1(\eta_3^s)$, $11_1(\eta_1^p)$, $11_2(\eta_2^p)$	$9_2(\eta_1^s)$
								•			•	$9_2(\eta_1^s)$, $12_1(\eta_1^p)$, $12_2(\eta_2^p)$	$9_1(\eta_3^s)$
									•	•		$10_2(\eta_3^s)$, $11_1(\eta_1^p)$, $11_2(\eta_2^p)$	$10_1(\eta_2^s)$
									•		•	$12_1(\eta_1^p)$, $12_2(\eta_2^p)$	$10_1(\eta_2^s)$, $10_2(\eta_3^s)$
										•	•	$11_1(\eta_1^s)$, $12_2(\eta_3^s)$	$11_2(\eta_4^p)$, $12_1(\eta_3^p)$

8.2.5 ERCA Classification Summary

Table 8.7 contains the availability and MTTF for all the ERCA classes discussed above. The availability is calculated from two different mean times to repair (MTTR). The MTTR of 2 hours is identical to that quoted for RAID systems, which includes replacing the faulty drive and rebuilding its information. For ERCA, this time includes a half hour reaction period to the failure, time required to replace the card, and the time needed to rebuild the data. A MTTR of 100 hours was also considered, which could equate to 4 hours a day of home use, 5 times a week for 4 weeks, before a domestic user would actually try to obtain a replacement card. Even with a MTTR of 100 hours, ERCA 4 has a MTTF of more that 50 million years.

Table 8.7: Summary of ERCA Availability and MTTF.

ERCA Class	Reliability			
	MTTR 2 hours		MTTR 100 hours	
	Availability	MTTF (years)	Availability	MTTF (years)
ERCA 1	$1.0 - 4.0 \times 10^{-11}$	5.7×10^{6}	$1.0 - 1.0 \times 10^{-8}$	1.1×10^{5}
ERCA 2	$1.0 - 8.4 \times 10^{-11}$	2.7×10^{6}	$1.0 - 2.1 \times 10^{-7}$	5.4×10^{4}
ERCA 3	$1.0 - 3.1 \times 10^{-10}$	7.3×10^{5}	$1.0 - 7.8 \times 10^{-7}$	1.5×10^{4}
ERCA 4	$1.0 - 1.8 \times 10^{-15}$	1.3×10^{11}	$1.0 - 2.2 \times 10^{-10}$	5.2×10^{7}

Table 8.8 gives the redundancy figure and recovery class for each of the ERCA levels.

Table 8.8: Summary of ERCA Classifications.

ERCA Class	No of Cards	Tag	Redundancy	Single Card	Double Card
ERCA 1	5	–	25%	ADOR	—
ERCA 2	7	–	75%	ADAR	—
ERCA 3	13	8 bits	44.4%	ADAR	—
ERCA 4	12	4 bits	41.2%	ADAR	ADOR

Figure 8.2 shows an impression of a Flash array containing 1.875 GByte of user-available storage (assuming 80 MByte cards and ERCA 4). The cards need a base of approximately 390mm by about 50mm. They only need 37.2mm of width if they were to be placed tightly together; however, doing to would make removal and insertion of cards difficult.

8.3 Supporting Non-perfect Flash Devices

> ... semiconductor companies think too conventionally. The largest Flash EEPROM chips, currently some 16 MBytes/chip, are guaranteed 100 per cent functional which makes large

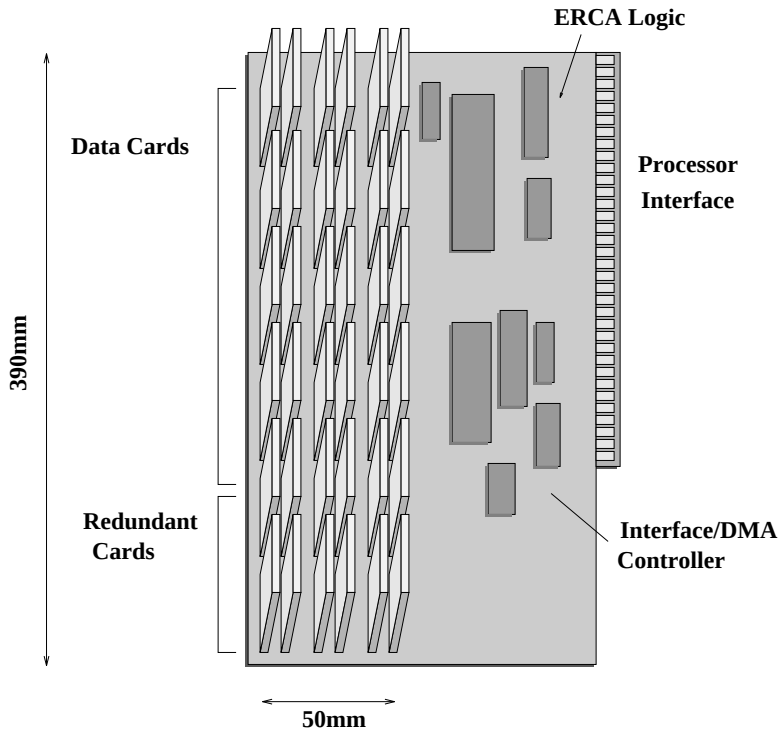


Figure 8.2: Three Rows of ERCA 4 Implemented on a PCB.

> memory arrays build from them expensive. The concept of wafer scale integration is some 25 years old but deserves rediscovery. In the same way that bad disk sectors are mapped and excluded, a similar redundancy EDC system could be made for large, low cost silicon storage. I would like to buy silicon memory arrays with their own built-in operating system rather than as individual chips.
>
> *— Frank Ogden, Editor of Wireless World [Ogden, 1994]*

The ERCA design lends itself to the design of Flash arrays constructed using non-perfect Flash chips. ERCA 4 for example offers automatic error detection and recovery for single Flash card failure, which could correspond to a failed chip somewhere in that array row. Thus if the system allowed one failed device to exist in each row, ERCA 4 could allow normal reading and writing to the array while still being able to detect a second failure.

A Flash array using imperfect devices would be initially formatted (*e.g.* all the free memory areas stored in a master free list). During the formatting procedure, each row in the array would be exercised, and checks made as to what types (if any) of errors exist in that row. If more than a single error exists, then that row is marked as bad and left out of the free list. Rows with either no errors or single errors can be used for normal data storage, with critical information (such as the free lists themselves) stored in the areas with no detected errors. In this way, critical data is protected by the double card-failure immunity of ERCA 4, while less critical information remains protected from subsequent single card failures.

If during normal operation a card failure occurs, then a check is made to evaluate the data held there. If the data is critical, then it is copied to another area which contains no failed devices. If the data is not critical and there is only a single failure on this row, then the error could be ignored. If this is the second failure on a row, then the data stored their must be copied elsewhere, and the failed row marked as bad.

Reliability can be improved by not using the areas with a single failure until the areas with no errors have been fully utilized. Single-failure areas can also be used for holding data which can be recreated if necessary (*e.g.*hashing tables used to find pages quickly, garbage collection temporary data, *etc.*). Even with all rows in the array containing a single card failure, the MTTF for the system is 10379 years (availability of $1.0 - 2.200 \times 10^{-8}$). This corresponds to a probability of less than 0.001 that a failure will occur over 10 years continuous use. With a rate of single failures spread more thinly throughout the store, and the automatic transfer mechanism and bad-row marking in place, this figure should be improved on significantly.

8.4 Flash Store Management

Flash memory presents very different performances for read and write operations. The speed at which it can be read is more or less the same as for RAM. The speed at which it can be written is considerably slower. Before it can be rewritten the current data must be erased by a separate operation. Individual words cannot be erased. Instead, whole blocks, with a minimum size of 64K in current technology, must be erased at once.

Once a block has been erased, individual words can be written. These writes are relatively slow, taking a minimum of 10 microseconds using current technology. There is thus a serious mismatch between Flash write speed and normal microprocessors memory access cycle times.

A mechanism is needed that will at once minimize the number of blocks to be erased and allow memory write cycles to proceed at normal speed. It is proposed that existing virtual memory techniques be modified to this end.

Machines would be equipped with both Flash and battery backed SRAM cards in their memory array. All pages currently resident in Flash store could be marked as read only. Read cycles would be transparently directed to the Flash store. Write cycles directed at a Flash-resident page would trigger a protection error. The fault handler would copy the recalcitrant page from Flash onto SRAM cards, re-map it at the same virtual address and enable writes before restarting the instruction. Subsequent writes would go to the page in SRAM. A standard least recently used algorithm could select dirty pages to be written back to the Flash store.

The pages need not be written back to their original location. The system could maintain a list of free page slots on recently erased Flash memory blocks, from which slots could be allocated. The page frames in Flash memory would cycle between 3 states

1. Mapped with valid data.

2. Unmapped but dirty.

3. Erased but still unmapped.

Once all of the page frames in a block were in state 2, the block as a whole could be erased and the page frames moved to state 3.

The write operation, which copies a whole page of data from SRAM to Flash, can be performed in the background by DMA.

8.4.1 Block Erasure Limitation

A concern with Flash card storage is their limitation of about 1×10^5 erase/program cycles on each Flash block.[3]. It would cause premature card failure if some blocks were continuously erased and programmed, while other blocks were rarely altered. To avoid this, the store manager monitors the number of erasures made on each block, and attempts to balance erasures throughout the Flash-cards. Flash blocks which have not changed for a long time are relocated onto blocks which have a high erasion count. In this way, the life of the store can be greatly extended.

Consider a Flash store based on ERCA 4, using a Flash-card capacity of 80 MBytes. Every row in the store would contain 320 Flash blocks, amounting to 640 MBytes of storage (ignoring tag bits). A three-row version of this store is used as a replacement for a file server's hard drive, providing 1.875 GBytes of data storage. Even taking as an average that one Flash block will be erased every 5 seconds, then the time required to perform 1×10^5 erasures (all of the minimum quoted limit on erasure cycles per Flash block) on every block would be over 15 years. It is predicted that this time would be extended by at least an order of magnitude through intelligent RAM caching. In envisaged applications, long-term data held on such a store would be rarely modified.

8.5 Analysis

With the replacement of comparatively slow-access disk storage with fast Flash cards, the performance of disk-bound applications should increase dramatically. It should also be possible, considered that data can still be read even when stored on the Flash array, to implement a computer system with much less `main' memory than that previously required. This saving in cost, space, and power consumption should help to alleviate the higher cost of the Flash cards. This is especially significant since the main memory would preferably be implemented using battery-backed SRAM chips (making data persist over power failures), which are much more expensive than their DRAM counterparts.

A simulator was constructed to measure the paging overhead as the amount of main memory was reduced. This was used to analyse the paging effects which would occur during execution of a number of benchmarks. In the simulator, main memory starts completely empty. If a read is attempted on a page which is currently unmapped, then it is considered to be a previously defined page (such as data or program instructions), and is identified as being mapped in Flash memory. If the page was mapped, either in main memory or Flash, then the read can proceed normally.

[3] This is the figure quoted in Flash-card specifications, but manufactures estimate that erasure should still be possible even after ten times as many cycles.

If an attempt is made to write to a page which is currently unmapped, then it is considered to be a page of new data. This page can therefore be mapped directly into memory without cost. If instead the page is to be found in the Flash store (and thus read-only), it is first transferred to main memory. The transfer incurs the cost of loading the entire page from Flash and storing it in memory.

If at any time a page needs to be mapped in main memory but there is insufficient room to hold the page (*i.e.* main memory is fully utilized), then a page is transferred from main memory into the Flash store. While the Flash array is being written to, the current process is suspended. Initially, an intermediate buffer was used, which transferred the page in the background, provided that the number of pages needing to be so transferred did not exceed the size of the buffer. If the buffer became full, the current process was suspended until there was at least enough space for a single page. To be fair in the analysis, this buffer was considered to be part of the total amount allocated as main memory, thus making the buffer size bigger removed space from the main memory area. Unfortunately, the buffer had a negligible effect on the overall performance (less than 0.1% generally). It is therefore recommended that data to be saved from memory to Flash occurs in the foreground, as the circuit complexity in supporting write suspension to allow reads from the Flash outweighs the advantages. Routines which do not access the Flash store can still be executed in parallel with the transfer (*e.g.* some kernel routines). There was always considered to be a page in the Flash store which could be written to immediately.

The analysis was performed on four benchmarks, VirTEX, DeTEX, zoo, and fig2dev. The simulator itself was constructed using SHADOW. The cost of loading a page from Flash was put at 1000 processor cycles, with the cost of saving a page at 10000 cycles. These figures were derived from the number of bytes per page (4096), divided by the width of memory. Since the array is to be used with DAIS-256, the width was also set to 256 bits. If the array was to be used with a smaller-width processor, then it would be advisable to use memory interleaving to maintain a 256 bit wide main memory design. Reducing the width to 64 bits had a considerably detrimental effect on the overall performance (*i.e.* three to four times worse[4]), implying than the extra complexity in using a wide memory size would be a worthwhile investment.

To offer some degree of comparison, the performance effects of replacing the Flash store with disk were also analysed. Here, the time to transfer a page between the memory and disk was set to 5 ms, and with a 50 MHz machine, this corresponds to 250000 cycles. It is impossible to map a page of the drive into memory directly in the same way as was possible with the Flash array, and thus a page which is currently held on the disk must be loaded into memory first (independently of whether the access was a read or a write). The analysis ignores the effects of multipage transfers between disk and memory, as this spectulative type of transfer has little advantage in large object stores where object access is highly dynamic. Work has been made in clustering related objects together into the same disk page, but the success rate is highly dependent on the application.

It could be argued that the 250000 cycles needed to transfer information between memory and a disk store would be used by other processes. This is only partially true, as in general there are not a large number of processes which run at the same time, and that object faults are also possible within

[4] But still significantly better than disk-based storage.

all of the other processes during the time the system is waiting for the first transfer to complete.

The simulator was given a range of pages which were used as main memory, and it produced figures giving the number of cycles used in loading and storing memory pages. The percentage of memory available with respect to the amount needed to run the program was calculated, as was the percentage overhead (number of cycles used in memory transfers against the number needed to execute the program in question). Figure 8.3 shows the results for the Flash-based store, while figure 8.4 depicts the results using the disk-based store. Note that the second graph has an order of magnitude more range depicted for the overhead axis.

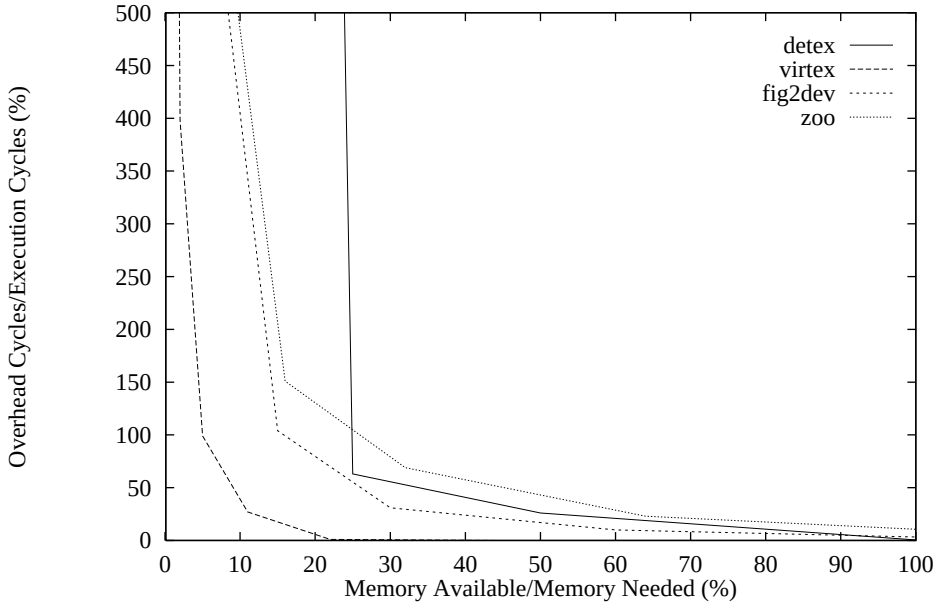


Figure 8.3: Results of the Flash-based Simulations

In the results, the overhead measured at the 100% memory available line is the overhead in loading the predefined data into memory (*e.g.* instructions, constants, *etc.*). In DeTEX, zoo, and fig2dev, the run-time of the benchmarks is small, and thus in comparison the overheads in actually loading in the instructions and initial data is high. However, the Flash store still outperforms the disk store by a considerable margin. VirTEX is the largest of the benchmarks, taking tens of seconds to complete even when running outside SHADOW. In this program, the initial load time is small in comparison to the run time. With a disk store, reducing the amount of main memory available to this benchmark significantly affects its performance. With a Flash-based design, VirTEX runs only half maximum speed even with only 5% of the amount of main memory which it actually needs. This is highly significant, since if scalable it should mean that an application which needs, say, a 100 MBytes of memory to run completely in memory will run at half maximum speed in a Flash-based system with only 5 MBytes of main memory.

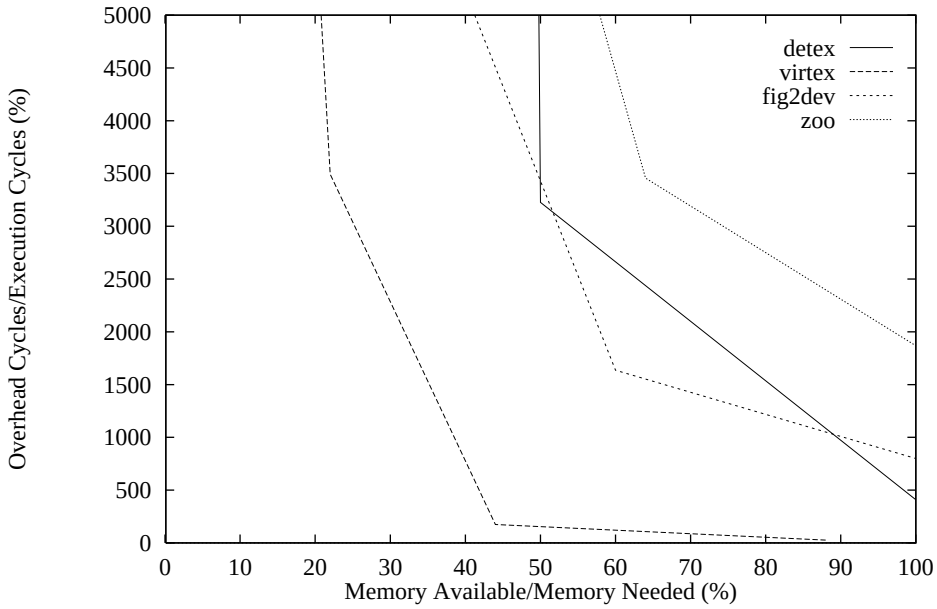


Figure 8.4: Results of the Disk-based Simulations

8.6 Conclusions

Flash-based storage, although currently expensive when compared to disks, is over time becoming more attractive for general-purpose applications. The performance gains to be had are significant, both in terms of reduced data-access times and with new algorithms feasible only with a direct-mapped store (*e.g.* searching and management of high-performance databases containing GBytes of data).

In an object-store, the loss of any single object can result in many more objects being made inaccessible. To minimize information loss, some form of redundancy is required. RAID techniques, although useful in block-based devices such as disks, do not offer the ability to automatically recover data without loading in the entire block surrounding the failure. ERCA proposes a methodology by which automatic recovery schemes can be created, such that a single card failure in the store can be both detected, corrected, and the requested data returned in a single read action on the store. Using this scheme, a number of initial ERCA-based models have been constructed, including ERCA 4 which also allows for double card failures to be detected.

It has been suggested that the requirement to use 100% perfect Flash chips in the store are one reason for the store's resulting cost. One way to avoid the need for such perfection is to use ERCA to automatically repair single card failures, and thus a store with only a maximum of one failure in each array row can be accessed as if no errors exist. Under ERCA 4, the facilities also exist to detect a second error occurring on a row, and for the data contained therein to be transferred to a more reliable area of the store. Through a combination of Flash management and ERCA, it should be possible to maintain a highly-reliable store even with the use of Flash components with less than perfect construction.

In comparison with a disk-based store, Flash stores result in significant speedups. With a disk system, it is normally considered essential to allow as much of an application to exist in main memory as possible, in order to avoid `thrashing'. The analysis of a number of benchmarks indicates that applications with run-times greater than a few seconds incur serious performance degradation once the available memory is decreased beyond 40-60% of that required. In a system using Flash, the critical requirement for main memory is significantly lower. During a simulation of TEX, the program ran at half speed with only 5% of the needed memory available. With disk, the same speed was not possible below about 45%, while at 5% the program took many orders of magnitude more time to run.

Provided that the cost of Flash storage is reduced as predicted by the economic investigation, and perhaps through the use of non-perfect devices, then this technology could easily spell the end of the disk-based storage medium. In relation to DOLPHIN, it offers an excellent basis for providing object persistence.

An interesting next step in this research would be to combine the performance and reliability of a Flash store with the cost effectiveness of disk storage. Flash could be used to cache frequently accessed or rapidly changing data (*e.g.* transaction files and shadow pages), while disks could be used to hold information which is rarely accessed by the current application.

Chapter 9

The Process Environment

Every process in DOLPHIN executes within the network-wide object address space. In this way, all objects are directly addressable from any process in the system. This differs radically from the more traditional Unix-like model where each process executes within a unique virtual address space. A per-process address space protects other processes' data by rigidly separating each processes data-space, while an object-oriented approach prevents malicious changes to data by making OIDs unforgeable.

A Unix process is often referred to as a *heavyweight thread*, where a thread implies an executing chain of instructions (*i.e.* the executing process in Unix). Some Unix implementations (*e.g.* Sun's lwp library [Sun, 1990] and Sequent's parallel library [Seq, 1987]) also support multiple *lightweight threads*, which run within a single heavyweight thread. These threads execute within the same address space, and therefore can access the memory of any other thread within the same process group.

A lightweight threads model has a number of advantages over heavyweight Unix-based processes:

- The application programmer has more direct control of thread scheduling. For example, it is possible to specify threads which, when runnable, entirely block other threads.
- Threads synchronization is cheaper to implement in shared memory than that possible using kernel-based mechanisms. Kernel routines are normally executed in a kernel mode, which requires a context-save to enter and a context-load to leave. These routines involve physical memory accesses (rather than virtual memory), and may require processor cache flushes or cache invalidation.
- Inter-thread communication can be achieved by passing pointers to messages, rather that physically copying data between address spaces.
- Lightweight thread context switches do not involve changes in the virtual memory mapping function, as all threads share the same virtual memory. This can result in a lower context-switching overhead than that of a heavyweight thread.

The process model used in DOLPHIN is very similar to the philosophy of Unix-based lightweight threads, and should exhibits similar advantages. Its scheduling algorithm should allows a greater degree of programmer control then that typical of heavyweight threads scheduling. Inter-thread communication can be performed via shared objects. Thread synchronization can be implemented using object locking combined with shared objects. However, context switches may require some virtual mapping maintenance, *e.g.* where object locking is implemented using a combination of object descriptor modifications and virtual memory protection. Fortunately, most threads will have few locks (they only have to lock something to protect it from being shared), and therefore the overhead in the thread context switch should be low.

One of the main objectives in the construction of DOLPHIN`s process manager was to write all the code in a portable language. This deviates from the standard approach in kernel design of writing the task control code in native assembler. The reason for choosing a portable language was to avoid any dependencies on the underlying processor's instruction set, since such dependencies would limit the speed at which DOLPHIN could be retargetted to machines based on processors other than DAIS.

9.1 MultiThreads

The first implementation of DOLPHIN's lightweight threads package (named MultiThreads), was written to run in a Unix environment. This allowed the package to be debugged in a fully equipped programming environment. In order to avoid creating dependencies for a Unix-like environment, it was decided to use as few system calls as possible. In fact, only four calls are used; **setjmp**, **longjmp**, **sigvec**, and **setitimer**.

9.1.1 Saving and Restoring Context

To allow multiple threads to execute direct co-operative scheduling, some way must be found to save the current processor state of the currently executing thread, and then load in the state of the next thread to execute. One way of doing this is via the standard C function **setjmp**. This saves all the necessary information to allow its sister function, **longjmp**, to restart execution at the point of the previous **setjmp**.

setjmp and **longjmp** were originally conceived to allow error recovery routines to be written by the programmer. However, provided that no attempt is made to return from the routine which made the **setjmp** call before the **longjmp** call is issued, then these functions can be used to directly store the processes' internal state. Context switching can be achieved by first saving the current thread's state, and then **longjmp**-ing to the previously saved state of the next thread to schedule, provided that the states were saved on separate stacks. The only complication is in setting up a stack when the thread is first created, and this is in turn the only factor of MultiThreads which is not completely portable. The problem is that no two manufacturers can decide on a standard way of laying out the **setjmp** buffer structure (known as *jmp_buf*). Fortunately, the structure of jmp_buf can found either in the manual, by examining the setjmp.h include file, or by investigative programming. Note that **setjmp** does not

necessary save floating-point registers (exactly what is saved is implementation specific). In DAIS this is not a problem, since there are no floating-point registers present (floating-point calculations are performed using the general-purpose registers). In addition, this is generally not a problem for any other processor, since all compilers currently examined save the floating-point registers onto the stack before calling any library routine compiled within a separate module, and thus every floating-point register is saved by the compiler prior to a call to **setjmp** or **longjmp**.

9.1.2 Scheduling Threads

MultiThreads contains a thread scheduling system. This runs every so often in the background, and decides when the currently executing thread will be replaced by another. Without a scheduler, task-switching would have to be visible to each thread, with the current thread physically selecting both what thread should be executed next and how long the current thread should continue executing.

In order to regularly reschedule threads, the scheduler operates off an interrupt. This interrupt must occur frequently enough to meet any real-time constraints which may exist, but the more context switches which occur per second, the higher the percentage of CPU time will be spent performing context switches. In Unix, a regular interrupt timer can be activated using **setitimer**. When the timer expires, a signal is generated (SIGALRM), which can be trapped using **sigvec**. The event handler is designed to use the currently executing thread's stack, and thus a switch by the scheduler to another thread is accomplished by saving the scheduler's state on the preempted thread's stack, and then jumping to the scheduler instance which exist on the next thread's stack. The multiple instances of the scheduler actually make the scheduling algorithm easier to implement, but can be confusing to think about. Figure 9.1 gives a graphical representation of a possible scheduling scenario.

Within this figure, a number of individual stacks can be seen. In fact, only the original Unix process' stack can be grown automatically as needed (*i.e.* as in standard Unix processes). All other stacks are of fixed size, so the application writer must ensure that these stacks are of sufficient size to hold all necessary information. Although stack overflow can not be detected immediately without virtual memory control (not generally available in standard Unix), it is detected in software by the scheduler. If a stack overflow does occur within any thread then that thread is destroyed.

The scheduler is constructed hierarchically, using *elements* as a building block. An element can either be a *thread* or a *group*. A group element can contain any number of elements, and if any of these elements are themselves groups then these too can contain further elements. Each group is linked to a particular scheduling algorithm. When the threads package is first initialized, there is one group (DEFAULT_GROUP), and this contains one thread element (main). The DEFAULT_GROUP is scheduled by PREEMPTIVE_SCHEDULER, and in this design the element with the highest priority is selected first. If there is more than one element with the high priority, then the element found first in the scheduler's internal list is selected. If the element selected is a group, then the scheduling algorithm for that group is used to select a element from the list assigned to that group. This process continues until a thread is selected by the schedulers.

Another scheduling type exists in MultiThreads (although more can be added by the user), that of

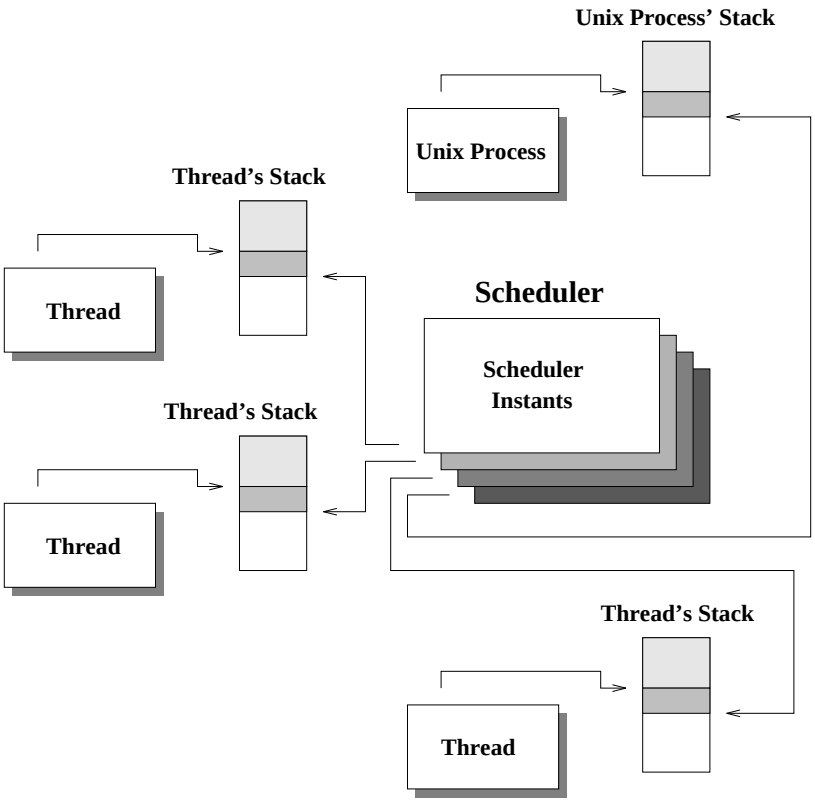


Figure 9.1: Overview of Thread Interaction

TIMESLICED_SCHEDULER. This scheduler selects elements from its internal list in such a way that, over time, all runnable threads assigned to that group receive:

$$\frac{\text{Element's Priority}}{\text{Total of all that Group's Priorities}} \times 100\%$$

of the CPU time supplied to that group.

This hierarchical view of the scheduling algorithm allows powerful process models to be implemented quickly and efficiently. It also means that thread scheduling in one group is logically isolated from all other groups. It is therefore possible to use a TIMESLICED_SCHEDULER at the top kernel level of a system, which schedules one group element per user. Each user can then define their own scheduler, process hierarchy, *etc.*, independently from the kernel and all the other users. An example of the CPU-time division amongst multiple schedulers is shown in figure 9.2.

In the PREEMPTIVE_SCHEDULER, only the first element of a set of elements of equal priorities will ever be selected to run (unless the first element is somehow suspended). It may be desirable to reorder the set, so that the first element becomes the last element, with all other elements shifted along (*i.e.* second element becoming first, *etc.*). This can be achieved via the **mt_switch(***group***,***prio***)** primitive. The *group* parameter specifies the id number of the scheduling group within which the set is stored, while *prio* selects the priority to reorder.

