Cherub: A Hardware Distributed Single Shared Address Space Memory Architecture

Aarron Gull
Systems Architecture Research Centre
City University

March 1993

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

ABSTRACT

Increased computer throughput can be achieved through the use of parallel processing. The granularity of a parallel program is the average number of instructions performed by the tasks constituting it. Coarse-grained programs typically execute huge numbers of instructions per task ($\approx 10^5$). The tasks in fine-grained programs are typically short ($\approx 10^3$). In general, the finer the program grain, the greater the potential for exploiting parallelism. Amdahl's Law shows that *in the absence of overheads*, the more *potential* parallelism that is realised in an algorithm, the faster it will be. The economical granularity of tasks is determined by the intertask communications overhead. Break-even occurs when processing is approximately equally divided between useful work and overhead.

The two common parallel programming paradigms are shared variable and message passing. Shared variable is, in general, the more natural of the two as it allows implicit communication between tasks. This encourages the programmer to make use of fine-grained tasks. The message passing paradigm requires explicit communication between tasks. This encourages the programmer to use coarser-grained tasks.

Two kinds of parallel architecture have become established. The first is the *multiprocessor*, which is built around a shared bus giving broadcast communications and a shared memory. This is characterised by low communications overhead, but limited scalability. The second is the *multicomputer*, which is based on point-to-point communications with larger communications overhead, but good scalability. Quantitatively, the low overhead of the multiprocessor is well matched to fine-grain tasks and, hence, to supporting the shared variable paradigm, while the high overhead of the multicomputer matches it to coarse-grain parallelism and, hence, to the message passing paradigm.

Currently, there appears to be no middle ground in parallel computing; an architecture which can support both several hundred medium-grained ($\approx 10^4$ instructions) parallel tasks and the shared variable programming paradigm would be advantageous in many applications.

This thesis asserts that it is possible to implement a new computer architecture, *Cherub*, which has at least 200 processors and is able to support shared variable programming with an optimal task granularity of around 10^4 instructions. This can be achieved through the combination of a hardware-based distributed shared single address space and a wafer-scale communications network.

To support the thesis, the dissertation first specifies a programmer's interface to *Cherub* which is simple enough to implement in hardware. It then designs algorithms which provide this interface, allowing the requirements of the underlying network to be estimated. Finally, a wafer scale communications network is outlined, and simulations are used to demonstrate that it can provide the performance required to successfully implement *Cherub*.

Contents

List of Figures

Acknowledgements

I would like to extend my thanks to the following people:

- **Peter Osmon**, my supervisor, for his help and encouragement.
- **Tim Wilkinson** for his ideas, constructive criticism and LaTeXmacros.
- **Mum** and **Dad**, who have subsidised me for most of my academic career.

Dedication

This is dedicated to those who care.

Declaration

Chapter 1

Introduction

1.1 An Informal Perspective

The existence of a radically new systems architecture often has landslide effects throughout the computer industry. For example, Atlas's demand paging [Kil61, Kil62], IVY's distributed shared memory [Li 86, Li 89] and Chorus's microkernel [Gui82, Arm86, Her88] have all had profound effects over subsequent computer design.

This dissertation proposes a new architecture, *Cherub*, which uses hardware to create a high performance distributed single shared address space and allows the use of a scalable medium–grained shared variable programming paradigm. It is hoped that this may lead to a significant advance in parallel computing.

1.1.1 Parallel Programming

Increased computer throughput can be achieved through the use of parallel processing. Two main issues are of importance in parallel systems:

- Granularity of Processing and Intertask Communication Overhead
- Programmability

Granularity of Processing and Intertask Communication Overhead

The granularity of a parallel program is the average number of instructions performed by the tasks constituting it. Coarse–grained programs typically execute huge numbers of instructions per task ($\approx 10^5$). The tasks in fine–grained programs are typically short ($\approx 10^3$). In general, the finer the program grain, the greater the potential for exploiting parallelism. Amdahl's Law shows that *in the absence of overheads*, the more *potential* parallelism that is realised in an algorithm, the faster it will be. The economical granularity of tasks is determined by the intertask communication overhead. Break–even occurs when processing is approximately equally divided between useful work and overhead.

Typical intertask communication overheads include setup, data sharing, synchronisation, result transfer and termination, as illustrated in figure 1.1.

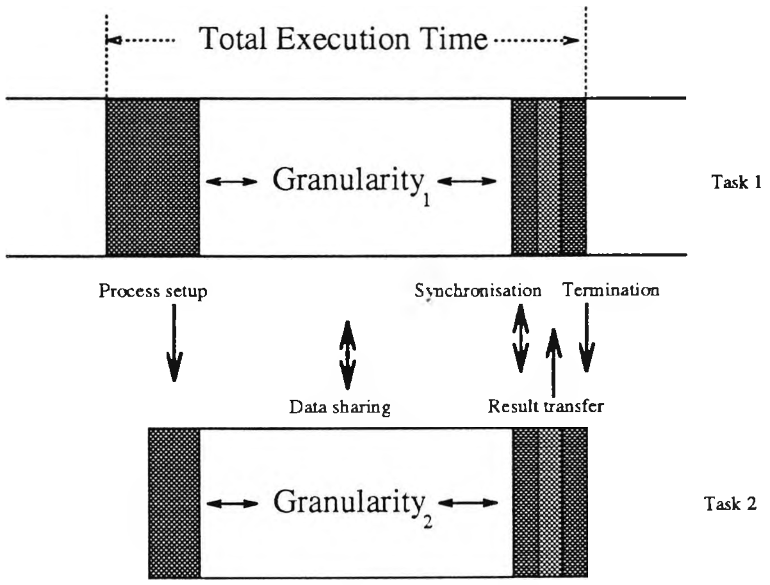


Figure 1.1: Typical Task Communication Overheads

Ideally, a sequential program of execution time ℓ can be decomposed into N tasks, all of which can all be performed in parallel. The time taken to execute the program will be:

$$\frac{\ell}{N}$$

If the decomposition is performed in a binary tree fashion, such that each task creates another, then the decomposition will take a logarithmic factor of the intertask communication overhead, $t_{overhead}$. The time taken to decompose and execute the program will, therefore, be:

$$t_{overhead} \cdot \log_2 N \quad + \frac{\ell}{N}$$

Differentiating with respect to N allows us to determine the optimum number of processors for a given value of ℓ:

$$\frac{dt}{dN} = \frac{t_{overhead}}{N \cdot \ln 2} - \frac{\ell}{N^2} \quad (\rightarrow 0 \;\; as \;\; N \rightarrow \infty)$$

The optimum number of processors, N_o, is at this curve's minimum:

$$N_o = \frac{\ell}{t_{overhead}} \cdot \ln 2$$

This is illustrated in figure 1.2, which shows the time taken to execute programs of various lengths which have been perfectly decomposed into parallel tasks, for a given intertask communication latency. It should be noted that it is not cost effective to parallelise a program whose sequential execution time is similar to the intertask communication latency ($t_{overhead}$). In addition small programs ($4 \cdot t_{overhead}$) can only be efficiently parallelised for a couple of processors and even quite large programs ($64 \cdot t_{overhead}$) don't show significant performance improvement when run on more then 15 processors.

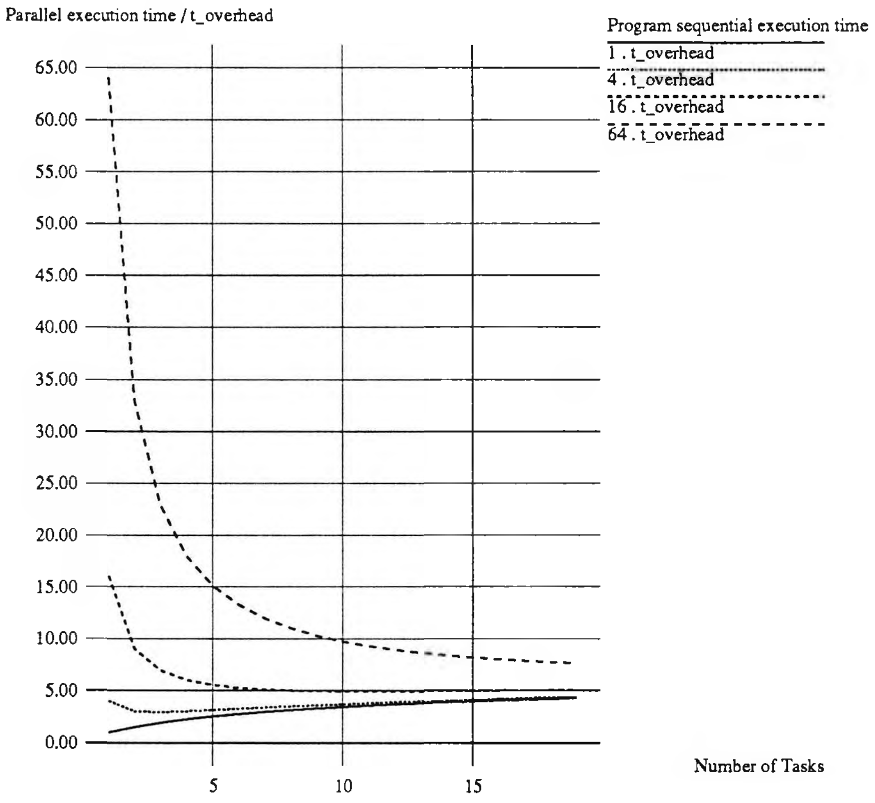


Figure 1.2: The Time Taken to Execute A Program Using Multiple Parallel Tasks

Given that an optimal number of processors exists, then the optimal grain size of a task, G_o, can also be calculated:

$$G_o = \frac{\ell}{N_o} = \frac{\ell}{\frac{\ell}{t_{overhead}} \cdot \ln 2} = \frac{t_{overhead}}{\ln 2}$$

Therefore, if a medium optimal grain size of, say, 10^4 instructions is required, then rearranging this gives:

$$\begin{aligned} t_{overhead} &= 10,000I \times \ln 2 \\ &\approx 7,000I \\ &\approx G_o \end{aligned}$$

Clearly, for a fine level of granularity to be cost effective, the task overheads must be similarly low. These are determined by the underlying architecture.

Programmability

A further tradeoff exists between the performance and the programmability of a parallel system; a programmer will only use parallelism when it can be realised easily using the programming paradigm.

The two most common parallel programming paradigms are shared variable and message passing:

- In the shared variable paradigm tasks communicate and synchronise implicitly through the use of common data structures.
- In the message passing paradigm tasks communicate explicitly through primitives for the protected sending and receiving of messages. The sequential arrival of messages inherently synchronises the actions of the tasks.

The shared variable paradigm is, in general, the more natural of the two to program as its simplified communication mechanism does not force the explicit partitioning of data between tasks. Furthermore, it allows parameters to be passed by reference rather than by value. This ease of use has two implications:

- The near transparent overheads of communication encourages the use of finer–grained tasks and, hence, more parallelism.
- It takes relatively little programming effort to coordinate a large number of parallel tasks.

Programmers, therefore, generally prefer the shared variable programming paradigm.

For reasons of simplicity, the shared variable programming paradigm is typically implemented using a shared memory computer architecture; shared data structures are simply held in shared memory, allowing the underlying architecture to perform the communication transparently. Two shared memory architectures are common:

- Tightly Coupled Multiprocessors with Physically Shared Memory
- Loosely Coupled Multicomputers with Distributed Shared Memory

These are described in the following two sections.

1.1.2 Tightly Coupled Multiprocessors With Physically Shared Memory

Multiprocessor architectures, also known in the literature as parallel random–access machines (PRAMs), are the most conventional way of implementing a shared variable parallel programming paradigm. These machines typically have one or more shared broadcast busses by which the processors are connected to a single globally accessible memory. A typical multiprocessor system is illustrated in figure 1.3.

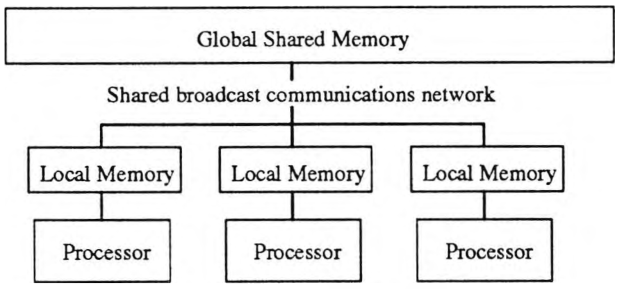


Figure 1.3: A Typical Multiprocessor

Multiprocessor systems are said to be tightly coupled because the processors must coordinate access to the shared memory. To reduce shared memory accesses, the processors often have local memories in which they cache data. The Sequent Symmetry [seq87], for example, has up to 30 Intel 80386 processors, each with a 64 Kbyte two–way set associative cache. Memory coherence is maintained invisibly to the overlying software through snooping hardware. This mechanism depends upon the ability to broadcast information efficiently to all of the processors in the system.

The main advantage of multiprocessor systems to the programmer is that the data sharing is implicit and its effect on performance is generally small because of the broadcast and snooping properties of the hardware. They, therefore, encourage the use of fine–grained tasks, typically of around 10^3 instructions. Their major problem, however, is that contention for shared bus and memory bandwidth results in a performance bottleneck.

This bottleneck may be reduced by:

- Using more busses to reduce contention. Physical size, however, severely limits the number of busses that can be connected to a single processor.
- Using wider busses to transfer larger words and so reduce the number of times the processors need to access the shared memory. Eventually, however, physical size limits bus width.
- Using faster busses and memory to reduce contention. Ultimately, the bus speed is limited by that of the memory because only one processor may access the memory through the bus at a time.

Although these techniques can be very effective, most high–performance multiprocessor systems are limited to around 30 processors[1]. *Scaling up shared memory architectures is, therefore, an important problem in computer architecture and is attracting considerable attention [Bel92].*

One interesting attempt to produce a scalable multiprocessor is the BBN Butterfly [bbn86]. The machine has 256 processors, each with a local memory. Each processor has the ability to directly access the memory of other processors via a non–broadcast butterfly network.

[1]The Elxsi 6400 is limited to 12 processors, the Encore Multimax to 20, the Flex/32 to 20, the Sequent Symmetry to 30 and the IP–1 to 33 [Don91].

This creates a scalable shared global memory, but strongly encourages the programmer to make use of local memory as remote memory accesses are five to ten times slower. This results in relatively poor peak performance and has led to the machine's commercial failure.

1.1.3 Loosely Coupled Multicomputers With Distributed Shared Memory

Most attempts to break out of the multiprocessor bus and memory saturation bottleneck have used multicomputer platforms. These comprise large numbers of processors linked by a point-to-point network and communicating with each other by message passing. This type of design is described as loosely coupled, meaning that each processor is almost entirely self sufficient. A typical multicomputer system is illustrated in figure 1.4.

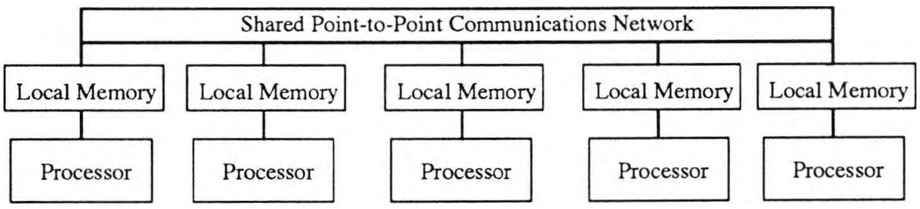


Figure 1.4: A Typical Multicomputer

Loosely coupled multicomputers have two main advantages over tightly coupled multiprocessors:

- Higher Scalability

  A bus-based system provides high-performance when relatively small numbers of processors are involved. Such machines are not scalable however. Multicomputers use communication networks with overall bandwidths proportional to the number of processors in the system, albeit at the cost of reduced connectivity. This provides scalability over many hundreds of processors[2].

  Winterbottom's Topsy [Win89] is a typical multicomputer. This employs high performance communications network and efficient caching to create a scalable system over 256 processors.

- Lower Cost

  Multicomputers are always cheaper than shared memory multiprocessors for the same amount of raw, not necessarily realisable, processing power. This is because the shared buses of multiprocessor systems are more complicated and expensive than the networks of multicomputers, especially when large numbers of processors are involved.

[2]The Intel iPSC/2 is limited to 128 processors, the CYBERPLUS to 256, the NCUBE to 1,024 and the J-Machine to 64K [Don91].

In order to use a shared variable programming paradigm on multicomputers a mechanism called distributed shared memory (DSM) [Li 86] has been devised. This uses a distributed memory management system to copy pages of data between the physical memories of the individual processors on demand. The selectivity of the mechanism minimises overloading of the network by ensuring that only those processors which need an updated data page get it.

DSM potentially offers a scalable shared memory architecture. Unfortunately, as the message handling latency in multicomputers is relatively high, these machines can only support coarse-grain ($\approx 10^5$ instructions) parallelism efficiently.

A notable alternative mechanism to DSM for the implementation of shared variable programming is employed in the Linda paradigm [Zen88, Gel89, Bor88, Car86b, Ahu88]. This uses an associative shared store called a *tuple space* which is implemented on a loosely coupled multicomputer by hashing data items to processors. Data items can be accessed either by name or by type, thus allowing a logical separation to be made between the location of an item and its value. This also affords a degree of data type polymorphism. Unfortunately, the latency of performing associative lookups considerably limits the granularity of sharing possible.

1.1.4 Hardware Single Shared Address Space Architectures

When intertask communication and DSM mechanisms are performed in software, the latency of setting up a new task is high (typically milliseconds) and so restricts these systems to coarse-grain parallelism. Ideally, it is desirable to implement these functions in hardware, thus reducing their latency. However, the intertask communication mechanisms in traditional operating systems are very complex and consequently difficult to implement in hardware.

A single shared address space (SSAS) memory model, where all tasks occupy a shared address space rather then private ones, appears to simplify several memory management problems so as to offer the potential to reduce latency (and hence grain size) and also make hardware memory management (HDSM) more feasible. The main advantages this has over the logically separate address spaces conventionally presented by operating systems are:

- The shared memory eliminates the redundancy among many traditional operating system mechanisms, such as file systems and inter-process communication. Potentially, this allows these software mechanisms to be coalesced into hardware, thus substantially reducing their latencies.