Each element can exist in one of seven states; NEW, ACTIVE, SLEEPING, HELD, MONITORED,

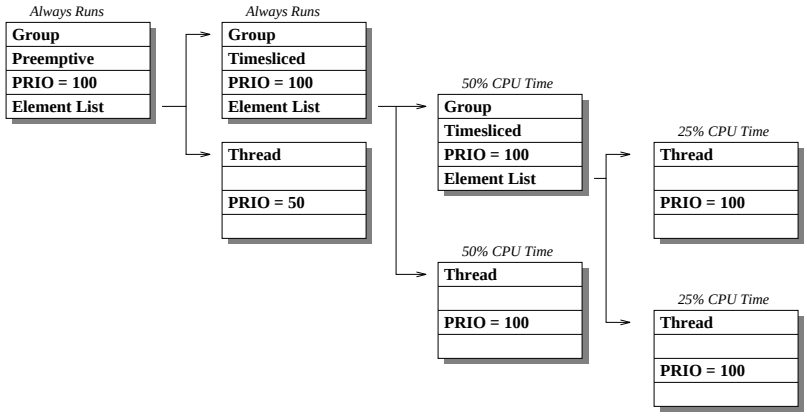


Figure 9.2: An Example of Multiple-Schedulers

WAITING, and DELETE. If a thread to be scheduled is in state NEW, its stack, program counter, and parameters are set up, and its state set to either ACTIVE or HELD (specified at thread creation time). This is done by the scheduler, rather than the parent thread, since in practice an optimizing compiler may reorder instructions to after the stack has changed, which may result in a segmentation fault. An element marked as ACTIVE can be executed normally. Threads can be put to sleep for a period of time using the **mt_sleep** primitive, and such threads are marked as SLEEPING. Elements can also be held (*i.e.* suspended indefinitely) by using the **mt_hold** primitive, and their state is set to HELD. If a thread is blocked in a monitor or in the exception handler, then its state is either MONITORED or WAITING respectively. Lastly, any element which is to be destroyed is marked as DELETE. Threads can not be deleted by the scheduler instant running on that thread's stack, and instead such threads are deleted by the next scheduler instant to run. An element in state HELD or SLEEPING can be restored to state ACTIVE by calling **mt_restart** with the element id in question.

9.1.3 Critical Code Protection

It may be desirable for certain code segments within application to be executed within which a thread switch can never occur. One example of this would be **malloc**, where it is essential that the internal structures which maintain the free list of unallocated memory spaces always remains in a consistent state. Thus, no two instants of **malloc** can overlap.

One way of providing support for critical code protection would be to switch off the scheduler while such code is being executed. This is achieved through the use of **mt_critical** and **mt_endcritical**. In order to support modular code, these two primitives can be nested, guaranteeing that no switch will occur until the end of the outermost **mt_endmon**.

The monitor is implemented by using two global variables. The first of these, *critical count*, is the current monitor depth (*e.g.* zero if the thread is not within a monitor). If the scheduler is initiated and discovers that the critical count is greater than zero, then the second global variable, *time expired* is set

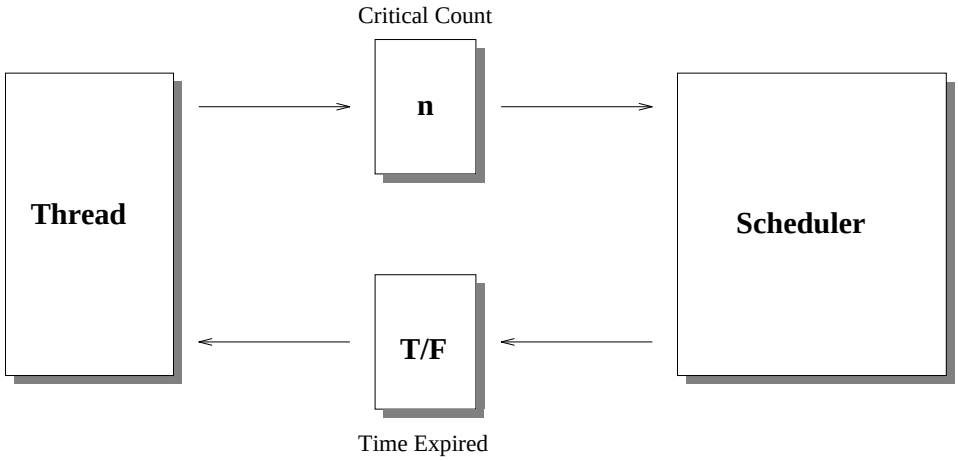


Figure 9.3: Scheduler Suspension for Critical Code

to true, and the scheduler suspends itself until invoked by the protected thread. When the thread executes the last **mt_endcritical** call, a check is made on the *time expired* flag. If the flag is false, then the program continues normally, otherwise the scheduler is invoked. Figure 9.3 graphically demonstrates this interaction between scheduler, thread, and monitor state.

This scheme means that a thread could spend more than its allotted share of execution time by performing a complex system call. A much more preferable scheme is to use monitor-based protection. A monitor is created using **mt_makemon**, and is entered and exited by **mt_monitor** and **mt_endmon** respectively. These calls manage a code monitor, such that only a single thread can exist between the start and end markers on a monitored area. If a second thread attempts to enter a monitored area, then that thread is held until the first thread exits that area. It is also possible to make a conditional monitor entry; the primitive **mt_cmonitor** will return 0 if it was unable to enter the monitor (*i.e.* instead of blocking as in **mt_monitor**, or 1 if it was. A monitor should be used in preference to a critical code area, since this permits a greater degree of fair scheduling.

9.1.4 Supporting Input/Output

The main problem with a scheduler executing transparently from the kernel-based heavyweight-thread scheduler is that involving blocking system calls. If a system call blocks (*e.g.* a read is performed from the keyboard, but no key-presses are waiting) then the entire heavyweight thread is suspended until the call un-blocks. This has the effect of blocking all the lightweight threads within the Unix process as well, rather than the more desirable result of just blocking the thread which made the system call.

In DOLPHIN, this is not a problem since all system calls can be written with the understanding that multiple threads exist, and the scheduler is alerted when the currently executing thread is waiting on an event (thus giving the opportunity to select a new thread). Under Unix, however, there is no perfect solution except for rewriting all the system calls, which would be a time-intensive task.

In Unix, most process blocks are caused by I/O. These are generally controlled by **write** and **read** primitives. In many Unix systems, a prediction of whether a read or write will block can be given

by the **select** routine. It is generally also possible to indicate that, whenever an I/O channel changes state (*e.g.* data arrives or its write-buffer becomes empty), it produces the SIGIO event. Threads can be instructed to wait for an event by using the **mt_waiton** primitive. It is therefore possible to construct complex arbitration systems by using a combination of **select** and **mt_waiton(***SIGIO***)**, allowing **read** and **write** to be implemented such that only the calling thread becomes blocked.

As an example of this, consider a network communication library which, when given a message by a thread, blocks that thread until a reply to the message is received by the library. Once received, the reply is passed back to the blocked thread and the thread restarted. The library would be controlled by a network handler, which would wait on a SIGIO event (linked to incoming network packets). If a packet is received, the hander wakes up, identifies which thread is to be the packet destination, and restarts that thread. A diagram containing the data and control links needed to implement this library is shown in figure 9.4

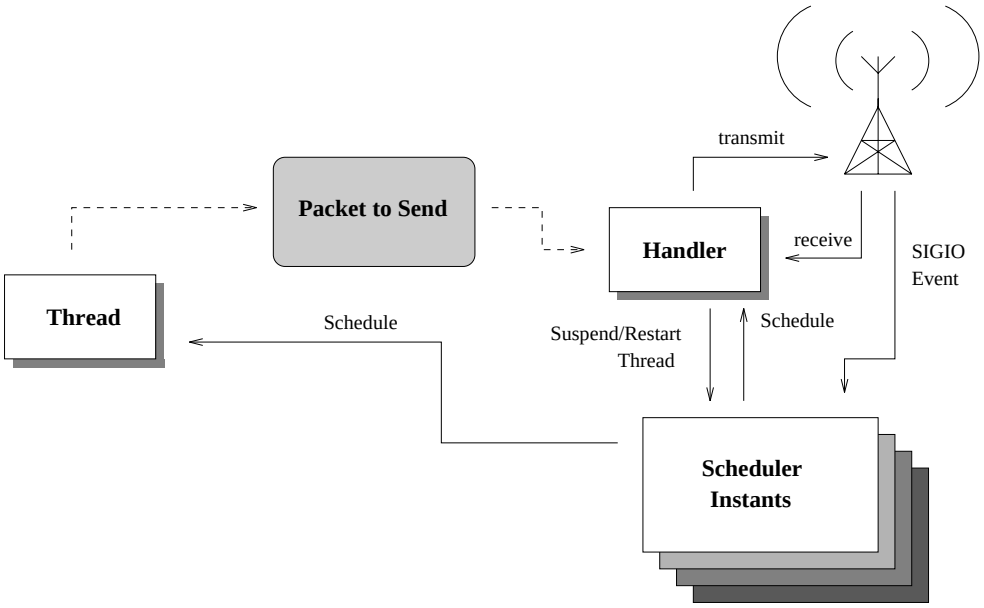


Figure 9.4: Example of a Network Communication Handler

9.1.5 Signal Handling

MultiThreads contains the ability to handle Unix signals using a thread-based philosophy. In MultiThreads, signals are known as *exceptions*. These exceptions can be generated by interrupts, the Unix kernel (through kill signals), and by other threads (using the primitive **mt_signal(***exceptions***)**).

If a thread wishes to be informed about a particular set of exceptions, it must issue the primitive **mt_exception_handler(***exceptions***)**. This specifies a set of exceptions which will be passed to the thread which issued this primitive. Whenever an exception occurs, a search is then made for the highest priority handler which has indicated a desire to receive that exception. Once a handler is found, the exception is queued for that thread. A handler can wait on exceptions by using the **mt_waiton_exception**

primitive. This removes the first exception from this handler's queue, or blocks if none are currently queued. Once a handler has received an exception, it may either accept it using **mt_accept_exception**, or pass it to the next-highest priority handler's queue using **mt_forward_exception**.

The system is designed such that, once a particular instance of an exception has been forwarded, that particular handler can never again receive it. It is also designed such that no handler can ever miss an exception, even if it is busy doing something else, since the exceptions are queued until needed. If there is no handler for a particular exception, or if all handlers have forwarded the exception, then the exception is silently discarded.

9.1.6 A Programming Example

Given the MultiThread subsystem, just how easy is it to construct more complex primitives? In this section, the construction of a synchronized message port is examined. Here, the task is to provide two primitives, **send** and **restore**:

1. **send** inserts a message into a particular message port, and then suspends the current thread until the inserted message has been accepted by the receiver.

2. **receive** waits for a message on the selected port.

The structure used in implementing a message port is split into three fields; *owner*, *message_list*, and *sender_list*. The first field contains the thread id of the port reader if the reader is currently waiting for a message, otherwise this field is zero. Field two contains a linked list, representing a queue of messages. The third field forms a linked list of the message-sender's thread-IDs, stored in the same order as that found in field two. The code to perform the **send** and **restore** primitives are presented in pseudo-Ada form in program 9.1. Critical code protection is used instead of monitors in order to maximize performance and minimize program complexity. However, the code could easily be written to use monitors if this was required.

In the code, use is made of Ada-style comments, which in turn represent one or more lines of code which are not included in the program listing. The code in question are standard linked-list manipulation routines, and have been excluded in order to increase readability. A reference guide to MultiThread's primitives is given in appendix C.

9.1.7 The Scheduler Implementation

The TIMESLICED_SCHEDULER algorithm can be implemented in a number of different ways. In most Unix kernels, the algorithms used are geared towards having few runnable processes at any one time. This may not be the case with lightweight threads. Indeed, popular kernel-based algorithms (*e.g.* Linux and BSD386) the priority of a process has little effect on the frequency of execution, but instead effects the length of time which a process will run before the next process is scheduled. This is sufficient for a small range of priorities, but once the range gets large much of the priority information must be ignored. As the number of runnable processes increase, then the time between subsequent runs

```
SEND:
        mt_critical();
        -- Append message to end of message_list
        if ( message.owner != 0 ) -- Owner is blocked
          mt_restart(message.owner);
        else
          id = mt_thread_id();
          -- Append this thread id into end of sender_list
          mt_suspend(id);
        endif;
        mt_endcritical();

RECEIVE:
        mt_critical();
        if (message.data == NULL) -- No Data Waiting
          message.owner = mt_thread_id();
          mt_suspend(message.owner);
          mt_endcritical();
          mt_critical();
        endif
        message.owner = 0;
        -- Extract the data from head of message_list
        -- Extract id, the id at head of sender_list
        mt_restart(id);
        mt_endcritical();
```

Program 9.1: Program to Implement Synchronized Messages

of a particular process increases in proportion, independent of its priority. This effect can be clearly seen when a Unix machine is heavy loaded; interactive programs begin to run in spurts rather than simply degrade slowly. This type of algorithm is undesirable for the MultiThreads system, since it is likely that many more threads will exist than the number of runnable processes which exist in a typical Unix environment.

MultiThreads instead implements the TIMESLICED_SCHEDULER using a probability based scheme. If we define T_r to contain the set of all runnable threads, with the priority of thread i given by ξ_i, then ideally, given that each time a thread is selected to run it does so for a fixed period of time, the probability P_i of thread i being selected by the scheduler should be given by:

$$P_i = \frac{\xi_i}{\sum_{j \in T_r} \xi_j} \tag{9.1}$$

This is in fact the equation used by the scheduler. When the scheduler is deciding which thread to select, it first generates a random number. This number is created in the range 0 to the total of all the runnable element priorities held within the scheduler. Then it is a simple matter to step through the list of runnable elements, subtracting away each element's priority from the random number until the number becomes less than or equal to zero. Once this has occurred, the element which caused this condition is the one to be selected next.

The main problem with this probabilistic scheme is that the search for the next element to schedule involves a linear scan through the active element list. In traditional schedulers, the first element of the list is always used, which is moved to another list once that element had executed for its timeslice. The worst case search depth for our scheduler is $n = |T_r|$ (the number of runnable threads). Provided that the threads have a distributed priority, and that the list of runnable threads is ordered $\xi_i \geq \xi_{i+1}$, then the search depth is $<< n/2$.

MultiThreads, when used properly, will have a tendency to split elements amongst the available scheduling groups. With hierarchical timesliced scheduling, the search-depth will be further reduced. For example, consider an arrangement of m timesliced scheduling elements each holding n/m thread elements with distributed priorities. If each of the schedulers has a single parent which itself is a timesliced scheduler, then the search depth is on average $<< (n/(2m) + m/2)$, even with all the threads of equal priority. Further layering of the schedulers is possible, allowing search time to become logarithmic in terms of n.

In systems where search time is a major factor, remember that the PREEMPTIVE_SCHEDULER has a search depth of one. For programmers who strive for a better scheduling algorithm, they can add their own scheduling code simply by passing a pointer to it in place of the scheduler field at thread creation time.

9.1.8 Analysis

The performance of MultiThreads is hard to quantify, as areas such as timesliced scheduling overhead depends on both the number of threads and also how they are mapped onto the available schedulers. All that can really be said is that under heavy loading, the runnable processes will still get a proportional

share of the available cpu time with timely (but short) periods to run. In traditional list-based schemes, heavy loading means that although the threads still have a fair share of the cpu, the periods between subsequent run instances can be significant.

In this section, some of the measurable information regarding MultiThread performance is presented. These are compared against the results obtained from another lightweight threads package, both running on a SPARC IPX. This second package, Pthreads [Mueller, 1993][1], has the aim of implementing POSIX compatible threads. This is achieved through a combination of C code and low-level assembly language. A POSIX thread interface could easily be implemented using the MultiThreads kernel, but this would not be useful in DOLPHIN, since the needs of POSIX threads diverges from the needs of an object-oriented kernel is certain areas (*e.g.* signal handling, stack management, *etc.*). The results of this comparison is shown in table 9.1. This table also includes some comparison information concerning the performance of the IPX's Unix kernel. Measurements are made using dual-loop analysis.

Table 9.1: Performance Metrics

Performance Matrix	Time (μsecs)	
	CThreads	MultiThreads
Enter and Exit a Critical Region	1	0.6
Enter and exit the Unix Kernel	18	
Enter and Exit the Threads Kernel	0.4	0.6
Create a Thread	12	10
Setjmp/Longjmp pair	29	
Thread Yield	37	34
Unix Process Context Switch	123	
Thread Signal Handler (internal)	52	37
Unix Signal Handler	154	
Thread Signal Handler (external)	250	193

The metrics given include the overhead in using setjump/longjump. This gives an indication for the minimum time to make a context switch. The metric for entering the Unix kernel was taken by timing a **getpid()** process call. In both systems the time to create a thread does not include stack allocation, since in Cthreads this is from a pre-allocated stack cache and MultiThreads creates the stack when the thread is first scheduled. Thread signal handling of external signals (*e.g.* SIGIO) is made using standard Unix signal handling routines in both implementations. This time should be reduced significantly with the replacement of the signal handler with an interrupt handler written specially for DAIS. Incidentally, Mueller also compared Cthreads against Sun's lwp, and showed that Cthreads has a significantly higher performance.

The results of this analysis suggest that MultiThreads outperforms Cthreads in all categories except kernel entry and critical region protection. Cthreads uses atomic instructions to implement critical protection, while MultiThreads uses two global variables to implement semantically identical (but slower) operation. MultiThreads uses this algorithm to protect entry into the thread kernel, which

[1] Work on Cthreads was conducted independently from MultiThreads, with an initial publication of MultiThreads [Russell, 1993] occurring slightly before Mueller's.

accounts for the loss of performance. While atomic instructions are faster, the MultiThreads algorithm is machine independent and written entirely in C.

Part of the reason for MultiThread's performance advantage in the other metrics lies in the signal handling. Pthreads is constrained to use POSIX signal handling, which means that each thread must have its own handler. In MultiThreads, the handler is global to all processes. If there is a requirement to implement handlers for each process individually, there is no reason why this can not be implemented on top of the available routines, thus not incurring any overheads unless it is particularly required.

Overall, the implication is than MultiThreads does not suffer unduly from being written without the benefit of assembly code. This serves as an indication that non-portable code need not necessarily lead to a higher performance when compared to a portable version.

In the unix implementation of MultiThreads, system calls need not be protected in a `blanket' fashion. If a signal occurs while a system routine is in progress, it is aborted automatically by the operating system. When the signal handler returns, the system call is then automatically restarted. Only those routines which contain side-effects need special management (*e.g.* strtok uses an internal static variable). In a multiprocessor unix environment, the current implementation of MultiThreads would make use of only a single processor. The main reason for this omission is the lack of standards for controlling and communicating between multiprocessor processes.

9.2 The DOLPHIN Implementation

Mapping MultiThreads from Unix to DOLPHIN removes much of the uncertainty in the correctness of resource management. In the Unix system, monitors are deleted when the creating thread dies. This is no longer a requirement, since of the monitor information is stored within an object, then the monitor can be automatically deleted by the garbage collector once the threads in the system have lost all pointers to that monitor.

The element IDs, used to identify an element by the programmer, could be set to be the OID of the element's data description (*i.e.* the object where the scheduler holds all relevant data on that element). There would be no problem with deleting elements, since a deleted element's data will persist until all OIDs to that data have been abandoned. There is no problem with element id reuse, since while an id is remembered by an application the element's data OID can not be reused. In a system with large OIDs, this may not be attractive. A scheme could be designed to use short thread IDs, which are in turn hashed in a table to locate the thread's stack object. If the garbage collector detects that the only reference to the stack is through the hashing table, then destroyed thread's stacks could still be detected and destroyed without the overhead of long thread IDs.

A thread's stack can be allowed to grow automatically when needed by introducing a new object type into the object manager, that of *stack objects*. These objects start with a small valid indexable range (say p indexes), accessed using an index of between q (the maximum object index) and $q - p$. Accessing this object with an index less than $q - p$ causes p to grow as required, increasing the object's size dynamically. This can be easily implemented in the object management system. Contraction of stack objects has not been considered an important issue (and is complex to control automatically),

although having the ability to set a maximum stack size is considered essential (to catch infinitely-recursive functions).

The search-depth problem of scheduling a large number of runnable elements within a timesliced scheduler is minimized, since it is likely that each user on the system will be given their own preemptive scheduler to hang their elements from, with this scheduler's parent being a timesliced scheduler controlled by the operating system. Thus the operating system's timesliced scheduler has a search-depth linear in the number of users on the system. If a particular user creates a large number of elements, then only that particular user's search-time is effected.

Note that all DOLPHIN system calls must understand that the thread scheduler may run at any time, and if necessary protect themselves against being rescheduled.

9.3 Conclusions

This chapter has demonstrated that a threads library written entirely in C is achievable and not significantly inefficient. This library supports all the standard thread requirements of critical code protection, scheduling, thread creation, sleeping, and signal handling.

MultiThread diverges from the POSIX signal-handling mechanism, since not only is it not generally needed in thread code, but is also produces significant overheads in the signal handling and context switching routines of the thread kernel [Mueller, 1993].

It is the intention of the author to release the Unix version of the MultiThreads library as freely distributable code. This should allow programmers on Unix systems to enjoy the flexibility of programming with threads, without incurring the non-portability traditionally related to thread use.

With respect to DOLPHIN, MultiThreads offers a fully-functional process handling scheme to both operating system and application program designers. It should contain all the routines needed to construct thread-based code within the DOLPHIN environment.

Chapter 10

Network Communications

In order to allow the users of DOLPHIN to experiment with networking, it is necessary to provide them with an interface to the network hardware. This interface should permit the user to implement a number of object-networking strategies both simply and efficiently. Clearly, this requires an interface which co-operates with MultiThreads (*e.g.* when a thread waits for a network packet, another runnable thread can be executed until the packet arrives).

It is not only the interface to the network package which is important, but also the protocol under which it sends data. If a machine needs an object, then the request for that object should be sent reliably. When the object is actually sent, it too must be sent reliably, otherwise object data could be lost. Such reliability could be built onto an unreliable protocol by the user, but since it is likely that reliable data transmission will be part of many object networking algorithms, reliability would be best implemented at the kernel level. Another factor is connection vs connectionless protocols. Under a connection-based scheme, a connection between communication end points must be constructed via handshaking packets before any data can be sent[1]. Once all data has been transmitted, the connection should then be terminated (as maintaining a connection normally consumes system resources).

Within this chapter, the requirements for the networking protocol are examined, and then compared against two standard network protocols; datagrams and sockets. Both of these can be found in almost every Unix system available, and many non-Unix system implementations also support them. It is also argued that the facilities provided by both of these approaches are either inadequate or overly constraining for use in supporting a reliable object-oriented heap on a network. An efficient networking protocol, MultiPackets, is then presented. This scheme was designed from the requirements for implementing efficient object management presented in the early stages of the chapter. The performance of MultiPackets is also examined, in a comparison with both datagrams and sockets.

The MultiPackets protocol is also applied to multicast packets and data routing, with an aim of

[1] Although some protocols allow the connection packets and the data to be sent within a single packet, provided that the data can be sent atomically.

reducing the number of data packets required to achieve common network-management instigated network operations.

10.1 Object Management on a Network

There are a number of different ways to implement object networking. Currently, there is no clear indication as to which method is the ideal, mostly because the design goalposts change almost with every application. Take a multiple-reader/single-writer approach, where read copies are invalidated when a write is initiated. Some applications can be expressed in ways so that their shared data access needs are mostly read-only, and this type of application runs reasonably efficiently using this networking model. Other applications perform random read/write actions to shared memory, which can lead to invalidation/request thrashing (and therefore performance degradation) during execution under the multiple-reader with invalidation scheme. Some of the general types of networked object management are listed below:

- Multiple-reader/Single-writer. Either a single write copy or n read-only versions of object data exists. To perform a write when possessing only a read copy, ownership of the data is first obtained, and then invalidation requests are sent to all of the read-copy holders of that data. Once all copies have been deleted, the write can be performed by the owner. Subsequent requests for read copies of the data are directed to the owner, and the first such request converts the owners read/write copy to read-only.

- Multiple copy modification broadcasts. All nodes interested in a piece of data possess a copy of that data. Any modifications to the data are sent to a centrally managed message sequencer, which then broadcasts the changes to all nodes with the particular data. In this approach, the node which made the changes should also receive a request to change the data from the sequencing node, and thus this message serves as an acknowledgement for the writer. This scheme frequently makes use of broadcast packets. These packets are received by all nodes on the network, each of which must use processing time to examine every broadcast message; this reduces node performance. The sequencing node is the main bottleneck in the design. Amoeba [Tanenbaum *et al.*, 1992] is an object manager which uses this scheme. Analysis shows that 700-800 shared object writes can be done in a second on standard ethernet, which is poor since this number is a constant independent on the number of nodes or the number of networks employed.

- Central Transaction: Clouds [Dasgupta *et al.*, 1991] uses a number of central object servers, on which all object changes are performed. The node on which a user sits uses its processing power only to request object data or object changes from the central servers. This means that high-speed networks are only required between the servers. Unfortunately, such an approach does not make use of the free processing power in all available machines, but only that found within the servers. Central transaction models are best suited to database-type transactions, and Clouds in no exception. Clouds uses a heavyweight object system, where each object is is fact a whole program (containing instructions, data, local variables, *etc.*). The servers themselves share data

using a one-copy approach, where only one version of object data exists at any one time, and this is transferred between servers as needed.

In DOLPHIN, where the exact network management strategy (or even the network type) is unknown, it is important not to make design decisions which will make certain strategies either impossible or inefficient to implement. Without this flexibility, the aim of providing a true testbed for further research into object systems could be compromised. From the methods examined above, and for maximum flexibility, DOLPHIN's network interface should:

- Support multicast transmissions. Multicast allows a single packet to be delivered to a number of networked machines. This saves of network bandwidth and networking interfacing overhead. Multicasts are useful in broadcasting invalidation messages, or in sending data modifications to object-data holders in the modification broadcast approach without disturbing other nodes. It should also allow multicasts to machines which do not all lie on the same piece of network cable, whether separated by a single router or by multiple or internet routers.

- Support packet sizes of at least 4 KBytes. Object sharing on the network will probably be instigated by page faults, and since DAIS uses a page size of 4 KBytes, it should be possible to transmit an entire page with a single command. Since ethernet packets are only about 1500 bytes, this would suggest that the network protocol should support fragmentation and reconciliation to and from network packets into higher-level data packets. Note that restricting DAIS to a fixed page size may have been an error, as this could restrict the usefulness of the design. In a later version of DAIS, a more flexible approach to defining the page size may prove useful.

- The interface must be compatible with the MultiThreads system, in that waiting for a packet does not cause all threads to become suspended, and that multiple threads running network commands do not cause data inconsistencies.

- The transmissions should be reliable. Re-implementing reliability for each object consistency scheme would slow down development time, whereas implementing reliability at the protocol level would allow optimizations to take place. Such optimizations include piggybacking of handshaking messages on valid data, including data sent with a multicast. It would also be attractive to piggyback handshaking information on any data bound for the target node, not just data being send to the particular thread responsible for the data actually being acknowledged. Such optimizations would be difficult unless implemented on a per-node bases within the DOLPHIN kernel.

- The transmissions should be connectionless and fast. In a connection-oriented system, the initial data transmission is delayed while a logical link is set up. Once created, the link must be managed and eventually closed down. Every open link usually requires some degree of system resources, and as such active links are normally considered a limited resource. Knowing when to close a link down is an added complication, as is recovery when one end of a link is broken. The complexity of connection-oriented protocols should therefore be avoided.

10.2 Standard Transmission Protocols

On Unix systems, the main way of addressing machines on a world-wide scale is through the use of the internet. This is a network addressed using 32 bit addresses. Communicating over this network is achieved via the TCP/IP protocol (Transmission Control Protocol/Internet Protocol). This supports both a connectionless (datagram) and a connection-oriented service. Both of these can be considered to lie in the bottom four levels of the OSI[2] protocol layer (shown in figure 10.1). The layers above this point control when messages are transmitted, what form the messages actually take, and the actions to take when messages arrive. In this chapter, we concentrate on the Transport Layer downwards.

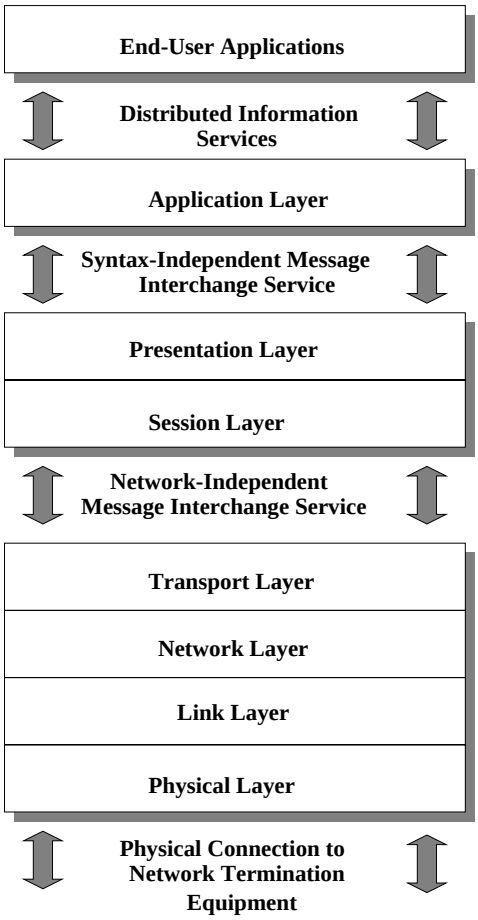


Figure 10.1: OSI Network Levels

10.2.1 Datagrams

Datagrams are one of the simplest form of transmission protocol. It is based on a connectionless, unsequenced, and unreliable transmission approach. A connectionless protocol does not require any

[2] In fact the TCP protocol is not covered by OSI, but instead by DARPA (Defence Advanced Research Project Agency). However, the protocol can still be described as lying between the transport to physical layers of the OSI model.

time to set up a connection between two machines, as data can arrive at any time from any machine without a prior message to indicate that data is about to be sent. Similarly, there is no indication that the last data packet received is in fact the last packet to be sent by a particular application, so there is no semantic interpretation to the idea of closing a data link. This has a number of advantages, in that data routes are stateless and that data can be transferred without the overhead of link setup. Stateless connections require no setup time after a system failure at the data transmitter, receiver, or at one of the routers in between.