- As there is no address aliasing, and protection is implemented above the processor cache, there is no need to flush and subsequently reload the processor caches and address translation look-aside buffers (TLBs) on context switches. This will greatly speed up restarting processes after DSM accesses.

- The overhead of process creation and termination is very low as there is no need to create address space tables for the threads. This makes it more economic to employ fine-grained processes.

- Greater reuse of the page tables in the system is possible, thus freeing memory for more useful data.

Combining SSAS and HDSM, therefore, has the potential for reducing the latency of shared memory access so that network communications performance dominates overhead and hence determines granularity.

1.1.5 Wafer–Scale Integration

For a HDSM to be useful, a communications network with a comparable latency and bandwidth is required. Fibre optics, one of the most modern communication mediums, has the bandwidth to match HDSM, but compatible low latency switching elements are not available.

Wafer–scale integration (WSI), the process by which very large–scale integration (VLSI) circuits are packaged as whole wafers rather than as individual chips, can be used to embed processors and memory dies in a silicon substrate which contains a communications network. The main performance advantages wafer–scale integration has over conventional chip and PCB technologies are:

- Higher Speed

  Conventional VLSI technology uses chains of output transistors to drive the pins of chips. These prove to be slow. The interconnected tiles on a wafer do not need these circuits and hence the speed of internal wafer communication is considerably higher then conventional chip to chip communication.

- Increased Wire Density

  As Dally suggests [Dal87], VLSI chip technology is severely limited by the number of pins that can be placed on a chip. If the number is high, the package must be large to accommodate them. Consequently the chip is expensive and its PCB density is low. In addition, space considerations make it hard to route very large numbers of tracks away from a chip; expensive multi–layer PCBs are required. Therefore, data–paths in conventional computers are severely limited in width.

  This is not so much of a problem in wafer–scale devices as these can employ very dense tracking. Consequently, communication data–paths in wafers can be much wider, by a factor of four say, then those in conventional systems.

It is asserted that a network constructed using WSI can have sufficient performance to support a HDSM with a medium task granularity of 10^4 instructions over several hundred processors. WSI also offers other advantages, chiefly lower cost per function and higher reliability.

1.2 The Thesis

Currently, there appears to be no middle ground in parallel computing; an architecture which can support both several hundred medium–grained ($\approx 10^4$ instructions) parallel tasks and the shared variable programming paradigm would be advantageous in many applications.

This thesis asserts that it is possible to implement a new computer architecture, *Cherub*, which has several hundred processors and is able to support shared variable programming with an optimal task granularity of around 10^4 instructions. This can be achieved through the combination of a hardware–based distributed shared single address space and a wafer–scale communications network.

1.3 A Plan of the Dissertation

To support the thesis, three pieces of work are required:

- First, a specification of the properties expected of the new architecture must be written, showing that it will be useful for a significant set of applications.
- Next, the algorithms used to implement the architecture must be designed, allowing the requirements of the underlying network to be estimated.
- Finally, it must be shown that a wafer-scale integrated communications network is able to provide the required performance.

This work is divided between six chapters:

- Chapter two examines issues regarding distributed shared memory construction. It suggests a SSAS as appropriate for constructing a medium-grained HDSM.
- Chapter three introduces *Cherub*, a proposed computer architecture based on a HDSSASMA and defines its appearance to the programmer.
- Chapter four describes the algorithms required to implement *Cherub*. The latencies of these algorithms are estimated to allow the requirements of the underlying architecture to be defined.
- Chapter five asserts that only wafer–scale integration will be able to provide the performance required to successfully implement *Cherub*. A wafer–scale network is designed and is shown, through simulation, to achieve that performance.
- Chapter six summarises the results achieved in the dissertation and draws some conclusions.
- Appendix A contains a glossary of terms.
- Appendix B examines a large example application which would benefit greatly from the scalable medium–grained shared variable programming paradigm *Cherub* supports.
- Appendix C describes the network simulations which were performed.

1.4 Contributions to Knowledge

This dissertation is believed to be the first detailed description and investigation of a HDSSASMA, and certainly the first HDSSASMA relying on WSI!

Chapter 2

Reducing Intertask Communication Latencies

"You asked for information; you need background to understand it."

(Silver Tower — Dale Brown)

2.1 Introduction

In chapter one it was asserted that, currently, there is no middle ground in parallel computing and an architecture which can support both several hundred medium-grained ($\approx 10^4$ instructions) parallel tasks and the shared variable programming paradigm would be advantageous in many applications. It was suggested that such an architecture, called *Cherub*, could be constructed.

We have seen that two kinds of parallel architecture have become established. The first is the *multiprocessor*, which is built around a shared bus giving broadcast communications and a shared memory. This is characterised by low communications overhead, but limited scalability. The second is the *multicomputer*, which is based on point-to-point communications with larger communications overhead, but good scalability. Quantitatively, the low overhead of the multiprocessor is well matched to fine-grain tasks and, hence, to supporting the shared variable paradigm, while the high overhead of the multicomputer matches it to coarse-grain parallelism and, hence, to the message passing paradigm.

Multicomputer based *distributed shared memory* can support a scalable shared variable programming paradigm, but its high intertask communication latency must be significantly reduced if a medium grain of parallelism is to be supported. This chapter, therefore, examines the design of DSM systems and looks at ways of reducing their high intertask communication latencies.

2.2 Principles of Distributed Shared Memory

In order to use a shared variable programming paradigm on multicomputers a mechanism called distributed shared memory (DSM) has been devised. This provides the semantics of a shared memory on top of multicomputer hardware. A large enough grain of data sharing is employed to balance the amount of communication with the capabilities of the network.

For a DSM to provide the illusion of a physically shared memory, it must automatically transform shared–memory accesses into inter–processor communication which locates and retrieves the required data. This is usually performed by a DSM server on each processor which is responsible for moving shared memory pages among the processors on demand. A typical DSM system is illustrated in figure 2.1.

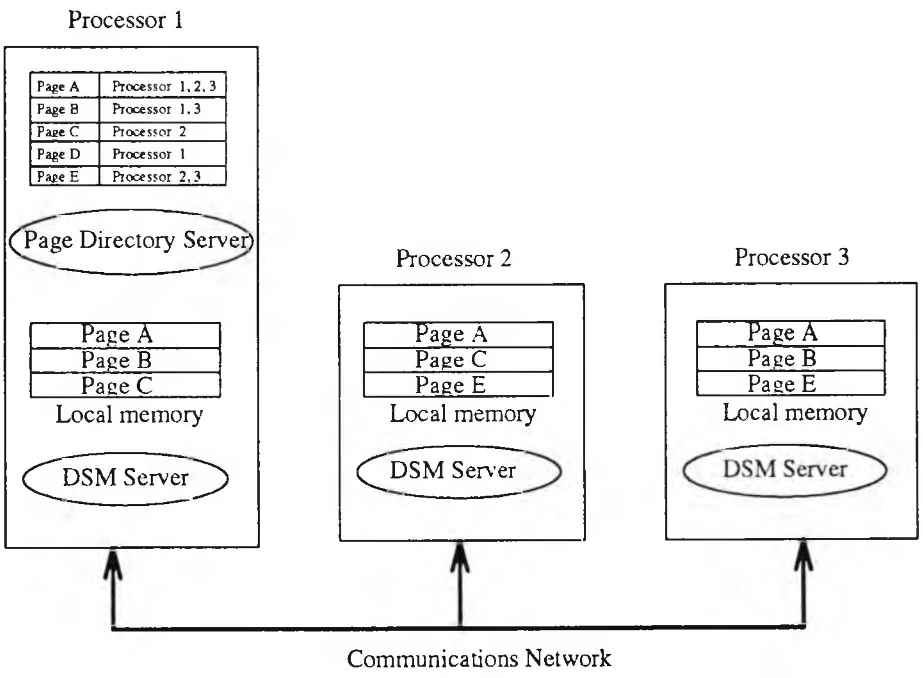


Figure 2.1: The Structure of a Typical DSM System

To be able to share the pages in the DSM, the servers must be able to locate them. This may be accomplished by one or more page directory servers. These maintain page directories which contain, directly or indirectly, the locations of the data items and their copies. The entries in the directories must be updated as pages are moved among the processors. Usually the functions of the shared memory server and the page directory server are combined in a single server.

One of the first DSMs was implemented by Li [Li 86]. He proposed three different directory schemes:

- Centralised Directory Server

  In this scheme a central directory server maintains lists of the pages owned by each node. Whenever a node requires a page, it communicates with its owner through the centralised server. As a consequence of all page requests passing through the centralised server, it is a performance bottleneck.

- Fixed Distributed Directory Server

  In this scheme several directory servers are employed to reduce the bottleneck experienced in the previous system. Each server is assigned a predefined subset of the pages. Every node in the system can identify the server responsible for each page and consult the appropriate one when page faults occur.

  This scheme generates a considerable amount of network traffic; at least three messages are produced on a read fault and potentially many more on a write fault.

- Dynamic Distributed Directory Server

  In the dynamic distributed server scheme every node has its own directory server. If a node receives a page request for a page it no longer owns, then it forwards the request according to where it believes it resides. A node's hints are updated according to the page faults it generates, the requests it receives and the periodic update broadcasts made by other nodes. Fowler [Fow86] proved that this algorithm always terminates by finding the true owner of the page.

  The number of messages required to locate an owner approximates to the logarithm of the number of hosts sharing the page. This reduces the network overhead of sharing infrequently modified pages.

Li's work was so fundamental that, even after nearly a decade, DSMs still employ his fixed and dynamic distributed page management algorithms.

2.3 DSM Performance Issues

Three main issues related to performance must be considered when designing a DSM: its granularity, its coherence mechanism and its synchronisation mechanism.

2.3.1 Data Granularity

Data granularity is the unit of data shared between processes. For implementational efficiency, the designers of distributed shared memories usually make the main memory page the unit of data granularity. Five competing issues determine the performance of a given page size:

- Locality of Reference

  A program will typically exhibit *locality of reference*, meaning that around a given time it will only use a small fraction its instructions and data. The pages which contain a program's active instructions and data are called its *working set*. These pages must be held in memory if *thrashing* is to be avoided. Unfortunately, the pages in the working set will also contain inactive instructions and data — in general, the larger a page, the greater the proportion of its contents that will be inactive. This results in inefficient memory usage.

- Internal Fragmentation

  Typically programs will not fill an integral number of pages; on average, half the final pages of their text, data and stacks will be empty. This residue space is wasted. Clearly, the smaller the page size, the lower the wastage through such *internal fragmentation* will be.

- False Data Sharing

  False data sharing can occur in shared memories where two or more unrelated variables, used by different programs, are located in the same page. The page appears to be shared even though the variables are not. The smaller the page, the less false data sharing will occur. False data sharing can have a profound effect on performance. For instance, some applications have shown speedups of 50% when false sharing was eliminated [Hag91b].

- Paging Overheads

  Servicing a DSM page fault can incur a considerable latency, typically milliseconds, the majority of which is independent of the page size [Whi92]. Therefore, the larger the page size, the fewer the number of page faults required to transfer a given amount of data and, consequently, the lower the latency incurred.

- Size of Directories

  It is necessary to keep directory information about the pages in the DSM. This can affect the performance of the system as the larger the page size, the smaller the directory required and the greater the memory available to hold useful data.

Clearly, a performance tradeoff exists, mainly between the amount of false sharing caused when the page size is too large and the paging overheads and excessive directory tables caused when it is too small. Experiments with the Psyche operating system [Bol] show that 256 byte pages appear to achieve a good compromise, although sometimes page sizes as small as 64 bytes are optimal.

Generally, hardware implementations of DSMs (called HDSMs) are able to employ much smaller page sizes than their counterparts implemented in software[1]. This is because the latency of servicing a page fault is much lower.

Unfortunately, due to their complexity, current HDSMs often sacrifice functionality in order to simplify the hardware: MemNet only allows one DSM region per machine; the J–Machine performs coherence in software and is implemented on low performance processors; while Dash and PLUS do not employ scalable coherence algorithms.

[1]Typical software implemented DSMs such as Mach [Acc86, Tev87b, For89], IVY [Li 86] and Mirage [Fle89a, Fle89b] employ pages of 4,096, 1,024 and 512 bytes respectively, while HDSMs such MemNet [Del86a, Del88a, Tam90], J–machine [Dal89], Dash [Len90, Len92] and PLUS [Bis90] enjoy pages of 32, 32, 16 and 4 bytes respectively.

2.3.2 Coherence Mechanisms

For reasons of efficiency, the DSM servers allow the processors to cache data in their local memory. This creates the logical memory hierarchy shown in figure 2.2. This, however, introduces data coherence problems. Generally updates to the shared memory must be propagated to the local copies cached at the processors. Data coherence can be maintained by restricting memory accesses while these propagations take place.

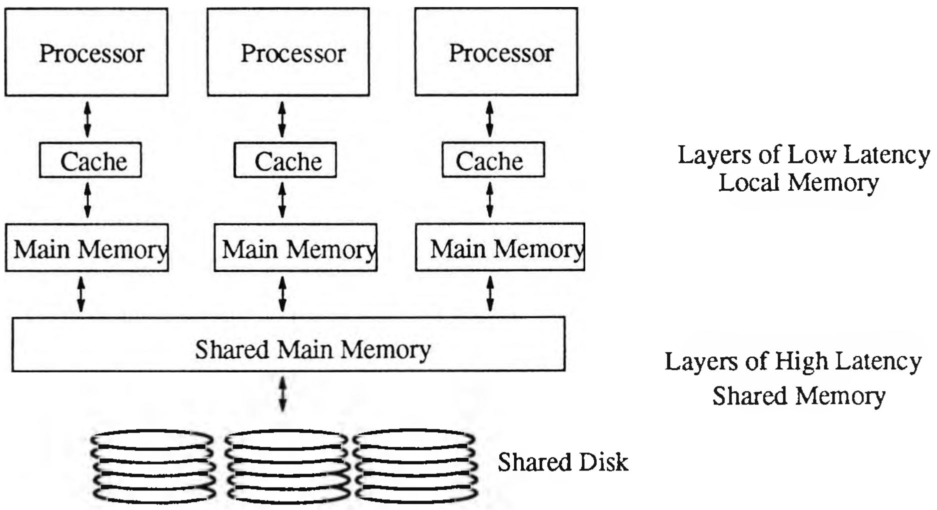


Figure 2.2: The Logical DSM Hierarchy

Coherence policies and associated mechanisms have evolved significantly over the last decade:

- Strong Coherence

  Early implementations of DSMs, such as IVY [Li 86] and MemNet [Del86a], emulated true shared memory architectures by providing strong, or sequential, data coherence. Using strong coherence semantics, a read operation performed by a processor returns the most recently written value.

  Strong coherence is typically implemented by allowing the existence of either a single writable copy of a given page, or multiple readable copies. In Li's IVY access to each page is strictly controlled by a write invalidate protocol; if a node Q faults when writing to a page p, its fault handler:

  - invalidates all copies of p,
  - obtains a copy of p from another node, if Q does not already have one.
  - changes the access permissions of p to *write*,
  - returns to the faulting instruction.

  Upon returning, node Q is said to own page p. Node Q is then allowed to read and write to the page freely. If a node Q faults when reading from page p, then its fault handler:

  - changes the access permissions of p to *read* on the owning node,
  - obtains a copy of p and sets its access permissions to *read*,
  - returns to the faulting instruction.

  Upon returning, node Q is free to read the page until it is invalidated by another node.

  A good example of an operation which requires strong coherence is synchronising using a lock variable; all processes must see the most up–to–date status of the lock.

  It was soon discovered that traditional shared memory algorithms often perform badly when using strong coherence. This is usually due to remote contention for data. However, strong coherence is still the most commonly implemented coherence strategy in DSMs.

- Relaxed Coherence

  The designers of second generation DSMs, such as Dash [Len90, Len92] and PLUS [Bis90], noted that strong coherence semantics are not necessary in most distributed applications. Consequently, they devised DSMs with a relaxed form of memory coherence. Using relaxed coherence semantics, a read operation performed by a processor will not necessarily return the most recently written value. If used correctly, this scheme can substantially increase program performance. However, if used incorrectly, it can cause programs to break unpredictably, thus giving incorrect results.

  A good example of a relaxed coherence scheme is that employed in the PLUS system [Bis90]. To minimise the cost of cache misses this uses a protocol which updates the cache copies of other nodes rather than invalidating them. When a processor generates a fault by reading a non–cached page:

  - The local coherence manager requests a copy of the page from the remote coherence manager responsible for the master copy.
  - If the page is currently being written, the remote coherence manager waits until the write completes.
  - The remote coherence manager then adds the requesting node to a list of nodes with copies of the page.
  - The required page is then sent to the local coherence manager.
  - Finally, the faulting process is restarted.

  When a processor writes to a page, it does not block, but rather:

  - The local coherence manager sends details of the write to the remote coherence manager responsible for the master copy of the page.
  - The remote coherence manager updates its copy of the page accordingly.
  - The remote coherence manager then sends an update request to the nearest node with a copy of the page. This propagates the request to any other nodes which also contain copies.

  - The last node in the copy list sends an acknowledge to the local coherence manager which originated the write operation.

  This protocol guarantees write consistency since writes are applied in the order that they are received at the remote coherence manager. However, a given read does not necessarily obtain the most up-to-date value of a memory location. *Special synchronisation primitives are, therefore, required for when data consistency is important.* In the case of PLUS, this takes the form of a fence operation which causes the coherence manager to block writes to a page until all the earlier ones have completed.

  Relaxed coherence semantics typically provide more efficient shared access than strong coherence as they require fewer synchronisations and less data movement. The Dash multicomputer system, for example, combines relaxed data coherence with a data acquire and release mechanism [Len90, Len92]. It was found that this gave a performance increase of between 10 to 40 percent over conventional strong coherence.