Datagrams are unsequenced in that if a particular node transmits two packets one after another to a single destination, there is no guarantee that the first packet will arrive before the second packet. This is especially true if there is multiple routes between the two nodes over which the packets could travel; the first packet could easily have travelled over a longer route than the second. In fact, since the datagrams are unreliable, there is no guarantee that a packet will arrive at all. If an application requires that a packet sent to a particular node must be received by that node (and therefore not lost), then it is the application which must provide such reliability as part of a higher-level protocol.

On the internet, network packets are limited in size to only about 1500 bytes in size. Datagrams posses the ability to split data to be transmitted over a number of datagrams. Where such splitting is required, each datagram is said to be a fragment. Unfortunately, the loss of a single fragment also means the silent disposal of all the other fragments. Since the probability of a lost packet is constant for each fragment, the combined probability of data loss increases with the number of fragments. This normally leads to a higher data retransmission requirement by the application.

Although datagrams have their limitations, they do have a high data throughput. Transmissions are unencumbered by link connect/disconnect protocols, sequencing overheads, or handshaking. This makes datagrams attractive for real-time applications. It is the speed of datagrams which should be preserved in the new protocol.

10.3 Connection-Oriented Sockets

The second type of TCP communication is based on a sequenced, lossless, connection-based protocol. These require a logical connection to be made between source and destination nodes before any data can be transmitted. When the data has been completely transmitted, then this link must be closed. Holding a link open to every possible destination site would be attractive, but each link needs system resources, and in Unix systems there is a physical limit to both the number socket links for each process and, in some cases, to each user.

Each packet transmitted is given a sequence number, which is incremented by one for each packet sent along that link. If the destination node receives a packet with a sequence number greater than than the last packet's sequence number, then that packet is clearly out-of-sequence and in error. A sliding window recovery system is often utilized, where a number of packets can be held awaiting previous packets which has not yet been received. Once this buffer is full, packets are silently disposed of until the missing packet(s) arrive. Socket protocol sends acknowledgements to packets which were received but not disposed of, and thus the transmitter detects packets which have not been successfully

delivered by retransmitting unacknowledged packets after a timeout period. Generally, the timeout is derived from previous retransmissions to that destination.

If a packet acknowledgement is lost, then the transmitter will retransmit (after the timeout) the packet which corresponded to that acknowledgement. When this is received by the destination node, it is acknowledged and disposed of. Such packets can be detected by the fact that their sequence number is smaller than the last successfully received packet.

With sequenced packets sockets lend themselves well to supporting true variable-sized data transmissions. Data sets larger than a single network packet are spread across sequential socket packets. Since all the packets are lossless, reconstructing large data sets is simply a matter of reading out network packets in their sequence number order. The loss of a single packet which is itself part of a larger data set requires only the retransmission of that packet (unless the shifting window has been over-run).

The connection-oriented basis in socket communication is the main drawback in an object-based system. When packets must be transmitted quickly between nodes, the requirement to first make a connection could slow down the data transmission. Since each active link uses system resources, connections have to be closed regularly. Connections may also have to be closed when a new link is required but there are insufficient capacity left in either the sender of receiver machine (*i.e.* there are too many other links still connected). Should either sending or receiving machine fail during the period when a link is open, then the link must be terminated and restarted before more data can be transferred. Links may be terminated not only by endpoint machine failure, but by temporary network failure, routers closing down, or heavy loading on the network which causes buffer over-runs.

From the specification of the needs in implementing object-based networking, sockets support the need for lossless communication and packet sizes of practically any size. Sequenced packets are not a feature of the specification. The requirement for sockets to set up links prior to data transmission is a needless overhead, and should be avoided in the protocol designed for DOLPHIN.

10.4 The MultiPackets Protocol

MultiPackets is a protocol designed to meet the needs of flexible object management on a network identified earlier, while attempting to incorporate the features identified above as being attractive for a network protocol.

The handshaking protocol using in MultiPackets is based on a three phase process. In phase 1, data is transferred to the destination node. This data is uniquely numbered within the network by source address, packet id number, and fragment number. Once a fragment is received, and *ACK* message is sent in response. If the transmitter does not receive an ACK message before a timeout expires, then the fragment is retransmitted and the timeout timer is reset. Once an ACK is received, then the fragment is removed from the transmission queue and an *ACKACK* message is sent back to the receiver. The receiver continues to send ACK messages until acknowledged by an ACKACK. All ACKs received by a node are responded to with an ACKACK message, while ACKACKs received when no corresponding ACK is queued are simply ignored.

In order to minimize the number of ACKs and ACKACKs required with fragmented data packets,

the ACK reply is delayed for a time. If further fragments are received, then these are incorporated within the same ACK message. Similarly, ACKACKs are delayed for a time, and a single ACKACK packet can acknowledge all of a data packet's corresponding ACKs. A zero-error message transfer is depicted in figure 10.2.

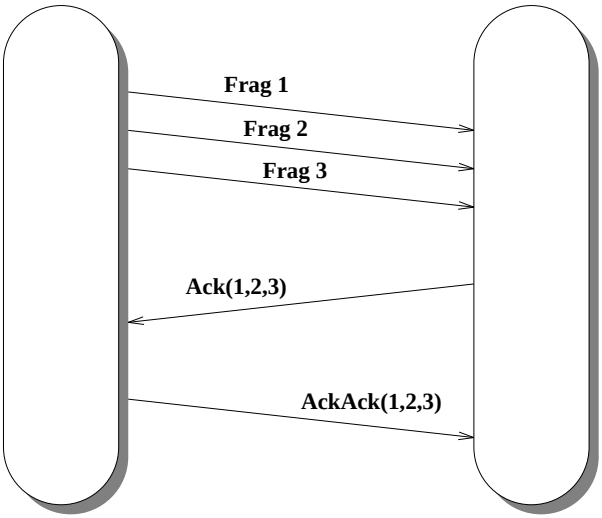


Figure 10.2: Overview of MultiPacket's Transmission Protocol

10.4.1 MultiPacket Retransmissions

There are four possible error scenarios to consider using the MultiPacket's transmission protocol. All possible errors are a function of these four:

1. All the network packets corresponding to a data packet are lost during transmission
2. Some of the network packets corresponding to a data packet are lost during transmission
3. An ACK message is lost
4. An ACKACK message is lost

It is also possible that a packet was received but was corrupted. In MultiPackets, packets failing a checksum test are silently disposed of and therefore can be considered to be lost.

If all network packets are lost during transmission, then once time $PACK_t$ has expired all packets which have not been ACKed will be retransmitted. It is important that this time period be short enough to get the data to the destination even on error-prone connections, while at the same time a long time period allows delayed packets (*e.g.* delayed by network loading or physical distance between nodes) time to reach the receiver. It would be beneficial if this timeout period was calculated at transmission time, based on the results of previous messages sent to that destination.

If some network packets get to the destination, then these will be acknowledged after time period ACK_t with an ACK message. After the $PACK_t$ timeout unacknowledged packets will then be retransmitted. It is therefore important that ACK_t is less than $PACK_t$ at the transmitter minus the round-trip transmission time (including software overheads). In reality, setting ACK_t to between 60-70% of $PACK_t$ forms a good compromise.

If an ACK message is lost, then the ACK will be retransmitted after the ACK_t timeout. To allow ACKs and ACKACKs to be delayed, and since the ACK timeout value has no direct effect on actual data throughput, their timeouts are deliberately long. This has the effect that a missing ACK message will result in the $PACK_t$ timeout expiring before ACK_t. In effect, a lost ACK message will result in a retransmission of the data which corresponds to the missing ACKs.

Finally, the loss of an ACKACK message results in the retransmission of the corresponding ACK message after the ACK_t timeout period. Duplicate ACKs are collapsed if the ACKACK message has not yet been sent, otherwise a new ACKACK message is queued. Duplicate ACKACKs are simply ignored.

The only error which affects network loading or data throughput is the loss of an ACK message. Fortunately, this too is minimized by the use of look-ahead piggybacking, which is discussed below. This mechanism allows ACK messages to be transmitted ahead of time, provided that there is already something else going to that node (*e.g.* a data packet).

10.4.2 Piggybacking

To save on network traffic, MultiPackets makes use of extensive piggybacking. ACK messages to be sent to the same destination as a data packet are attempted to be sent with that data instead. Since ACK messages are requeued after transmission, only those ACK messages to be sent in the next m milliseconds are examined. For implementation reasons, m is currently set to 0.5 seconds, although this could be easily changed if necessary. ACKACK messages are also piggybacked. Since such messages are only queued once, all queued ACKACKs are checked for piggybacking eligibility.

ACKs and ACKACKs can also piggyback on packets which contain no data. This occurs when an ACK or ACKACK timeout expires before it could be successfully piggybacked on an earlier transmission.

For use in the ethernet environment, the largest unit of user data which can be sent in a single MultiPacket packet is set to 1024 bytes, with 1450 bytes being the maximum network packet size after headers and piggybacked data. The maximum packet size should really be 1518 bytes, but for testing purposes MultiPackets uses datagrams as the network protocol, and thus must include the overheads in the datagram header information. In future versions of MultiThreads, use will be made of more low level interfaces to the network[3]. With this in mind, the performance realized by MultiPackets will be lower than that possible using low-level network interfacing alone.

[3] Much of the testing of MultiPackets was performed on Unix systems, where network access at a lower level than datagrams, *e.g.* raw sockets, is restricted to the superuser.

10.4.3 A MultiPacket Packet

Each MultiPacket packet is made up of six component parts; a header and checksum, the destination list, ACK/ACKACK directory, the ACK list, the ACKACK list, and the data itself. These have the following functions:

- Header and checksum. This contains data describing the internal structure of the remainder of the packet, such as number of ACKs and ACKACKs, size of the data part, the data's fragment and id number, and the source's machine address and thread id. There is also a checksum, which serves as a confirmation that the packet has been successful received without transmission errors or buffer-induced truncation.
- Destination list. This contains an entry for each destination machine for this packet. This will either be a single machine for a point-to-point transmission, or multiple entries in the case of a multicast. Each entry contains the network address of a node, and the destination thread id for the data with respect to that node.
- ACK/ACKACK directory. Each entry in the directory contains three fields. Field 1 contains an offset taken from the beginning of the destination list. This offset indicates the machine to which this entry corresponds. The remaining two fields are offsets from the beginning of the ACK and ACKACK lists respectively. Combined together, this indicates which ACK and ACKACK entries correspond to which machine.
- ACK list. This contains a list of the packet IDs and fragment numbers for each packet the sender is acknowledging.
- ACKACK list. This contains an packet id and fragment numbers for each ACK being ACK-ACKed.

Figure 10.3 shows the internal structure of a MultiPacket packet pictorially, including the links from the directory to the ACK and ACKACK lists.

10.5 Performance

In order to examine the performance of the MultiPackets algorithm, and also to allow its use in the DOLPHIN environment, the algorithm was coded into the SubKernel. The complete programming reference for the MultiPackets routines are listed in appendix 10. This has been completely integrated with the MultiThreads process environment, thus a thread using the network interface never causes other threads to be suspended while the requests are performed (except when required to avoid critical regions). This is true both in the DOLPHIN environment and when using MultiThreads and MultiPackets in a Unix machine.

In all analyses performed, simulations were executed under what we have named light, medium, and heavy loading. This implies that the simulations were executed in parallel in groups of either 10,

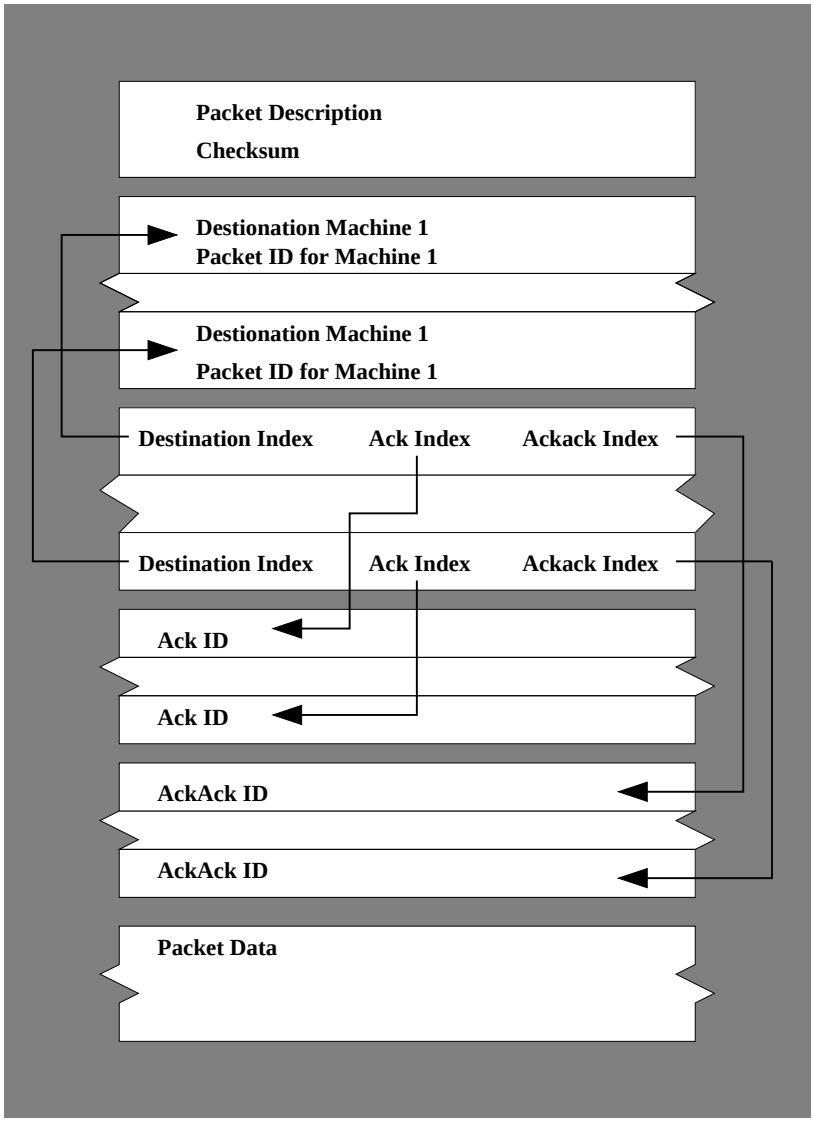


Figure 10.3: Internal Structure of a Packet

20, or 30 instances. This was found to have a clear effect on network loading, similar to that expected under 30%, 60%, and 100% network saturation. All experiments were performed on a single ethernet link, using ten SPARC ELCs. No other users were using either the network or the SPARCs during the simulations. Each simulation was executed ten times to give average (as opposed to one-off) results.

10.5.1 Retransmissions

Of initial interest in the setting up of MultiPackets were the timeout values to be used. These were investigated by running a simulation of packet echoing between two machines in the network. The simulator used datagrams, and if a packet sent by the transmitter was not received by the transmitter within a timeout value, then that packet was transmitted again. Duplicate packets were ignored by both the receiver (which echoed all non-duplicates back to the transmitter) and the transmitter. The timeout was varied between 0.125 and 8.0 seconds. Each retransmission was considered an error, and the number of errors were displayed at the end of the simulation. The results can be seen in table 10.1.

Table 10.1: Error Conditions against Retransmission Timeout

Timeout	Light Loading			Medium Loading			Heavy Loading		
(secs)	$\overline{x}$	σ	Error%	$\overline{x}$	σ	Error%	$\overline{x}$	σ	Error%
0.125	172.6	45.5	1.726%	467.2	101.1	4.672%	859.4	169.3	8.594%
0.250	58.4	18.8	0.584%	111.4	31.3	1.114%	178.0	36.6	1.780%
0.500	34.6	13.8	0.346%	53.9	19.3	0.539%	66.8	16.0	0.668%
1.000	24.1	9.1	0.241%	46.4	15.6	0.464%	59.7	15.5	0.597%
2.000	12.4	6.6	0.124%	28.4	11.1	0.284%	47.0	13.4	0.470%
4.000	6.6	4.0	0.066%	15.9	7.4	0.159%	27.2	10.9	0.272%
8.000	3.7	2.4	0.037%	8.4	4.6	0.084%	13.8	6.5	0.138%

The results indicate that, as the timeout increases, the number of retransmissions made by the transmitter decreased. High timeouts also mean that a lost packet will not be detected as such for a considerable period, which may interfere with real-time constraints. If a user typing on the keyboard caused a message to be sent, then a wait of 8 seconds for a journey needing only 0.125 seconds may be intolerable. However, a low timeout means that a number of transmissions targeted to a machine which is down will begin to swamp the network.

DOLPHIN uses a logarithmic backoff, with a maximum retransmission period of 128 seconds. The initial retransmission time is based on either prior retransmissions to that node or the predicted network distance to that node. Previous network retransmissions are held in a small cache using a direct-mapped algorithm (for simplicity and speed). The predicted distance uses programmer knowledge for machines on local networks, with all other machines getting a larger initial timeout (currently 6 seconds)[4]. In the same simulation as that depicted above, MultiPackets performed similarly to the results given for a datagram timeout of 1 second.

[4] We have not yet investigated internet-based communications, and thus this timeout value is selected by intuition.

10.5.2 Throughput

The total throughput suggested during the description of the algorithm should be better than a socket-based transmission protocol, though not as good as a datagram system. In order to investigate this claim, the datagram simulator used to investigate retransmissions was adapted so that instead of just datagrams, the simulator could used either datagrams, sockets, or MultiPackets. The simulator was then used in the same setup as described above, and the results were plotted in three graphs. Figure 10.4 shows the results with light loading, figure 10.5 under medium loading, and figure 10.6 under heavy loading. The graphs contain all three types of transmission, with sockets marked with circles at the control points, MultiPackets marked with squares, and datagrams left unmarked. With respect to datagrams, the lines all relate to different timeouts (the same timeouts used in the previous experiment), with the smallest timeout nearest the bottom of the graph and the largest nearer the top (and all lines in between ordered accordingly).

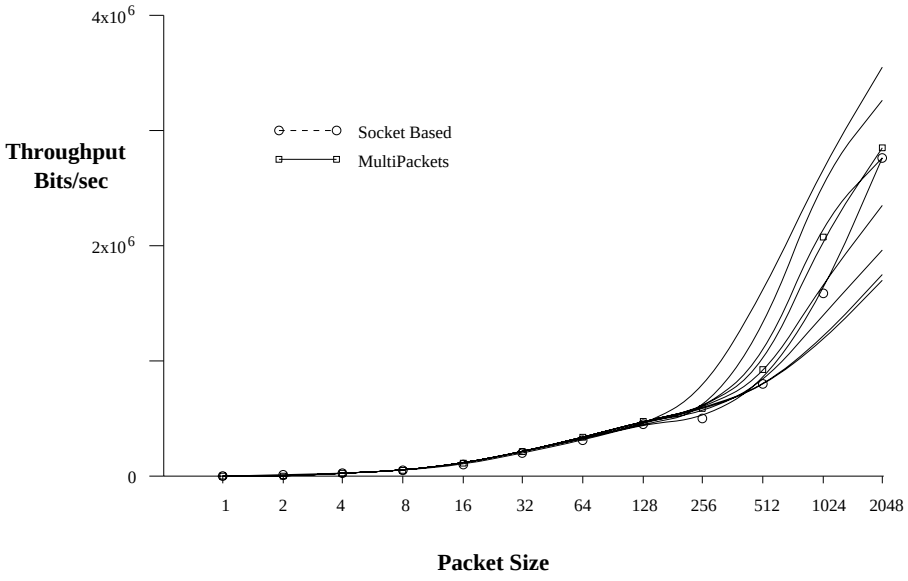


Figure 10.4: Network Performance under Light Loading.The unmarked lines in the graph correspond to datagrams with timeouts of 0.125, 0.25, 0.5, 1, 2, 4, and 8 seconds, corresponding from the unmarked line nearest the bottom to that nearest the top respectively.

These graphs all show that datagrams with timeouts of between 1 and 2 seconds consistently outperforms MultiPackets and sockets, while in turn sockets are consistently beaten by MultiPackets. At packet sizes smaller than approximately 128 bytes, there is little difference to see at the scale of the graph. Performance of MultiPackets can only increase once the overhead of interfacing to the network via datagrams is removed, using instead a lower-level interface. In addition, all the experiments were made with only one thread using the network in each program, with each program communicated with only one node in the network. This nullifies the ability of MultiPackets to piggyback with information involving other threads in the system, which is something which should happen in a multiuser environment (especially if many of the users are running parallel programs over a number of machines).

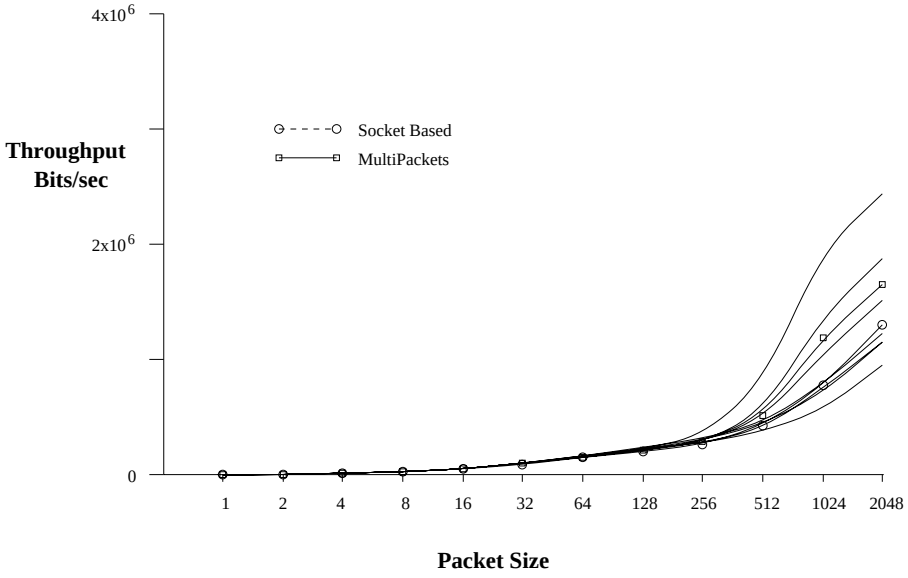


Figure 10.5: Network Performance under Medium Loading.The unmarked lines in the graph correspond to datagrams with timeouts of 0.125, 0.25, 0.5, 1, 2, 4, and 8 seconds, corresponding from the unmarked line nearest the bottom to that nearest the top respectively.

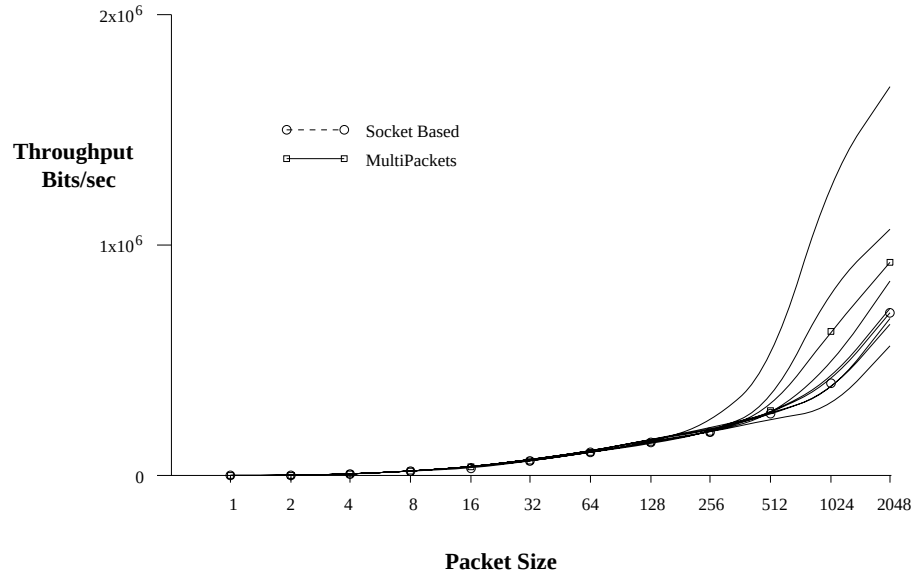


Figure 10.6: Network Performance under Heavy Loading.The unmarked lines in the graph correspond to datagrams with timeouts of 0.125, 0.25, 0.5, 1, 2, 4, and 8 seconds, corresponding from the unmarked line nearest the bottom to that nearest the top respectively.

MultiPackets is completely integrated with the MultiThreads environment. For DOLPHIN, each thread is identified by a thread id. This id is an OID containing data which uniquely refers to that thread via information contained within the thread scheduler running on the thread's processor. By using an OID in this way, unique thread ids can be generated using the standard object creation routines, and the location of the thread (which machine it is running on) can be found via the object data referred to by that thread's OID.

10.6 Packet Routing under MultiPackets

Multicast transmissions to machines on the same network can be handled either by multiple point-to-point communication or by an ethernet broadcast packet. The selection of which method to use is made by a costing algorithm in the transmission subsystem, which estimates the disruption to all nodes with a broadcast packet when compared to the network overhead used by multiple point-to-point transmission. However, in the case of multiple networks when the cost algorithm decides to use a broadcast, things become slightly more complicated.

In the ideal implementation, a broadcast packet would be picked up by all nodes on the first network, and a separate packet would be delivered to the router needed to deliver the packet to all other nodes not directly connected to the current network. The router packet would contain only the addresses of machines to which the packet has yet to be delivered. This router would then retransmit the packet in the same way as the initial sender, using either point-to-point or broadcasting techniques to reach nodes on local networks, and forwarding the packet (with successfully delivered addresses blanked out) to other routers. In this way, the amount of network loading could be minimized. Whenever a route would mean transmitting to a router which does not understand MultiPacket's multicast protocol (*e.g.* the packet includes nodes reachable only over the internet), then the packet would be split into multiple point-to-point transmissions.

This scheme relies on writing routers intelligent enough to perform this splitting, costing, and retransmission. This should be regarded as future work, since at present the use of low-level network interfaces and the ability to run programs on the author's departmental routers is restricted to system support staff.

10.7 Conclusions

MultiPackets allows the researchers involved with object-based networking on DOLPHIN to experiment with a variety of network coherency schemes. Its performance, while slower than that of datagrams, outperforms sockets by a measurable degree (between 5% and 10%). More important than just performance, MultiPackets is constructed with knowledge of MultiThreads, and thus even under the Unix version it allows multiple threads in the current process to use the network without causing other threads to block.

The MultiPackets supports both single node and multicast transmissions in its standard protocol.

It also indicates a way where the basic TCP/IP routers could be adapted to increase the overall performance of multicasts. This would certainly be a useful achievement, since many network coherency protocols can use multicasts to good effect. In any case, by supporting multicasts by default, their actual implementation can be isolated from all other routines involved with the network. This permits further research to be performed in this area without disturbing other levels of DOLPHIN.

Overall, MultiPackets offers a faster and more flexible network interface than that possible using standard sockets. It also provides a higher-level interface than that available with standard datagrams.

Chapter 11

Memory Management

Chapter 8 demonstrated how a Flash-based backing store and battery-backed main memory (RAM) could be unified into a single persistent virtually-addressed heap. However, this heap must still be managed to support object allocation and deallocation. This support is provided by the SubKernel memory management system, which is discussed in the remainder of this chapter. The memory hierarchy can be seen graphically in figure 11.1

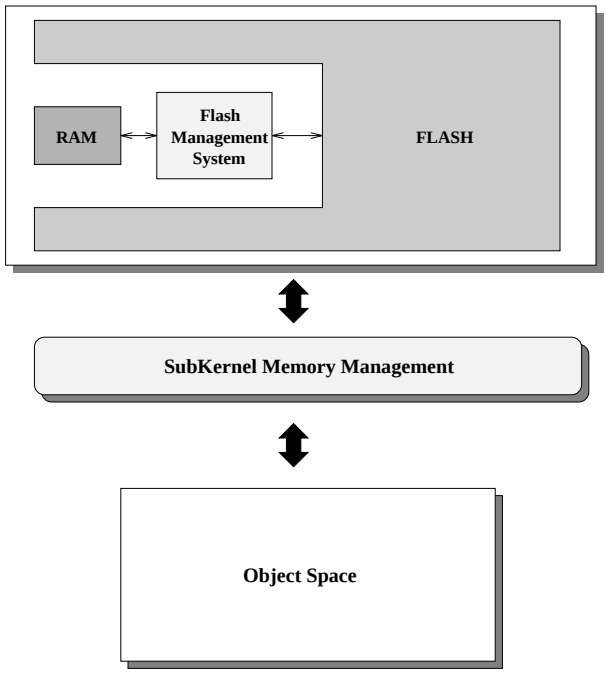


Figure 11.1: DOLPHIN's Memory Layout.

The SubKernel object allocation and deallocation routines support an area known as the *object*

space, within which all programs and data outside the SubKernel reside. The basic scheme used in managing this area is based on a single allocation chain, and this is discussed first. Subsequent discussions indicate how the single allocation chain algorithm is used in object-space management, and how object management is provided.

11.1 Allocation Using a Single Free List

There are basically two types of memory management algorithms used within current operating systems; ordered free lists and memory buckets. With an ordered free list, free memory areas are held in a single list sorted by size (largest first). Allocation is performed by checking the first element in the list. If the element is too small, then the allocation failed (since the first element is the largest available). If the element is too big, then either the list is scanned until a closer match to the required size is found, or the element is split into one part of the required size, and the other holding the extra space (which is then returned to the free list).

The memory-bucket approach uses an array of free lists, each holding a particular range of memory block sizes. A common arrangement is for the array to be indexed with the log of memory sizes (rounded up). When an allocation is requested, the correct array index is calculated and its free list scanned; if the free list is empty it is filled with a number of correctly-sized blocks taken from a larger free area (found in a free list with a larger index). Typically, once an element is found the whole of it is returned, even if the area is slightly too big. This helps to reduce fragmentation, where splitting blocks would result in all elements eventually being held in the smallest index free list. None of the free lists need to be sorted, since all elements in each list must be of the required size.

Both schemes can be coupled with a reconciliation process, where areas previously split up can be joined back together if the areas are free simultaneously. This helps to reduce fragmentation. Reconciliation in a pure bucket algorithm is complicated by the management of the arrays, and may interfere with bucket refilling[1]. The single sorted free list is easier to implement, but this must be offset by the cost of maintaining the sort.

Our scheme, which we have nicknamed *Space Manager*, is based on analysis performed on a sorted free list allocator. Here, the depth into the free list within which the sort was maintained was reduced, and the depth required to probe for a memory request was measured. Over time, the probe depth was one element when the sort depth was maintained at a minimum of ten elements, and averaged between one and two for sort depths of as low as two elements. Naturally, sort depths smaller then two (*i.e.*unsorted) behaved many of orders of magnitude worse. Space Manager uses a single free list sorted with a depth of two. In this way the largest and second largest elements are commonly at the head of the list, while the remainder of the list remains (in general) unsorted.