  Typically asynchronous algorithms, such as producer-consumer relationships, are suited to relaxed coherence. For example, a consumer process will only access the contents of a shared memory buffer after a producer process has filled it and performed a synchronisation operation. By using PLUS's relaxed, rather than IVY's strong, coherence protocol, one invalidation message can be saved per page of data passed in the buffer.

- User Provided Coherence

  The designers of third generation DSMs, such as Mach [For89] and the J-machine [Dal89], observed that, ideally, applications should be able to customise the DSM server according to their intended data access patterns. By employing a well defined interface, they allowed applications to provide their own DSM servers. For example, in the J-machine:

  - All memory locations are tagged with a state s.
  - Each memory operation can optionally specify a precondition, x, and a post condition, y, on the state.
  - If prior to accessing a memory location $s \neq x$, an exception handling process specific to s and x is created.
  - When the operation completes s is assigned the value y.

  The main problem with such schemes is that although their flexibility can be useful, DSM servers are difficult to write. As a result most programmers relied on the system provided default. The scheme, therefore, introduced additional software latencies, often without benefit.

- User Assisted Coherence

  The designers of the latest generation of DSMs also believe that it is best if an application makes its own coherence decisions, but they also understand that it is important to minimise the associated overhead on the programmer:

- The Munin system [Ben90] categorises nine types of shared memory by their pattern of access. A different cache coherence mechanism — some strong, others relaxed — can be employed for each. By logging the memory access patterns of a number of programs, it is possible to suggest which coherence mechanism best suits each. For example:
  - *Producer–Consumer Relationships*
    The producer does not need to access data pages after they have been written and the consumer does not need them again after they have been read.
  - *Burst–Write Pages*
    These should be held on the processor while being written, but can be released for another processor once the burst is over.

  The main problem with this approach is that a page isn't necessarily used in the same way throughout its life. A good example of this is a page on the stack. In addition, not all languages are amenable to this type of analysis. For example, single assignment languages, such as the functional language Paragon [And91b], do not reuse data space memory.
- Hill et al [Hil92] suggest that the programmer or compiler should be able to bracket shared data accesses with annotations which indicate their intended use. Unlike the schemes employed in Munin, the annotations are only advisory and so can improve the performance of the shared memory without altering its coherence semantics.

  Hill's Check–In and Check–Out (CICO) shared–memory model uses special Dir_1SW hardware to support the following annotations:

check_out_X	Expect exclusive access to block
check_out_S	Expect shared access to block
check_in	Relinquish a block

  In addition, two additional annotations are supported to allow data prefetching to be overlapped with computation:

prefetch_X	Expect exclusive access to block in near future
prefetch_S	Expect shared access to block in near future

  Hill hand annotated several parallel applications from the SPLASH benchmark suite [Sin92] to simulate the effects of the CICO model. In most cases it was possible to avoid competitive page sharing almost entirely. The programs that relied upon unsynchronised data sharing (data races) were notable exceptions, but even so, their data faults were significantly reduced.

2.3.3 Synchronisation Mechanisms

In most systems the execution of processes must be delayed while constraints imposed on the ordering of actions are satisfied. This is usually achieved through the use of synchronisation mechanisms which guarantee mutual exclusion among processes within certain critical regions of code.

The conventional shared variable programming paradigm provides a number of synchronisation mechanisms such as locks, semaphores [Dij65] and monitors [Hoa74]. These are typically implemented using one or more shared variables, which are continually read, or 'spun' on, by processes waiting for the synchronisation conditions to occur. Not only does such spinning waste processor time, thus reducing throughput, but in DSMs it also incurs excessive communication costs as the pages containing the variables must be frequently moved between the processors running the processes. This is called *thrashing.* It is, therefore, necessary to provide synchronisation primitives which are efficient in the distributed shared memory environment.

Two examples of DSMs which have specialised synchronisation mechanisms are:

- Mirage

  The Mirage system [Fle89a, Fle89b] allows a process to lock a page, such as one containing a synchronisation variable, into its processor's memory for a given time quantum called a *delta.* All other processes accessing the page are blocked until either the *delta* expires, or the process gives up the page voluntarily. Thus thrashing is prevented.

- Linda

  The Linda programming paradigm provides an inherent distributed synchronisation mechanism through the presence or absence of keyed data items, tuples, in a region of associative, distributed shared memory called the tuple space (TS). Tuples are placed in the TS using the Linda *out* primitive. They are removed by the *in* primitive. This searches the TS for the tuple and, if it is present, removes it. If it is not present, the primitive blocks until the tuple is placed into the TS.

  Gelernter has demonstrated that it is possible to use Linda to construct spin-free locking mechanisms [Car89]. A tuple with a well-known key is associated with a critical region of code. Whenever this tuple is in the tuple space, the code region is unlocked. Before a process can enter the critical code region it must perform an *in* primitive. This locks the critical region. When the process leaves the critical region of code, it re-inserts the tuple into the TS using the *out* primitive. This has the effect of unlocking the region.

  Similar techniques can be used to construct efficient semaphore and message passing mechanisms.

2.3.4 Implementing Intertask Communication Mechanisms in Hardware

When intertask communication and DSM mechanisms are performed in software, the latency of setting up a new task is high (typically milliseconds) and so restricts these systems to coarse-grain parallelism. Ideally, it is desirable to implement a DSM in hardware (HDSM), thus reducing its latency. Although HDSMs have already been tried — notably MemNet, J-machine, Dash and PLUS — typically they have compromised their functionality because of the complexity of intertask communication mechanisms. Some way must, therefore, be found of simplifying these mechanisms.

One way of doing so is using a cache–only memory architecture (COMA) such as the DDM [Hag91a, Hag92] and the KSR1 [Bur92]. These are distributed architectures whose processors' memories are organised as large set–associative caches. Hardware coherence hardware is used to create the illusion of a single global layer of cache. Complex memory management schemes which create a virtual memory can then be implemented relatively easily above the level of the global cache. Unfortunately, this still leaves the process creation and termination mechanisms implemented in relatively slow software.

This idea has evolved into the single shared address space (SSAS) architecture, where all tasks occupy a shared address space rather than private ones. This appears to simplify several memory management problems so as to potentially reduce latency (and hence grain size) and also make the coalescing of interprocess communication mechanisms into hardware more feasible.

2.4 Single Shared Address Space Architectures (SSAS's)

In conventional computer architectures, processes have logically separate address spaces, typically at least 32 virtual address bits in size. These provide processes with both address independence and data space protection. In recent years, however, there has been a dramatic increase in program size. Applications such as databases and multimedia are easily able to consume 32 bits of data virtual address [Kru89, Mas91a]:

- Databases

  To benefit from hardware DRAM caching, database designers often directly map entire databases into virtual memory. Many commercial databases already consume 40 address bits.

- Multimedia

  At 24 frames a second and around 4 Mbytes per frame, uncompressed video will consume 32 address bits in only 45 seconds. Similarly, at around 25 Mbytes each, 32 address bits can only hold 160 high quality A4 colour images.

In 1990, Hennessy and Patterson [Hen90] made the following observation:

> *Address–Consumption Rule:*
> The memory needed by the average program grows by a factor of 1.5 to 2 per year. That is, one address bit is consumed per year.

In the past, when the number of available address bits has been exhausted both hardware and software techniques have been tried to alleviate the problem:

- Segmentation

  The HP PA–RISC [Hew90], used in the Snake Workstation, and IBM RS/6000 [Jef90] have address spaces constructed of 32–bit segments. When an address is accessed, the type of machine instruction used determines the segment referred to. This effectively gives programs access to a number of 32–bit address spaces.

- Swizzling

  A notable software solution called swizzling [Wil91c], has been described by Wilson. In this scheme, programs on disk contain wide symbolic addresses. These are converted to actual 32-bit addresses at page fault time. Unfortunately, this means that all imported pointers must be translated, even those not used. In addition, memory must not only be assigned for the imported page, but also for the pages referenced by it. This makes the technique particularly inefficient.

Past experience, most notably the PDP-11 architecture, has shown that such solutions are hardly ever satisfactory and only act as stop-gaps until processors with larger addresses appear; either they are too difficult to program, or they incur very high performance overheads. Fortunately, a number of forward-looking companies have realised this; both the MIPS R4000 [MIP] and Digital Alpha [Dig92] have unsegmented 64-bit virtual address spaces. This step has had a revolutionary, rather than evolutionary, effect upon operating system design. A 64-bit address space is virtually unconsumable using *conventional* programming. It eliminates the traditional need to reuse the address space of programs. Consequently, there has been considerable interest in single shared address space (SSAS) architectures, most notably Psyche [Sco89a, Sco89b, Scoa, Cha, LeB89, Scob, Mar], Opal [Cha92a, Cha92b] and ANGEL [Wil91a, Wil92, Sti92].

SSAS computing is the logical extension of the light-weight processes found in conventional operating systems such as Chorus [Arm86, Her88], Mach [Acc86, Tev87a], Choices [Cam87] and the Synthesis Kernel [Mas89]. Light-weight processes have very little context of their own, typically only a stack segment. A number of light-weight processes share the address space of a heavy-weight process, thus making it very cheap to share data, switch contexts and perform optimised scheduling between them. The SSAS extends this concept by making all processes light-weight, residing within a globally shared protected address space. Single and multiple address space architectures are compared in figure 2.3. The dotted lines denote protection domains.

It is interesting to note that even 64 bits of virtual addressing is not really enough! If all of the computers in the world — many millions perhaps — are to be mapped into a single address space using wide-area networks, even more address bits — the next step is logically 128 — will be required.

Through the abstraction of a protected SSAS it is possible to unify, and hence simplify, many traditional operating system mechanisms such as memory, files, inter-process communication, and protection.

2.4.1 Using a SSAS to Simplify Operating System Mechanisms

SSAS architectures can be used to simplify many traditional operating system mechanisms, thus making it easier to implement them in hardware.

Unifying Caching Mechanisms

Traditionally operating systems have a plethora of caching and copying mechanisms, all of which are implemented in software. Often operations are replicated. Two examples of

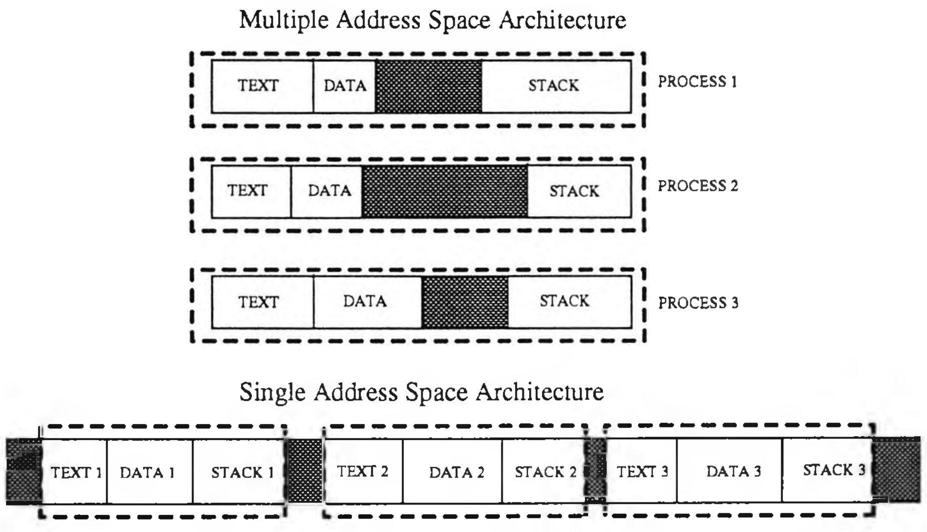


Figure 2.3: Comparing Single and Multiple Address Space Architectures

this from the UNIX System V operating system [Bac86] are:

- The File System Block and Inode Caches

  UNIX uses a cache to hold regularly used file inodes. These are not held in the normal block cache because of the sparse nature of inode accesses and the large size of file system blocks. However, the inode cache can still contain inodes which are also held in the file system block cache. This is clearly wasteful.

- The Page List and the File System Block Cache

  Program text, though read-only, is held in memory when being executed by programs but is discarded when not being used. UNIX keeps the pages of the program text in the file system block cache as well as the memory in case it is needed again soon. It is wasteful to have two copies of the same text in memory.

Multiple level caches like this waste memory, but more importantly add to software latencies. The problem is that UNIX was designed when main memory was scarce. Hence it employs hierarchical caches to optimise memory usage. Now that main memory is relatively inexpensive, such excessive caching is pointless.

A SSAS architecture is able to use its alias-free memory hierarchy to provide invisible caching, replacing individual caches with a single unified mechanism. This technique has also been used to coalesce other operating system mechanisms by unifying memory with:

- Files (memory mapped files in MULTICS [Cor65], Mach [Tev87b] and UNIX [Mey88, Lef89]);
- Inter-process communication (memory objects in Mach [Acc86]); and
- Processes (Killian's /proc file system [Kil85]).

This reduces the software overheads normally associated with many of the traditional operating system mechanisms and increases the system call, and hence intertask communication, bandwidth. The Synthesis Kernel shows how effective such a reduction in the layers of operating system software can be [Pu 88, Mas89].

There are two potential problems with this:

- Loss of High–Level Control

  Unifying mechanisms can sometimes result in a performance degradation due to the loss of high–level control. Yokoyama et al [Yok89] describe the situation with memory mapped files in which a process overwrites a whole page of data in a file which is on disk. A conventional file system call is told that the whole page of data is to be overwritten and so knows that it is not necessary to read its old contents from disk. A memory mapped file, however, will perform this pointless disk read.

- Unsuitability of Some Mechanisms

  Some operating system mechanisms are not suited to being treated as conventional memory. Serial output devices, such as terminals, are notable examples of this.

It is clear, therefore, that although unification benefits most data operations, applications should retain a degree of control for situations where it does not.

The coalescing of software levels, made possible by a well–designed SSAS architecture, has the potential to reduce the latency of intertask communication in a distributed shared memory and make finer–grained processing feasible.

Increased Memory Sharing

In general, memory sharing between processes is a good thing as it implies more economical memory usage. The history of the UNIX operating system illustrates how memory sharing has increased in importance over time:

- UNIX 3BSD (1981) allowed processes running the same program to share a copy of the text segment (which could not be altered), while each process had its own copy of the data segment. This saved main memory when multiple copies of a program were executed simultaneously.
- In System V R2V4 (1984) the fork system call was re-implemented to use copy–on–write (COW) techniques. These allow a process's memory to be copied at low cost by allowing processes to share pages while they are unmodified; pages are only copied when they are changed, thus postponing page copying and often eliminating it altogether. This significantly reduces the overhead of the fork system call in two ways:
  - The time required to create the new process image is minimised as there is no data copying involved initially.
  - The memory requirements of the processes are minimised through the efficient sharing of data.

COW also provides an efficient and relatively unobtrusive way of checkpointing processes for debugging [Fel89] or fault tolerance [Wil93].

- System V R3 (1987) introduced shared libraries [Jam86]. These go a step further by allowing different programs to share common routines from specially structured libraries. Dynamic linking is required to execute programs, but the saving in disk (around 96%) and main memory (around 34%) usually results in a net performance gain [Ros89].

  An additional advantage of shared libraries is that errors in them can be fixed without rebuilding executable files. By installing a new shared library, old executable files automatically use the updated library without relinking.

A SSAS architecture allows the logical extension of these techniques; everything in the address space can be potentially shared or copied using COW. It is hoped that this increased level of sharing will reduce the need for data movement between processors in a distributed shared memory system, thus improving performance.

Reducing DSM Context Switch Latencies

In physically shared memory architectures, where the shared memory latency is very low, processors typically block while waiting for data to be loaded from the shared memory. In DSM architectures the memory latency is much higher, thus making it cost effective to perform context switches when shared memory faults occur. Therefore, the context switch latency of a DSM is important in determining its performance and should be minimised.

One of the major factors in determining the latency of a context switch is the design of the processor cache. Normally processors have either one or more layers of local cache memory composed of associatively addressed SRAM[2]. Most processor caches are accessed using physical addresses. One of the main reasons for this is that it provides process address independence — different processes can use the same virtual addresses without clashing. If physical addressing is not used, each time a context switch occurs the cache must be flushed, as the new virtual addresses refer to different physical locations. This is a significant performance penalty as writing cache lines back to memory is slow. Furthermore, when an old process is restarted it must recapture its cache context, thus incurring an additional penalty.

Unfortunately, when accessing a physically addressed cache an address translation step is required to convert the virtual addresses used by processes into the physical addresses used by the cache and main memory. This takes time and requires another partially associative cache, the translation lookaside buffer (TLB). The number of entries in the TLB effectively

[2]SRAM caches are typically three to four times faster then DRAM based main memory. They are, however, very silicon expensive. This is because DRAM generally uses one-transistor memory cells, which require about one-fourth of the area used by the four and six-transistor (flip-flop) memory employed in SRAM. In addition each cache line has a number of tags and flags which are used by the associative addressing hardware. These consume extra silicon. Processor caches, therefore, tend to be small. The SPARC-2, for example, is one of the most modern RISC processors and it has a one Mbyte external SRAM cache. This is small compared to the main memory of most modern workstations which is at least eight Mbytes.

determines the smallest page size which can be employed by a processor; the smaller the page size, the smaller the memory the TLB addresses.

For performance reasons it is, therefore, desirable to construct virtually addressed caches. These eliminate the TLB translation from cache hits, allowing lookups to be performed in a single clock period. In addition, without the limitations of the TLB, a smaller page size can be employed. Unfortunately, to avoid having to flush the cache on context switches, the cache lines must be tagged with additional information[3]. This is silicon expensive, however, considerably reducing the size of the cache.

A SSAS architecture simplifies the construction of virtually addressed caches because it does not allow address aliasing. This reduces DSM context switch latencies, increases the speed of cache lookups, and allows larger caches to be constructed.

2.4.2 Disadvantages of a SSAS Architecture

Separate address space architectures do have a number of useful properties which are lost with a SSAS:

- Process Address Independence

  The separate address space architecture allows different processes to use the same virtual addresses without clash. Processes in single address space architectures must somehow avoid inadvertent address clashes.