[1] All the elements used to refill a bucket usually come from a single larger element. The instant the bucket is refilled, then all the elements in that list are candidates for reconciliation. Even if this is avoided, allocation and then freeing of a single element in that bucket could result in a reconciliation attempt, which would empty the bucket again.

11.1.1 Element Structure

Each area of memory controlled by the Space Manager is described by one or more Space Descriptors. The layout of these descriptors is shown in figure 11.2. Descriptors are split into two parts, one part which is always present, and another which is only present when the area is free (*i.e.* deallocated). The always valid area consists of two words; *size*, which is the size of the memory area in bytes (including the always valid area), and *last*, which is an offset from the address of the current descriptor to the descriptor immediately lower than it in memory. By adding the size to the current descriptor address, the next descriptor in memory can also be found. The valid while free part contains two pointers which form a two-way linked list of free descriptors. When the block is allocated, the *size* field of the always valid part is negated. With a maximum memory per block of 2^{32} bytes, the overhead per block caused by the descriptors is only two memory words (assuming 32 bit words).

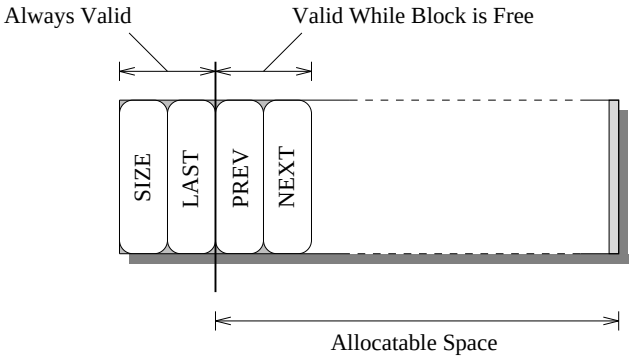


Figure 11.2: Structure of a Space Descriptor.

11.1.2 Free List in Action

The initial state of the free list is depicted in figure 11.3. Initially, two elements are needed for every contiguous block of memory. One element serves as a marker for the end of the list (it is the only element in the free list which has a negative *size*) and the other holds all of the remaining free space. Their *last* offsets link each descriptor to the other. Any allocation requests is made using area **A**.

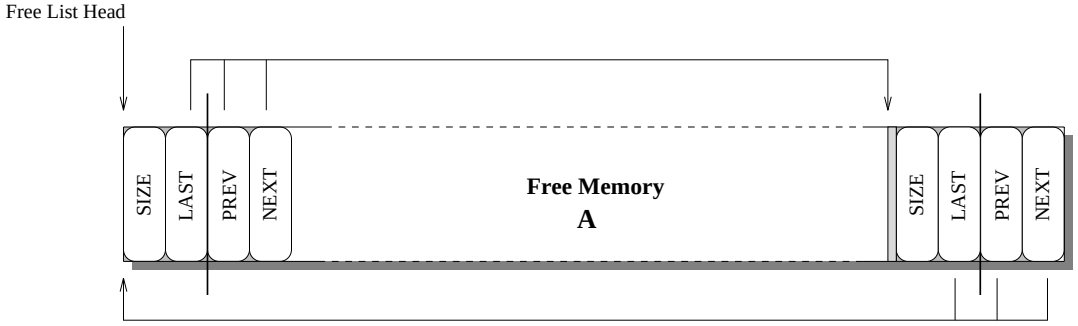


Figure 11.3: Initial State of the Allocation Chain.

When a allocation request is received, the amount requested (plus the size of the always valid part of the descriptor) is subtracted from block **A**. This space is then used to create the new area, which we will name **C**. The *last* pointer of the descriptor whose address is next highest in memory is changed to link to the new block, while the new block's *last* pointer is set to the address of the previous descriptor. This results in the state shown in figure 11.4. Allocation from the end of the block means that the free list pointers need only be changed when either the whole of the block is being allocated or the resultant size of the block violates the sort of the first two elements. This produces a saving of around 8% of the allocation overhead during simulations.

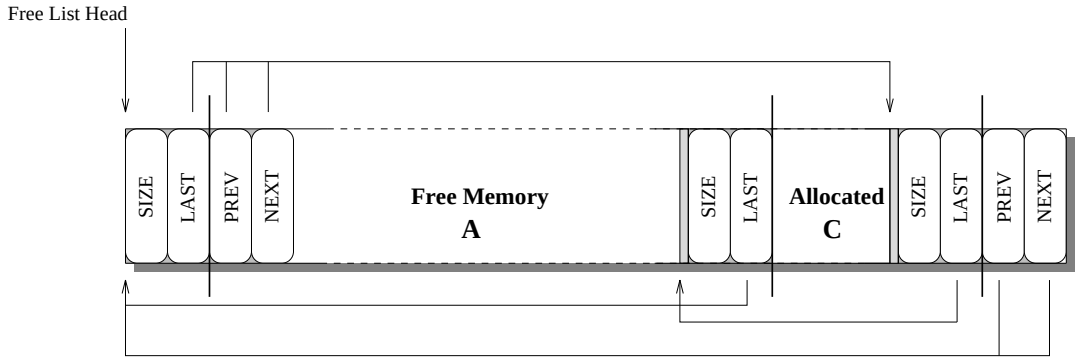


Figure 11.4: Allocation Chain after One Allocation.

Now consider another allocation request. Again, the requested space is subtracted from area **A**, and this is used to create the new area, named **B**. The *last* offsets are adjusted as before. The current state of memory can be seen in figure 11.5.

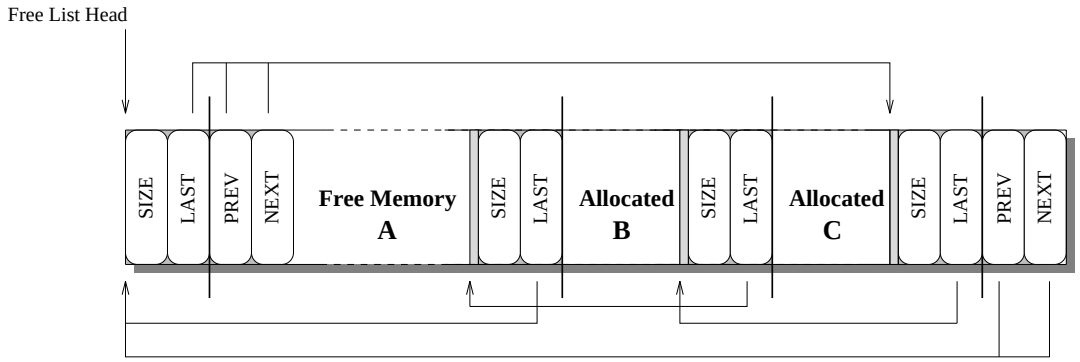


Figure 11.5: Allocation Chain after Two Allocations.

The Space Manager supports reconciliation of blocks; when an area of memory is being freed, a

check is first made to see if either of the blocks adjacent in memory are also free. Thus, if:

Previous Free	Next Free	Action
No	No	Insert freed block into free list
Yes	No	Use Join-in-place
No	Yes	Extract Next, join, and then insert
Yes	Yes	Extract Next, join, then Join-in-place with Prev

If the action is a simple insertion, then the block is marked as free and inserted into the list. To insert, the routine first compares the first element in the list with the element to be freed. If the freed block is bigger, it is inserted at the head of the list. If it is smaller, but bigger than the second element in the list, then it is inserted before the second element. If neither case applies, then the freed element is appended to the end of the free list.

If the action is Join-in-place, then the size of the element to be freed is added to the *size* of the physically previous memory block. If this block is now bigger than the first or second element in the free list, the grown element is extracted from the list and inserted as in the simple insertion case.

Finally, if the action is join, then the two elements specified are merged together. The extraction action merely removes the specified element from the free list, independent of where in the list that element currently lies.

This algorithm does a surprisingly good job at maintaining the free list, since it is much more likely that a block appended to the end of the list will be extracted by a join operation and inserted nearer the head for future allocation, than the case where an appended block will propagate to the head through normal allocation.

With this algorithm in mind, now consider area **C** being freed. First, a check is made on the surrounding descriptors; they both indicate that the surrounding areas are allocated, and thus no reconciliation is possible. Therefore the deallocated block is simply inserted at the end of the free list (but before the end-of-list descriptor), thus maintaining the list's sort. This produces the state shown in figure 11.6.

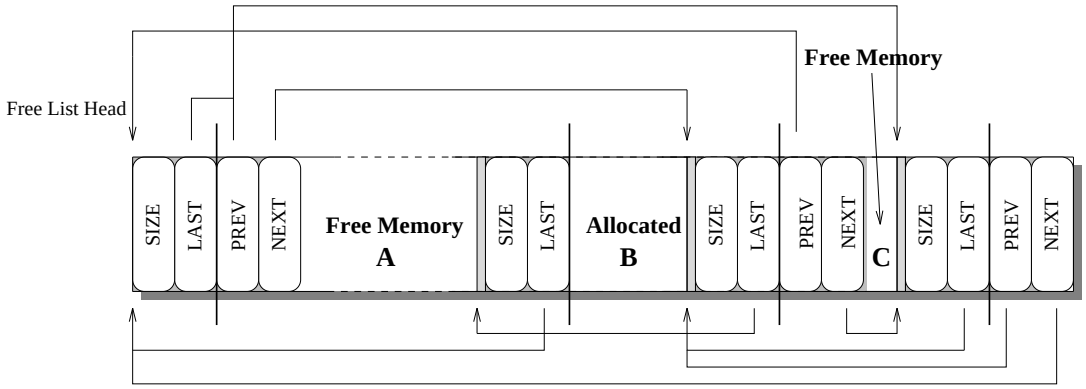


Figure 11.6: Allocation Chain after Two Allocations and One Free.

Finally, consider the freeing of area **B**. Both of the surrounding areas are marked as free (their *sizes* are both positive), and so all three blocks can be combined into a single area. Block **C** needs to be removed from the free list, but an optimization can detect that block **A** can remain in the list (since its position in the list will be unchanged even after the join). The *size* of block **A** is changed by adding the size of areas **B** and **C**. The *last* offset from the next descriptor in memory (in this case the end-of-list marker) is also updated to point to the start of the combined area (*i.e.* the start of area **A**). This leaves the memory structure in the same form as the initial state (figure 11.3).

11.1.3 Performance

The Space Manager algorithm was tried out using a number of different applications on both a Sun4 and a LINUX machine, and each showed a measurable performance gain in comparison to the memory allocated supplied with the compiler. To give an example of the speedup, two demonstration programs were selected; *random* and *network exerciser*. The random program allocated and freed a large number of memory blocks. This was based on a large array of pointers, whose elements were initially set to NULL. Random probes into the array were performed, and if the pointer there was NULL memory was allocated of random size (between 10 and 1000 bytes) with its address stored at that index point. If not NULL, the memory pointed to was freed and the pointer set to NULL. The array used has a size of 4096 entries. The network exerciser gave the MultiPacket system a workout, sending and receiving a large amount of information. MultiPacket itself uses Kernel Space for buffering data. The two programs were executed on three machines; a LINUX station, a Sun4, and a Sequent Symmetry. The results are shown in table 11.1.

Table 11.1: Analysis of the Performance of Space Manager

	Speedup using	
Architecture	Random	Network Exerciser
LINUX	1.74	1.18
SYMMETRY	0.90	0.95
SUN4	1.84	1.23

This results show that for random, both LINUX and SUN4 showed significant speedups, in comparison to the allocator supplied with the operating system. The random program performance is completely allocation/deallocation rate based, so this result may not be surprising. The network program also showed a performance gain with both machines of between 18% and 23%, which helped in turn to boost network throughput measured using this program by a similar amount. Note however that the SYMMETRY architecture performed consistently better than the Space Manager system. This architecture uses an allocator which is a non-compacting bucket system. Further investigations showed that although faster at allocation, it used almost twice (181%) as much main memory than the new algorithm under the random program (which is ideal for catching fragmentation-prone implementations).

Although it does not generally use twice as much memory in all applications, over time fragmentation would become a problem in the Kernel space, and this would be highly undesirable.

Further testing has shown this memory allocation system to be highly resistant to fragmentation. In one case, a constraint satisfaction problem was implemented which made significant use of dynamic memory allocation failed to complete after using 81 MBytes of virtual memory (running on a Sparc-Station 10). Using the Allocation Chain, the problem required only 21 MBytes of virtual memory. The actual working size of the program was 15 MBytes.

11.2 Object-Space Management

The object space is mapped in two distinct ways; one for object descriptors, and one for everything else. Object descriptors are of regular size, and are thus immune to the problems of fragmentation (if kept separate from other data allocations). The descriptors are also mapped into their own virtual memory partition, and are therefore controlled separately from all other object space data. Two functions are provided to allocate and deallocate descriptors, which use a single free list to hold available descriptor structures. If the list is emptied, then a new virtual page is requested from the SubKernel memory manager. The new page is used to construct more descriptors. The deallocation routines make no attempt to return pages to the SubKernel after use, since detecting that no descriptor on a page is actually being used is best left to a background process (*e.g.* the garbage collector).

Allocations not involving object descriptors are managed using a combination of a bucket allocator and the Space Manager. Compaction only involves blocks on a single virtual page, and when all blocks on a page have been freed, that page is unmapped from the data partition and returned to the SubKernel memory manager for reallocation. A pure space manager algorithm would not give good performance, since the amount of splitting that can be performed is smaller, due in turn to the smaller maximum block size of a single virtual memory page. Bucket allocation helps to minimize the splitting involved, although block splitting still occurs if the block to be allocated is significantly larger than that requested.

There are two separate sets of allocation routines for object data. One set is used to allocate space which is mapped read/write, and the other returns space which is read-only. This is used to control access to object data. In addition, another routine converts data areas from one state to the other. This is either done by remapping the page (where the page contains only that single data block) or by allocating a new block with the required protection and then copying the data. The allocator is designed such that a block which changes state often will have a smaller possibility of sharing its virtual page with another block. This probability is decreased with the size of the block in question. The main allocation process for object data is based on a single free list, adapted so that memory areas are not joined across virtual memory pages. Whenever a whole virtual memory page has been freed, that page is returned to a pool of free pages and its virtual address released (for possible reuse).

Allocation of data in SubKernel routines which operate below the level of objects, such as management parts of the Thread Manager, use the object data allocation routines. The read-only and read/write descriptions for object data have no effect on SubKernel routines, allowing them to write to any area of the store. With this in mind, allocation requests are catered for with a third interface, which attempts

to allocate first from the read-write free list, then from the read-only, before finally requesting a new virtual page (with any remains of the page after allocation going into the read-write free list).

11.2.1 Object Management

The management of objects is based solely on the allocation routines presented above. This isolates object management from the virtual memory interface, allowing flexibility with future changes to that level and below. Object management requirements are basically:

1. Allocation of object descriptors.
2. Mapping of object data on unmapped-page faults.
3. Solving access failures caused by insufficient protection.

The respective routines provide the support needed for local non-networked object management. They also allow ties to which other management routines can be added (such as the example object network manager presented in chapter 12). To permit a clear description of these three operations, the use of Ada-style exceptions is used. Basically, if an exception is raised, control is passed to an exception manager higher up in the execution tree. This is implemented in C code using a library based on **setjmp()**.

Allocating an object

Object allocation is a trivial operation, and is performed without mapping in the actual data pages for the object. This produces high object creation rates, and more importantly supports the idea of object stubs required to support object networking (see chapter 12). Object descriptors are allocated via the object-space descriptor allocation routines. When allocated, the length of the object is added to the descriptor, as is the checksum to the OID field. The object status is set to OIDVALID, meaning that the descriptor can now be referenced without an ERR_INVALIDOID exception being generated by the processor. The data pointer of the descriptor is set to NULL, and the object status marked as UNMAPPED.

Object deallocation is performed by setting the descriptor status to OIDINVALID, and then clearing the OID checksum field. A deallocation routine is called, which removes any required structures attached (*e.g.* the data information or networking structures). The descriptor is then returned to the descriptor free list.

Mapping in Object Data

When an object's data is accessed, the descriptor status field must have MAPPED set. If not, then the object data field must point to NULL and needs to be initialized. An exception handler updates this field to contain a virtual memory address. If the length of the object allows all of its data to lie within a single virtual page, then the object data allocator is used to create a virtual memory area of the correct

size. The virtual address of this area is stored in the data field of the descriptor, and the status field updated to MAPPED.

If the length of an object is greater than a single virtual page, then a virtual address for the whole of the object data part is generated using an unused part of the data partition, and this is stored in the data field (with the status field updated to MAPPED). The exception handler then maps in only the page of the data required for the current access, leaving all other pages unmapped.

On accessing an object which is MAPPED but longer than a single virtual memory page, it is possible to reference a page which is unmapped in virtual memory. This causes an interrupt, which in turn maps in a single virtual page at that location.

The current design is such that if an object length is smaller than a single page, then multiple objects can use that page for their data (as allocated via the object data allocator). If the object is greater than a page, then any left-over space on the last page of the data cannot be used for any other purpose. This makes management of the data area significantly simpler to understand and cleaner to code. The efficiency of this scheme is lower than that possible if the last page of a multi-page object could be used for other purposes, but since most object are predicted to be either much smaller than a page or will cover many pages, the amount of memory left unaccessible should be small.

Changing an Object Mapping

If an object data part has memory mapped as read-only, then a write request to that data will cause an exception (ERR_ACCESS). This allows for other features to be added to the based object system, such as object locking and network sharing, to be implemented more easily. Object data management functions support the conversion of any object's data from read-only to read-write (and vice-versa) as required. This functionality is used in the network manager described in the next part of this document.

11.2.2 Garbage Collection

The SubKernel provides a simple garbage collector, based on a three generation asynchronous mark-and-sweep algorithm[2]. Currently, data in the persistent store is not collected, but is still used to generate roots for younger generations. The routines which handle virtual memory only know about objects in the youngest generation, and so accessing an object in a older generation results in that object being moved into the youngest generation. This permits dynamic problems to be examined in the DOLPHIN environment, without running out of memory. This garbage collection mechanism is far from perfect, but performs well in the initial testing of the system. More work is required researching the garbage collection of objects in large object heaps, especially those involving networked objects. Further research into this area should be considered as future work (see chapter 13).

[2] An explanation of basic garbage collection techniques can be found in appendix B.

11.3 Conclusions

This chapter has presented the memory management facilities available to programmers in DOLPHIN. These should cater for the more common requirements of object researchers using DOLPHIN as a basis for their own work.

The space manager algorithm described surpasses the performance of many currently available implementations found on common systems, typically by approximately 20%. Where it does not outperform an algorithm in speed, it runs a close second, adding the advantage of memory compaction, and the resistance to fragmentation which such a system brings.

The object data management routines attempt to quickly return unused virtual pages back to the virtual manager. This is essential to a system designed to run continuously for days or weeks (or longer) without needing to be rebooted. The inclusion of such an aim can only help to improve the reliability of the final design.

Overall, the memory management facilities, coupled to the processor design of DAIS, help to provide a fast and flexible object-oriented architecture. When this is joined with the other parts of the SubKernel (*e.g.*MultiThreads and MultiPacket), the environment created meets the design objectives of DOLPHIN; to provide a strong basis for further research into the area of object systems. In the next part of this document, This environment is used to realize a simple exercise in object network management. This part also offers some conclusions on the DOLPHIN system, as well as presenting work identified for future research in this area.

Part III

Examples and Conclusions

Chapter 12

Example Applications in DOLPHIN

The main elements which make up the DOLPHIN environment were presented in the previous part of this document. During their presentation, and also in the general aims of the project, this environment was suggested to provide a flexible basis for implementing algorithms generated from research into object-oriented systems. In this chapter, we examine the accuracy of this suggestion by developing and implementing a simple networked object manager.

Object networking is a complex problem, and no-one has yet developed an algorithm which ideally solves all the inherent requirements. These include:

- Objects can be read and/or written by any machine with the authority to do so, and at all times there should be data consistency (an object can not exist in two different machines if each contains different data).
- Local garbage collection can proceed independently from global collection, at least with respect to the youngest generation. Local collection should not be delayed by network latencies or external nodes being down.
- Global collection should be able to remove garbage with links spanning across nodes. If information is required from a down node then only the garbage which spans through that node is not collected. Neither space nor objects should be lost due to node or network failures. Cycles of garbage spanning multiple nodes must be garbage collected. Note that references to objects may be present in packets in flight within the network, or within the network protocol's buffers, and nowhere else.
- The algorithm employed to read and write networked objects should be efficient. In order to minimize the communication bottleneck, it is generally preferable to load objects to be accessed into the local node's memory, and perform all accesses locally.

The network manager proposed here should be seen only as a possible scheme to implement object

networking in DOLPHIN. It uses a variant of a shared memory scheme proposed by Li and Hudak [Li and Hudak, 1989]. The current implementation does not implement networked garbage collection, but local collection can proceed as normal. It uses single writer, multiple reader approach with a distributed server scheme. Page invalidation (converting multiple read copies to a single write copy) uses MultiPacket's MultiCast facility. Note that, since all PIDs have their own network address describing which machine created them, all objects can be created (and allocated PIDs) using only local knowledge (*i.e.* no centralized PID allocator).

12.1 Network Manager

Li and Hudak, in [Li and Hudak, 1989], investigate a number of distributed memory-management protocols. These involve the control of fixed-sized pages, for use in shared virtual memory systems. DOLPHIN, with its object-based memory, differs from shared virtual memory in certain key areas. In this section, the algorithm proposed by Li and Hudak is presented, followed by the modifications needed to map this onto an object management scheme.

12.1.1 Dynamic Distributed Copy Sets

The Dynamic Distributed Copy Sets algorithm (DDCS) allows virtual pages to migrate around a network as required by running applications. There is no concept of a centralized record of where pages are currently stored, only of a *probable owner* of a page. Pages start off being owned by a single node (*e.g.* the node which first accesses that page). If other nodes request read-access to that page, then the owner's write access is reduced to read-only, and all requesting readers are given a read copy of the page to access. Each of the readers consider the first node to be the page owner, and their probable owner fields all point to it.

If any node wishes to gain write access to a page, it sends a request to the probable owner of that page. When such a request is received by the owner, it reduces its own access to that page to no-access, and then forwards that page to the requesting node, along with a list of all nodes which asked the owner for a copy of that page in the past. The requested node then updates its probable owner field for that page to point to the requesting node. On receiving the write copy of the page, the node sends invalidation messages to all nodes which are listed in the list of read-copy owners, and sets its own probable owner field for that page to itself.

If a node receives a request for write-access to a page to which it currently has no access, then it forwards the request to the node listed as the probable owner of the page, and then updates this entry to point to the requesting node (which helps to maintain short probable-owner chains). If a node receives a read request, and the node contains a read-copy of that page, then the receiving node sends a read-copy of the page back to the requester, and adds the requester to the local copyset list for that page.

When a node receives an invalidation message, the page in question is transferred to a no-access state. Any nodes listed in the local copyset list for that page are then forwarded the invalidation message. The probable owner field is then updated to point to the new write-copy owner.

Li and Hudak demonstrate that this algorithm is correct (*i.e.* it avoids deadlock and race conditions, and never loses a link between two nodes). They also demonstrate that, in general, the number of messages required to find a page averages at approximately two. This number was derived using an algorithm which requested a randomly selected page, and may be even smaller when used in a real-world application.

To make this algorithm easier to understand, a pseudo-code version can be found in program 12.1. The primitives **lock** and **unlock** provide atomic locking and unlocking, which is used to make unique changes to a particular page. The **invalidate** primitive sends page invalidation messages to all those in the copyset. A reply is not needed for the routine to return, since the worst that can happen is that some nodes have access to out-of-date read-only information. If such data is undesirable, then the routine must wait for all replies before continuing.

12.1.2 Object-based Networking

There are a number of problems with the DDCS algorithm presented above. Immediately obvious is that the algorithm is based on pages, while DOLPHIN uses objects. Before object data can be accessed, the object descriptor information must be loaded. We use a two-stage algorithm to solve this; when an object page is sent to a node, it is accompanied by a list of the OIDs which that page contains. If these identifiers do not exist on the receiving node, then an object descriptor page is created for each, with the probable owner set to the sender and the entire object (including descriptor) set to no-access. When any of these object descriptor stubs are referenced, the actual descriptor is requested over the network for read-only access. This allows range-checking to be employed locally. If data is needed from within a networked object, then the page on which that data resides is requested with the access permissions required. Should write access be required on the descriptor (*e.g.* to change the length of the object), then all pages of the object must be obtained (and locked until the operation is completed) with write permissions before the descriptor is requested. In this case, the requests are made from highest page to lowest, thus avoiding deadlock. To allow garbage collection on the network, all OIDs which have been exported to networked nodes are recorded locally.

To minimise memory usage, nodes should be allowed to remove all references to a networked object if that object is no longer needed locally. To remove a node's object-page owner information from the copyset tree, the node sends a message to those within its copyset, informing them of a closer probable owner (by using the node's own probable owner field). This is done using a protocol which allows termination of the modifications if a write-induced invalidation message is received by the node. If an object-page's owner information contains no entries in the copyset, then it can be deleted immediately. The same protocol is used to remove the object descriptor information. The removal protocol is executed as a last resort, since a write-request on the object page in question performs the same function. This algorithm deserves to be investigated more deeply, and this is discussed in chapter 13.

The invalidation messages are sent using the MultiPacket's multicast facility. This should allow invalidations to be made efficiently, invalidating a page on n nodes takes only a single priority message

```
Read fault handler:
    Lock( page.lock )
    ask page.probowner for read-access to page
    page.probowner = ReplyNode
    page.access = read-access
    Unlock( page.lock )

Read server:
    Lock( page.lock )
    if page.access ¡¿ no-access then
        page.copyset = page.copyset union RequestNode
        page.access = read-access
        send page to RequestNode
    else
        forward request to page.probowner
        page.probowner = RequestNode
    Unlock( page.lock )

Write fault handler:
    Lock( page.lock )
    ask page.probowner for write-access to page
    Invalidate( page,page.copyset )
    page.probowner = self
    page.access = write-access
    page.copyset = {}
    Unlock( page.lock )

Write server:
    Lock( page.lock )
    if page.probowner == self then
        page.access = no-access
        send page and copyset to RequestNode
        page.probowner = RequestNode
    else
        forward request to page.probowner
        page.probowner = RequestNode
    Unlock( page.lock )

Invalidation server:
    Lock( page.lock )
    if page.access ¡¿ no-access then
        Invalidate( page,page.copyset )
        page.probowner = RequestNode
        page.copyset = {}
    Unlock( page.lock )
```

Program 12.1: Hi and Hudak Algorithm for Shared Virtual Memory

on each necessary network (plus all the background acks required in the MultiPacket's protocol). In DOLPHIN, there is also the need to send messages between processes. While the actual message can be passed in a shared object, the synchronization of reading and writing to the object must be controlled. To allow this, a primitive is built into the networking system which returns to the calling thread as soon as the specified object has either been sent to another node or accessed by a local thread. The calling thread can then attempt to read the object, and thus read the message. Provided that whenever the thread attempting to send a message first locks the message-carrying object, then the object will stay accessible only by that thread until the object is later unlocked. This is used to provide message handling.

12.1.3 Protecting Objects in DOLPHIN

Identifying objects across the network is performed using 256 bit OIDs. These OIDs are never sent openly in their entirety, so as to avoid obtaining access to an object by simply listening in on packets being delivered to other machines. Firstly, let us consider that a machine on the network has access to a networked object. To request the data part of the object, its OID is embedded in the request structure, along with the requester's network number. This structure is then encrypted with the checksum field of the object id as a key, and sent to the owner of the object data (via the probable owner tree) along with an unencrypted version of the requester's network id and the desired OID, but this time with the checksum field blanked out.

When a request of object data is received, the object's checksum is retrieved from local memory. It is then used to unencrypt the request structure, and a check is made that the structure has been correctly unencrypted (using the host id and the embedded checksum). If the check fails, then the requesting node is considered hostile, and its request ignored. If the host id on the structure differs from that in the unencrypted part, then the unencrypted version of the host id is considered to belong to a hostile host, and is again ignored.

Provided that the request is considered valid, then the data requested is packaged up ready for transmission (along with the sender's network address and any other management data required). This is then encrypted with the requested OID's checksum, and then transmitted with a header containing the source node and the OID (again minus the checksum field).

On reception of a object's data packet, the packet body is unencrypted using the locally known checksum. Checks are made on the sending fields and the OID's checksums matching. Provided all is correct, the data is transferred to the required point in memory. Failure means that the sending node becomes considered hostile.

If a node becomes hostile, requests from that node are ignored from a short period of time (seconds). This serves to protect data from prying eyes, without interfering unnecessarily with legal data access. It is important that the lock-out period is not too severe, since it is possible either for an illegal request to be made accidentally, *e.g.* by an error in the operating system, or mischievously by another host which somehow manages to affix the blame for the illegal request on an innocent machine.

12.2 Examples

In order to demonstrate the correctness of the network manager a simple simulator was constructed. This was build on top of the SubKernel elements described in previous chapters, and used the network manager described above. The interface to the simulator is provided by C routines, which allow objects to be created and accessed. As the simulator was constructed specifically for this analysis, its syntax is not provided within this document. The simulator was executed within a Unix process. Using the simulator, two traditional parallel benchmarks were executed; travelling salesman[1] and matrix multiplication. It is also of interest to investigate the speedup achieved by using this network manager, as the number of processing nodes are increased. To offer further comparison, the results for these two algorithms were also obtained for an Orca [Ball *et al.*, 1992] implementation, running on a modified Amoeba [Tanenbaum *et al.*, 1990] kernel. This kernel uses reliable broadcast protocols to broadcast object writes to all nodes in the network, which it is suggested improves network performance [Tanenbaum *et al.*, 1992]. Unfortunately, this type of protocol does not scale well, since increasing the number of networks over which the problem domain executes does not increase data throughput. In this algorithm, all changes to objects must be sent to all nodes in the network, and thus all networks retransmit the broadcast message.