  This is trivial with different programs; they can be simply compiled to run at different addresses. Allowing multiple copies of a given program to be run simultaneously, however, or supporting UNIX fork semantics, is much more difficult:

  - Running Multiple Copies of a Program Simultaneously

    Executing a number of copies of a program simultaneously is difficult in a single address architecture because any static data accessed using absolute addresses is shared between all the programs. This problem is especially prevalent in shared libraries. Note that this is only important when a process modifies its static data; read-only data is not a problem.

    One way of solving the problem is to use a level of indirection. Modifiable static data is accessed via indirection vectors placed at unique fixed addresses on the bottom of the process's stack. When the process starts up it makes a copy of its data using COW. It then initialises the indirection vectors to the appropriate locations in the data copy. As processes have unique stacks, they can access their own static data by dereferencing the appropriate vectors.

    The main advantage of this technique is that it does not reduce the ability of programs to share data. It has two minor drawbacks however:

    * An indirection cost is incurred on each static data access. Fortunately, processes rarely write to static data. When this does happen, however, the

[3]Process address independence can be provided by tagging cache lines with the owning process's ID (PID). Caches then only need to be flushed when a PID is recycled. Tags are problematic, however, when different processes share common regions of memory.

indirection vectors will be automatically cached so that the performance cost is minimised.
The indirection overhead could be greatly reduced by inventing a new processor instruction which dereferences indexed locations on the bottom of the process's stack.

* The indirection vectors need to be managed so that every item of static data in the system is assigned a unique vector. As these vectors are relatively sparse, this is not a significant problem when process stacks are large.

Figure 2.4 shows how a shared library can be implemented using this technique. Jump vectors to the routines are placed at the start of the library. This enables the internal format of the library to be changed without having to relink the programs which use it.

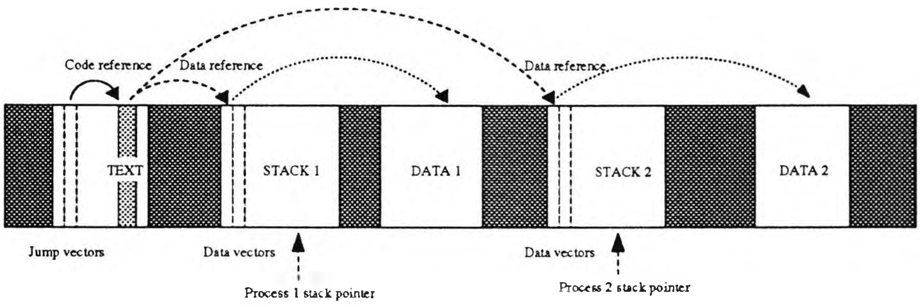


Figure 2.4: Using Indirection to Allow Code Sharing

– Supporting UNIX Fork Semantics

The UNIX fork system call creates a new child process with an exact image of its parent's address space. This mechanism relies on having an architecture which is able to provide address independence. SSAS architectures cannot provide this and so have difficulty supporting fork semantics.

The author has proposed software techniques for providing UNIX-like fork semantics on single address space machines [Gul89, Gul88]. Generally, these involve scheduling forked processes so that they never execute simultaneously. It is then possible to swap forked processes in memory so that the executing process always occupies the original memory. Clearly this is slow and, due to the migration of processes in memory, it severely limits the way addresses can be passed between processes. Similar techniques are employed in the Cedar [Tei84, Swi86] and Pilot [Red80] operating systems.

Hardware segmentation, often employed in conventional operating systems[4], is a faster and cleaner solution to this problem. This allows a child process to create local aliases for its stack and data, so that they overlay those of its parent. When a process wishes to pass stack or data addresses to another, it must first convert them to global addresses. The ANGEL operating system uses this scheme [Wil91a, Wil92], but this solution has two main disadvantages:

[4]For example, MS-DOS, MULTICS [Cor65], Hewlett-Packard's Precision [Lee89] and COSMOS [Hor].

* Child processes require an address translation before every cache access. This loses much of the speed advantage of the virtually addressed cache. Processes not resulting from a fork, however, do not need the address translation and so incur no performance loss.
* It discards many of the software advantages of only having one pointer type.

The operating system community is coming around to agreeing that compatibility with UNIX fork semantics is not important [Sco89b, Cha92b]. In reality, few UNIX programs make active use of the fork semantics[5] and it should not be difficult to reimplement those which do.

- Process Protection

  The logical separation of address spaces in conventional architectures provides processes with implicit protection from the malicious or accidental actions of other processes. This is not so in SSAS architectures. If the SSAS architecture is to be useful in a multi–user environment, somehow it must be possible to prevent processes from accessing logical address ranges to which they are not entitled. Although this protection mechanism must work at the cache level, it is important that it should not delay the cache lookup process.

  Protection can be provided in hardware by giving each process a set of domain registers. These are similar to the base and limit registers found in segmented architectures. A process is only able to access an address if it is within the range of one of its domain registers which has the appropriate access rights. The process shown in figure 2.5 can access the range of addresses shown in white. This mechanism does not incur any performance penalty as domain register and cache lookups can be made in parallel.

Domain Registers

Base	Limit	Access
3000	1000	R/X
5000	2000	R/W
9000	200	R/W

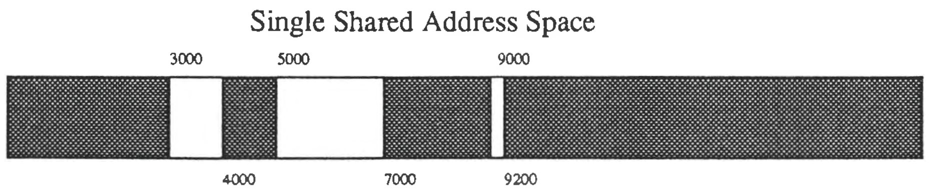


Figure 2.5: Providing Protection by Domain Registers

[5] *make* [Fel79] is a notable exception.

When a process attempts to access an address to which it is not entitled, an exception is generated. The process may be allowed to trap this error, giving it the opportunity to correct its domain registers and continue.

Assume that the SSAS is divided into a number of non–overlapping memory ranges, or objects[6]. Access to each object is strictly controlled by the operating system; it provides a mechanism with which a process can manipulate its domain registers, but permits it to access only those objects to which it is entitled. When the operating system grants a process access to an object, it loads the object's start address, limit and access rights into one of its domain registers.

Two different schemes for object access control have been used in previous, although not exclusively SSAS, systems:

- Passwords

  This scheme is employed by operating systems such as Chorus (ports) [Arm86], Psyche (keys) [Sco89b], Amoeba (capabilities) [Mul91] and Opal (protected pointers) [Cha92b]. When a process creates a new object it specifies a number of secret passwords corresponding to access rights such as read, write and execute. Passwords are simply random numbers with enough bits to make their forgery improbable. A process can get the operating system to enter an object into its domain registers by making a system call, quoting the object's start address and the appropriate secret passwords for the required access rights.

  It is envisaged that each user has a local name server object which contains mappings of hierarchical names to object and password pairs. A user's processes are given the password to enable them to read their name server object. One of the features of this approach is that a public program which uses an object which users are not allowed to access directly[7], must have the object's password hard–wired into its code. This can be obtained from the master name server when the program is compiled by a privileged user.

  The main advantage of the password scheme is that the operating system does not need to be concerned about the movement of passwords between processes, only their attempted use. This reduces communications overheads.

  The main disadvantage of the scheme is that a single process's rights to access an object cannot be easily revoked; changing the password has the undesirable side effect of also preventing other processes from accessing the object. It is also difficult to prevent a process from giving away a password to a process which is not entitled to it.

- Access Control Lists

  This scheme is employed by operating systems such as Mach [Acc86] and ANGEL [Wil92]. It provides a higher degree of security than password protection, but at the cost of an additional communication overhead.

  Each object has a secret access control list which is maintained by the kernel. This states which processes are able to access the object and in which ways. When a process accesses an object which is not in its domain registers, the

[6]These are called Windows in Ra [Ber88, Das88], Realms in Psyche and Segments in Opal.

[7]For example, the UNIX command *ps* accesses the kernel memory device /dev/kmem.

kernel examines the object's access control list. If the process has the required access rights to the object, then its details are entered into the domain registers. If not, then an exception is generated.

A process can obtain access rights to an object in only two ways: by initially creating the object; or by being granted them by another process, such as a name server. This server must both possess the access rights and be allowed to give them away. The server grants the access rights by issuing a system call which tells the kernel to add the new process to the object's access control list.

This scheme has two main advantages: it is impossible for a process to fraudulently obtain access rights to an object and a process's rights to access an object can be revoked by simply removing its name from the appropriate access control list.

Both of these schemes allow the traditional login process to be replaced with a mechanism by which the name and password of a local name server object are supplied and validated. This policy allows the traditional operating system concept of users to be discarded in favour of a more powerful sharing mechanism.

The number of domain registers determines the number of objects which can be mapped into a process's address space simultaneously. In effect domain registers are similar to UNIX file descriptors. Processes in SunOS 4.0, for example, are limited to 64 active file descriptors.

- Resource Reclamation

  Previously, it was claimed that it is virtually impossible to fill a 64–bit address space. Yet real systems have finite amounts of physical memory. At some point it becomes necessary to reuse it.

  When a process in a separate address space architecture terminates, its resources can be reclaimed easily because it has a private context. In SSAS architectures, as a side–effect of the increased sharing, it is more difficult to define process contexts; an item of data can only be reclaimed when there are no pointers to it anywhere in the system.

  The minimum level of support for the architecture requires applications to explicitly request the deletion of data from the address space. This places the onus on programs to clear up after themselves, but if necessary, automatic garbage collection routines can be provided.

2.5 Conclusion

This chapter has shown that, in order to provide the required scalability, *Cherub* must be implemented on a distributed architecture. A distributed shared memory (DSM) — a mechanism which provides the semantics of a physically shared memory on a distributed architecture — is therefore required to implement the shared variable programming paradigm. Unfortunately, when intertask communication and DSM mechanisms are implemented in software, the latency of setting up a new task is high. This restricts them to coarse–grain parallelism. Implementing a DSM in hardware (HDSM) would reduce its latency, but is difficult due to the complexity of the conventional memory model. Hence, a simplified memory model must be constructed.

Single shared address space (SSAS) architectures were then introduced. Although still very much in their infancy, with many issues concerning their structure and efficient use yet to be fully investigated, these appear to offer several operating system simplifications such as unifying caching mechanisms, increasing memory sharing and reducing DSM context switch latencies. *Combining SSAS and HDSM, therefore, has the potential for reducing the latency of shared memory access so that network communications performance dominates overhead and hence determines granularity.*

Chapter 3

Simplifying The Operating System Call Interface

"Compromise is the art of design"
(Unknown)

3.1 Introduction

Existing parallel architectures either support a few tens of fine-grained concurrently executing tasks communicating via shared variables, or hundreds of coarse-grained tasks communicating via message passing. Currently, there is little middle ground. In chapter one it was asserted that a significant number of applications exist which are well suited to a medium-grained parallel architecture with several hundred processors which can support the shared variable programming paradigm [Don91]. The *Cherub* architecture is an attempt to address this need.

Chapter two has asserted that by employing a SSAS memory model, *Cherub's* intertask communication mechanisms can be simplified to the extent that they can be easily implemented in hardware. This will greatly reduce communication latency, thus allowing medium-grained programming. The increased data sharing encouraged by the memory model will also result in a significant improvement in the efficiency of memory usage over conventional architectures.

This chapter deals with the problem of designing a new operating system interface which can be used to provide similar functionality to the system calls of a conventional operating system, but which are considerably simpler to implement in hardware. To do this we must first understand what services a conventional operating system provides. Then we must examine ways of unifying those services through the use of a SSAS. Finally, having defined an interface to *Cherub*, we must show that it is useful. This can be achieved by identifying classes of applications for which the interface is well suited.

The *Cherub* architecture presented in this chapter is the work of the author and is, in effect, an early form of the ANGEL operating system being developed at City University and Imperial College [Wil91a, Wil92, Sti92].

3.2 A Conventional Operating System Interface

Before we can decide what the *Cherub* interface should look like, we must determine what services conventional operating systems provide. For example, the system calls of UNIX Version 7 (V7) can be grouped into a number of categories:

- Process Management

  The UNIX V7 operating system supports multiprocessing — running processes concurrently. System calls are provided for the creation (*fork*), execution (*exec*), synchronisation (*wait*) and termination (*exit*) of processes.

- Persistent Data Storage

  Persistent, high latency, data storage is accomplished in UNIX through the file system:

  - Data is stored in files located within a hierarchical name space. System calls are provided for the creation (*creat, mknod, link*) and destruction (*unlink*) of files, as well as for the manipulation of the name space (*chdir, mount, unmount, chroot*).
  - Data access is provided though a set of system calls (*open, close, read, write, lseek, ioctl, stat, fstat).*
  - Protection is achieved through the concept of *user* and *group* identities. System calls are provided for the manipulation of identities (*getuid, getgid, setuid, setgid*) and for altering their associated access rights to files (*chmod, chown*).

- Non-persistent Data Storage

  Non-persistent, low latency, data storage takes place in memory. Text, data and stack memory regions, called segments, are automatically allocated upon a process's creation and deallocated on its termination. Processes allocate memory for their heaps manually (*brk*).

  - Memory locations are named using addresses comprising segment and offset components.
  - Memory locations are accessed using read and write machine instructions.
  - Protection is achieved by giving each process its own private address space.

- Interprocess Communication

  V7 provides two interprocess communication mechanisms:

  - Synchronous communication is performed via the file system. System calls (*pipe, dup*) are provided which allow the output of one program (*stdout*) to be fed into the input of another (*stdin*). Protection and naming is provided through the file system owner and group mechanisms.
  - System calls are also provided for the asynchronous sending (*kill, alarm*) and receiving (*signal, pause*) of events. Processes are named with unique identifiers (*getpid*) and protection is provided through the file system.

- Time Management

  V7 provides several system calls that allow the reading (*time*) and setting (*stime*) of the time-of-day clock, as well as for determining file access (*utime*) and process execution (*times*) times.

- Debugging

  V7 provides a system call which allows a process to control the execution of another for debugging purposes (*ptrace*).

- Booting and Shutting the System Down

  Finally, to shut the system down, the disk cache is first flushed (*sync*) and then the operating system is halted (*reboot*). When the operating system is rebooted, it creates a single user process called *init*. This is responsible for creating additional user processes which perform user logins.

Many system functions are provided by user level programs through the use of special devices which are accessed as normal files. These devices include:

- */dev/tty*

  These devices allow access to the character based terminals connected to the machine. They can be used to read input from keyboards and write output to screens.

- */dev/kmem* and */dev/mem*

  These devices can be used to access the kernel's virtual memory and the system's physical memory respectively. They are used by programs which require detailed process information, such as *ps*.

- */dev/di*

  These devices can be used to access the raw contents of the disk drives. They are used by programs which require detailed file system information, such as *df*.

We will use the UNIX V7 system calls as a checklist of the services the *Cherub* interface should provide.

3.3 The Cherub Interface

Like a conventional operating system, *Cherub* must provide programming abstractions such as processes, persistent and non-persistent storage and inter-process communication. However, this must be done in a way which lends itself to being implemented in hardware. This implies that these abstractions should be unified — and hence simplified — using a SSAS.

- The *Cherub* SSAS is seen by the programmer as a single 64-bit address space, called the Object Space.

- The Object Space contains protected non–overlapping address ranges called objects[1]. Objects unify programming abstractions such as processes, persistent and non–persistent storage and inter–process communication.
- All objects have the same fixed finite size; they neither grow nor diminish in size. This simplifies the address space management as objects are not allowed to overlap.

Cherub must provide mechanisms for management of objects within the Object Space. Explicit creation and deletion mechanisms are employed for their simplicity. Names must also be provided by which objects can be referenced.

- Objects are created using the *create_object* system call. This takes two parameters, *new_dr* and *image_dr*. These will be explained when protection and access semantics are discussed. The call unifies the UNIX process (*fork*) and file (*creat, link, mknod*) creation system calls.

  The system call allocates a unique identifier, *global_name*, by which the object can be referred to. This is actually the start address of the object within the Object Space, thus creating a flat name space. To allow objects to be referenced by address, the location of an object does not change throughout its lifetime. This mechanism unifies the UNIX naming schemes for files, memory and processes. The first twelve objects in the Object Space are reserved for the operating system. This will be explained later.

  It is envisaged that personalised user name servers will be employed to map hierarchical names, similar to those of UNIX files, onto the flat *global_names* employed by *Cherub*.

- Objects are persistent, that is, once created they exist until explicitly deleted. This simplifies resource allocation. Object deletion is performed by the *destroy_object* system call. This takes a single parameter, *old_dr*, which will be explained when protection is discussed.

  Once an object is deleted, its *global_name* becomes invalid and any attempt to use it will generate an exception. Garbage collection of unreferenced objects must be performed by the programmer.

  This system call unifies the UNIX process termination (*exit*), file destruction (*unlink*) and memory reclamation mechanisms.

Cherub must also provide some mechanism by which object information such resource usage and access times can be obtained.

- Statistics about an object can be obtained using the *object_info* system call. This takes two parameters, *old_dr* and *statistics_buffer_ptr.* The system call writes information regarding the object specified by *old_dr* into the *statistics_buffer.* The *old_dr* parameter will be explained later.

  This system call unifies the UNIX *stat* and *fstat* system calls with the /dev/kmem, */dev/mem*, and */dev/di* devices.

[1]These are not objects in the full object–oriented sense.

Objects in the Object Space must be protected from the accidental or malicious actions of processes. A password system will be employed because of its simplicity and low communications overhead.

- Objects are protected by passwords called *capabilities*. Each object has three capabilities: *read*, *write* and *execute*. An object's capabilities are assigned upon its creation and cannot be changed. By convention, the value 0 is both an invalid capability and address. This value, therefore, can be used by a process to relinquish an access right to an object. It also helps trap invalid pointer dereferences.

  Each process has a number of *protection domain registers*. To access an object in a particular manner, a process must first load the applicable *capability* to that object into one of its protection domain registers. A capability's validity is determined when it is first used to access an object. A data access made without the correct capability generates an exception, allowing the process to correct the error. The validation and exception process takes a finite amount of time, thus complicating attempts to crack capabilities systematically. This mechanism equates with the UNIX *open* and *close* system calls.

  Once a process has a capability to an object, it can use it for the object's lifetime. Giving another process an object's capability allows it to access the object in the appropriate manner. The set of capabilities possessed by a process, therefore, equates with the UNIX concept of ownership and access rights (*chown, chmod, getuid, getgid, setuid, setgid*). Consequently, logging-in is simply a mechanism by which the *global_name* and capabilities for a memory object, which contains the set of name mappings and capabilities associated with a given user, are presented to a name server program for verification and use.

  The *protection domain registers* are also used by the system calls:

  - The capabilities to be assigned to an object are passed to the *create_object* system call in the *protection domain register* specified by the *new_dr* parameter.
  - For a *destroy_object* system call to be successfully performed on an object, its *write* capability must be contained in the *protection domain register* specified by the *old_dr* parameter.
  - For a *object_info* system call to be performed, the object's *read* capability must be contained in the *protection domain register* specified by the *old_dr* parameter. The process must also possess the *write* capability to the *statistics_buffer* used.

All object are accessed as if they were conventional memory, thus providing a consistent interface to the programmer. This unifies the UNIX file (*read, write, lseek, ioctl*) and memory access mechanisms. However, object access semantics can differ, thus providing scope for various programming abstractions.

- The semantics of objects are defined upon creation. To maintain the simplicity of the SSAS, it is envisaged that there will be only six types of object: memory, process, sleep-wakeup, semaphore, rendezvous and hardware. (Other object types may be added in the future.)

The access semantics of an object are passed to the *create_object* system call in the *protection domain register* specified by the *new_dr* parameter.

- To reduce the overhead of data copying, the contents of an object may be initialised upon creation to those of an image object for which the read capability is held. This provides a cheap mechanism for copying objects. If the object is not initialised, it contains zeroes.

  The access semantics of some of object types, however, mean that it is not sensible to use them as images when making *create_object* system calls. For example, synchronisation objects contain personal state information, such as which addresses processors are sleeping on. This is meaningless when duplicated.

  The details of the object to be used as an image are passed to the *create_object* system call in the *protection domain register* specified by the *image_dr* parameter.

Thus, it can be seen that the plethora of system calls provided by conventional operating systems such as UNIX can be replaced with just three, operating on a global shared address space containing various types of objects. The access semantics of these objects will now be explained in the following four sections.

3.3.1 Process Objects

Processes are represented in the *Cherub* architecture as objects in the Object Space. They have the following properties:

- Process objects are created and terminated explicitly using the *create_object* and *destroy_object* system calls, although process termination may also occur implicitly through exceptions.

- Processes are light-weight in that they have a relatively small amount of context information. This is limited to some scheduling information, a number of exception vectors and a set of general purpose, debugging and protection domain registers.

- In principle, *Cherub* supports any number of concurrently executing processes, although several hundred, one per processor, is optimal.

- All processes are scheduled and run concurrently by *Cherub*, unless they are blocked on memory accesses, have reached a breakpoint, or have been halted.

Process objects must also provide conventional process related operating system primitives such as signal and exception handling (*signal, kill*), timers (*alarm*), debugging (*ptrace*), protection (*setuid, seteuid, getuid, geteuid*) and run-time statistics (*utime*). These primitives are provided through *offsets* in the process object which have specific roles. The format of a process object is as follows:

Execution Time	
Countdown Timer	
Status Word	
Exception Vectors and Exception Flag	
General Purpose Registers (including PC, USP, ESP and SR)	
Breakpoint Registers	
Protection Domain Registers each containing	Global_Name
	Read_Capability
	Write_Capability
	Execution_Capability
	Access_Semantics
Unused	

The roles of these *offsets* are listed below. All are 64–bits wide.

- *Execution Time*

  This field shows how much processor time, in μS, has been spent running the process. This field equates with the UNIX *utime* system call.

- *Countdown Timer*

  This field is used to generate timed exceptions. When a non-zero value is written to this field, it is used to initialised a μS countdown timer. Writing zero to the field disables the timer and reading it gives the number of μS remaining. When this reaches zero, an exception is generated.

  This mechanism equates with the UNIX *alarm* system call.

- *Status Word*

  This field contains a number of bit flags which relate to the current execution status of the process. These include information such as whether the process is running or blocked on a data access. Most of these flags are read–only.

  One of the most important flags is the *stop flag.* This can be modified. When set, this prevents the process from being executed. This flag is set by default when a new process is created. This allows a parent process to construct its child's environment before it starts executing.

- *Exception Vectors and Exception Flag*

  The *exception vectors* contain the addresses of exception handler routines. There is one vector per exception type. By default, when a process is created all of its exception vectors are empty.

  The *exception flag* bitfield is used to send an exception to a process. There is one bit per exception type. When a particular bit is set, the appropriate exception is generated. This causes the process to suspend the normal execution of instructions and, if the appropriate *exception vector* is not 0, to execute the specified handler. If the *exception vector* was not defined, the process is terminated by an implicit *destroy_object* system call.

The *exception vectors* are prioritised. An exception cannot preempt a higher priority one. The exception handler must clear the appropriate bit in the *exception flag* before the process can accept a further exception of this, or a lesser, level. Blocked exceptions are queued by *Cherub* and issued in priority order.

Upon returning from an exception handler, normal instruction execution is resumed.

These mechanisms equate with the UNIX *signal* and *kill* system calls.

- *General Purpose Registers*

  These are the general purpose processor registers. These include the *program counter*, *user stack pointer*, *exception stack pointer* and *status register.* The processor has instructions which manipulate these registers directly.

- *Breakpoint Registers*

  These registers are used for debugging purposes. These contain the addresses which, when accessed by the process, generate exceptions. These are akin to the debugging registers of the Intel 80386 processor [Int86]. Due to the uniformity of the object space, these addresses can refer to either instructions or data. This mechanism equates with the UNIX *ptrace* system call.

- *Protection Domain Registers*

  Each process has a number of sets of protection domain registers which enable it to access the contents of objects. The processor has instructions which manipulate these registers directly. As domain registers equate with UNIX file descriptors, it is likely that 64 sets[2] will be sufficient. Each set contains five registers:

  - *Global_Name*

    This holds the *global_name* by which an object is referred to. If this field contains 0, the register set does not hold valid data.

  - *Read_Capability*

    This contains the read capability for the object (if any). If this field contains 0, this capability does not exist.

  - *Write_Capability*

    This contains the write capability for the object (if any). If this field contains 0, this capability does not exist.

  - *Execute_Capability*

    This contains the execute capability for the object (if any). If this field contains 0, this capability does not exist.

  - *Access_Semantics*

    This register holds the access semantics which the object provides.

[2]The number of file descriptors currently supported by SunOS 4.0.

No policy is enforced on the use of the 64 domain registers, but by convention the following associations are used:

Domain Register	Use
0	The system process object.
1	The process object itself.
2	The process's text object.
3	The process's data object.
4	The process's stack object.
5	A rendezvous object for signaling the process's termination.
6, 7 and 8	The equivalent of UNIX's stdin, stdout and stderr.
9, 10 and 11	Semaphore objects associated with the previous three registers.
12 to 63	Additional objects used by the process.

Domain register one refers to the process object itself. This equates with the UNIX *getpid* system call. Domain registers two, three and four refer respectively to the text, data and stack objects that will be used by the process. Domain register five refers to a rendezvous object which is used for synchronising with the parent upon termination. Domain registers six to eleven equate with UNIX's stdin, stdout and stderr and allow their unification with the *dup* and *pipe* system calls. Additional objects, if any, can be placed in the remaining registers.

When a process object is created, it has no stack. A memory object must be created separately for this purpose. Similarly, it is necessary to create a rendezvous object with which the child process can synchronise with its parent upon completion. The details of these objects must be loaded into the appropriate domain registers prior to process execution.

Object one in the Object Space is reserved for a special process object, called the System Process. This object represents the *Cherub* operating system and is the first process created when the system is booted. It, therefore, equates to the UNIX *init* process. The System Process is able to ignore all capability restrictions and its domain registers initially refer to objects one through twelve.

The System Process's *execution_time* field corresponds to the current time–of–day clock. Most processes possess the *read* capability for the System Process in domain register zero, thus enabling them to read the clock (*time*). Processes which also possess the System Process's *write* capability can set the clock (*stime*), reboot the operating system by issuing a *destroy_object* system call, or halt it by setting its *stop flag* (*reboot*).

It can be seen that process objects provide access to a wide range of process and system control mechanisms. Due to the multitude of dissimilar process related mechanisms provided by conventional operating system interfaces, processes are the most complex type of object in Cherub. *Although, the number of mechanisms provided by* Cherub *is not significantly lower then conventional systems, they are accessed in a unified manner through the process object.*

3.3.2 Memory Objects

Memory objects are contiguous regions of virtual memory composed of a number of hardware memory pages. The contents of memory objects survive machine reboots, thus unifying UNIX's concepts of persistent (*lseek, ioctl, read, write*) and non–persistent storage.

Cherub will employ a 256 byte page size — experiments with the Psyche operating system have shown this to be a good compromise. The page size determines the optimal size for shared data structure elements. Pages are allocated automatically on demand and continue to exist for the lifetime of the object, thus replacing UNIX's *brk* system call.

As explained in chapter two, current theory dictates that a DSM should only provide strong coherence — a read returns the last value written to a location — as it is the easiest mechanism to program. However, it is also believed that applications should be able to give the underlying computer architecture hints about intended future memory usage. The processors in *Cherub* provide four new machine instructions for this purpose.

- *busy_read(address)* and *busy_read_write(address)*

  These instructions tell the architecture that the program will probably access the data page containing *address* heavily in the near future. The instruction type corresponds with the nature of the intended data access.

- *idle(address)*

  This instruction tells the architecture that the program will probably not access the data page containing *address* in the near future.

- *finish(address)*

  This instruction tells the architecture that the program will probably not access the data page containing *address* again.

The instructions are advisory only; they do not guarantee exclusive memory access.

Memory objects combine efficient fine–grained coherent data sharing with persistent storage.

3.3.3 Sleep–Wakeup, Semaphores and Rendezvous Objects

These synchronisation objects allow a number of processes to coordinate their actions without the high overheads associated with polling. The three types of synchronisation objects to be supported were chosen because of their success in UNIX:

- Sleep–Wakeup

  Processes reading from an offset within this type of object are blocked. When a process writes to the object, all processes reading from that offset are unblocked, returning the written value. Writes to offsets from which no processes are reading are lost. This resembles the UNIX kernel's sleep–wakeup mechanism [Bac86].

- Semaphores

  This object is similar to the sleep–wakeup object except that each write will only unblock one process. Furthermore, writes are not lost when no processes are reading; they are simply stored until the next read. Data items will be read in the order that they were written. This resembles Dijkstra's P and V semaphores [Dij65].

- Rendezvous

  The access semantics of this object resemble those of the sleep–wakeup object except that a write to an offset from which no processes are reading will block until a read takes place. This resembles the UNIX *wait* system call [Bac86].

By combining these three simple mechanisms, more complicated primitives such as Mellor–Crummey and Scott's locks and barriers [Mel91] may be constructed.

It should be noted that synchronisations can be performed using memory objects alone. For example, the strict coherence provided by memory objects can support spin–lock synchronisation. However, this mechanism is wasteful of CPU time and in distributed systems incurs excessive interprocess communication. Synchronisation objects are, therefore, provided for the sake of efficiency.

Synchronisation objects provide access to simple, efficient, synchronisation primitives.

3.3.4 Hardware Objects

Hardware objects map onto the control registers of the hardware devices, such as terminals and tape drives, which are connected to the architecture. They can be used by processes to control these devices using memory mapped I/O. These objects equate with UNIX devices such as */dev/tty*. The allocation and internal structure of hardware objects are implementation dependent.

Hardware objects create a simple, unified interface to dissimilar devices.

3.4 Programming Cherub

Having defined *Cherub's* system call interface and the access semantics of its objects, it is now possible to show how the new architecture will be programmed. Two of the areas previously identified as involving major operating system issues are process management and data storage. Examining these makes it possible to demonstrate that *Cherub* is easy to program and that it involves lower overheads then conventional operating systems, thus allowing it to support finer grained processing.

3.4.1 Process Management

The *Cherub* process life cycle is illustrated in figure 3.1. The following stages are involved in setting up, synchronising with, and cleaning up after, a new process:

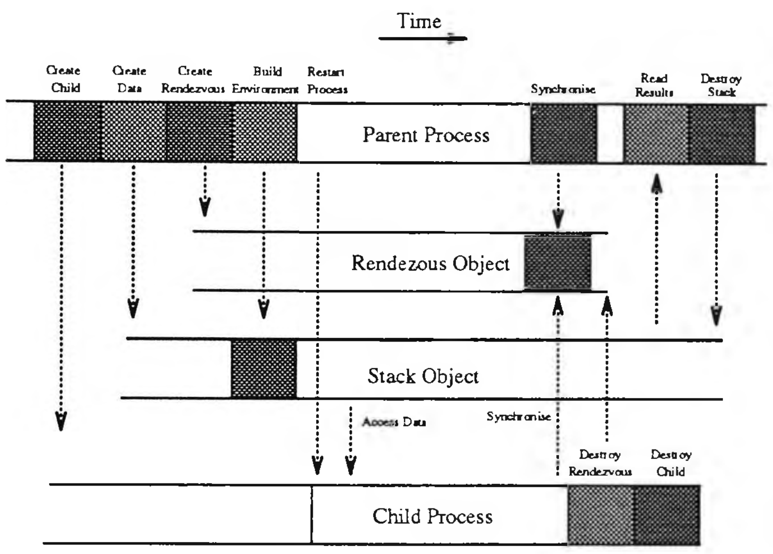


Figure 3.1: The Cherub Process Life Cycle

1. Create Child Process Object

   The parent process creates the child process object using the *create_object* system call.

2. Create Stack Object

   A memory object must be created using the *create_object* system call. This will be the new process's stack object. When the child process terminates, this object will be used to pass its results back to the parent.

3. Create Rendezvous Object

   A rendezvous object must be created using the *create_object* system call to enable the child and parent processes to synchronise upon termination.

4. Build Child Process's Environment On Stack

   The arguments and environment needed by the child process are first written into its stack object. Next, the child process's first twelve domain registers are set up. Finally, the address of the process object is placed in a well-known register of the new process.

5. Restart Process

   The parent process issues *finish* instructions on the pages of the child's process and stack objects. The process object is then started by resetting its *stop flag*.

6. Load Text And Data Pages

   The child process issues *busy_read_write* instructions on the pages of its process, stack and rendezvous objects. The process then demand loads the pages it needs for execution from its text and data objects. Similar overheads will also be incurred by a non-parallel function.

7. Synchronisation

   Once the child process has completed execution, a synchronisation operation must be performed to inform the parent. The address of the results is passed to the parent process using the rendezvous object. The parent process is then able to read the results from the child's stack object.

8. Termination

   Once the results have been transferred, the child process issues *destroy_object* system calls to terminate its synchronisation object and itself. The parent process is responsible for terminating the stack object once it has finished with the results. In this way the child process pays most of the overhead of the cleanup operation, not the parent.

The programming overhead of this mechanism can be significantly reduced by the provision of library calls such as:

- *pid = create_process(execution_address, capability_list, argument_list, environment_list)*

  This library call creates a process, a stack and a rendezvous object, sets up the process's domain registers, builds the environment on the stack using the argument and environment lists and starts the process at the given execution address. It returns a process identifier which can be used to refer to the process. This is equivalent to a unified UNIX *fork* and *exec* system call.

- *synchronise_process(pid)*

  This library call performs a read on a process's rendezvous object. This is equivalent to the UNIX *wait* system call.

- *terminate_process()*

  This library call first performs a write to the issuing process's rendezvous object. It then destroys the process object, along with its synchronisation object. The parent process is responsible for destroying the stack object once it has finished with its contents. This is similar to the UNIX *exit* system call.

These library routines provide functionality similar to their UNIX system call counterparts, but are based upon primitives which are simple enough to be implemented in hardware. Alternatively, process creation may be automatically performed by intelligent compilers with parallelisation constructs, thus further simplifying the programming overhead of using the machine.

As was shown in chapter one, if the intertask communication overheads associated with the process life cycle can be performed in under 7,000 instructions, then a task granularity of around 10,000 instructions is optimal. Figure 3.2 illustrates the maximum potential speedup which may be obtained when a sequential task is decomposed into subtasks of this granularity, all of which can be executed by processes in parallel.

To determine whether the desired level of granularity is feasible, the implementation of these mechanisms must be examined. This will be done in the next chapter.

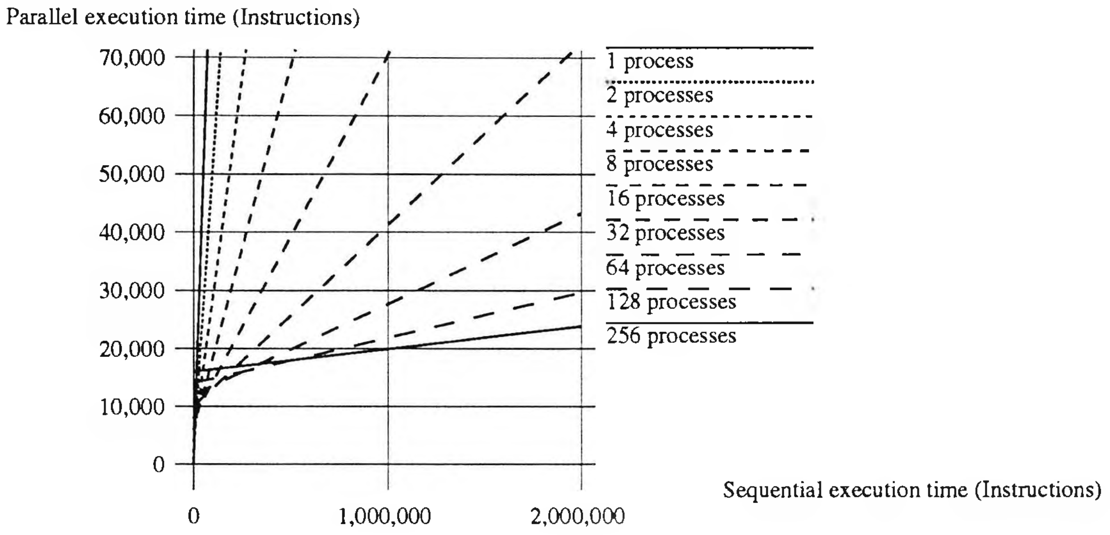


Figure 3.2: The Time Taken To Execute A Program In Parallel On Cherub

3.4.2 Data Storage

All data storage in *Cherub* uses the memory object type:

- Objects are created and terminated using *create_object* and *destroy_object* system calls. They are persistent.