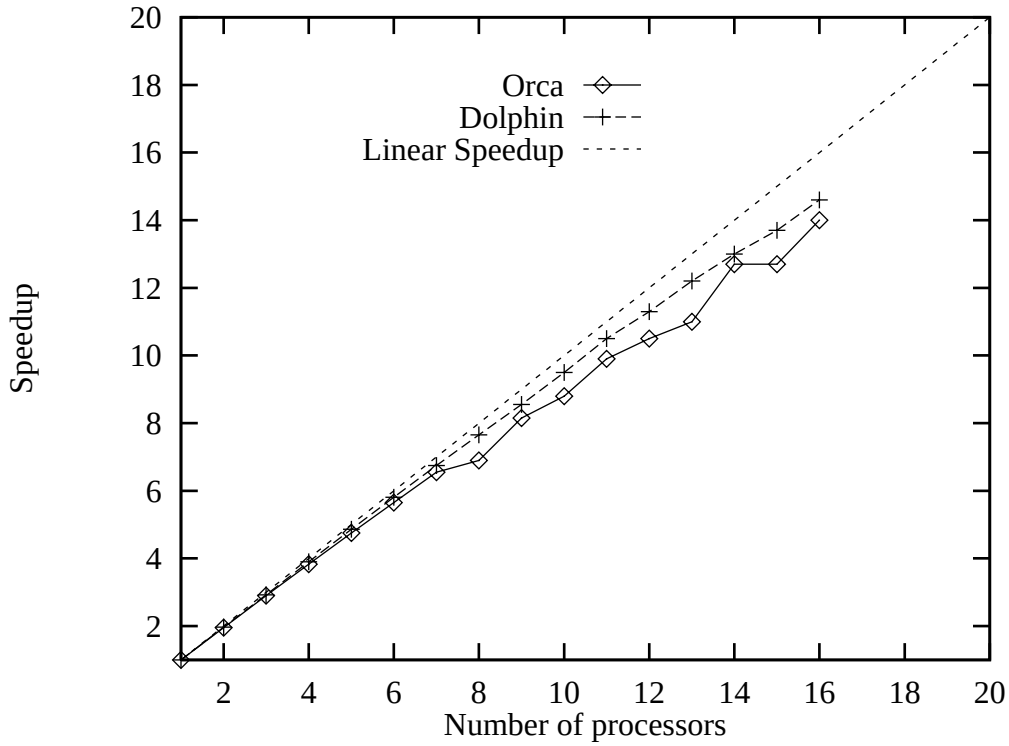


Figure 12.1: Matrix Multiplication Benchmark (250×250)

[1] In general, the brute-force algorithm for solving the travelling salesman problem is being replaced by other techniques, including genetic algorithms. However, the brute force implementation still returns the correct answer (as opposed to the GA which returns only the best answer found within a particular timespan).

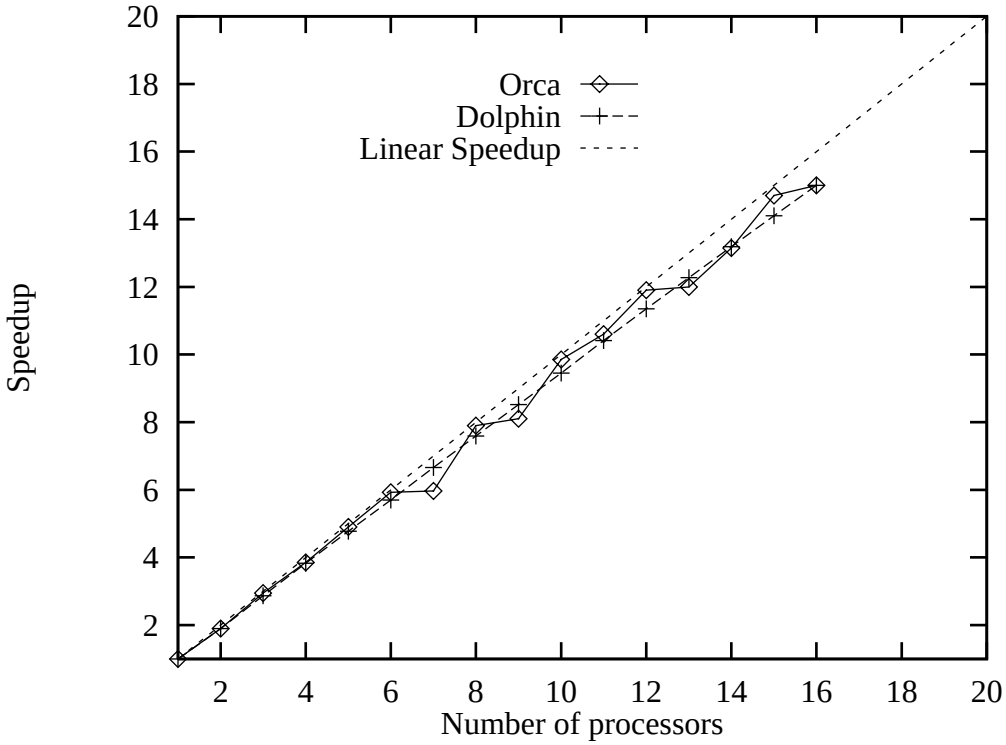


Figure 12.2: Travelling Salesman Problem (12 cities)

Figure 12.1 shows the results for the matrix multiply example, while figure 12.2 refers to the travelling salesman problem. In matrix multiple, the source matrices are of size 250×250. The travelling salesman problem is based on an average over three randomly generated graphs each containing twelve cities. In both cases, the speedup measured is near to linear. All benchmarks on DOLPHIN were executed twenty times each, with the average[2] speedup used in the graphs. The results for Orca were taken from [Tanenbaum *et al.*, 1992].

Both of the benchtests demonstrate the benefits which can be gained through shared memory managed via a network. In matrix multiply, the two source matrices are only ever read from (except when they are being initialized). Thus they are requested only once by each machine in the network. The results matrix is the only one which is written to. Each matrix is build from an 256 entry array of 256 entry vectors. The problem partitioner shares out the vectors between processors, and thus there is no competition in writes.

With the travelling salesman problem, the graph of the cities is accessed read-only, with each process storing its current working details in a local object. Another read-only object contained all combinations of starting points and second city. Only two objects are used read-write; one storing the shortest path found so far, and the other containing the index of the next starting point to investigate. When each process runs, it locks the starting point index object, stores that number in a local variable,

[2] The standard deviation measured during this experiment was negligible. The main reason for running the program multiple times was to remove effects caused by the programs used being demand-loaded from the central server or swapped to disk.

and then increments the index and unlocks the object. It then uses the starting points object to find the two cities to begin the search with. If a route shorter than that found in the shortest-route object, then that object is updated with the new distance. If a node's current route is longer than the best distance found so far, then a new route is selected. This algorithm allows up to 132 processors to be involved in the calculation.

Linear speedup will not always be the case. Worse than linear is always possible, since some problems which need a high degree of intertwined shared object reads and writes will suffer due to the latency involved in requesting data over the network. In the benchmarks presented above, the actual calculation time significantly outweighed the communications overhead. Superlinear speedup is also possible; but generally this only happens due to the effect of virtual memory. If a problem running on a single processor is continually swapping to disk, then a distributed implementation whose processes can run without swapping can, depending on the degree of intercommunication required, run with superlinear speedup.

12.3 Conclusions

The algorithm presented in this chapter is not meant as a guide to writing the ultimate network manager, but instead as a demonstration of how DOLPHIN can be used to support further research into the realm of object-oriented systems. However, this simple algorithm does remarkably well when compared to both the linear speedup ideal and that returned by Orca.

The transference of objects and object requests using this algorithm is fully protected from a wide range of mischievous acts. It is highly unlikely that any node in the network will be able to gain access to an object without first being given that OID. Random probing using a range of checksums, especially when coupled with the lock-out features of the protocol, suggests an average time to crack a code somewhat larger than the life of our sun. This is certainly better than using a public-key encryption mechanism, where once a single private key becomes known (*e.g.* by bribing someone who knows the code) the node to which that code belongs becomes compromised. Using the DOLPHIN scheme, each object is protected by a unique code, and cracking that code only gives access to objects from that point in the object tree onwards.

Further investigation into networking algorithms is certainly warranted. The algorithm described above, while not perfect, does offer a basis for future work. The mechanism behind removing nodes safely from a tangle of copysets and probable owners should be investigated, as should network-based garbage collection. In DOLPHIN implementations where the OIDs are not explicitly (*e.g.* by tags) marked (*e.g.* as in DAIS 256), locating the OIDs for maintaining object import/export lists can be computationally expensive. One avenue for examination in such systems is to use a process whereby objects exported to other machines have all OID aligned data words initially considered as valid OIDs. When the data is received at the requesting node, again all OID aligned memory words are considered as OIDs, and an object descriptor entry is created for each (with the probable owner field set to the sending node). Local garbage collection on the sending node will eventually eliminate the OID entries in the export table which are not truly object descriptors, using internal knowledge. OIDs in the import

table could be eliminated as a function of the networked garbage collection algorithm.

It is also be of interest to investigate techniques for obtaining the initial OID necessary to initiate legal object requests over the network. In the examples, this was done using a known message object, whose OID is always the same on each machine (except for the network address part). This was used to send a text string to the node, which corresponded to a name registered to a particular object. The OID was then send to the requesting node. For simplicity, this initial transfer was not encoded, but it is possible to encode such transmissions using a public key encryption scheme. Such techniques have standard implementations, and are not discussed further here.

Chapter 13

Future Work and Author's Comments

Throughout the body of this document, promising avenues of research have been followed and analysed. Yet there is much more research to be investigated in the area of object-oriented systems. It is hoped that DOLPHIN will be a useful tool in furthering these investigations. In the immediate future, a number of topics have been highlighted for scrutiny. These include:

- Flash array management.
- Implementing multiple register stacks without fixed heads.
- Network management, especially with respect to garbage collection.
- MultiPacket's advanced routing of Multicast messages.
- Object transaction support.

It should also be made clear that the DOLPHIN environment is not currently a real system implementation, but is instead a design proposal. Constraints, both financial and chronological, have helped to constrain DOLPHIN to a paper/software emulation implementation. However, the work presented above not only serves as pointers for future researchers, but also as the basis for a full DOLPHIN system. The next steps in its developments can be categorized as:

1. Full software simulator, from DAIS up.
2. Construction of the DAIS pseudo object-oriented C compiler.
3. Hardware-supported DOLPHIN.

In the remainder of this chapter, the topics listed above are described briefly. This should be of interest to those researchers looking for a new direction in their research. Note that some of the topics are already under investigation here at Strathclyde University. For more information please feel free to contact the author.

13.1 Management of the Flash array (GC and transfers)

The ERCA-based Flash array presented in chapter 8 was analysed using standard virtual memory management techniques. To use this system effectively in the DOLPHIN environment, a mapping algorithm must be produced to control transfers between Flash and the main-memory based object store.

An initial proposal to managing the store would be to start with an main-memory space free of objects. When an object is referenced, the virtual translation for that object's descriptor entry is loaded from Flash into memory, and it is added to the MMU's virtual page table. If object data is later referenced, then the corresponding translation entry is also loaded into memory. In this way, information such as the dirty and accessed bit in the translation entry can be updated, without the problems of trying to write to an entry which is held in the read-only store (*i.e.* the Flash array).

If data is being written to the Flash store, the data is transferred into main memory and its virtual translation entry adjusted to point to the new location. The original location of the data in the store is then marked as dirty. When the data is ready to return to the store, it is written to a different location, and once again the corresponding translation entry is updated. A similar scheme is used to allow virtual entries to be written back into the store. This part of the algorithm is similar to that described in the performance experiments on the Flash array.

Further research would be beneficial in producing a well-formulated algorithm for performing the swapping between the Flash store and main memory, and also in maintaining free-space in the store itself.

The area of garbage collection should also be examined. The current system uses a generation-type implementation, with only the first two generations being implemented. Fortunately, since there is not yet any software to support object transfers into the Flash store (and we do not actually have a Flash store built anyway), objects never age beyond the second generation. Thus the collector really functions as a two-generation mark-and-sweep system. At the same time as research into the store management is taking place, the problems of garbage collection over large persistent stores should also be investigated, as there is a likely to be a large degree of interdependencies.

13.2 Register stacks without fixed heads

A register file based on multiple stacks was presented in chapter 6. In this design, the top of each stack was implemented as a register, and only when the register was being either pushed or popped did the stack need to be accessed. It has been suggested that a performance gain could be realized by removing the register at the head, and instead operating directly on the stack itself.

When data is pushed onto a register, then the register to stack transfer must be performed immediately, since the register in question must be used to store the new data. However, when a register is popped there is no real requirement to stop the processor until the relevant stack transfers the new head-of-stack element into the register in question. Instead, the transfer could be delayed until the register is being read. Interestingly, if the first access after a pop is a push, then no data transfer is needed

anyway.

To implement this suggested scheme, the registers used as the stack heads could be removed, and the reads and writes performed directly on the stack heads. With the stacks able to support sizes between 0 and their maximum capacity, then only a pop on stack size zero or a push with the stack completely full would pause the executing program. This has the effect of apparently increasing the size of each stack by one cell, when compared to the multiple stack implementation of chapter 6. It also required more logic to control access to and management of the stacks. Whether the improvement in performance, or the reduced silicon utilization necessary for the same performance, justifies the added complexity is question which should be investigated by researchers interested in this problem area.

13.3 Object Network Management; a Linux implementation

It is intended that the object network manager presented in the examples chapter of this document will be investigated further. It has been suggested that an extension to the Linux operating system could be made to implement a pseudo-object based memory system. It is proposed that a collection of 486 machines connected by ethernet should be obtained. Each of these machines will be running a modified version of the Linux kernel, whereby part of the memory map will be standard local memory, and the other configured as networked memory. In this way, modifications to the networking strategy can be investigated, such as heavy-handed object id selection scheme suggested in the conclusions of chapter 12 for avoiding the need to find all object IDs in exported object data.

The networked memory of each machine can be allocated and deallocated through provided memory management primitives. Allocation is performed in one partition of this area. For example, the proposed system will have four processors, and thus the network memory is made up of four partitions of 0.25 GBytes each, equating to a GByte of memory reserved for networking. Each machine can only allocate space in its own partition, which avoids the need for a centralized allocation mechanism. Deallocation is handled by a network garbage collector, which runs regularly in the background.

If a machine in the group requires access to networked memory, that information is obtained using a network management algorithm similar to that shown in the examples chapter. There is already support for advanced virtual memory management techniques in the Linux kernel. This system will be used to implement some time-intensive tasks with which we are currently involved, including place-and-route problems and neural network simulations. It will also be used to investigate the problems involved in initiating contact with a machine for the first time, possibly using the ASCII string-based request mechanism suggested in chapter 12. The design of this system has already begun.

13.4 Multicast Routing

During the discussion on MultiPackets, a proposal was made to support multicasts across networks, using a low-bandwidth algorithm. This algorithm is based on allowing routers to forward multicast packets onto other networks, removing addresses of machines reached through other routes from the

packet header. This would allow network loading to be reduced during multicast actions, *e.g.* object page invalidation messages.

Multiple network implementations are common on the world-wide network (the internet). If the object-based memory model was to become popular, then it is certain that many institutions on the internet would convert of object-oriented systems. If these sites wish to share objects, then efficient mechanisms for passing object data about must be utilized. Failure to monitor network utilization would result in the internet becoming a severe bottleneck.

Using the advanced multicast routing technique, multiple sites across the globe could receive invalidation requests in a relatively small number of messages. In addition, there is no requirement to logically equate the ACK messages of MultiPackets to mean a confirmation of delivery. Instead these signals could be interpreted as a promise of delivery. For example, a packet sent through multiple routers would firstly be delivered to the first router, which would acknowledge the packet to the original sender. The packet would then be forwarded to the next router, which acknowledged the packet only as far as the previous router. In this way, failed packets only cause loading on the local network connecting the two routers, and not all the networks between the failed and the sender's network.

The routing-acknowledge technique in the form discussed above could only function correctly if the data held in each router was guaranteed to be safe from failures. For example, if a traditional router is power-cycled, the information contained therein is lost. However, in an object-oriented system such as DOLPHIN, all objects can persist even over power outages, thus indicating an application of such persistent systems in advanced routing algorithms. This algorithm is currently being investigated. In both basic MultiPackets and the multicast routing, it is also of interest to investigate the effects of network reconfiguration (such as node additions and removals), especially with respect to safety.

13.5 Transaction Support

In the world of object-based systems, the term Object-Oriented Database Management Systems (OODBMS) is often separated from the idea of a persistent object-oriented system. This is primarily due to the need in database applications to rely on transactions. A transaction is, put simply, a part of the program that, when entered, can only be left in two distinct ways; commit or abort. If the transaction commits, then all changes made to data within that transaction are committed to the long-term store (*e.g.* the actual image of the database as held on disk). If the transaction aborts, then all changes made to data within the transaction are unwound, returning the database to the state it was in before the transaction begun. In reality, transactions become more complex, with the problems of nested transactions and multi-threaded transaction support.

Implementing transactions in the DAIS processor is not significantly difficult, and this is under examination by the author. The processor is updated to hold two new registers; one containing a transaction depth and the other the currently executing thread's process id. These two pieces of information are also held on each virtual page of an object, with the fields initially set to some value to indicate that no transaction is currently operating there.

When DAIS accesses any object page, a check is made on its transaction depth. If the depth is set,

then a transaction is in progress. If the transaction id of the page is equal to DAIS' transaction id, then the page is in a transaction belonging to the current thread. In this case, the depth is checked against the current depth. If these are not equal, then a copy is made of the data, and the depth linked to the new copy set to the current depth. If data is accessed whose id does not match the current thread's id, then the thread is suspended until the transaction on that page either commits or aborts.

When a transaction is entered, then any page accessed which does not have the correct depth is duplicated (to allow recovery if the transaction aborts). If the transaction commits, then the version committed has its transaction depth reduced by one, destroying the duplicate (if any) of that data at the new depth. In this way, a page which is only accessed at transaction depth 1 and 3 will only have two duplicates, and a commit at depth 3 changes the data to a depth of two. A second commit reduces the page to depth 1, destroying the duplicate depth 1 page in the process. This avoids the need to take copies of data unnecessarily if the page is not accessed at intermediate depths.

The support for transactions in DAIS has been examined, but it was thought to over-complicate the implementation of DAIS and DOLPHIN, and so was left out of this document. It is also unclear as to how the complexity increase required to support the additional transaction-based hardware would effect the overall performance of the system, and whether transactional support in hardware is needed. For the most parts, most general problems do not need transactions to support their implementations, and even those which do could be implemented using the object-locking systems provided in DOLPHIN. A fuller investigation of transaction support is certainly warranted.

13.6 Second Level development: Full Software Simulator

Up to this point, there has been no complete simulation of DOLPHIN constructed. All the analysis has been performed using simulations which separate the DAIS processor from the subkernel and higher level modules. A software simulation of DAIS would be a useful stepping-stone towards a hardware implementation. For example, a compiler for the system should be constructed before a hardware version of the processor is ever produced, allowing compiler-designer recommendations on the instruction set to be incorporated in the design.

For this reason it is hoped to construct a full software-based simulation of DAIS, coupled to an implementation of the subkernel. This should allow investigations to be made as to the specific form of the instruction set used, and how it differs from that envisaged during the examples given in chapter 6.

13.7 Compiler

With the development of a software simulation of DAIS, the compiler design can be investigated. In this design, there is two main areas of research to examine. The first of these areas is that posed by the stack-based register set. Intuitively, these registers should allow for interesting optimization techniques with respect to both the allocation of registers and the implementation of stack-oriented problems (such as that found in recursive routines). However, the complexity in implementing these techniques may

be high.

Secondly, the design of a language for use as the main system language in DOLPHIN should be examined, and how this language maps to an object-based heap. The author envisages a language derived from C++, since this both offers the medium-level of complexity found in C with the links to object-based addressing. This language would preferable be revamped, removing the remnants of virtual-based addressing still present in the current implementation of the language. It has been suggested that Object Oriented C may also be a worthy candidate in the selection of the language, though at first glance this language may appear too complex to match the requirements for having a fast and efficient system programming language.

13.8 Third Level Development: Hardware Construction

It has always been hoped that the DAIS processor could be implemented in silicon, and set up as as nodes in a network of object-oriented systems. Due to the cost and time involved in such an undertaking, this currently appears unlikely. It may however be possible to reach a compromise situation, using a fast modern RISC processor to either interpret DAIS instructions or run-time compile them into a cache of recently executed pages (in a similar way to SELF). This processor could utilize hardware support for converting object addresses to their virtual equivalents, *e.g.* by using a hardware cache. In this way a fast implementation of DOLPHIN could be created, which in turn could increase the interest in the DOLPHIN approach in particular and the realm of object-systems in general. The implementation of DAIS in hardware is a long-term goal of the author.

13.9 Author's Comments

This thesis covers a large number of topics, with the aim of providing a basis for future research into Object systems. In retrospect, two of the most important areas of research were the object processor (DAIS) and the Flash-card reliability work. I intent to concentrate future research on DAIS. This will involve writing an object-oriented language compiler for the DAIS instruction set, and producing an FPGA-based DAIS computer system (using a combination of battery-backed SRAM, Flash memory, and disk drives to support persistence).

Acting with hindsight, it is also clear that DAIS-256 was a practically unrealistic though theoretically correct suggestion. It makes much more sense to use shorter OIDs, and alias them to network PIDs when (and if) required.

Objects may be constructed so that they have two sizes (positive and negative size), with OIDs restricted to appear only in the negative part. This modification not only speeds up garbage collection, but also simplifies OID aliasing techniques and permits network import/export tables to be maintained without difficulty. For this reason the current version of DAIS, on which I am now working, uses only 32 bits for the OIDs. This has the added benefit of reducing the per-object overhead to only 16 bytes (plus 8 bytes for objects smaller than a virtual page). This will help to support the next generation

of object languages efficiently, where average object size is dropping towards 40-50 bytes per object. Object aliasing is also useful to support hierarchical network addresses, where it would be possible to have a range of OID sizes, depending on whether the object was only local to a single machine, used in a LAN, or available world-wide.

Part IV

Appendices

Appendix A

DAIS Stack Simulation Results

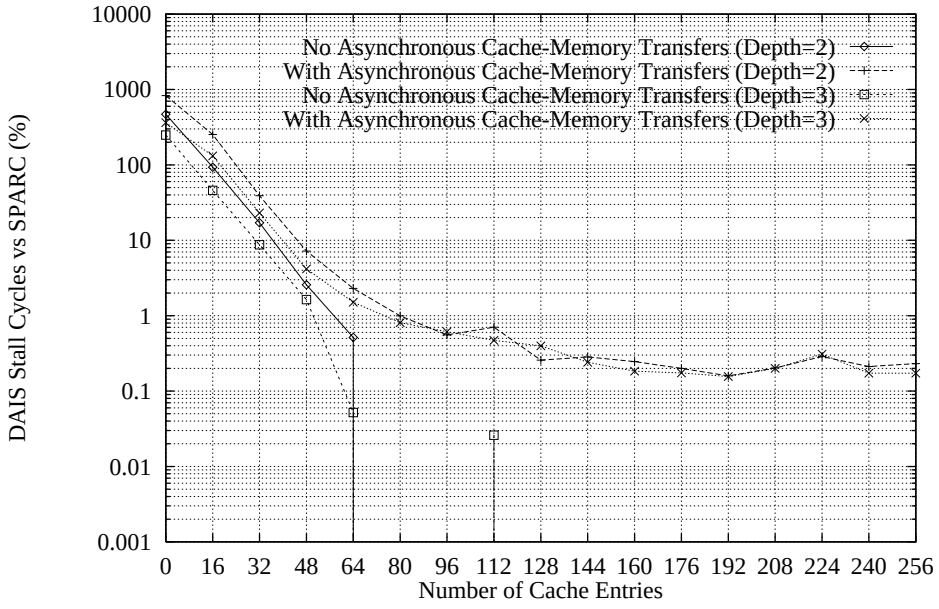


Figure A.1: Analysis of depth 2 and 3 in fig2dev

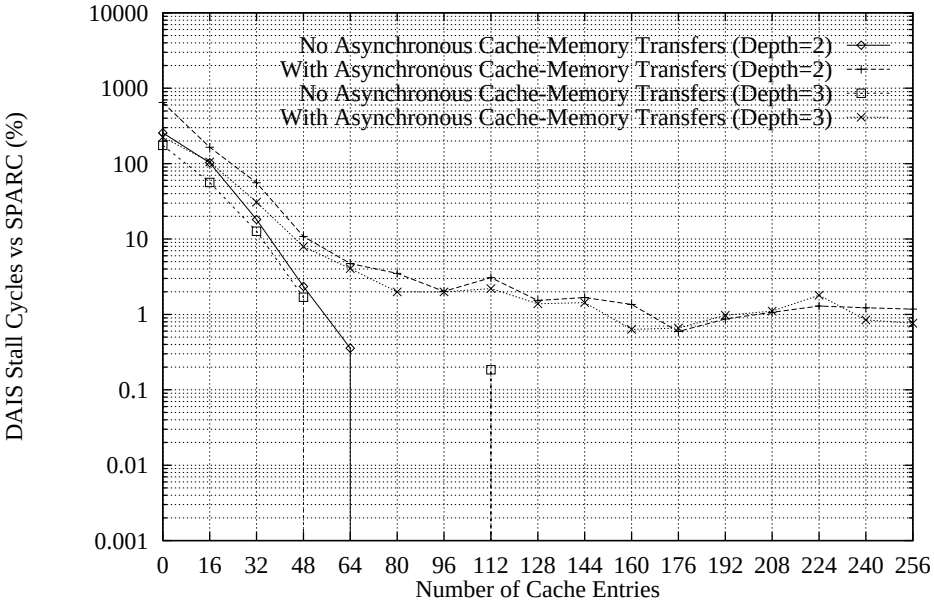


Figure A.2: Analysis of depth 2 and 3 in zoo

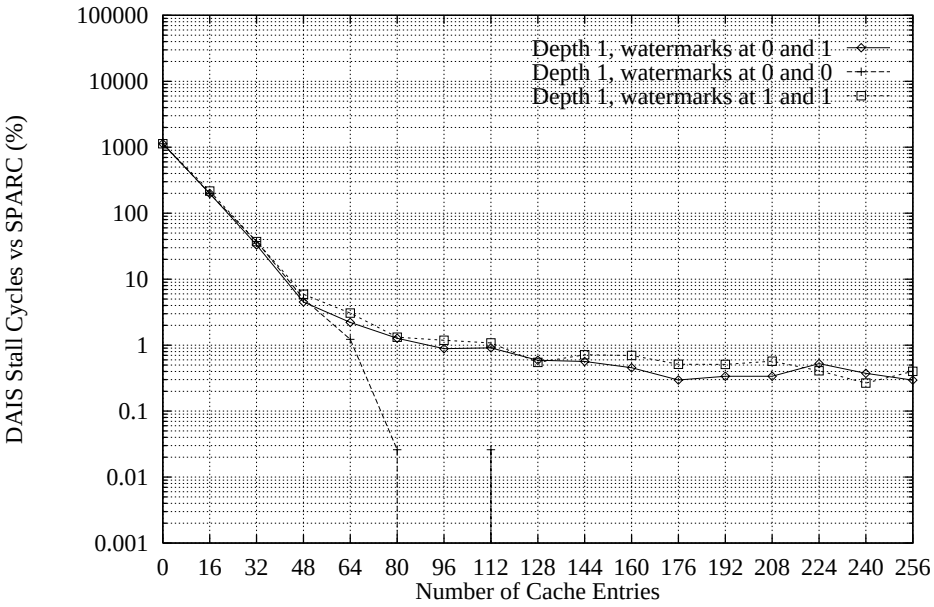


Figure A.3: Stack Depth 1 in fig2dev (no asynchronous transfers)

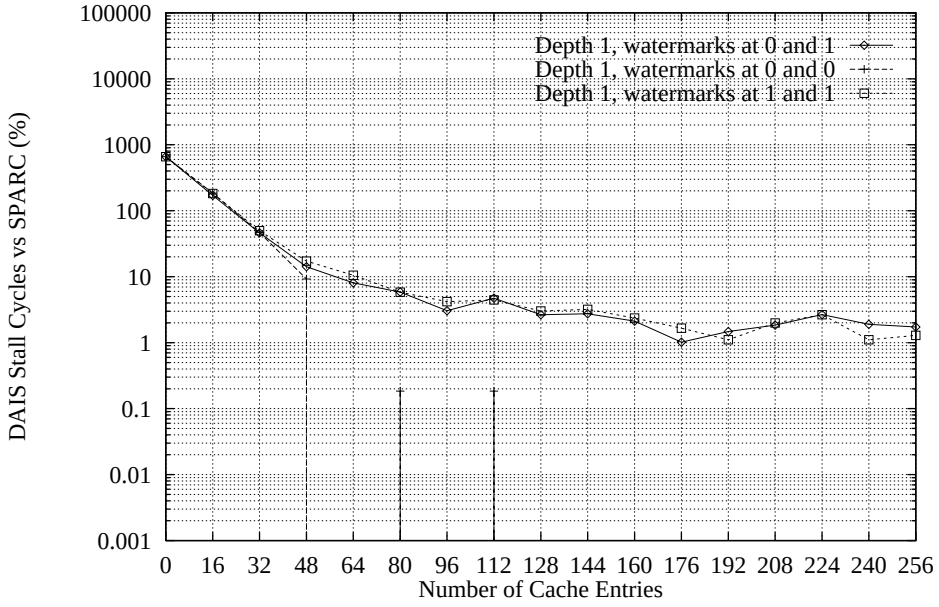


Figure A.4: Stack Depth 1 in zoo (no asynchronous transfers)

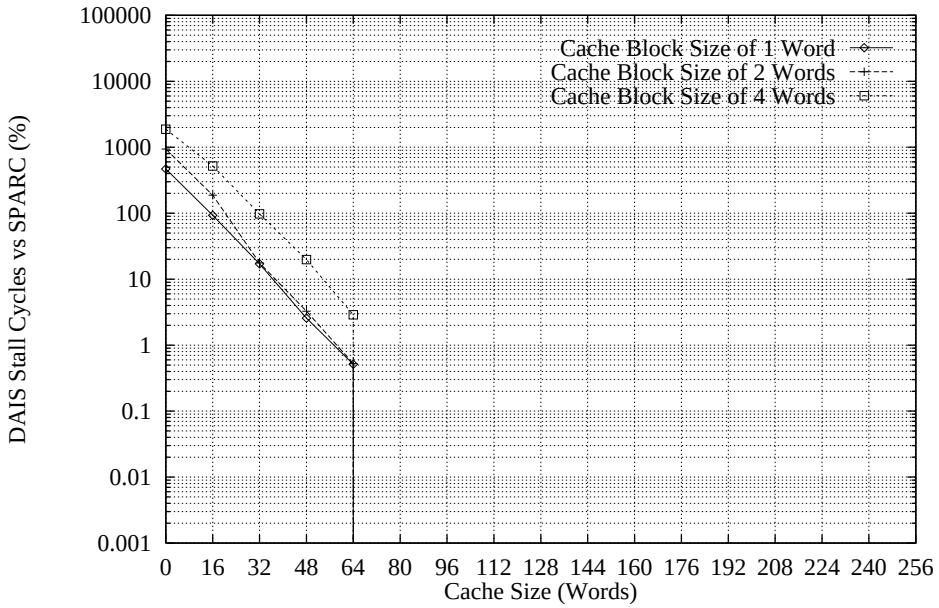


Figure A.5: Stack Depth 2 With Varying Stack Cache Line Length (fig2dev)