- Upon creation, an object is named with a *global_name* assigned by *Cherub*. This corresponds to its start location in the Object Space. If required, user-level personalised name servers can be implemented to map hierarchical names onto the flat ones employed by *Cherub*.

- The contents of an object are accessed using conventional read and write machine instructions. The strong coherence semantics of the object ensures that data written to it is immediately visible to other processes.

  Shared code libraries may be constructed using the indirection vectors technique described in chapter two.

- An object can be accessed by any number of processes which possess its capability, thus allowing parameter passing by reference.

This is clearly a simpler and more efficient storage mechanism then UNIX's explicit, coarse-grained, *read* and *write* system calls. Consequently, it is much better suited for implementation in hardware.

3.5 Applications Suitable for Cherub

Finally, having seen how Cherub would be programmed, it is possible to suggest the properties which would make an application particularly well suited to the new architecture:

- The application should be able to make use of concurrently executing medium-grained processes, each consisting of around 10,000 instructions. Ideally no more then several hundred of these should be running concurrently, one per processor.
- The application should use the shared variable parallel programming paradigm. The shared data structures should have the following properties:
  - The data structures should be shared for read and write, thus requiring *data coherency.*
  - Synchronisation by record locking should be necessary when modifying shared data structures to maintain *data consistency.*
  - False data sharing during record locking should be unacceptable. Hence, only one record should be stored in each page. This is the case in real-time applications. Hence the page size should be small — around 256 bytes. It should be undesirable to use an architecture with a larger page size. This could be because either:
    - The database contains a large number of records; or
    - A fast record transfer time is very important, as is the case in many real-time applications.

A non-trivial real application which has these properties, an airborne early warning system, is examined in appendix B. This demonstrates how *Cherub* can help to provide an efficient and elegant solution for this application.

3.6 Conclusion

In this chapter we have introduced the *Cherub* system call interface, which is programmed through a SSAS called the Object Space. It was decided that only three main operating system mechanisms are required:

- Conventional UNIX operating system mechanisms such as processes, persistent and non-persistent storage and inter-process communication are unified as six types of object within the Object Space: process, memory, sleep-wakeup, semaphore, rendezvous and hardware.
- Object protection is provided by password-like capabilities. The set of capabilities possessed by a process replaces the UNIX concept of users and groups.
- A flat naming scheme, based on the start addresses of objects within the Object Space, is employed. It is expected that private user name servers will be employed to map hierarchical UNIX-like file names into object names and capabilities.

It is asserted that these mechanisms are simple enough to be implemented in hardware, thus reducing intertask communication latencies to below 7,000 machine instructions and resulting in an optimal task granularity of 10,000 instructions. This will enable *Cherub* to provide a degree of parallelism which is useful for a significant range of applications, while maintaining a natural data sharing mechanism which simplifies its usage.

The next chapter will examine this assertion, showing how *Cherub* can be implemented in hardware and estimating its performance.

Chapter 4

Implementing the Cherub Architecture

4.1 Introduction

In chapter three the interface to a new distributed computer architecture, *Cherub*, was defined. This will be programmed through a single shared address space, with the intention that many conventional inter–task communication mechanisms will be simplified to the extent that they can be implemented in hardware. It is hoped that this will reduce the latency of the process life–cycle to below 7,000 machine instructions. This chapter will determine whether this is possible.

It is important to appreciate that good implementation mechanisms are just as critical as models of computation when designing parallel architectures; almost any model of computation can be implemented on a machine if the right mechanisms are employed. In Chapter three it was stated that *Cherub* should provide the following:

- Support for several hundred concurrent light–weight processes.
- A granularity of data sharing of around 256 bytes.
- A medium granularity of processing; the total time spent setting up, providing data for, synchronising and scheduling a 10,000 instruction process should be similar to that needed to execute about 7,000 instructions.
- A multiple access policy, to a globally shared address space.

These properties can only be provided by combining high performance state–of–the–art hardware with low overhead mechanisms. This chapter is, therefore, divided into three parts. First the underlying hardware needed to support *Cherub* is determined. Secondly, mechanisms are designed which implement *Cherub* on that hardware. Lastly, the performance of the mechanisms is considered.

4.2 The Cherub Hardware

Before mechanisms for the implementation of the *Cherub* Object Space can be suggested, the architecture's hardware must be defined. Fortunately, many MIMD (Multiple Instruction, Multiple Data) [Fly66] multicomputer architectures conform logically, if not physically, to a single model: a number of processors, each with one or more layers of local memory, connected by a network to layers of shared memory and disk storage. This logical organisation is illustrated in figure 4.1.

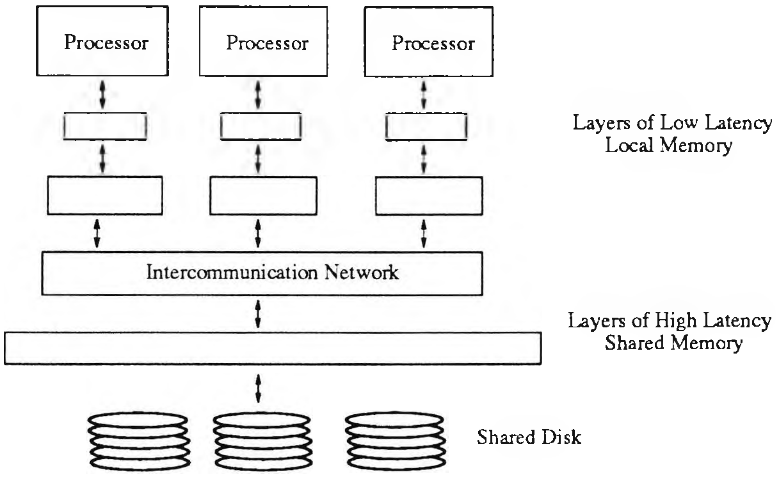

Processor
Processor
Processor
Layers of Low Latency Local Memory
Intercommunication Network
Layers of High Latency Shared Memory
Shared Disk

Figure 4.1: Cherub's Logical Hardware Organisation

The main factors affecting the performance of such an architecture are:

- The number and speed of the processors;
- The amount and speed of the local memory;
- The latency, bandwidth and scalability of the network; and
- The amount and speed of the shared memory.

From a philosophical standpoint, as it may take several years to implement *Cherub*, it is important to employ mechanisms which are designed not for existing but rather future hardware[1]. On the other hand, as it is virtually impossible to accurately predict the speed or direction of technological advance, there is little point in trying to look too far ahead. Consequently, the proposed implementation of *Cherub* assumes hardware which should be available within the decade. It is, therefore, necessary to predict the capabilities of such hardware:

- Storage

  In 1990, Hennessy and Patterson [Hen90] observed three trends in memory hierarchies:

[1]This process of designing with future technology in mind is sometimes called Technology Intercept Planning.

- DRAM-Growth Rule:
  Density increases at about 60% per year, roughly quadrupling in 3 years.
- Disk-Growth Rule:
  Density increases at about 25% per year, roughly doubling in 3 years.
- Address-Consumption Rule:
  The memory needed by the average program grows by a factor of 1.5 to 2 per year.

This has several interesting implications for *Cherub*.

- Silicon Storage
  If the DRAM–Growth trend observed by Hennessy and Patterson continues, one Giga-bit silicon memory devices will exist within the decade[2]. As the cost of storage is typically inversely proportional to its density, these will cost under $25 each[3].
  Compared to its cost, however, the access time of DRAM is dropping relatively slowly, about 7% per year [Hen90]. As DRAMs with 60 ns access times are now starting to appear, this implies that at the end of the decade, the average DRAM access time will be no better then 40 ns.
  Two recent development in silicon–based storage devices include Flash Electrically Erasable and Programmable ROM (Flash–EEPROM) and Ferroelectric RAM (FRAM).
  * Flash–EEPROM [Mas91b, Hes90, Lah90] is reprogrammable nonvolatile memory, a blending of both EPROM and EEPROM. It has a EPROM–like density, EEPROM–like reprogrammability and an access time of around 120 ns. Flash memory has two main disadvantages. Firstly it has a limited write life–time of around 10,000 cycles. Secondly, it has to be erased in sectors. These make it more useful as a silicon disk then as a replacement for DRAM. It is currently possible to house four to sixteen megabytes of flash memory in a credit–card form. In fact, as removable media, these card–sized modules already exceed the storage capacities of conventional floppy disks.
  * Ferroelectric RAMs [Moa90, Gna89] are similar to conventional DRAMs in design, but use newly developed ferroelectric cells to provide nonvolatility and very high switching speeds (typically around 1 ns). They offer the potential to build nonvolatile memories with the speed of static RAMs and the density and cost of dynamic RAMs. They have a life–time of at least 10^{10} write cycles

  If FRAM is able to achieve a DRAM–like capacity within the decade, it will be desirable for *Cherub* to employ it instead of SRAM and DRAM. However, it is doubtful whether the technology will mature this quickly.

[2]Hitachi's electron beam based lithography system, the HL-700F, is currently able to draw 0.1μm lines at a rate of one 4 inch wafer per hour. Hitachi have simulated 0.1 micron devices and are convinced of the feasibility of Giga-bit memory chips [hit90].

[3]DRAM currently retails for around $3500 per Giga-bit of 70 ns memory. A 510 Mbyte magnetic disk can be purchased for $979, around $245 per Giga-bit.

It is important to realise that in the near future a single DRAM chip could provide 128 Mbytes of storage. Similarly, a large 16 Mbyte SRAM processor secondary cache might only require one chip[4]. It is, therefore, assumed that *Cherub* will have several Gbytes of DRAM storage and its processors' secondary caches will contain at least 16 Mbytes of SRAM. In comparison, primary caches are relatively small as they constitute a major processor yield hazard. A typical future processor's primary cache might only be 512 Kbytes in size.

– Magnetic Storage

When Hennessy and Patterson made their memory hierarchy predictions it appeared that silicon-based storage would eventually become cheaper then magnetic storage. In addition, it appeared that optical storage might a provide an even cheaper, if semantically different, backing storage medium [Pin89].

These predictions now look doubtful; recently significant breakthroughs have been made in magnetic disk technology. Oxide magnetic heads and recording media have been replaced by metallic magnetic materials, allowing a recording density approaching 100 Mbit per square inch. Furthermore, newly evolving techniques, such as vertical storage [Ron90, Die89], could potentially increase the capacity of magnetic disks up to 30 times that of the present generation. Meanwhile optical storage density has been limited by the physical size of the read/write laser.

As processors increase in performance, the high latency and low bandwidth of magnetic storage becomes more marked. Techniques such as RAID disk arrays [Pat88, And91a, Ols89] and interleaving [Kim86] are being investigated to increase the bandwidth of magnetic disks. Disk arrays have also been used to provide fault tolerance against drive or media failure [Gib89]. Smaller disks, faster rotation speeds, multiple read/write heads per track [Mit89] and intelligent disk heuristics [Ric89, Sel90, Kin90] are being investigated to improve the data access latency. However, it still remains a significant bottleneck.

If the current trends in disk storage continue, it is probable that Cherub *will employ a RAID-like disk array providing Tbytes of storage. This array is likely to have a latency of around 10ms, a transfer rate of several hundred Mbytes per second, and be accessed on a sector basis of at least 16 Kbytes in size.*

The predicted properties of the *Cherub* memory hierarchy are listed in table 4.1. It is necessary to decide which of these technologies should be used as local memory and which should be shared. For reasons of performance, it is necessary to achieve a reasonable balance between the access time of a memory block and the time taken to transfer it. A tradeoff, therefore, exists when implementing data coherence at a given level in the memory hierarchy:

– Due to false data sharing, the lower in the memory hierarchy coherence is performed, the greater the number of invalidations that have to be percolated up to the higher levels.

[4]This can be deduced from the assumption that 1 Git-DRAMs will exist. SRAM is between 4 and 6 times more expensive in transistors then DRAM. Therefore, 1 Gbit of DRAM $\approx$ 16 Mbytes of SRAM, assuming that cache line tags incur a 25 percent transistor overhead.

- Alternatively, the higher in the memory hierarchy coherence is performed, the greater its latency is in relation to the time needed to transfer a memory block. In addition, the smaller blocks require more directory table entries, resulting in increased memory wastage.

Memory Technology	Cycle Time	Predicted Block Size	Capacity
Primary Cache	few ns	256 byte lines	Kbytes
Secondary Cache	10s ns	256 byte lines	Mbytes
DRAM	1,00s ns	16 Kbyte pages	Gbytes
Disk	1,000s ns	16 Kbyte blocks	Tbytes

Table 4.1: Predicted Properties of Cherub Memory Hierarchy

In chapter three it was stated that Cherub *should support a granularity of data sharing of around 256 bytes. This implies that the sharing, and hence the data coherence mechanism, should occur in either the primary or the secondary cache. However, in order to provide data coherence, lists must be maintained of the pages held on each processor. Due to its relatively low yield, it is too expensive to hold these tables in primary cache. Therefore, data coherence will be implemented in the secondary cache.*

- Processors

  In 1990, Borg et al [Bor90] made some predictions regarding workstations of the near future. Adapting them slightly for *Cherub* we get:

  - Processors will have a RISC architecture and two levels of cache;
  - The primary cache cycle time will be 2 ns;
  - The additional time to go to the second level cache will be 16 cycles (32 ns);
  - The main memory (DRAM) latency will be 140 cycles (280 ns); and
  - The main memory transfer rate will be 16 bytes every 10 ns.

  These predictions give some idea of the performance the *Cherub* global shared memory must achieve to create a balanced memory hierarchy.

  When silicon storage was discussed it was shown, by extrapolating Hennessy and Patterson's memory hierarchy trends, that a Borg-like processor might reasonably have 512 Kbytes of on-chip primary cache and 16 Mbytes of off-chip secondary cache. Studies of such very large virtually addressed caches [Prz90, Bug90, Sho88, Wan89] generally agree that it is most cost effective to make primary caches direct mapped and second level caches set associative. A write-through policy is usually adopted in the primary cache to maintain coherence and a write-back policy in the secondary cache to reduce memory traffic. The cache miss ratios reported in the literature depend largely upon the behaviour of applications simulated, but total cache miss ratios below 1.0×10^{-3} are often quoted [Bug90, Sho88].

Assuming that the primary cache miss ratio is 0.1, the secondary cache miss ratio is 0.01 [Bug90] and the main memory is large enough to avoid disk accesses, an average processor cycle will take around 6 ns:

Memory Level	Hit Ratio	Cycle Time (ns)	Average Cycle Time (ns) (Hit Ratio × Cycle Time)
Primary Cache	$1 - 0.1$	2	1.8
Secondary Cache	$0.1 \times (1 - 0.01)$	32	3.2
Main Memory	0.1×0.01	$280 + 256 \div 16 \times 10$	0.4
			5.4

Thus, ignoring overheads, a 10,000 instruction *Cherub* process will typically take under $60\mu S$ to execute.

Given that *Cherub* can have at least 200 concurrent tasks, it is implied that the architecture, should have at least that number of processors. This suggests that network addresses should have at least eight bits.

In order to support process objects, as defined in the previous chapter, the processor should have at least the following registers:

- 64 general purpose registers (64 bits each)
- 64 protection domain registers (5 × 64 bits each)
- 8 breakpoint registers (64 bits each)

A *Cherub* processor accesses its primary and second level caches using an on–processor intelligent cache controller. This is responsible for locating lines of data, if they are in the caches, and if not, communicating with the cache controllers of other processors to find them. The controllers associate unique request identification numbers with the data requests they issue to other controllers so that they can match the requests to the incoming replies.

- Network

  The exact form of the network joining the processors is largely irrelevant. However, it must be able to support communication among many processors, which implies that it supports point–to–point rather then broadcast communication.

  On initial inspection, a circuit switched communications network appears to be well suited to *Cherub's* needs. This mechanism allows cheap, exclusive, two–way communication between cache controllers.

  The implications of using other types of networks are examined in chapter five.

To summarise, it has been predicted that if *Cherub* is constructed within the decade, it could reasonably consist of the hardware components illustrated in figure 4.2:

- At least 200 high performance RISC processors with a 6 ns instruction cycle.
- Each processor could have a 512 Kbyte local on–chip primary SRAM cache, organised as 256 byte lines.

- Each processor could have a 16 Mbyte local secondary SRAM cache, organised as 256 byte lines. Data coherence will be provided at this level.
- A scalable point-to-point communications network.
- Around a Gbyte of DRAM storage.
- Around a Tbyte of optical or magnetic disk storage.