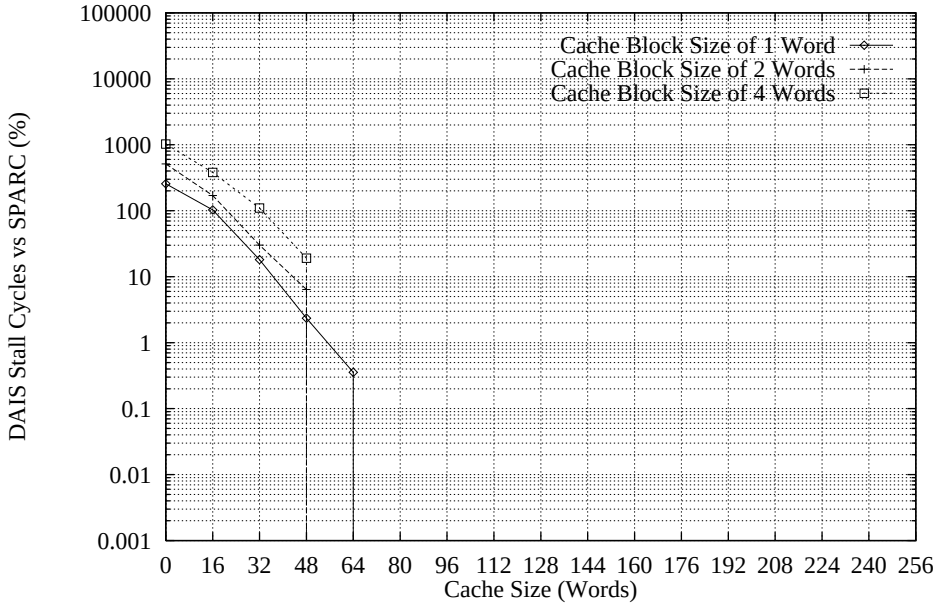


Figure A.6: Stack Depth 2 With Varying Stack Cache Line Length (zoo)

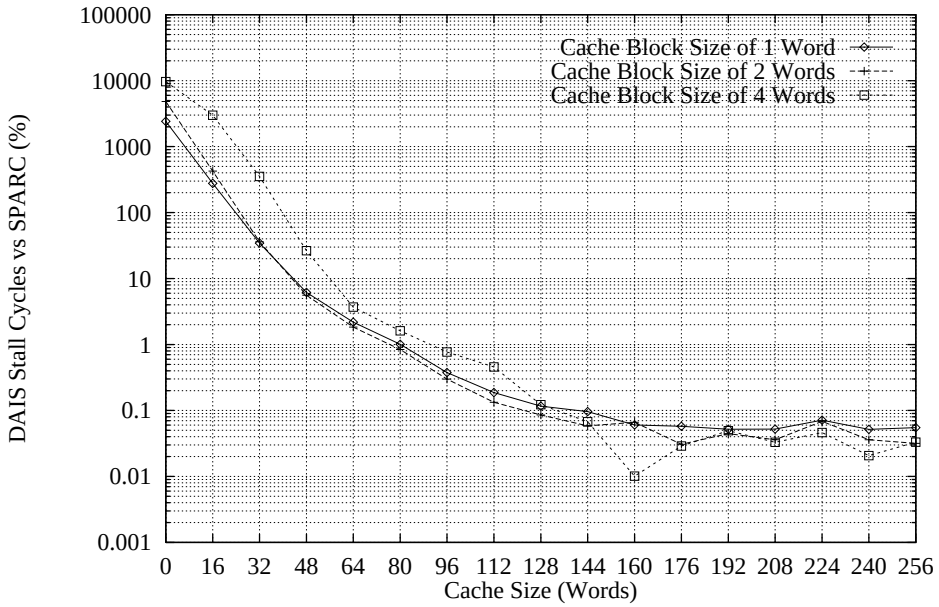


Figure A.7: Stack Depth 1 {0,1} With Varying Stack Cache Line Length (TEX)

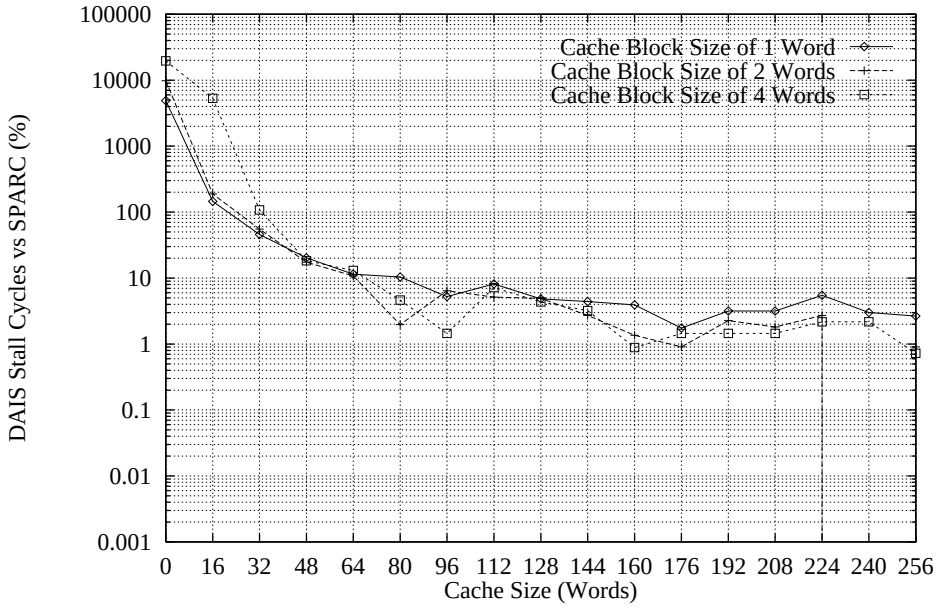


Figure A.8: Stack Depth 1 {0,1} With Varying Stack Cache Line Length (DeTEX)

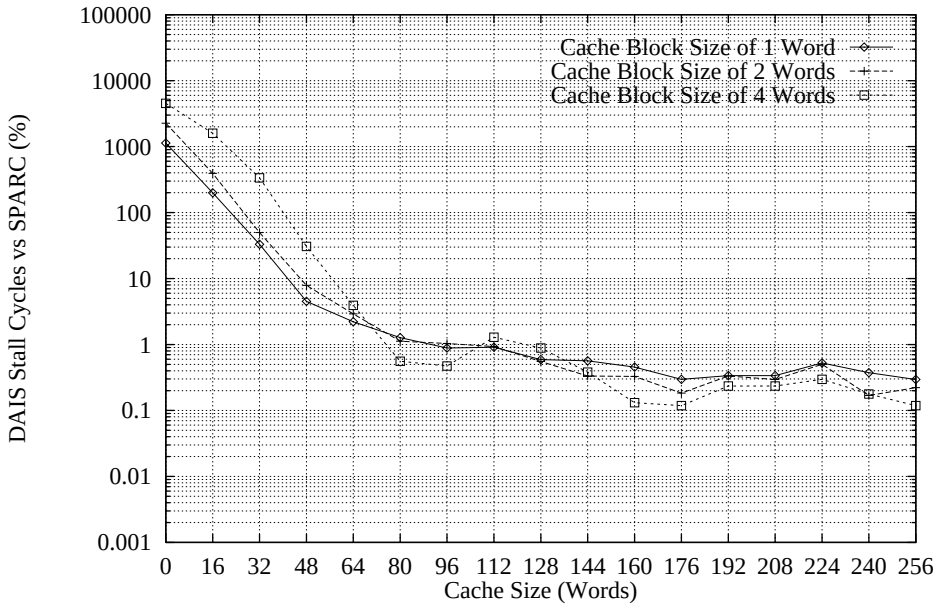


Figure A.9: Stack Depth 1 {0,1} With Varying Stack Cache Line Length (fig2dev)

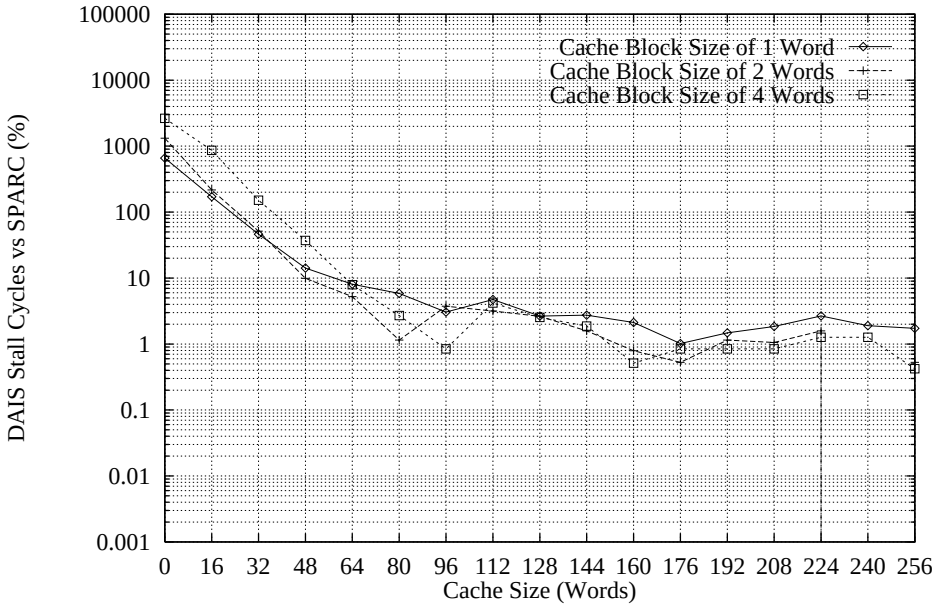


Figure A.10: Stack Depth 1 {0,1} With Varying Stack Cache Line Length (zoo)

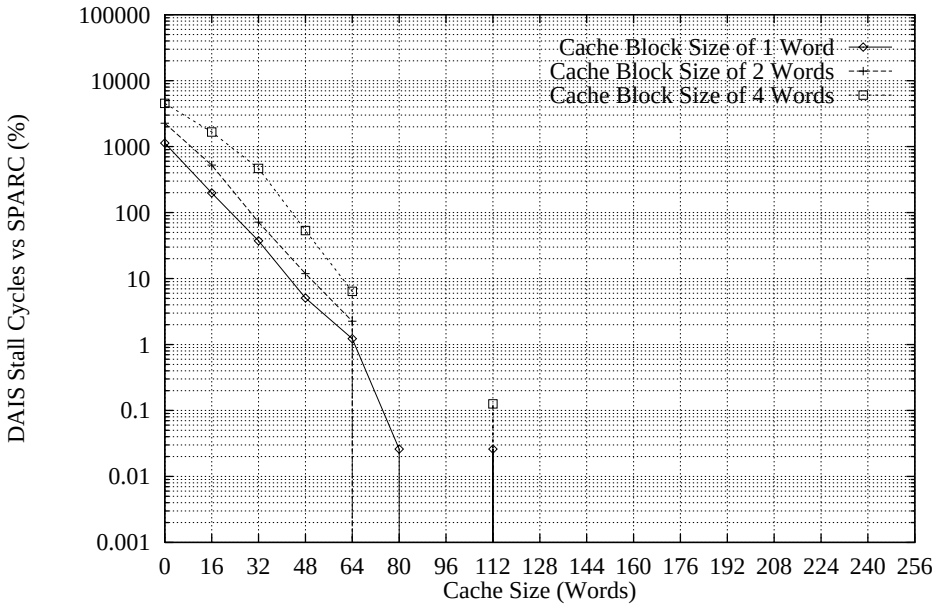


Figure A.11: Stack Depth 1 {0,0} With Varying Stack Cache Line Length (fig2dev)

Table
A.2 A.3 A.4 A.5 A.6 A.7 A.8 A.9 A.10 A.11 A.12 A.13 A.14 A.15 A.16 A.17 A.18 A.19 A.20 A.21 A.22 A.23 A.24 A.25
6.5
6.6
A.1
A.2
6.7
6.8
A.3
A.4
6.10
6.12
A.5
A.6
A.7
A.8
A.9
A.10
6.9
6.11
A.11
A.12
Figure

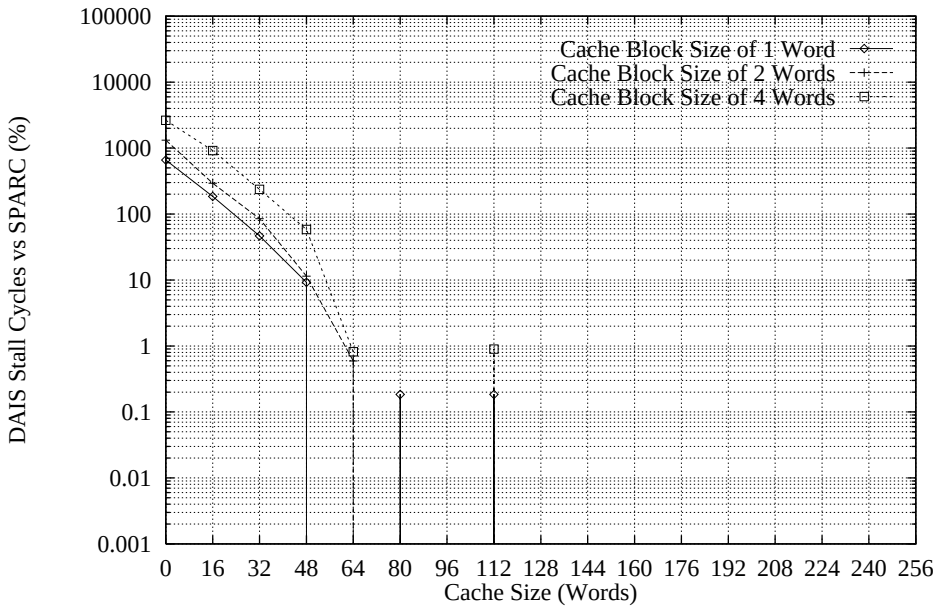


Figure A.12: Stack Depth 1 {0,0} With Varying Stack Cache Line Length (zoo)

Table A.1: SPARC Spill/Refill Analysis

Benchmark	Stall Cycles	
	Spills	Refills
TeX	386880	386592
DeTEX	1920	1728
fig2dev	27072	26880
Zoo	4128	3840

Table A.2: Stack Spill/Refill Analysis for TEX (depth 2)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	8313623	7750170	13005673	13569039	–	2548950	2548860
16	93.7	748983	1556794	1039352	1372198	98.3	407221	587363
32	99.2	110121	239511	136285	159518	99.8	50490	79596
48	99.9	21217	43778	22602	25771	100.0	10875	17614
64	100.0	5348	11727	5410	6064	100.0	4718	7349
80	100.0	2405	5363	2569	2654	100.0	2170	3516
96	100.0	1117	2081	924	839	100.0	416	757
112	100.0	403	806	423	267	100.0	75	138
128	100.0	240	439	248	206	100.0	67	118
144	100.0	219	410	157	145	100.0	48	72
160	100.0	79	185	148	101	100.0	33	34
176	100.0	170	176	113	77	100.0	26	52
192	100.0	68	129	109	62	100.0	26	27
208	100.0	67	117	118	49	100.0	20	40
224	100.0	91	188	157	86	100.0	28	72
240	100.0	90	200	149	74	100.0	21	14
256	100.0	103	234	114	76	100.0	20	14

Table A.3: Stack Spill/Refill Analysis for DeTEX (depth 2)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	79293	81918	130659	127977	–	42180	42120
16	98.7	1422	4371	2046	1087	99.3	1518	2557
32	99.6	442	1475	670	307	99.8	399	651
48	99.8	179	652	318	172	100.0	70	146
64	99.9	72	293	180	62	100.0	7	20
80	99.9	42	169	105	36	100.0	0	0
96	100.0	32	79	66	12	100.0	0	0
112	100.0	52	179	81	19	100.0	0	0
128	100.0	17	102	67	8	100.0	0	0
144	100.0	41	64	36	6	100.0	0	0
160	100.0	18	77	52	12	100.0	0	0
176	100.0	18	42	24	0	100.0	0	0
192	100.0	17	54	39	15	100.0	0	0
208	100.0	17	45	32	4	100.0	0	0
224	100.0	16	86	68	11	100.0	0	0
240	100.0	9	89	61	13	100.0	0	0
256	100.0	15	69	34	6	100.0	0	0

Table A.4: Stack Spill/Refill Analysis for fig2dev (depth 2)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	159110	288433	404836	275456	–	126690	126630
16	87.7	39288	96917	47761	47269	96.6	19377	30875
32	98.4	5679	15385	6062	4553	99.5	3299	5898
48	99.7	890	2999	1035	502	99.9	420	965
64	99.9	264	976	387	103	100.0	84	193
80	100.0	93	443	173	68	100.0	0	0
96	100.0	85	215	132	33	100.0	0	0
112	100.0	88	291	101	21	100.0	0	0
128	100.0	25	114	80	4	100.0	0	0
144	100.0	44	109	47	8	100.0	0	0
160	100.0	36	97	69	21	100.0	0	0
176	100.0	32	76	45	14	100.0	0	0
192	100.0	25	61	59	36	100.0	0	0
208	100.0	45	64	39	20	100.0	0	0
224	100.0	27	128	78	15	100.0	0	0
240	100.0	18	96	73	14	100.0	0	0
256	100.0	32	93	38	3	100.0	0	0

Table A.5: Stack Spill/Refill Analysis for zoo (depth 2)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	19885	29213	190067	180652	–	9750	9660
16	97.6	3082	9403	4118	1929	96.7	3027	4787
32	99.3	806	3429	1258	572	99.8	529	848
48	99.8	162	657	307	118	100.0	77	101
64	99.9	81	278	143	35	100.0	7	20
80	99.9	45	220	108	30	100.0	0	0
96	100.0	38	115	67	9	100.0	0	0
112	100.0	55	178	57	12	100.0	0	0
128	100.0	21	95	56	8	100.0	0	0
144	100.0	38	89	46	8	100.0	0	0
160	100.0	23	80	40	15	100.0	0	0
176	100.0	17	28	18	0	100.0	0	0
192	100.0	23	43	33	13	100.0	0	0
208	100.0	32	49	30	7	100.0	0	0
224	100.0	20	78	57	11	100.0	0	0
240	100.0	14	79	49	10	100.0	0	0
256	100.0	20	69	29	6	100.0	0	0

Table A.6: Stack Spill/Refill Analysis for TEX (depth 3)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	2091355	2779883	5990801	5302186	–	1178910	1178820
16	91.5	323041	721854	484457	604851	98.0	173818	256816
32	98.9	51772	116445	62999	68850	99.7	29817	46903
48	99.8	10027	23366	11794	12264	99.9	6735	11036
64	99.9	3685	8477	3815	4143	100.0	3111	5546
80	100.0	1726	3900	1777	1894	100.0	1326	2414
96	100.0	522	980	478	455	100.0	90	210
112	100.0	246	587	335	244	100.0	104	206
128	100.0	123	333	196	112	100.0	35	93
144	100.0	169	347	210	134	100.0	54	61
160	100.0	88	187	92	83	100.0	27	47
176	100.0	132	191	109	114	100.0	56	81
192	100.0	42	110	97	55	100.0	21	40
208	100.0	127	153	100	62	100.0	7	7
224	100.0	89	202	113	97	100.0	27	27
240	100.0	67	128	104	31	100.0	27	20
256	100.0	72	124	106	42	100.0	21	21

Table A.7: Stack Spill/Refill Analysis for DeTEX (depth 3)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	40847	28880	43921	55831	–	3120	3060
16	98.1	788	2781	1381	440	98.7	974	1496
32	99.4	259	1026	399	203	99.7	280	501
48	99.8	103	331	184	57	100.0	28	105
64	99.9	50	188	90	16	100.0	0	0
80	99.9	17	115	88	22	100.0	0	0
96	99.9	32	116	66	14	100.0	0	0
112	99.9	38	141	102	17	100.0	0	0
128	99.9	16	109	61	13	100.0	0	0
144	99.9	21	71	42	5	100.0	0	0
160	100.0	13	37	29	5	100.0	0	0
176	100.0	10	35	25	0	100.0	0	0
192	100.0	13	39	36	10	100.0	0	0
208	100.0	31	53	25	3	100.0	0	0
224	99.9	26	92	65	18	100.0	0	0
240	100.0	3	73	53	9	100.0	0	0
256	100.0	12	49	44	7	100.0	0	0

Table A.8: Stack Spill/Refill Analysis for fig2dev (depth 3)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	77380	118769	181562	140116	–	67170	67110
16	86.7	21810	49272	18764	17629	96.3	10025	14797
32	97.8	3215	9256	3477	2110	99.4	1539	3165
48	99.6	417	1835	682	234	99.9	335	548
64	99.9	184	630	229	62	100.0	14	14
80	99.9	95	341	164	38	100.0	0	0
96	99.9	83	248	99	23	100.0	0	0
112	99.9	57	197	139	18	100.0	0	0
128	100.0	30	185	89	11	100.0	0	0
144	100.0	31	99	60	5	100.0	0	0
160	100.0	23	76	47	20	100.0	0	0
176	100.0	26	68	44	2	100.0	0	0
192	100.0	24	60	46	10	100.0	0	0
208	100.0	38	70	39	20	100.0	0	0
224	100.0	36	131	76	13	100.0	0	0
240	100.0	11	82	66	8	100.0	0	0
256	100.0	17	76	53	7	100.0	0	0

Table A.9: Stack Spill/Refill Analysis for zoo (depth 3)

Cache Entries	With Asynchronous Transfers					Without Asynchronous Transfers		
	Hit Rate	Stall Cycles		Asynch Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	Spills	Refills	%	Spills	Refills
0	–	6436	10591	16448	12206	–	6660	6570
16	82.0	1612	6380	2857	837	92.5	1633	2633
32	95.2	408	1925	733	206	98.5	322	640
48	98.8	113	490	188	71	99.8	35	93
64	99.3	63	245	105	12	100.0	0	0
80	99.6	18	133	80	16	100.0	0	0
96	99.6	33	118	65	12	100.0	0	0
112	99.5	34	133	92	18	100.0	0	0
128	99.7	19	86	51	10	100.0	0	0
144	99.7	24	85	53	5	100.0	0	0
160	99.9	17	31	25	4	100.0	0	0
176	99.9	16	34	19	1	100.0	0	0
192	99.8	16	58	40	11	100.0	0	0
208	99.8	31	53	25	3	100.0	0	0
224	99.7	24	112	60	10	100.0	0	0
240	99.8	7	57	35	4	100.0	0	0
256	99.8	17	41	39	8	100.0	0	0

Table A.10: Spill/Refill Analysis of TEX, Stack Depth 1 (no asyn.), varying Hysteresis

Cache Entries	Watermarks 0 and 1			Watermarks 0 and 0			Watermarks 1 and 1		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills	%	Spills	Refills
0	–	9352830	9352740	–	9352830	9352740	–	9352830	9352740
16	91.3	812497	1337176	96.9	801095	1491423	95.4	962423	1498790
32	98.9	96929	169874	99.7	88717	175844	99.5	113988	194653
48	99.8	17395	30353	99.9	16126	32503	99.9	21889	37132
64	99.9	6188	10689	100.0	4967	10389	100.0	7294	12219
80	100.0	2891	4897	100.0	2111	4480	100.0	3353	5991
96	100.0	1022	1884	100.0	606	1424	100.0	1099	1856
112	100.0	553	896	100.0	117	122	100.0	588	922
128	100.0	364	537	100.0	27	66	100.0	406	692
144	100.0	350	394	100.0	54	53	100.0	371	501
160	100.0	224	242	100.0	27	21	100.0	252	320
176	100.0	210	235	100.0	14	14	100.0	224	323
192	100.0	189	214	100.0	21	66	100.0	210	215
208	100.0	203	201	100.0	28	40	100.0	217	319
224	100.0	238	315	100.0	41	47	100.0	217	228
240	100.0	175	226	100.0	7	7	100.0	161	179
256	100.0	161	264	100.0	21	34	100.0	210	273

Table A.11: Spill/Refill Analysis of DeTEX, Stack Depth 1 (no asyn.), varying Hysteresis

Cache Entries	Watermarks 0 and 1			Watermarks 0 and 0			Watermarks 1 and 1		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills	%	Spills	Refills
0	–	108420	108360	–	108420	108360	–	108420	108360
16	97.7	2506	3905	99.0	2596	4385	98.7	2695	4044
32	99.3	798	1229	99.8	672	1424	99.5	994	1592
48	99.7	364	530	99.9	209	416	99.8	420	567
64	99.8	217	288	100.0	0	0	99.9	266	480
80	99.8	168	291	100.0	0	0	99.9	203	291
96	99.9	119	112	100.0	0	0	99.9	182	245
112	99.9	161	199	100.0	0	0	99.9	161	225
128	99.9	91	123	100.0	0	0	100.0	91	115
144	99.9	91	104	100.0	0	0	100.0	112	143
160	99.9	77	96	100.0	0	0	100.0	70	95
176	100.0	42	35	100.0	0	0	100.0	63	69
192	99.9	70	70	100.0	0	0	100.0	42	42
208	99.9	70	70	100.0	0	0	100.0	56	94
224	99.9	105	137	100.0	0	0	100.0	91	91
240	99.9	70	63	100.0	0	0	100.0	42	42
256	100.0	49	68	100.0	0	0	100.0	70	96

Table A.12: Spill/Refill Analysis of fig2dev, Stack Depth 1 (no asyn.), varying Hysteresis

Cache Entries	Watermarks 0 and 1			Watermarks 0 and 0			Watermarks 1 and 1		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills	%	Spills	Refills
0	–	304860	304800	–	304860	304800	–	304860	304800
16	86.5	41601	65880	93.5	37979	68917	92.8	46200	70764
32	97.8	6615	11227	99.1	6361	13778	99.0	7448	12589
48	99.7	938	1501	99.9	888	1853	99.8	1190	1993
64	99.8	497	696	100.0	182	478	99.9	595	1058
80	99.9	273	414	100.0	7	7	100.0	308	405
96	99.9	217	261	100.0	0	0	100.0	245	395
112	99.9	210	280	100.0	0	0	100.0	252	332
128	100.0	140	178	100.0	0	0	100.0	133	163
144	100.0	140	166	100.0	0	0	100.0	168	218
160	100.0	126	119	100.0	0	0	100.0	147	229
176	100.0	77	83	100.0	0	0	100.0	126	151
192	100.0	91	91	100.0	0	0	100.0	119	157
208	100.0	91	91	100.0	0	0	100.0	119	189
224	100.0	126	158	100.0	0	0	100.0	98	123
240	100.0	98	104	100.0	0	0	100.0	56	88
256	100.0	70	89	100.0	0	0	100.0	105	111

Table A.13: Spill/Refill Analysis of zoo, Stack Depth 1 (no asyn.), varying Hysteresis

Cache Entries	Watermarks 0 and 1			Watermarks 0 and 0			Watermarks 1 and 1		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills	%	Spills	Refills
0	–	25020	24930	–	25020	24930	–	25020	24930
16	80.1	5061	7907	98.2	5154	8907	94.4	5446	8400
32	94.7	1358	2153	99.6	1096	2448	99.0	1484	2285
48	98.4	399	676	99.9	189	513	99.8	518	789
64	99.1	245	368	100.0	0	0	99.9	336	462
80	99.3	161	284	100.0	0	0	99.9	196	245
96	99.6	119	112	100.0	0	0	100.0	140	177
112	99.4	154	205	100.0	0	0	100.0	140	197
128	99.7	91	110	100.0	0	0	100.0	105	124
144	99.7	91	117	100.0	0	0	100.0	112	131
160	99.7	84	77	100.0	0	0	100.0	77	102
176	99.9	42	35	100.0	0	0	100.0	70	56
192	99.8	56	56	100.0	0	0	100.0	42	42
208	99.8	70	70	100.0	0	0	100.0	63	88
224	99.7	98	104	100.0	0	0	100.0	91	110
240	99.8	56	88	100.0	0	0	100.0	42	42

Table A.14: Spill/Refill Analysis of TEX, Stack Depth 2 (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	5097900	5097720	–	10195800	10195440
16	98.1	636591	1034278	95.0	1895545	4245820
32	99.9	64185	112253	99.2	532774	1136378
48	100.0	13049	23761	100.0	36268	75188
64	100.0	4310	8405	100.0	8098	16542
80	100.0	1931	3859	100.0	3495	7662
96	100.0	561	1131	100.0	1566	3881
112	100.0	116	196	100.0	508	914
128	100.0	96	152	100.0	64	130
144	100.0	84	121	100.0	175	359
160	100.0	77	171	100.0	0	0
176	100.0	26	44	100.0	81	124
192	100.0	45	39	100.0	137	174
208	100.0	64	64	100.0	19	25
224	100.0	64	77	100.0	57	99
240	100.0	39	64	100.0	0	0
256	100.0	26	26	100.0	100	100

Table A.15: Spill/Refill Analysis of DeTEX, Stack Depth 2 (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	84360	84240	–	168720	168480
16	99.2	2241	4345	94.9	21667	52341
32	99.8	541	1099	99.8	1335	3137
48	100.0	85	203	100.0	254	772
64	100.0	0	0	100.0	0	0
80	100.0	0	0	100.0	0	0
96	100.0	0	0	100.0	0	0
112	100.0	0	0	100.0	0	0
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Table A.16: Spill/Refill Analysis of fig2dev, Stack Depth 2 (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	253380	253260	–	506760	506520
16	96.2	37694	63365	92.3	81980	198436
32	99.7	3343	6173	99.0	17064	35646
48	100.0	595	1174	99.8	3326	7351
64	100.0	98	185	100.0	463	1094
80	100.0	0	0	100.0	0	0
96	100.0	0	0	100.0	0	0
112	100.0	0	0	100.0	0	0
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Table A.17: Spill/Refill Analysis of zoo, Stack Depth 2 (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	19500	19320	–	39000	38640
16	98.5	4404	8475	97.6	7731	21086
32	99.8	731	1559	99.6	2697	5631
48	100.0	150	335	99.9	400	1046
64	100.0	0	0	100.0	0	0
80	100.0	0	0	100.0	0	0
96	100.0	0	0	100.0	0	0
112	100.0	0	0	100.0	0	0
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Table A.18: Spill/Refill Analysis of TEX, Stack Depth 1 {0,1} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	18705660	18705480	–	37411320	37410960
16	90.7	849457	2439849	61.0	3397345	19708564
32	99.2	71323	208662	95.2	437381	2301991
48	99.9	10934	32356	99.6	30954	173176
64	100.0	3640	10568	100.0	4284	24221
80	100.0	1631	4870	100.0	1918	10549
96	100.0	560	1757	100.0	917	5042
112	100.0	294	736	100.0	525	3016
128	100.0	210	453	100.0	147	798
144	100.0	147	305	100.0	105	418
160	100.0	133	380	100.0	28	50
176	100.0	70	168	100.0	49	175
192	100.0	112	233	100.0	77	306
208	100.0	84	194	100.0	49	206
224	100.0	154	375	100.0	77	281
240	100.0	84	194	100.0	35	125
256	100.0	84	162	100.0	56	200