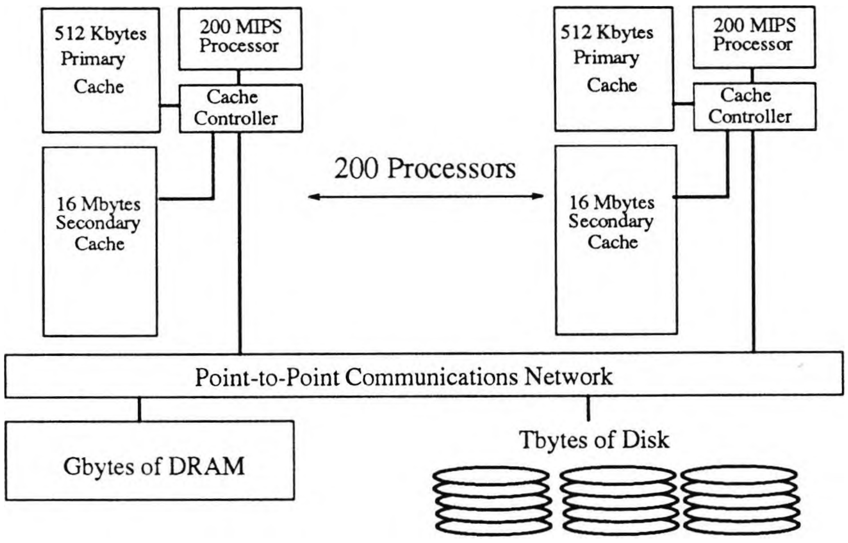


Figure 4.2: Cherub's Physical Organisation

Having predicted the *Cherub* hardware, it is now possible to examine mechanisms for implementing the Object Space upon it.

4.3 Implementing the Object Space

Chapter three specified that the 64 address bit *Cherub* object space is divided into ranges of virtual address which are allocated to individual objects. Furthermore, all objects are the same size, are named by their unique *global_name* and their contents are addressed using *offsets*.

As all objects are the same size, it is possible to use the upper n bits of a virtual address, where $0 < n < 64$, to denote the *global_name* of an object and the bottom $64 - n$ bits to give the *offset* within it, thus simplifying the design of the addressing hardware. This allows any combination of 2^n objects, each containing 2^{64-n} bytes of address space. A compromise is clearly necessary:

- A small value of n gives large objects, thus simplifying the management of large data structures. However the system will then have few objects and much of the virtual address space will be wasted through internal fragmentation.

Files in the the UNIX operating system are effectively limited to four Gbytes in size, given that the file size field in the inode is 32 bits. This has proved to be a severe limitation when handling large databases [Rob92], where at least 40 bits are often required.

- A large value of n gives small objects, thus making more efficient use of the virtual address space. However, the small object size will complicate the management of large data structures.

  By default, the UNIX operating system's *newfs* command allocates an inode structure per 2,048 bytes of disk space. Although, often this number of inodes proves to be generous, it is not unusual for systems such as news servers to have several million files.

It is, therefore, asserted that a reasonable balance is to have 2^{24} objects, each 40 address bits in length. This creates 16,777,216 objects, each containing 1 Tbyte of virtual address space.

Chapter three specified that objects are persistent[5]. They must, therefore, be stored on disk or some other permanent storage medium. It should be noted that an object can potentially be larger then the capacity of any single disk drive currently available. Consequently, it must be possible to store an object over multiple disks, so that its size is not limited by the physical capacity of any one media component. A logical to physical mapping scheme is, therefore, employed to convert logical disk addresses into the appropriate physical block and devices. This also allows techniques such as sparse storage, disk striping [Kim86] and RAID [Pat88, And91a, Ols89] to be employed. The cache controllers contain tables which enable them to perform this mapping. Logical to physical disk mapping is illustrated in figure 4.3.

The disk is logically divided into 1 Kbyte blocks, although the disk drives will probably be physically accessed using much larger, say 16 Kbyte, blocks. This small logical block size is very important since it reduces the latency of block transfers and zero-fills. It also helps reduce internal disk block fragmentation, albeit at the cost of larger disk block indexes. This will be explained later.

The layout of the logical disk address space is illustrated in figure 4.4. It is mainly divided into two parts — the free lists and process run-queues will be explained later:

- Object Descriptors

  The information about objects is held on disk in object descriptor structures. These are 256 bytes long, the granularity of data sharing, thus eliminating false data sharing. They have the following fields:

[5]Object persistence is only addressed in this thesis in a rudimentary manner since it is not considered that true persistence — fault tolerance — is necessary. The design only guarantees that the contents of memory objects will survive reboots if the system is shut down cleanly.

Cherub's ability to make inexpensive copies of memory objects, provides a convenient checkpointing mechanism. In his thesis, Wilkinson [Wil93] shows that it is possible to provide fault tolerance by regularly checkpointing objects and monitoring the access dependencies among them. If fault tolerance is required, therefore, the implementation must be redesigned to support the copying of objects other than memory.

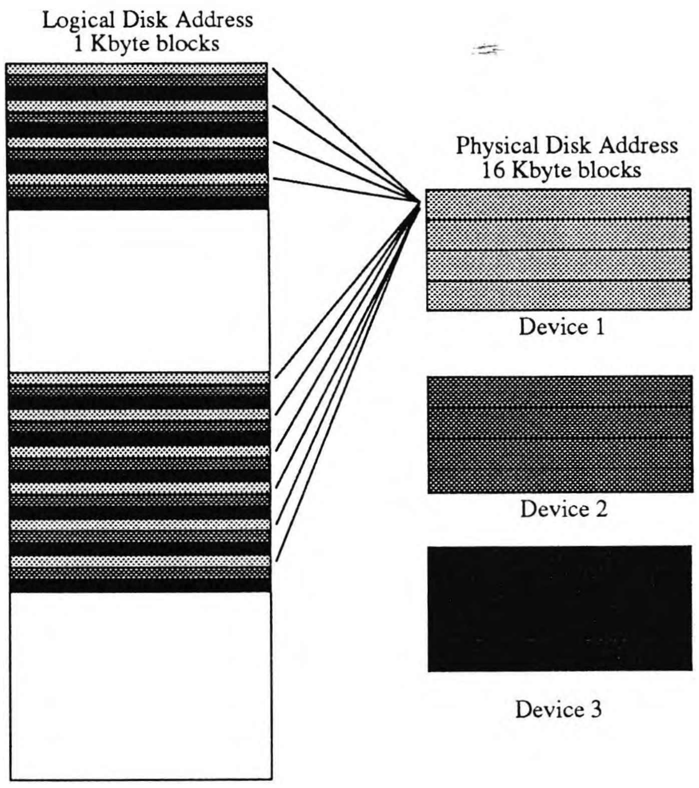


Figure 4.3: Logical to Physical Disk Address Mapping for Sparse and Striped Storage

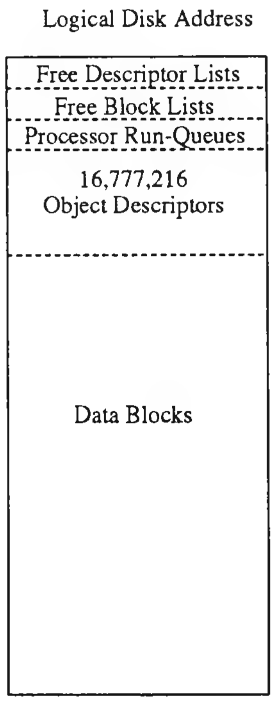


Figure 4.4: Logical Disk Layout

Field	Size (Bytes)
Valid Bit Access Semantics	1
Read Capability	8
Write Capability	8
Execute Capability	8
Creation Time	4
Last Modified Time	4
Last Accessed Time	4
Object Dependent	219

It should be noted that this structure limits capabilities to 64 bits[6] and restricts the number of possible object access semantics to 128[7], leaving 219 bytes for object dependent data. These sizes are thought to be sufficient.

For speed, the Object Space's descriptors are held in a preallocated four Gbyte table in the logical disk address space; the actual physical storage devoted to this table, however, is likely to be small as the disk address mapping scheme allows the logical disk address space to be sparse. An object's *global_name* is used as an index into the table to obtain its descriptor.

- Data Blocks

  The remainder of the logical disk address space is divided into one Kbyte blocks. These are used to hold object indexes and data. This will be explained further when memory objects are discussed. The logical block size chosen compromises between minimising the data block transfer and zero fill latencies, reducing internal fragmentation, and minimising the size and depth of the data block indexes required.

Object descriptors and data blocks are allocated from linked lists of free disk blocks, each one Kbyte block containing up to 128 64-bit free descriptor or data block numbers. A large number of free lists are maintained to reduce contention; a processor is initially assigned its own free object and data block lists. When these are exhausted, it allocates from the other processors' lists.

The mechanisms used to implement the different object access semantics are described in the following four sections.

4.3.1 Process Objects

Process objects should be assigned to processors selectively so as to maximise the throughput of the system, while minimising the overheads of creating and terminating individual processes.

[6]An attempt to systematically break a 64-bit capability will, on average, take approximately 30 thousand years, assuming each attempt only takes 100 ns!

[7]If more then 128 object types are required, Cherub's design goal of simplicity through unification has probably been compromised.

Many conventional process placement algorithms work by load balan-ing [Jac89, Win89, Ber91]. Such schemes attempt to distribute processes so that currently executable processes are spread evenly among the processors in the system. In addition, some algorithms detect when load imbalances occur and compensate by migrating running processes to less loaded processors [Jul87, Bar86, Che88a, Dou89].

Cherub performs neither load balancing or process migration; it is asserted that the overheads of such mechanisms are not cost effective when the average process life–time and the number of processors involved are considered. The justification for this is as follows:

- Load Balancing

  The major problem when balancing load in *Cherub* is that as the number of instructions executed by a process is typically very small, processor loads change frequently. Most load balancing mechanisms reported in the literature such as bidding [Sta84], drafting [Ni 85] and gradienting [Lin86] are not effective in such situations. This is because they are so complex that by the time a placement decision has been made, the load on the receiving processor may have changed significantly. Furthermore, the performance advantage that a complex scheduling algorithm may have over a simpler one when employed on small systems is often significantly reduced on larger systems [Gri91].

  Cherub employs a random process placement scheme. Each processor is assigned a unique doubly linked list in the disk address space, which represents its run–queue. When a new process object is created, its address is placed at the head of a randomly selected queue. Busy processors time–slice between their executable processes. As the average process life–time in *Cherub* is expected to be short, it is reasonable to employ relatively long time–slices, such as 12,000 instructions, which is just over the average process life–time. Idle processors check their run–queues periodically, say every 100 instructions, for new work. This polling frequency is an important component of the latency of starting a new process.

  This random placement scheme is intended to reduce contention by avoiding a single global process run–queue. Experiments have shown that such random placement [Eag84] often performs well compared with more complex strategies when small processes are involved. Furthermore, when load imbalances do occur, *Cherub's* processes are so short–lived that they should not persist.

- Process Migration

  Another of *Cherub's* disadvantages is that process migration is far less profitable then in coarser grained systems. This is because, even on a heavily loaded fine-grained system, the time taken to migrate a process and its cache context is likely to greatly its life–time. Process migration, therefore, is not supported in the implementation.

A process object descriptor is very similar to that of a memory object, except that, since a complete process register file can be held in four 1 Kbyte logical disk blocks, only four data block pointers are required. In addition, as COW sharing of process objects is not supported, the level of indirection is not necessary. A process object descriptor is illustrated in figure 4.5.

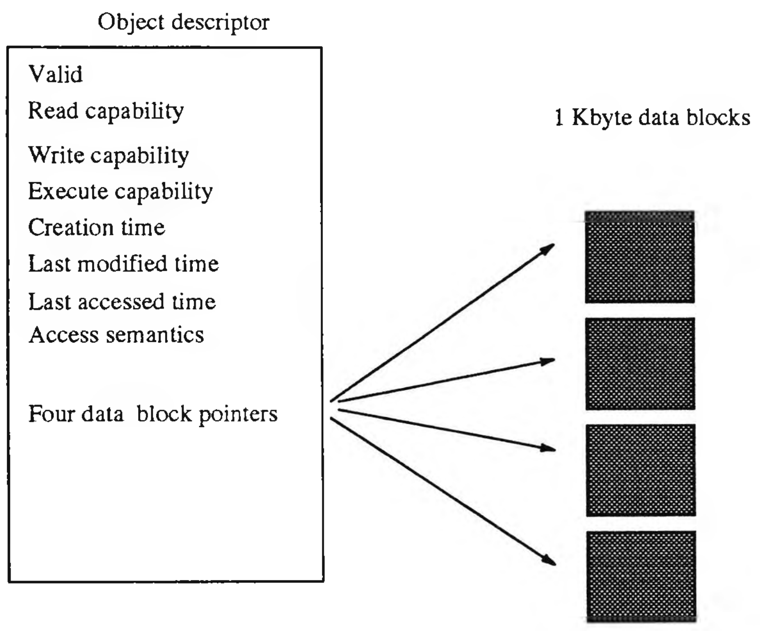


Figure 4.5: Process Object Representation on Disk

The contents of the process object data block, and hence the process register file, are as follows:

64 domain registers	×	40 bytes each	=	2,560 bytes	
64 general purpose 64–bit registers	×	8 bytes each	=	512 bytes	
8 breakpoint registers	×	8 bytes each	=	64 bytes	
8 other registers	×	8 bytes each	=	64 bytes	
				3,200 bytes	(4 disk blocks)

For consistency with the other types of object, the cache controller allows a process object's remaining address space to be accessed, but it always reads as zeroes.

The cache controllers translate accesses to certain process object addresses into special actions. For instance, setting a process's *stop flag* causes it to be removed from its processor's run–queue. Similarly, setting the System Process's *stop flag* results in all of the processors' run–queues being cleared. Alternatively, destroying the System Process object causes the run–queues to be cleared and the System Process object to be restarted.

4.3.2 Memory Objects

Chapter three specified that memory objects have the following properties:

- **They are are large, potentially sparse, regions of persistent storage.**

  As was explained in section 4.3, all objects will be held on disk to provide persistence. A memory object's data blocks are indexed, thus allowing common sparse objects to be represented efficiently, while still supporting rare non-sparse objects. Typical examples of memory objects are illustrated in figure 4.6.

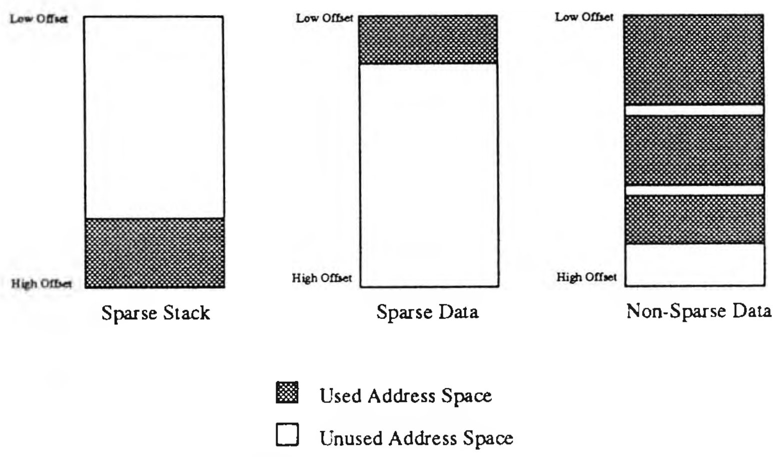


Figure 4.6: Examples of Typical Memory Objects

- **They can be used to hold both program data and stacks efficiently.**

  To enable memory objects to contain both program data and stacks, it must be possible to use both ends of their *offset* range efficiently. Therefore, their indexes are balanced as illustrated in figure 4.7. Only a single indirection is required to access data in the first and last eight Kbytes of an object, although up to five levels of indirection are needed to access data in its middle. It should be noted that, as a result of this balanced structure, some of the pointers in the final quadruple indirection index are wasted.

- **It is possible to duplicate their contents cheaply.**

  Efficient memory object duplication is implemented through copy on write. The memory object indexes point to data block entries which contain the disk addresses of the data blocks holding their contents. These allow unmodified data blocks to be shared between one or more related objects. As an example, assume that object B in figure 4.8 has been created using object A as an image. Each block's COW counter indicates the number of objects which are sharing it; in this case they will initially contain the value two. While a data block remains unchanged, it can be shared between the objects. If a data block is modified, a copy is made of it, a new data block entry is created for it and the original block's COW counter is decremented.

4.3.3 Synchronisation Objects

Synchronisation object descriptors are similar to those of process objects, except that they contain pointers to two linked lists, sleep and wakeup. This is illustrated in figure 4.9.