Table A.19: Spill/Refill Analysis of DeTEX, Stack Depth 1 {0,1} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	216840	216720	–	433680	433440
16	97.9	2373	5934	65.2	36141	198476
32	99.4	644	1796	99.2	882	3875
48	99.8	168	599	99.9	196	612
64	99.9	133	342	99.9	84	493
80	100.0	35	52	100.0	49	156
96	99.9	77	207	100.0	14	50
112	99.9	84	143	100.0	49	268
128	99.9	63	155	100.0	42	150
144	100.0	42	78	100.0	42	100
160	100.0	21	39	100.0	14	25
176	100.0	14	26	100.0	14	50
192	100.0	35	65	100.0	14	50
208	100.0	28	52	100.0	14	50
224	100.0	42	78	100.0	21	75
240	100.0	0	0	100.0	21	75
256	100.0	14	26	100.0	7	25

Table A.20: Spill/Refill Analysis of fig2dev, Stack Depth 1 {0,1} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	609720	609600	–	1219440	1219200
16	81.5	56518	155308	53.8	137235	725228
32	97.7	6713	20195	90.6	26082	154921
48	99.6	1001	3251	99.1	2576	14019
64	99.9	357	1214	99.9	350	1765
80	99.9	168	438	100.0	70	231
96	100.0	147	413	100.0	49	206
112	100.0	154	349	100.0	105	592
128	100.0	98	201	100.0	84	393
144	100.0	63	117	100.0	56	150
160	100.0	42	135	100.0	21	50
176	100.0	28	71	100.0	14	50
192	100.0	63	117	100.0	21	106
208	100.0	49	110	100.0	21	106
224	100.0	77	194	100.0	35	125
240	100.0	14	77	100.0	21	75
256	100.0	42	78	100.0	14	50

Table A.21: Spill/Refill Analysis of zoo, Stack Depth 1 {0,1} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	50040	49860	–	100080	99720
16	81.9	4739	11717	57.1	10535	55087
32	95.8	1078	2817	92.1	2065	9431
48	99.2	203	550	98.1	525	2295
64	99.6	119	278	99.6	84	518
80	99.9	35	52	99.8	49	156
96	99.7	77	207	100.0	14	50
112	99.7	84	156	99.8	49	268
128	99.8	63	136	99.9	42	150
144	99.9	42	78	99.9	42	100
160	99.9	21	39	100.0	14	25
176	100.0	14	26	100.0	14	50
192	99.9	35	52	100.0	14	50
208	99.9	28	52	100.0	14	50
224	99.9	42	78	99.9	21	75
240	100.0	0	0	99.9	21	75
256	100.0	14	26	100.0	7	25

Table A.22: Spill/Refill Analysis of TEX, Stack Depth 1 {0,0} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	18705660	18705480	–	37411320	37410960
16	96.3	1409890	3315249	85.4	4503585	20499648
32	99.7	138120	298102	98.2	1004495	3232730
48	100.0	24039	48592	99.9	92022	255014
64	100.0	8048	15622	100.0	16545	42390
80	100.0	4044	8293	100.0	6433	16874
96	100.0	1066	2071	100.0	3735	8723
112	100.0	0	0	100.0	1238	2996
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Table A.23: Spill/Refill Analysis of DeTEX, Stack Depth 1 {0,0} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	216840	216720	–	433680	433440
16	99.2	3336	7163	85.7	41464	205458
32	99.8	958	2181	99.7	2122	6146
48	100.0	256	485	100.0	360	819
64	100.0	7	38	100.0	19	43
80	100.0	0	0	100.0	0	0
96	100.0	0	0	100.0	0	0
112	100.0	0	0	100.0	0	0
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Table A.24: Spill/Refill Analysis of fig2dev, Stack Depth 1 {0,0} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	609720	609600	–	1219440	1219200
16	90.6	78497	204390	78.4	160734	739354
32	99.0	11293	27426	95.7	51463	200224
48	99.8	2018	4400	99.6	7166	21671
64	100.0	341	878	100.0	890	2563
80	100.0	0	0	100.0	0	0
96	100.0	0	0	100.0	0	0
112	100.0	0	0	100.0	0	0
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Table A.25: Spill/Refill Analysis of zoo, Stack Depth 1 {0,0} (no asyn.), varying Cache Block Size

Cache Entries	Block Size of 2 Words			Block Size of 4 Words		
	Hit Rate	Stall Cycles		Hit Rate	Stall Cycles	
	%	Spills	Refills	%	Spills	Refills
0	–	50040	49860	–	100080	99720
16	97.9	6931	15269	95.5	13808	55861
32	99.6	2109	4344	99.3	4726	13230
48	99.9	310	560	99.8	1080	3339
64	100.0	7	38	100.0	19	43
80	100.0	0	0	100.0	0	0
96	100.0	0	0	100.0	0	0
112	100.0	0	0	100.0	0	0
128	100.0	0	0	100.0	0	0
144	100.0	0	0	100.0	0	0
160	100.0	0	0	100.0	0	0
176	100.0	0	0	100.0	0	0
192	100.0	0	0	100.0	0	0
208	100.0	0	0	100.0	0	0
224	100.0	0	0	100.0	0	0
240	100.0	0	0	100.0	0	0
256	100.0	0	0	100.0	0	0

Appendix B

Garbage Collection

With no explicit deletion of objects, the object-oriented heap must be able to automatically detect whenever an object becomes inaccessible. Once a dead object has been located, the space which it has been utilizing should be returned to the heap-space allocation pool. This is the task of the *garbage collector*.

An object is considered to be inaccessible once it can not be reached, either indirectly or directly, through a root object. These root objects can be defined by the operating system, but typically represent the single node at the top of the structure which connects all objects together on the non-volatile storage medium, and the object identifiers contained within the machine's registers and residing on the stack.

B.1 Garbage Collection Algorithms

Garbage collectors are generally partitioned into two phases; detection (phase 1) and removal (phase 2). Sometimes a third phase is also required, that of compaction. Consider the effect of randomly allocating and deallocating areas of the heap: eventually, the heap will contain a large number of allocated areas, each of differing sizes, with random distribution. Now the largest area which can be allocated is not equal to the total amount of free space, but instead is the size of the largest free block (which is probabilistically speaking radically different). Under such circumstances the heap is said to be *fragmented*. It may therefore be convenient to rearrange the segments into a compact mass at one end of the storage area (*i.e.* perform heap *compaction*), thus increasing the maximum size of any future allocations, and also reducing the effort in identifying unallocated areas of the heap. It may possible to avoid (or at least minimize) the need for compaction by providing the deallocator with a method of joining together contiguous unallocated regions. Attempting to avoid compaction is certainly attractive, so as to avoid the overhead in copying objects about the heap. Additionally, since the object addresses may change at any time during compaction, any change in object location must be immediately notified to the function responsible for converting identifiers into addresses. This could cause complications if

an object address had been loaded by an object access primitive, and then before the address could be utilized, the object was relocated.

Note that in garbage collection, it is essential that the object identifiers can be easily located amongst the other data (*e.g.* integers and floats) which may be contained within the objects. With a system which provides tagging of identifiers, this is not a problem. In other systems, some sort of higher level knowledge on each object must be obtained (*e.g.* the application may be able to inform the garbage collector on the location of object identifiers within its objects through a procedural interface). It is also possible to consider all data elements as object identifiers, which results in all reachable and some unreachable objects being considered as live (*i.e.* non-garbage). However, those unreachable objects which were mistakenly left uncollected may be numerous, and in the long term use up valuable resources (such as non-volatile storage space) [Wentworth, 1990].

There are a number of different approaches used in the design of garbage collectors, of which the main classes are: namely...

- Reference Counting
- Marking and Sweep
- Semispace Copy collection
- Generation Scavenging

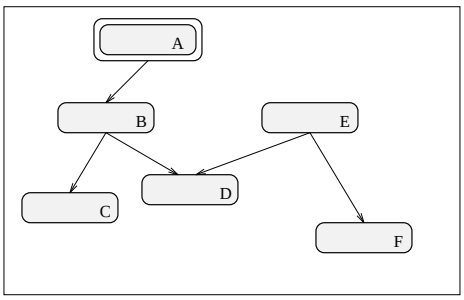
In discussions concerning garbage collection, the term *mutator* is used to describe the application which makes all object creation requests, while the garbage collector itself is referred to as the *collector*. More information on garbage collection techniques can also be found in [Cohen, 1981].

B.1.1 Reference Counting

In reference counting, the heap management system maintains a record of the number of times that an object identifier appears within the heap. If the count for any identifier should ever fall to zero, then the object to which it refers must be inaccessible, and therefore ready for deletion.

In order to maintain the identifier occurrence record, every store operation is interrupted and its effect examined. If the operation would overwrite an object identifier, then the count for that identifier is decremented by one. If the data to be stored is an object identifier, then the identifier's corresponding count is incremented by one. Once these two checks have been made, the store operation is allowed to continue.

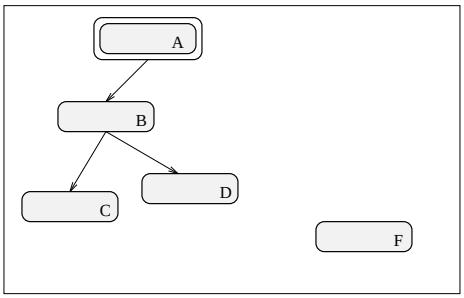
If any count is decremented to zero, then that object is marked for garbage collection. Objects so marked are recursively scanned (following all object identifiers), and the count for each identifier found is decrementing. This operation may result in even more garbage being detected. In typical implementations of reference counting, objects marked for collection are not scanned immediately, but deferred until a second program can be activated, which can process a number of object removals in one operation.



Node Identifier	Reference Count
A	(Root)
B	1
C	1
D	2
E	0
F	1

Figure B.1: Initial Heap for Reference Count

An example of reference count garbage collection can be seen in figure B.1. This combined diagram and table arrangement provides a simplistic, object based pictorial representation of the heap and the corresponding reference count. Object A is defined as a root object (*i.e.* it is never garbage collected), thus forming a permanent anchor to which all persistent heap objects should be directly or indirectly connected. In the heap depicted, object E is not referenced by any object. Its corresponding table entry shows that the reference count for object E is zero, therefore indicating that object E should be removed from the heap. In deleting the object, all references contained within the object are examined, and the entries within the reference table are updated. This new state is shown in figure B.2.

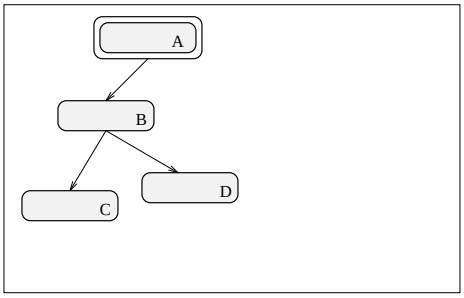


Node Identifier	Reference Count
A	(Root)
B	1
C	1
D	1
E	(Deleted)
F	0

Figure B.2: First Pass for Reference Count

At this point, another entry within the reference table is found to be zero (that of object F), and so this too can be deleted. With no more counts equal to zero, there is no more detectable garbage to collect. The final state of the heap is shown in figure B.3.

The main advantage of reference counting is that the memory required to perform the garbage collection is generally the smallest out of all other available mechanisms. This must be balanced with the considerable overhead in maintaining the reference table, remembering that for each store the old contents of the object must be read (allowing the count for any overwritten identifier to be decremented) before the new reference can be written (which requires the entry for any written identifier to be incremented). This can consume 15% of the available CPU time. Once a reference count reaches zero, it must be scanned so that all objects which it references can have their reference count decremented.



Node Identifier	Reference Count
A	(Root)
B	1
C	1
D	1
E	(Deleted)
F	(Deleted)

Figure B.3: Second Pass for Reference Count

This is a recursive operation, which can consume an additional 5%. This overhead totals up to a fifth of the target architecture's processing power [Ungar, 1984].

A variation on general (immediate) reference counting was generated by Deutsch and Bobrow [Deutsch and Bobrow, 1976] to counteract the overhead of maintained reference counts. Instead of interrupting the store primitives, a record of which objects have been altered is maintained. Before any attempt is made to identify dead objects, the system halts and the reference count updated to reflect all object changes made since the last update. This technique, which is known as *deferred reference counting*, utilizes only a marginal larger amount of memory than the immediate reference counting approach.

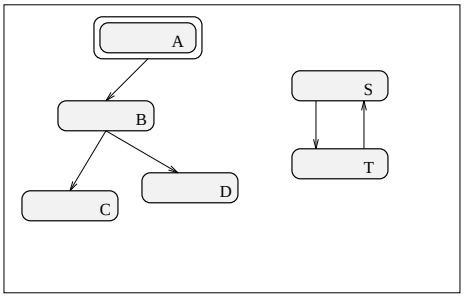
Deutsch and Bobrow also indicate that a high percentage of objects will only be referenced once. Taking advantage of this fact, it is possible to construct a hashing table, which takes object references as an input, and contains an entry for every object which has a reference count greater than one (thus saving memory).

From measurements taken using a deferred reference system, about 3% of the processor time is taken up with manipulation of reference counts, another 3% for updating the reference table, and 5% for the recursive freeing of collected objects. This totals in at about 11%, or roughly half the time required by the immediate reference counting algorithm [Ungar, 1984].

The performance could be improved by using a hardware device which can indicate whether a particular object has been written to since the last deferred count update [Appel *et al.*, 1988].

Unfortunately, reference counting contains a more serious disadvantage; although no object which could be accessed by another object can been accidentally removed through this method, not all of types of garbage can be identified. Consider the example shown in figure B.4. In this case, two objects (namely S and T) have no link with the root object, and are therefore garbage, but the table shows that all reference counts are greater than zero. Objects S and T are indeed referenced, although only by each other. Such garbage in known as *cyclic*. Since many modern programming languages allow the use of cyclic constructs (*i.e.* doubly-linked lists), failure to allow cyclic garbage to be identified a clear drawback with this mechanism. It may be possible to instead use reference counting for the majority of the time (since most garbage is not usually circular data structures), and use (less frequently) other more complex (and less memory efficient) collection techniques which can identify these circular structures

[Deutsch and Bobrow, 1976].



Node Identifier	Reference Count
A	(Root)
B	1
C	1
D	1
S	1
T	1

Figure B.4: Cyclic garbage failure in reference count

B.1.2 Marking and Sweep

In order to overcome the problem or reclaiming cyclic data structures, a new approach was undertaken, that of *Mark and Sweep*. In this system, a three stage mechanism was introduced.

1. Mark phase
2. Deallocation sweep
3. Compression phase

In the first stage of the mark and sweep algorithm, all objects reachable from root nodes (*i.e.* the objects which are not garbage) are traversed in a breadth or depth first arrangement, marking all examined objects in some manner. The advantage of a depth first scan is the simplicity of the design, since it relies on recursion. However breadth first, although more complex to construct, has the advantage of requiring the least stack space to operate, unlike the depth first scan which could need large areas of storage to hold the recursion stack.

Next, the garbage collector sweeps through the entire heap space, searching for objects which have not been marked by the initial traversal routine, and any such objects deallocated. Once all deallocations have been performed, the heap can be compressed (if required). Note that the traversal of all objects requires that other applications which may modify the heap be suspended until the end of the marking phase. This could be a problem in a real-time (*e.g.* interactive) environment.

B.1.3 Semispace Copy Collection

With the *copy collector* design [Baker, 1978], two distinct areas (or *semispaces* of the heap are maintained. During normal program execution, all objects are located within only one of these semispaces.

In a garbage collection cycle, the mechanism starts by searching through all reachable objects. Whenever a reachable object is located, it is copied to the second semispace, and the object address is updated as in the compaction mechanisms. Once all reachable objects have been copied, the system

can restart program execution. However, the semispace which was originally being used as the heap (the *from-space*) now becomes unused, while the semispace to which the objects were copied to (*i.e.* the *to-space*) becomes the object heap.

A slight variation on this type of collector is to, at the start of garbage collection, copy only the root objects to to-space. From then on any new object is created within the new to-space area, and a background task handles the remaining from-space to to-space transfers. If an attempt is made to access an object before it has been moved into to-space, then the relocation of that object is handled immediately. This gives a faster response time for application programs than that possible if all copying is done in one action.

One problem with this scheme of garbage collection is that a large amount of copying is necessary, even when no compaction is required. As a result, the locations of all objects are changed during every collection cycle (unless both of the semispaces somehow map onto the same physical addresses, *e.g.* using memory management). In addition, at any time only half of the total heap space is available for object allocation.

B.1.4 Generation Scavenging

For the garbage collectors discussed above, the amount of time and effort used in processing each object is largely independent of object life-span. However, analysis and observation indicates that, in the general case, young objects tend to die young, while old objects tend to continue to live. It is therefore desirable to construct a collection algorithm which can take advantage of this fact. Thus the creation of *Generation Scavenging* [Lieberman and Hewitt, 1983].

With this type of garbage collection, the heap is split into a number of *generations*, each containing objects of a similar range of ages. Live objects age within their particular generation until they are of a sufficient age to be *tenured* (*i.e.* promoted to the next generation). Objects residing within any particular generation are garbage collected more frequently than those in an correspondingly older generation, while the oldest generation is rarely (or just more slowly) collected. The process of performing garbage collection on a generation is known as *condemning*.

In order to avoid scavenging regions of older generations during a condemn cycle, object pointers which directly refer from older to younger generations are not directly supported. Instead, pointers which travel forward in time are handled through entry tables. When a generation is condemned, its own generation space and entry table, as well as those of any younger generation must be scavenged.

To better understand generation based garbage collection, consider the objects depicted within figure B.5. The objects {A,B,C,D,E,F,G} are coded in order of creation (with A being the oldest and G the youngest), with {D,C,F} representing unreachable (*i.e.* dead) objects. These objects have been created in a generation based heap (implemented using three generations; *Young*, *Mature* and *Venerable*). The object distribution for this heap arrangement can be seen in figure B.6. Notice that two references exist from older to younger objects (*i.e.* A to F and C to G), and these are implemented via an entry table. The entry table for the mature generation is empty, and for clarity is not shown in the diagram.

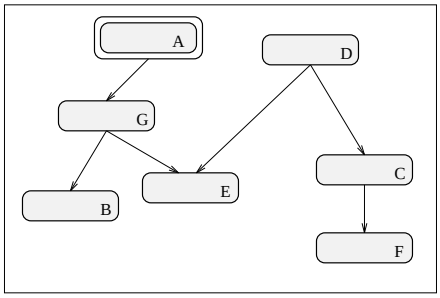


Figure B.5: Objects at the Start of Generation Scavenging

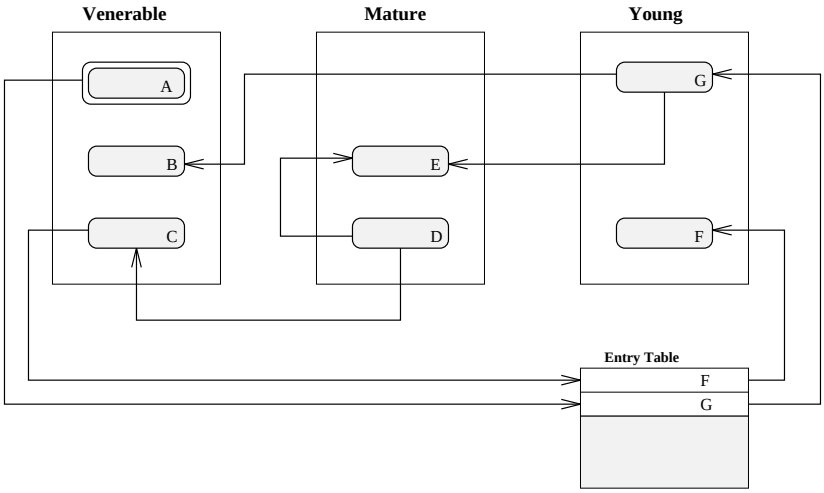


Figure B.6: Generation Mapping of Figure B.5

None of the garbage present in the heap of figure B.6 can be detected using an Young-based collection, but during a Mature scavenge, object D can be recognized as not being referenced from either another Mature object, a younger object, or the Mature generation entry table. With the removal of object D, the generation heap will look like figure B.7.

From the position shown in figure B.7, no more garbage can be detected until a Venerable scavenge is performed. Since no references to object C can be detected, this object can be safely deleted. Remember that whenever a collection takes place, all younger generations must be collected at the same time, and this results in the removal of object F from the Young generation (and from its entry table). Now all garbage has been removed from the heap (figure B.8).

Note that if dead object is tenured, it may cause other dead objects to be tenured instead of collected. This problem is known as *nepotism*. Analysis has shown that under normal circumstances, nepotism represents an insignificant effect on efficiency [Ungar and Jackson, 1988]. It is important, however, to make sure that any changes to the traditional garbage collection mechanism does not exacerbate this problem.

The CPU time required to perform this type of collection tends to be much smaller than the other systems previously examined. By implementing such generation based systems, large performance

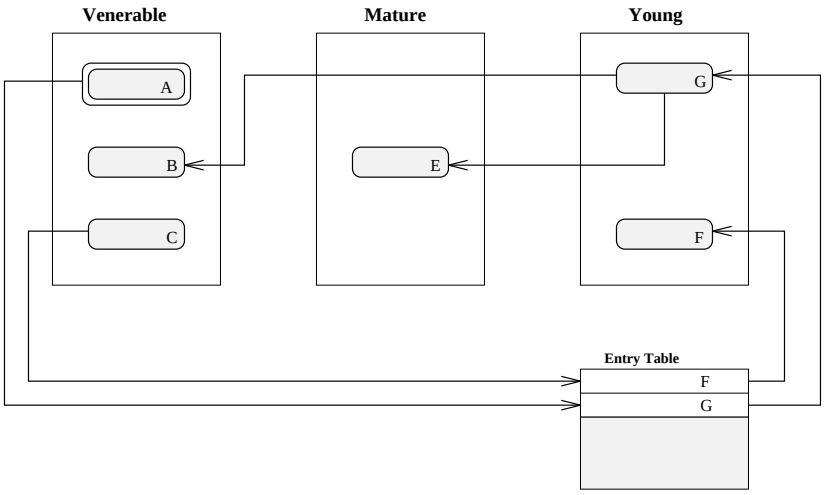


Figure B.7: Generation Mapping After Mature Collection

gains can be obtained over other mechanisms. Generation By way of an example, Berkeley Smalltalk was converted to generation garbage collection from reference counting, and as a result programs executed under this system showed an average 70% speed increase. It is probably on such performance improvements that generational heaps are becoming highly favoured by many researchers, especially systems using three or more generations [Wilson, 1991, Ungar and Jackson, 1988] (fewer than three generations require a significant larger amount of CPU time [Wilson and Moher, 1989]).

A number of researchers have suggested enhancements to the standard generational collection model depicted above:

1. Object tenuring need not be based on a fixed period of time (*e.g.* as is done in [Lieberman and Hewitt, 1983]). Instead, it is possible to use a feedback-mediated tenuring policy (where the system makes judgement on the objects to be tenured), permitting a high degree of flexibility, both with respect to pause times and also in limiting tenured garbage [Ungar and Jackson, 1988].

2. During marking phases, a significant performance increase can be realized if objects which do not contain object identifiers are dealt with separately, thus reducing the amount of information which must be scanned to detect all reachable objects [Ungar and Jackson, 1988].

3. Instead on basing the tenuring process on actual age, objects can be managed using a *bucket brigade* mechanism [Wilson and Moher, 1989, Shaw, 1988, Wilson, 1991]. In this scheme, a generation consists of a number of buckets, each holding objects of approximately the same age. Objects are created in the first bucket, and are transferred from bucket to bucket at every scavenge. All buckets within a generation are scavenged together, and those objects stored within the oldest bucket are tenured into the youngest bucket of the next-older generation.

4. Using a separate generation space for the youngest generation, which is reused and emptied at every collection cycle, improves locality [Ungar, 1984], increasing the effectiveness of both cache and virtual memory [Wilson, 1991].

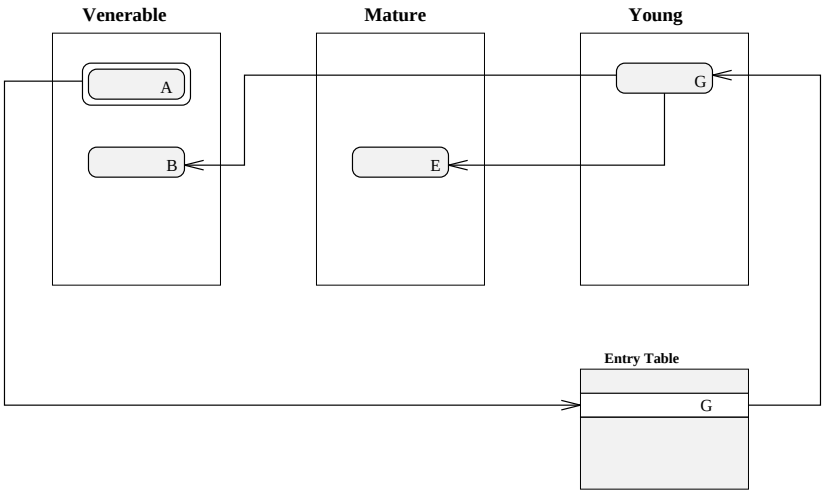


Figure B.8: Generation Mapping After Venerable Collection

Unfortunately, all the collection schemes examined this far have been developed with the belief that all objects can be accessed during garbage collection. This however incurs heavy I/O penalties, since some objects may have been moved onto non-volatile storage. If the target system also supports a networked heap (where remote machines can contain internally the object identifiers of local objects), then additional problems are created.

Appendix C

MultiThreads Programming Reference

The information presented below lists all available primitives in the MultiThreads library. The descriptions refer to the actions of the Unix version of MultiThreads. In DOLPHIN, no information is ever deleted by MultiThreads, and so data such as monitors, deleted elements, stacks, persist until all reference to them has been lost. All the programming functionality remains unchanged. THIS_THREAD always refers to the element id of the currently executing thread.

```
procedure mt_arm(
    INTEGER: interval
)
```

This call activates all the thread functions. It converts the current heavyweight process to a lightweight process of maximum priority. It places this within the DEFAULT_GROUP, which is defined to be preemptive. All the exception handlers are set to their defaults (ignore exceptions).

The parameter *interval* defines the resolution in microseconds of the MultiThread clock, on which all time-dependent facilities are based. Scheduling instances are activated every *interval*. and the time period over which a process sleeps is rounded up to the nearest *interval*. An interval of 100 will suffice for most problems.

function mt_thread(
    VOID : (*task_addr)()
    ELEMENT_ID : group
    INTEGER : priority
    VOID : *param
    MT_FLAGS : flags
) **return** ELEMENT_ID

This primitive creates a thread within the selected *group*. To insert the thread into the default group, *group* can be set to DEFAULT_GROUP. The address of the new thread is specified by *task_addr*, which is passed *param* as a single parameter. The thread runs at the specified *priority*, which if the parent thread is in the same group as group must be equal to or lower than the parent priority. Parent threads can create threads in child groups with any desired priority (up to MAXPRIO).

Threads may either be defined to run as soon as possible (*flags* set to MT_READY) or to start off in the HELD state (*flags* set to MT_HOLD). For more information on the HELD state, see **mt_hold()**.

function mt_create_group(
    ELEMENT_ID : parent
    ELEMENT_DATA : *(*scheduler)()
    INTEGER : priority
) **return** ELEMENT_ID

A new scheduling scheme can be introduced at any time, and is scheduled itself in an identical way to a thread. Creating a scheduling group of *priority* 50 in a timesliced scheduled which also contains an thread of priority 50 will result in the thread running for 50with the remaining time being shared amongst the threads contained within the scheduling group, as defined by that scheduler.

By default, two schedulers are currently defined; PREEMPTIVE_SCHEDULER and TIMESLICED_SCHEDULER. The *parent* is the element id of the scheduling group within which the new scheduler will execute. *priority* is simply the scheduling priority of that group within the scheduling scheme of the parent. Further scheduling types can be added at any time, since the *scheduler* parameter is simple a pointer to the scheduling function required.

Scheduling groups can be layered to any required depth. You can not create a group within a parent group; group can only be created within a child. Group priority is protected in the same way as for thread creation.

function mt_element_id **return** ELEMENT_ID

This function returns the element id of the currently executing thread.

```
function mt_setprio(
    ELEMENT_ID    : id
    INTEGER       : new_priority
) return INTEGER
```

The priority of an element, independent of whether it is a group or a thread, can be read or modified using the **mt_setprio** primitive. If *new_priority* is PRIO_INVALID, then the priority is not changed. If an error occurs during the primitive`s execution, PRIO_INVALID is returned, otherwise the old priority is returned.

An element`s priority can not be changed by a thread within a child group, and threads within the current group can only change the priority to between 0 and the priority at which the thread was created. Threads within parent groups can freely change the element's priority to any desired value (up to MAXPRIO).

```
procedure mt_hold(
    ELEMENT_ID    : id
)
```

Threads can exist in a number of states. Runnable threads have the state ELEMENT_ ACTIVE. This state can be changed to ELEMENT_HELD for any child element or element executing in the current group, as specified by *id*. Once in this state, the element is never scheduled. If the element is a group, then all elements within that group are suspended. Once held, an element can only be restarted using the **mt_restart()** primitive.

```
procedure mt_restart(
    ELEMENT_ID    : id
)
```

A thread *id* in state ELEMENT_HELD (see **mt_hold()**) or ELEMENT_ASLEEP (see **mt_sleep()**) can be restored to a runnable state (ELEMENT_ACTIVE) through the use of the **mt_restart()** primitive. Any thread can restart any element. If the thread is in some other state, then there is no effect.

```
procedure mt_switch(
    ELEMENT_ID    : group
    INTEGER       : priority
)
```

In a scheduling group using preemptive scheduling, the element with the highest priority is executed until that thread leaves the state ELEMENT_ACTIVE. If there are two or more elements with the highest priority, then one of the elements will be selected using some internal algorithm, while the

others will be treated as if they have a lower priority.

It may be desirable to reorder the selection scheme used to select the runnable element within a group of equal priority ones. This can be done using **mt_switch**, which reorders the elements contained within a particular scheduling *group* at the selected *priority*. This reordering is done in such a way that the previously selected element at that priority is moved to the end of the element list used to decide which element to execute at that priority. If there is only one element of that priority in the selected group, the call has no effect. This primitive allows simple scheduling algorithms to be implemented on top of the preemptive scheduling scheme. Reordering the elements of groups other that the preemptive case have scheduler specific results.

```
procedure mt_sleep(
    STRUCT TIMEVAL : *period
)
```

Elements can be suspended for a period of time using the **mt_sleep()** primitive. Once called, this primitive suspends the current thread for at least the time period specified by *period*. *period* is rounded to the next highest interval as specified by the **mt_arm()** primitive. Once the period has expired, the thread is returned to state ELEMENT_ACTIVE.

```
procedure mt_destroy(
    ELEMENT_ID    : id
)
```

Any thread within the current group or a parent group of element *id* can kill that element, independent of its current state or type. If a scheduling group is killed, then all elements contained therein are also killed. Killed threads which lie within monitors or that are exception handlers are handled as gracefully as possible, with the emphasis on keeping the system running (rather than causing deadlock). All resources assigned to that element (or those below it) are returned to the system (monitors which that element or a child element created, stack space, etc). However, the programmer should endeavour to clear up the system before killing off elements, as the system-based cleanup may not result in the desired actions.

```
function mt_get_element_data(
    ELEMENT_ID    : id
) return *ELEMENT_DATA
```

This primitive is available to allow implementation of low-level features based on the internal structures of MultiThreads. It is not intended to be used for setting any information unless the programmers know what they are doing. The data returned is a pointer to the actual data on element *id* used by the thread manager, and unregulated changes to its contents will almost certainly result in a crash.

function mt_makemon **return** *MT_MONITOR

Monitors are the preferred way to protect an area of code from being executed more than once concurrently. Within a monitored area, it is guaranteed than at most one thread instant can execute. Monitors can be created using the **mt_makemon()** primitive. Monitors die with the creating thread, releasing all other threads waiting for entry to that monitor.

```
procedure mt_monitor(
    MT_MONITOR *monitor
)
```

At the beginning of a monitored area, the **mt_monitor** primitive is specified, along with the address of the *monitor* in question (as created by **mt_makemon**). If there is already a thread executing within the specified monitor, then the current thread enters the state ELEMENT_WAITING, and is suspended until selected for entry into the monitor. If more that one thread is waiting, then the highest priority thread is selected first for entry into the monitor.

```
procedure mt_endmon(
    MT_MONITOR : *monitor
)
```

At the end of a monitored area started by **mt_monitor()**, **mt_endmon()** should be called. This indicates to the system that the monitored area is now clear of threads, and any waiting threads can be scheduled for entry into the area. If the thread is killed while in a monitored area, then **mt_monitor** is called on the threads behalf by the thread manager.

```
function mt_cmonitor(
    MT_MONITOR : *monitor
) return INTEGER
```

mt_cmonitor works in a similar way to **mt_monitor**, except that if the thread would have been suspended, the **mt_monitor** primitive is aborted, and the function returns 0. If the monitor is successfully entered, the function returns 1 instead. This allows non-blocking entry to monitors to be achieved.

```
procedure mt_exception_handler(
    EXCEPTION : exceptions
)
```

The MultiThreads environment supports a number of exceptions. For the unix implementation, there are 32 possible exceptions, which match one for one onto the standard Unix signals. If there

are no exception handlers for a particular exception, then that exception is ignored. If there are one or more handlers for that exception type, then the highest priority handler is selected which has not already rejected that exception.

Exception handlers are created as normal threads, and are linked into the thread manager as exception handlers using the **mt_exception_handler()** primitive. This takes a single parameter, *exceptions*, which contains a mask of the exception numbers which the current thread is interested in receiving.

function mt_waiton_exception **return** EXCEPTION

Once a thread is defined as an exception handler using **mt_exception_handler()**, exceptions to which the thread is interested in are queued until that thread calls **mt_waiton_exception()**. If there are no exceptions queued, then this primitive suspends the current thread until an exception of interest is queued. The actual exception which triggered the handler is returned.

procedure mt_accept_exception

Once a thread returns from **mt_waiton_exception()**, the thread must decide if that exception which restarted the thread was actually intended of itself. If it was, then **mt_accept_exception()** will remove the exception from the exception queue, and that particular exception occurrence will not be sent to any other interested handlers.

procedure mt_forward_exception

If a handler is restarted with an exception which was not intended for that particular thread, then the exception can be returned to the exception queue using the **mt_forward_exception()** primitive. The exception is then passed to the next highest priority handler. Once a particular exception instance is forwarded by a handler, it will never return (even if the handler reduces its own priority in an attempt to catch the exception up). After every exception handler registered to receive that exception has received and then forwarded the exception, then that exception instance is disposed of.

```
procedure mt_exception(
    EXCEPTION : exceptions
)
```

Exceptions may be instigated at any time using the **mt_exception** primitive. This causes all the exceptions ANDed into *exceptions* to be treated as exception instants. These instants are then queued for processing by all interested handlers.

procedure mt_critical

In rare circumstances, the protection offered by the monitor constructs may be insufficient. It is possible to physically stop all threads from being scheduled by issuing the **mt_critical** primitive. This

has the effect of switching off the scheduler. *The use of this call should be avoided where ever possible*.

procedure mt_endcritical

At the end of a critical region (as started by **mt_critical()**), the scheduler can be restarted using the **mt_endcritical()** primitive. The use of critical code protection using **mt_critical** is *not recommended*, and instead the programmer should make use of the monitor constructs.

Appendix D

MultiPackets Programming Reference

This appendix list the primitives available to the programmer within the MultiPackets library. This library is completely integrated with MultiThreads, and as such does not cause process blocking under Unix during either a send or receive action. Any routine used which blocks until completion will block only the thread initiating that routine.

procedure mp_arm()

This routine initializes the structures used internally by the MultiPackets library. It also creates the threads necessary to the current handling of MultiPackets. Three threads are created; the first thread handles incoming packets, the second handles retransmissions of packet data, while the last performs internal memory management in the package.

```
function mp_multicast(
    ADDRESS_CHAIN       : *where
    char                : *data
    int                 : length
    TX_OPTIONS          : options
) return MP_ID
```

Data to be send to more than one destination should be sent using a single call to **mp_multicast**. This routine takes a linked list *ADDRESS_CHAIN*, containing *node* (a destination address), *thread* (the destination thread at that address), and *next* (the pointer to the next element in the list). Note that it is also possible to send a message to the server on a particular node, instead of a specified thread, by using the thread id *MP_SERVER* (see **mp_server** below). The data pointed to by *data* of length *length* is then transferred to the nodes specified in the *where* linked list. The call itself is non-blocking, and

returns as soon as the packets are queued for delivery. Delivery is guaranteed. *TX_OPTIONS* can be either *TX_NORMAL* or *TX_DEALLOCATE*. Using the latter option, the data pointed to by *data* is freed once the data has been successfully sent to all parties, while under the former the data is preserved and must be handled by the programmer. The routine returns a message id for the request, which can be used in checking the transmission's progress.

```
function mp_send(
    ELEMENT_ID          : thread
    MACHINE_ADDRESS     : dest
    char                : *data
    int                 : length
    TX_OPTIONS          : options
) return MP_ID
```

mp_send functions in an identical way to **mp_multicast**, but instead of specifing a linked list of destination nodes for the data, a single node specified by *dest* and *thread* is directly specified in the parameter field. There are no other differences between the two routines.

```
function mp_sync(
    MP_ID               : id
    MACHINE_ADDRESS     : dest
    struct timeval      : *timeout
) return INTEGER
```

The **mp_sync** primitive is used to check on the progress of a previous transmission. Using this primitive with the *MP_ID* and *dest* of a previous transmission causes the current thread to block either until the relevent node has confirmed its reception of the packet, or (in the case of *timeval* not being a NULL pointer) until the specified timeout is reached (whichever occurs first). If the thread is restarted due to a timeout, the routine returns 0, otherwise it returns 1. If the specified message has already been delivered successfully before the routine was called, then no block is entered and the routine returns 1 immediately.

```
function mp_receive(
    char                : **data
    int                 : *length
    MP_ID               : *id
    MACHINE_ADDRESS     : *source_node
    ELEMENT_ID          : *source_thread
    struct timeval      : *timeout
) return INTEGER
```

Using the **mp_receive** primitive, any thread may wait for messages directed specifically at that thread by another party via **mp_send** or **mp_multicast**. The calling thread is blocked until either a message is received or, in the case of *timeout* being non-NULL, the specified timeout being reached. Once again, termination of the routine by timeout returns 0, while all other cases returns 1.

Once the data has been received, the information is returned to the calling thread via the parameter fields. *data* contains the information, using space allocated within the library itself. This space can be used indefinitly, or freed by the programmer via **deallocate_space**. The length of the information is contained in *length*, with the message's *MP_ID*, sender's network address, and the senders thread id stored in *id*, *source_node*, and *source_thread* respectively.

Messages send to a thread which is not currently waiting for messages are queued until required (or until the death of the thread). All messages received for a thread are maintained until read or the death of the thread occurs.

```
function mp_server(
    char                : **data
    int                 : *length
    MP_ID               : *id
    MACHINE_ADDRESS     : *source_node
    ELEMENT_ID          : *source_thread
    struct timeval      : *timeout
) return INTEGER
```

As was indicated in **mp_multisend**, it is also possible to send a message to a node without knowing a particular destination thread, by specifying the thread id as *MP_SERVER*. Such messages can be read by any thread via the **mp_server** primitive. This primitive functions exactly as **mp_receive**, except that messages specifying the current thread's id as a destination are delayed (until the use of **mp_receive**), and instead only those messages addressed to this node with thread id *MP_SERVER* are detectable. All other functionality is identical to **mp_receive**.

Bibliography

[Abramson, 1981] D. Abramson. Hardware management of a large virtual memory. In *Proceedings of the 4th Australian Computer Science Conference*, pages 1–13, Brisbane, 1981. Cited on page 17.

[Adv, 1987] Advanced Micro Devices, 901 Thompson Place, PO Box 3453, Sunnyvale, California 94088. *Am29000 User's Manual*, 1987. Cited on page 77.

[Almes, 1980] G. T. Almes. *Garbage Collection in an Object-Oriented System*. PhD thesis, Carnegie-Mellon University, June 1980. Cited on page 32.

[Appel *et al.*, 1988] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20. ACM Press, June 1988. Cited on page 208.

[Atkinson *et al.*, 1983] M. P. Atkinson, P. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computing Journal*, 26(4):360–365, 1983. Cited on page 5.

[Baker, 1978] Henry G. Baker, Jr. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978. Cited on page 209.

[Balch *et al.*, 1989] P. Balch, W. P. Cockshott, and P. W. Foulk. Layered implementations of persistent object store. *Software Engineering Journal*, pages 123–131, March 1989. Cited on page 9.

[Ball *et al.*, 1992] H. E. Ball, M. F. Kaasgoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions*, 18(3):190–205, March 1992. Cited on page 166.

[Bayliss *et al.*, 1983] John A. Bayliss, John A. Deetz, Chun-Kit Ng, Scott A. Ogilvie, Craig B. Peterson, and Doran K Wilde. The interface processor of the intel VLSI 432 32-bit computer. In Gupta and Toong [1983a], pages 175–182. Cited on page 20.

[Berstis *et al.*, 1982] B. Berstis, C. D. Truxal, and J. G. Ranweiler. The IBM SYSTEM/38: Addressing and authorization. In Seiwiorek et al. [1982], pages 540–544. Reprinted from IBM SYSTEM/38 Technical Developments, pp. 55–58, 1978. Cited on page 17.

[Blaauw and Brooks, 1964] G. A. Blaauw and F. P. Brooks. The structure of the SYSTEM/360. *IBM Systems Journal*, 3(2):119–135, 1964. Cited on page 13.

[Budde *et al.*, 1983] David L. Budde, Steven R. Colley, Stephen L. Domenik, Allan L. Goodman, James D. Howard, and Michael T. Imel. The execution unit for the VLSI 432 general data unit. In Gupta and Toong [1983a], pages 175–182. Cited on page 20.

[Chambers *et al.*, 1989] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically- types object-oriented language based on prototypes. In Norman Meyrowitz, editor, *OOPSLA'89*, pages 49–70. ACM Press, October 1989. Cited on page 8.

[Chang and Mergen, 1988] A. Chang and M. F. Mergen. 801 storage architecture and programming. *ACM Transactions on Computer Systems*, 6(1), February 1988. Cited on page 17.

[Cockshott *et al.*, 1993a] W. P. Cockshott, A. J. Cowie, G. W. Russell, and D. McGregor. Memory resident databases: Reliability, compression and performance. Research Report ARCH-11-93, University Of Strathclyde, Department of Computer Science, 26 Richmond Street, Glasgow. G1 1XH. Scotland, UK, July 1993. Cited on pages 4 and 225.

[Cockshott *et al.*, 1993b] W. P. Cockshott, A. J. Cowie, G. W. Russell, and D. McGregor. New persistent memory technologies and relational databases. Submitted to VLDB Journal. This article is also available as a technical report, [Cockshott *et al.*, 1993a]., 1993. Cited on page 100.

[Cockshott, 1990] W. P. Cockshott. Design of POMP - a persistent object management system. In *Persistent Object Systems*. Springer Verlag, 1990. Cited on page 69.

[Cockshott, 1992] Paul Cockshott. Performance evaluation of the Rekursiv object oriented computer. In Veljko Milutinovic and Bruce D. Shriver, editors, *Proceedings of the Hawaii International Conference on System Sciences*, pages 730–736. IEEE Computer Society Press, January 1992. Cited on page 39.

[Cohen, 1981] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3), September 1981. Cited on page 206.

[Colwell *et al.*, 1988] Robert P. Colwell, Edward F. Gehringer, and E. Duglas Jensen. Performance effects of the architectural complexity in the intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, August 1988. Cited on page 22.

[Corradi and Natali, 1983] Antonio Corradi and Antonio Natali. Using the iAPX-432 system as a support for chill parallel constructs. *Microprocessing and Microprogramming*, 12((3,4)):159–166, October/November 1983. Cited on page 19.

[Cox *et al.*, 1983] George W. Cox, William M. Corwin, Konrad K. Lai, and Fred J. Pollack. Interprocess communication and processor dispatching on the intel 432. *ACM Transactions on Computer Systems*, 1(1):45–66, February 1983. Cited on page 24.

[Dahlby *et al.*, 1982] S. H. Dahlby, G. G. Henry, D. N. Reynolds, and P. T. Taylor. The IBM SYSTEM/38: A high-level machine. In Seiwiorek et al. [1982], pages 533–536. Reprinted from IBM SYSTEM/38 Technical Developments, pp. 47–50, 1978. Cited on page 15.

[Dasgupta *et al.*, 1991] Partha Dasgupta, Jr. Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *IEEE Micro*, 24(11):34–44, November 1991. Cited on page 137.

[Deutsch and Bobrow, 1976] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976. Cited on page 208.

[Edwards *et al.*, 1980] D. B. G. Edwards, A. E. Knowles, and J. V. Woods. The MU6-G. a new design to achieve mainframe performance from a mini sized computer. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 161–167, 1980. Cited on page 17.

[Fabry, 1974] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974. Cited on page 27.

[Farby, 1974] R. S. Farby. Capability-based addressing. *Communications of the ACM*, 7(17), July 1974. Cited on page 15.

[Glass, 1991] Brett Glass. SPARC revealed. *Byte*, 16(4):295–302, April 1991. Cited on page 76.

[Gupta and Toong, 1983a]
Amar Gupta and Hoo-min D. Toong, editors. *Advanced Microprocessors*. IEEE Press, 1983. Cited on page 224.

[Gupta and Toong, 1983b] Amar Gupta and Hoo-min D. Toong. An architectural comparison of 32-bit microprocessors. In *Advanced Microprocessors* [1983a], pages 155–174. Cited on page 20.

[Harland, 1988] David M. Harland. *REKURSIV, Object Oriented Computer Architecture*. Ellis Horwood Limited, 1988. Cited on pages 34 and 58.

[Hennessy and Patterson, 1990] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1990. Cited on pages 18, 75–76, 93, and 95.

[Henskens *et al.*, 1990] Frans A. Henskens, John Rosenberg, and Michael R. Hannaford. Stability in a network of MONADS-PC computers. In *Computer Architecture to Support Security and Persistence of Information*, pages (16–1)–(16–10). University of Bremen, May 1990. Cited on page 30.

[Hoffman and Soltis, 1982] R. L. Hoffman and F. G. Soltis. The IBM SYSTEM/38: Hardware organization of the SYSTEM/38. In Seiwiorek et al. [1982], pages 544–546. Reprinted from IBM SYSTEM/38 Technical Developments, pp. 19–21, 1978. Cited on page 18.

[Houdek *et al.*, 1981] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM SYSTEM/38 support for capability-based addressing. In *The Eighth Symposium on Computer Architecture*, pages 341–48, May 1981. Published in SIGARCH Newsletter, Vol 9, No 3. Cited on page 15.

[Hunter *et al.*, 1985] Colin Hunter, Erin Farquhar, and James Ready. *Introduction to the Intel iAPX432 Architecture*. Reston Publishing Company inc., 1985. Cited on pages 19 and 59.

[Intel, 1981a] Intel. *Engineering Specification for the iAPX432 Extentions to Ada*, 1981. Cited on page 19.

[Intel, 1981b] Intel. *iAPX 43201 iAPX 43202 VLSI General Data Processor*, 1981. Revision A. Cited on page 20.

[Intel, ND] Intel. Introducing FLASHFILE™ memory from Intel: Enabling a new generation of mobile computers and dedicated systems. Memory Components Division, Folsom, California. Cited on page 103.

[Jones *et al.*, 1979] Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsen Schwans, and Steven R. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *The Proceedings of the 7th Symposium on Operating System Principles*, pages 117–127. ACM, December 1979. Cited on page 24.

[Kahn *et al.*, 1981] Kevin C. Kahn, William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, and Frew J. Pollack. iMAX: A multiprocessor operating system for an object-based computer. In *Proceedings of the Eighth Symposium on Operating System Principals*, volume 15, pages 127–136. SIGOPS, December 1981. Cited on page 19.

[Kaiser and Czaja, 1990] Jörg Kaiser and Karol Czaja. An architecture to support persistence in object-oriented systems. Available from the authors at kaiser@gmdzi.gmd.de or czaja@gmdzi.gbx.de, 1990. Cited on pages 31–32, 58, and 68.

[Kaiser, 1988] Jörg Kaiser. MUTABOR, a coprocessor supporting memory management in an object-oriented architecture. *IEEE Micro*, 8(5), October 1988. Cited on page 31.

[Kaiser, 1990] Jörg Kaiser. An object-oriented architecture to support system reliability and security. In *Computer Architecture to Support Security and Persistence of Information*, pages (9–1)–(9–15). University of Bremen, May 1990. Cited on pages 31–32 and 58.

[Kane, 1987] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice-Hall, 1987. Cited on page 95.

[Keedy and Rosenberg, 1989] J. Leslie Keedy and John Rosenberg. Uniform support for collections of objects in a persistent environment. In Bruce D. Shriver, editor, *Proceedings of the Twenty Second Annual Hawaii International Conference on System Sciences*, volume 2, pages 26–35. IEEE Computer Society, 1989. Cited on page 25.

[Keedy, 1980] J. L. Keedy. A comparison of two process structuring models. Technical report, Monash University, 1980. MONADS Report Number 4. Cited on page 25.

[Keedy, 1984] J. Leslie Keedy. An implementation of

capabilities without a central mapping table. In *Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences*, pages 180–185, 1984. Cited on pages 25, 28–29, and 59.

[Kilburn *et al.*, 1956] T. Kilburn, D. B. G. Edwards, and G. E. Thomas. A transistor digital computer with a magnetic drum store. *Proceedings of the IEE*, 103B(sup 1-3):390–406, 1956. Cited on page 13.

[Kilburn *et al.*, 1962] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level storage system. *IRE Transactions*, 2(EC-11):223–235, April 1962. Cited on page 13.

[Kilburn, 1989] Tom Kilburn. The manchester university digital computing machine. In M. R. Williams and Martin Campbell-Kelly, editors, *Charles Babbage Institute Reprint Series for the History of Computing*, volume 14, pages 138–141. MIT Press, 1989. Cited on page 75.

[Koch and Rosenberg, 1990] David Koch and John Rosenberg. A secure RISC-based architecture supporting data persistence. In *Computer Architecture to Support Security and Persistence of Information*, pages (10–1)–(10–14). University of Bremen, May 1990. Cited on page 25.

[Lauer and Needham, 1979] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *ACM Operating Systems Review*, 13(2):3–19, 1979. Cited on page 25.

[Leg, Spring 1991] Legacy Storage Systems Inc, 200 Butterfield Drive, Unit B, Ashland, MA 01721. *Report on Fault Tolerance*, Spring 1991. Cited on page 105.

[Li and Hudak, 1989] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989. Cited on pages 4 and 162.

[Lieberman and Hewitt, 1983] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983. Cited on pages 210 and 212.

[Liskov, 1985] B. Liskov. *The Argus Language and System*, volume 190. Springer-Verlag, New York, 1985. Lecture Notes in Computer Science. Cited on page 5.

[Liu, 1991] Yu-Cheng Liu. *The M68000 Microprocessor Family*. Prentice-Hall International, 1991. Cited on page 31.

[Malhotra and Munroe, 1992] Ashok Malhotra and Steven J. Munroe. Support for persistent objects: Two architectures. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE Press, 1992. Cited on page 17.

[Morrison and Atkinson, 1990] R. Morrison and M. P. Atkinson. Persistent languages and architectures. In J. Rosenberg and J. L. Keedy, editors, *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 9–38. Springer-Verlag and the British Computer Society, 1990. Cited on page 5.

[Motorola, 1985] Motorola. *MC68020 32-Bit Microprocessor User's Manual*. Prentice-Hall, second edition, 1985. Cited on page 104.

[Mueller, 1993] Frank Mueller. A library implementation of posix threads under unix. In *1993 Winter USENIX Conference*, pages 29–42, January 1993. This and other papers on Cthreads is available by anonymous ftp (including the library itself) from ftp.cs.fsu.edu (128.186.121.27), in the directory /pub/PART. Cited on pages 133 and 135.

[Ogden, 1994] Frank Ogden. A nasty case of slipped disks. *Electronics World and Wireless World*, 100(1694):1, January 1994. Cited on page 116.

[Organick, 1972] E. I. Organick. *The MULTICS System: An Examination of its Structure*. MIT Press, 1972. Cited on page 13.

[Ousterhout *et al.*, 1980] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980. Cited on page 24.

[Patterson and Séquin, 1982] David A. Patterson and Carlo H. Séquin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, 1982. Cited on page 76.

[Pinnow *et al.*, 1982] K. W. Pinnow, J. G. Ranweiler, and J. F. Miller. The IBM SYSTEM/38: Object-oriented architecture. In Seiwiorek et al. [1982], pages 537–540. Reprinted from IBM SYSTEM/38 Technical Developments, pp. 51–54, 1978. Cited on page 15.

[Pollack *et al.*, 1981] Fred J. Pollack, Kevin C. Kahn, and Roy M. Wilkinson. The iMAX-432 object filing system. In *Proceedings of the Eighth Symposium on Operating System Principals*, pages 137–147. SIGOPS, December 1981. Published in SIGOPS, Vol 15, No 5. Cited on pages 21, 31, and 68.

[Pollack *et al.*, 1982] Fred J. Pollack, George W. Cox, Dan W. Hammerstrom, Kevin C. Kahn, Konrad K. Lai, and Justin R. Rattner. Supporting ADA memory management in the iAPX-432. In *The Proceeding of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–131. ACM, 1982. Cited on page 22.

[Richardson and Carey, 1989] J. E. Richardson and M. J. Carey. Implementing persistence in E. In *Proceedings of the Third International Workshop on Persistent Object Systems*, pages 175–199. Springer-Verlag, 1989. Cited on page 5.

[Richardson and Ganapathi, 1989] Stephen Richardson and Mahadevan Ganapathi. Code optimization across procedures. *Computer*, 22(2), February 1989. Cited on page 75.

[Rosenberg and Abramson, 1985] John Rosenberg and David Abramson. MONADS-PC - a capability-based workstation to support software engineering. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pages 222–231, 1985. Cited on pages 25 and 59.

[Rosenberg *et al.*, 1990] John Rosenberg, James Leslie Keedy, and David Abramson. Addressing mechanisms for large virtual memories. Technical report, St Andrews University, 1990. CS/90/2. To appear in the Computer Journal, August 1992. Cited on page 30.

[Rosenberg, 1989] J. Rosenberg. Pascal/M - a pascal extention supporting orthogonal persistence. Technical Report 89/1, Department of Electrical Engineering and Computer Science, the University of Newcastle, 1989. Cited on pages 30 and 40.

[Rosenberg, 1991] J. Rosenberg. Architectural support for persistent object systems. In *Proceedings of the International Workshop on Object-Orientation in Operation Systems*, Los Alamitos, CA, 1991. IEEE Computer Society Press. Cited on page 5.

[Russell and Cockshott, 1993a] Gordon Russell and Paul Cockshott. Architectures for persistence. *Microprocessors and Microsystems*, 17(3):117–130, April 1993. Cited on page 4.

[Russell and Cockshott, 1993b] Gordon Russell and Paul Cockshott. A survey of architectures for memory resident databases. Research Report ARCH-10-93, University Of Strathclyde, Department of Computer Science, 26 Richmond Street, Glasgow. G1 1XH. Scotland, UK, April 1993. Cited on page 4.

[Russell and Shaw, 1993a] Gordon Russell and Paul Shaw. An object-based processor cache. Research Report ARCH-12-93, University Of Strathclyde, Department of Computer Science, 26 Richmond Street, Glasgow. G1 1XH. Scotland, UK, August 1993. Cited on page 4.

[Russell and Shaw, 1993b] Gordon Russell and Paul Shaw. Shifting register windows. *IEEE Micro*, 13(4):28–35, August 1993. Cited on pages 4, 78, and 80.

[Russell and Shaw, 1993c] Gordon Russell and Paul Shaw. A stack-based register set. Research Report ARCH-9-93, University Of Strathclyde, Department of Computer Science, 26 Richmond Street, Glasgow. G1 1XH. Scotland, UK, April 1993. Cited on pages 4 and 78.

[Russell *et al.*, 1995] Gordon Russell, Paul Shaw, and Paul Cockshott. Dais: An object-addressed processor cache. Accepted for the Sixth International

Workshop on Persistent Object Systems. Proceedings to be published by Springer-Verlag, 1995. Cited on page 4.

[Russell, 1993] Gordon Russell. Lightweight threads for portable software. In *Proceedings of the UKUUG/SUKUG*, January 1993. Cited on pages 4 and 133.

[Schmidt, 1977] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, 1977. Cited on page 5.

[Seiwiorek *et al.*, 1982] Daniel P. Seiwiorek, C. Gordon Bell, and Allen Newell, editors. *Computer Structures: Principles and Examples*. McGraw-Hill International, 1982. Cited on pages 75 and 224.

[Seq, 1987] Sequent. *Guide to Parallel Programming*, 1987. Cited on page 123.

[Shaw, 1988] R. A. Shaw. *Empirical Analysis of a LISP System*. PhD thesis, Stanford University, February 1988. Cited on page 212.

[Soltis and Hoffman, 1987] Frank G. Soltis and Roy L. Hoffman. Design considerations for the IBM SYSTEM/38. In Gerald E. Peterson, editor, *Tutorial: Object-Oriented Computing*, volume 2. IEEE Computer Society, 1987. Cited on pages 15 and 59.

[Soltis, 1977] G. Soltis. Design of a small business data processing system. *IEEE Computer*, pages 77–93, September 1977. Cited on pages 15, 17, and 59.

[Sun, 1990] Sun Microsystems. *Programming Utilities and Libraries*, March 1990. Pages 17–51. Cited on page 123.

[SUN, 1992] SUN Microsystems. *Introduction to SHADOW*, April 1992. Cited on pages 66, 83, and 97.

[Tanenbaum *et al.*, 1990] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990. Cited on page 166.

[Tanenbaum *et al.*, 1992] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):11–19, August 1992. Cited on pages 137 and 166.

[Tevanian and Smith, 1989] Avadis Tevanian, Jr. and Ben Smith. MACH: the model for future UNIX. *Byte*, pages 411–415, November 1989. Cited on page 2.

[Ungar and Jackson, 1988] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA'88 Proceedings*, pages 1–17, September 1988. Published in SIGPLAN Notices, volume 23, number 11, November 1988. Cited on page 211.

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclemetion algorithm. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, May 1984. Published in Software Engineering Notes, volume 9, number 3, and in SIGPLAN Notices, volume 19, number 5. Cited on pages 207–208 and 212.

[USDoD, 1983] United States Department of Defence. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1915 A-1983. Cited on page 19.

[van Rumste, 1983] M. van Rumste. The iAPX432, a next generation microprocessor. *Microprocessing and Microprogramming*, 12(2):69–106, February 1983. Cited on pages 18 and 59.

[Wall, 1987] David W. Wall. Register windows vs. register allocation. Technical report, Digital Research, December 1987. Also published in *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988. Cited on page 75.

[Wallace and Pose, 1990] C. S. Wallace and R. D. Pose. Charging in a secure environment. In *Security and Persistence*. Springer Verlag, 1990. Cited on

page 69.

[Wentworth, 1990] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software, Practice and Experience*, 20(7):719–727, July 1990. Cited on page 206.

[Whitfield and Wight, 1973] H. Whitfield and A. S. Wight. EMAS - the Edinburgh multi-access system. *Computer Journal*, 16(4), 1973. Cited on page 13.

[Wilson and Moher, 1989] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In Norman Meyrowitz, editor, *OOPSLA'89*, pages 23–35. ACM Press, October 1989. Cited on page 211.

[Wilson, 1991] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. *ACM SIGPLAN Notices*, 26(3):45–52, March 1991. Cited on page 211.

[Wulf *et al.*, 1981] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental System*. McGraw-Hill, 1981. Cited on pages 31 and 68.

[Zeigler *et al.*, 1981] Stephen Zeigler, Nicole Allegra, David Coar, Robert Johnson, James Morris, and Greg Burns. The intel 432 ada programming environment. In Gerald E. Peterson, editor, *Tutorial: Object-Oriented Computing*, pages 286–291. IEEE Computer Society, 1981. Cited on page 19.