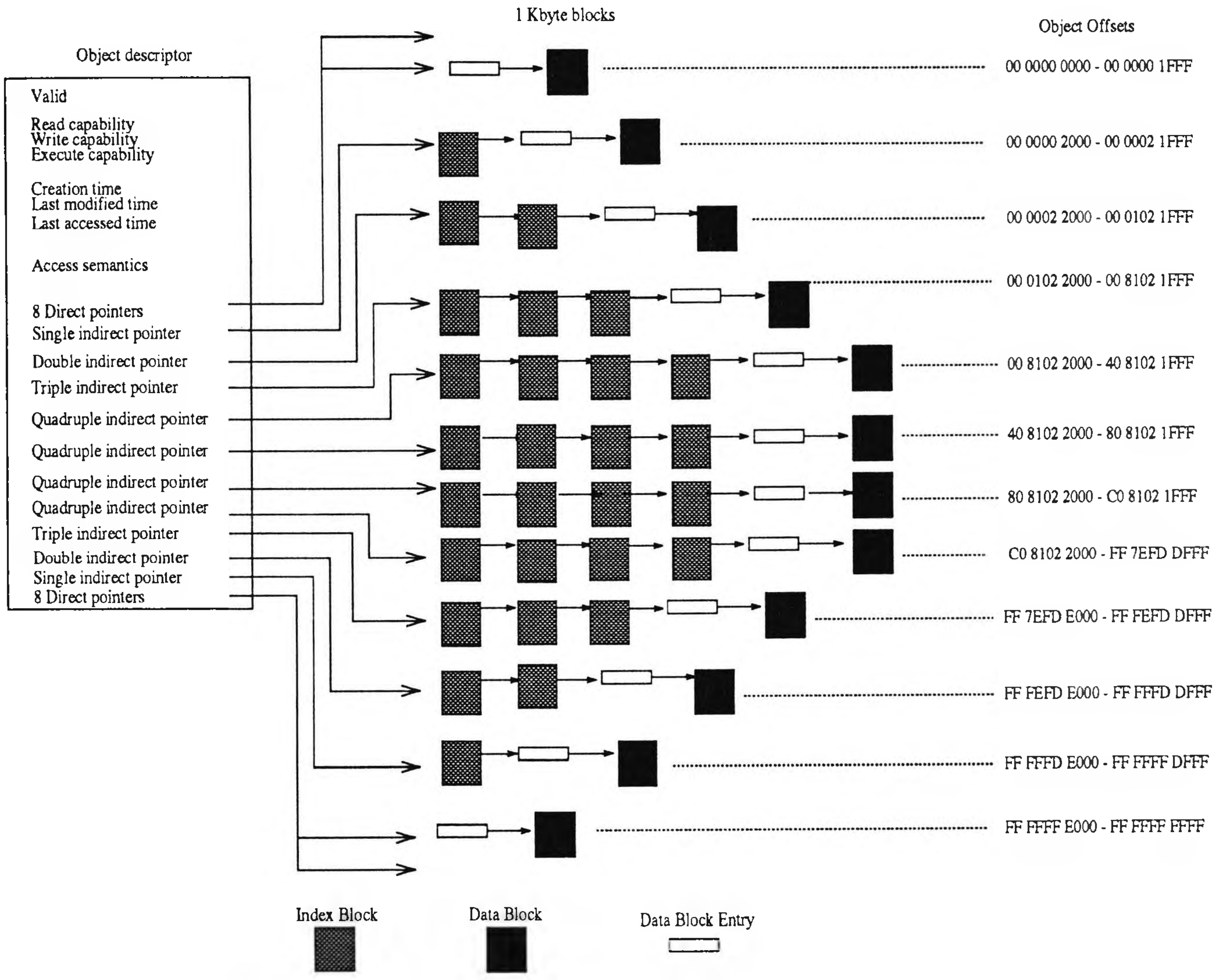

1 Kbyte blocks
Object Offsets
Object descriptor
Valid
Read capability
Write capability
Execute capability
Creation time
Last modified time
Last accessed time
Access semantics
8 Direct pointers
Single indirect pointer
Double indirect pointer
Triple indirect pointer
Quadruple indirect pointer
Quadruple indirect pointer
Quadruple indirect pointer
Quadruple indirect pointer
Triple indirect pointer
Double indirect pointer
Single indirect pointer
8 Direct pointers
00 0000 0000 - 00 0000 1FFF
00 0000 2000 - 00 0002 1FFF
00 0002 2000 - 00 0102 1FFF
00 0102 2000 - 00 8102 1FFF
00 8102 2000 - 40 8102 1FFF
40 8102 2000 - 80 8102 1FFF
80 8102 2000 - C0 8102 1FFF
C0 8102 2000 - FF 7EFD DFFF
FF 7EFD E000 - FF FEFD DFFF
FF FEFD E000 - FF FFFD DFFF
FF FFFD E000 - FF FFFF DFFF
FF FFFF E000 - FF FFFF FFFF
Index Block
Data Block
Data Block Entry

Figure 4.7: Memory Object Representation on Disk

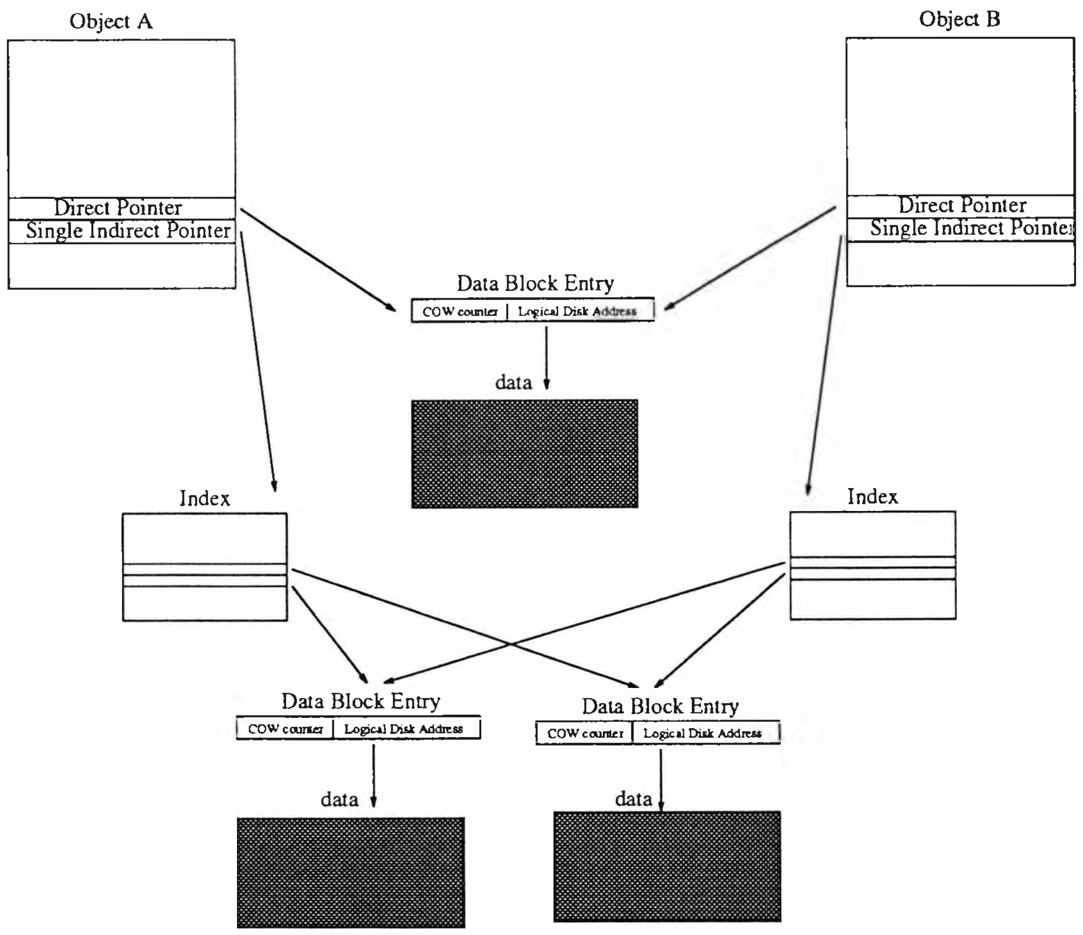


Figure 4.8: Implementing Copy On Write In Memory Objects

The use of these lists depends upon the exact type of synchronisation supported:

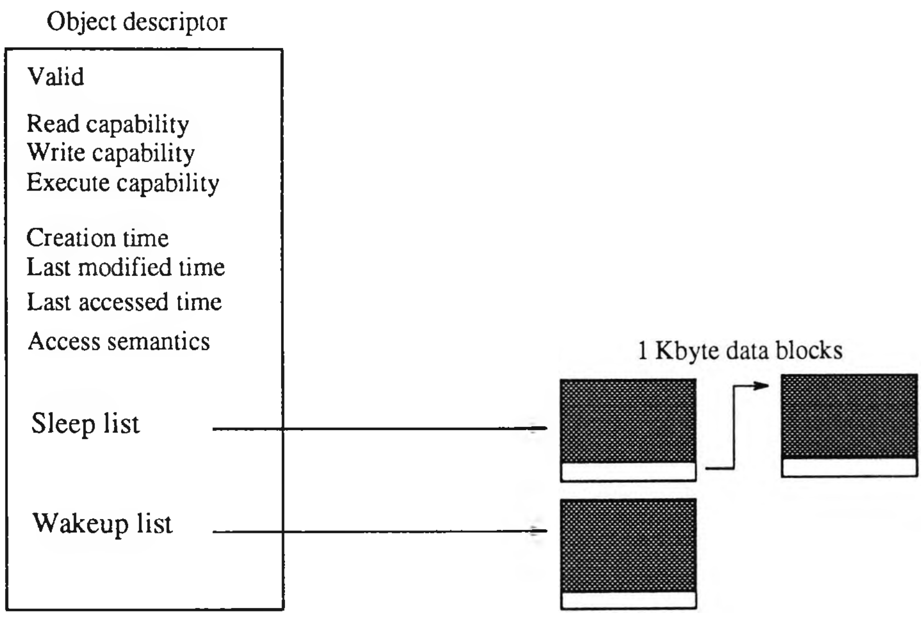


Figure 4.9: Synchronisation Object Representation on Disk

- Sleep-wakeup

  Sleep-wakeup objects only use the sleep list. When a process reads from one of these objects, its processor's number and the address accessed are added to the head of the sleep list. The process is blocked by being removed from its processor's run-queue.

  When a process writes to a sleep-wakeup object, the address written is compared with the items in the sleep list. Each process blocked on the address is removed from the list. The processes are placed back on their processor's run-queue, their reads giving the stored value.

- Semaphores

  When a process reads from a semaphore object, the address accessed is compared with those in the wakeup list. If a matching item is found, it is removed from the list, its read giving the written value. If no match is found, the processor number and the address accessed are added to the tail of sleep list and the process is removed from the run-queue.

  When a process writes to a semaphore object, the address written is compared with those in the sleep list. If a match is found, it is removed from the list and the process is placed back on the appropriate run-queue, its read giving the written value. Otherwise, the processor number, address and the value written are added to the tail of the wakeup list. The process is left on the run-queue.

  These mechanisms allow very low overhead synchronisations. Similar mechanism were employed by Ahuja in the Linda machine [Ahu88].

- Rendezvous

  When a process reads from a rendezvous object, the address accessed is compared with those in the wakeup list. If a matching item is found, it is removed from the list and both the new and the old process are placed back on the appropriate run–queues, the read giving the written value. If no match is found, the processor number and the address accessed are added to the tail of sleep list and the process is removed from the run–queue.

  When a process writes to a rendezvous object, the address written is compared with those in the sleep list. If a match is found, it is removed from the list and the process is placed back on the appropriate run–queue, its read giving the written value. Otherwise, the processor number, address and the value written are added to the tail of the wakeup list and the process is blocked by being removed from its processor's run–queue.

4.3.4 Hardware Objects

The object descriptor for a hardware object contains the network address the device physically occupies. Accesses to these objects are translated directly into transactions with the appropriate devices. The exact form these transactions take depends upon the hardware devices involved.

4.4 Implementing Object Space Caching and Coherence

In section 4.3 it was shown how the Object Space could be constructed using the lowest level of the *Cherub* memory hierarchy, the shared logical disk address space. As was previously explained in section 4.2, it is intended that the higher levels in the memory hierarchy shall be used as successive levels of cache for the logical disk address space, with data coherence taking place in the secondary cache. This hierarchy of caches is illustrated in figure 4.10.

The levels in the memory hierarchy will now be discussed in reverse order. The lowest two levels are relatively uninteresting and are only examined superficially:

- Disk

  The lowest layer in the *Cherub* memory hierarchy is constructed from persistent mass storage devices such as magnetic or optical disk drives. It is expected to contain Tbytes of storage.

  As disk seek times[8] and rotation latencies[9] are typically high, a large physical block size, 16 Kbytes, is employed to balance them against the transfer time[10]. A mapping scheme is used to translate the one Kbyte logical disk block addresses used by *Cherub* into the appropriate physical device and block numbers used by the hardware.

[8]Current seek times range from 12 ms to 20ms.

[9]Typically 8.3 ms on a 3,600 RPM drive.

[10]Current transfer rates range from 1 to 4 Mbytes per second.

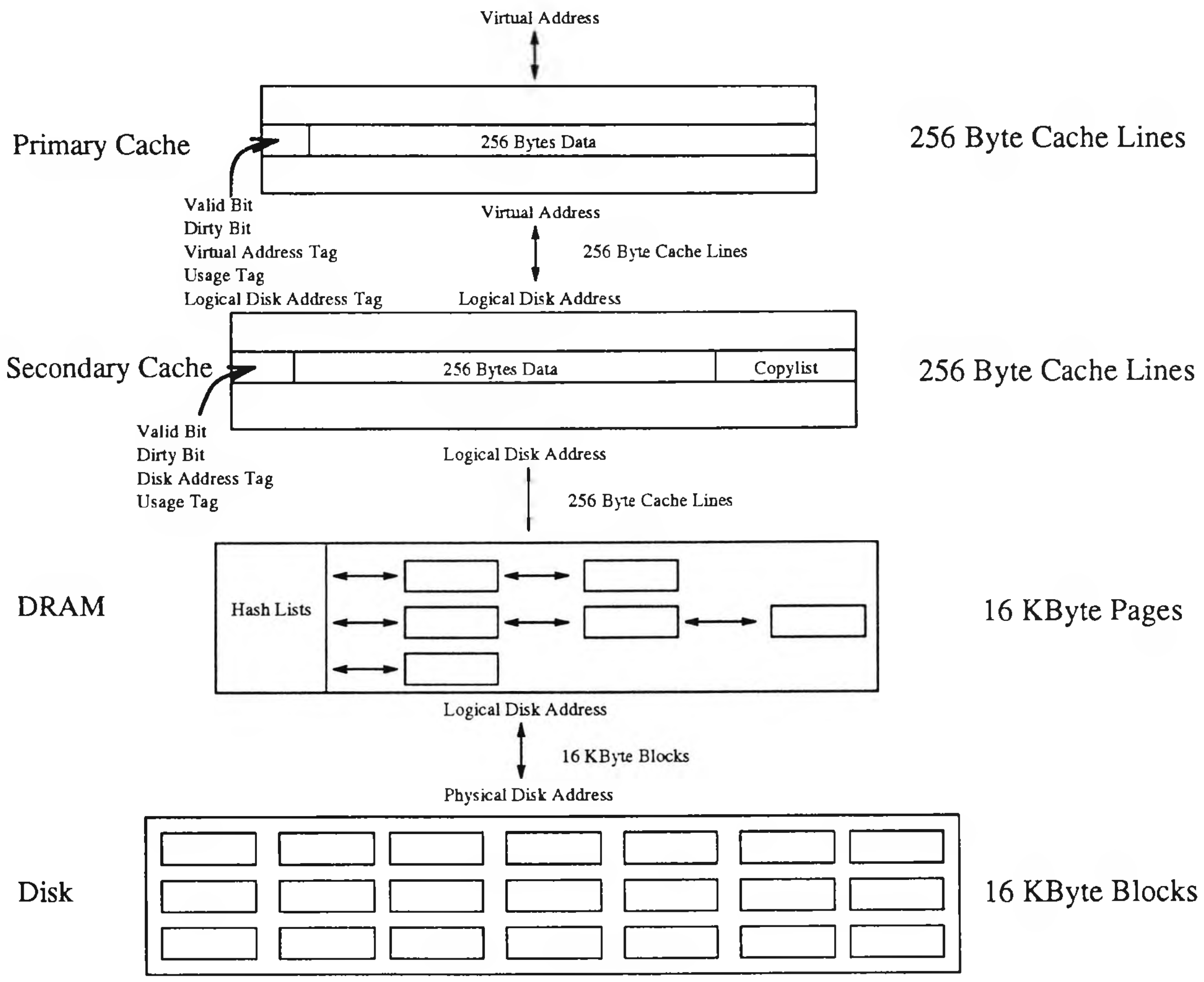


Figure 4.10: The Cherub Cache Hierarchy

- DRAM

  The next layer in the *Cherub* memory hierarchy is constructed from DRAM and is expected to contain Gbytes of storage. This is employed as a cache for the disk memory layer and is accessed using logical disk block addresses.

  The DRAM is logically divided into a number of pages, each comprising 16 contiguous logical disk blocks. This allows whole disk blocks to be transferred to and from the DRAM. Logical disk blocks are located in the cache using hash lists.

  To minimise contention, each processor is assigned a unique set of pages it is responsible for. These pages are initially placed on a free list. When this is empty, they are chosen at random for replacement[11].

The top two layers in the *Cherub* memory hierarchy are very interesting and are examined in some detail:

- Secondary Cache

  The second level in the memory hierarchy is constructed from off-processor SRAM. It is expected that each cache will contain around 16 Mbytes of storage, organised as 256 byte lines — the granularity of data sharing. This level of the memory hierarchy has two main roles: firstly it is responsible for translating virtual Object Space addresses into logical disk addresses; and secondly it performs the data sharing and coherence mechanisms.

  - Virtual Object Space to Logical Disk Address Translation

    In section 4.2 it was decided that *Cherub's* data sharing, and hence coherence, mechanisms will be implemented at the level of the secondary cache. However, due to the possibility of virtual address aliasing through COW, it is also necessary to perform virtual Object Space to logical disk block address translation at the layer of data sharing. The secondary cache is, therefore, addressed using logical disk block addresses. As these are more evenly distributed then virtual addresses, the cache can be set associative, thus reducing its cost.

    The address translation is performed by first using the virtual address to locate the appropriate object descriptor entry. Depending upon the access semantics of the object involved, the offset can then be used to locate the logical disk block containing the data accessed.

  - Data Sharing and Coherence

    For simplicity, data coherence is maintained using Li's fixed distributed server algorithm. Each logical disk block is hashed to a *home* cache controller. This is responsible for maintaining the coherence of the data in its blocks. Each cache line has a *valid* bit and a *permissions* bit. Together these determine the access rights to its data. There are three possibilities:

    * NIL

      A cache miss; the data is not in the cache.

[11]Studies have shown that for large cache sizes there is little performance difference between intelligent replacement policies, such as least recently used (LRU), and non-intelligent schemes, such as random [Hen90].

  * READ
    A read–only copy of the data is in the cache.
  * WRITE
    A read–write copy of the data is in the cache.

  Li's coherence protocol allows either multiple readable copies of a cache line, or a single writable copy. When a permission violation occurs, the processor's cache controller recovers by communicating with the *home* cache controller of the required line. Table 4.2 summarises these communications.

  Cherub keeps track of the copies of a given cache line using a list stored at its *home* address. The list is implemented using a bitfield which has one entry per processor. An extra bit marks whether the list represents read or write copies. Although, the approach of using bitfields to represent the lists is not scalable for very large systems, it will work reasonably for 200 processors and it is simple to implement in hardware. Furthermore, in view of the increased sharing promoted by the SSAS memory model, it is assumed that many pages (especially in shared libraries) will be shared by a large number of processors in the system, thus making more elaborate representations, such as linked lists, less efficient.

Each cache line has two bits which are used in selecting lines for replacement. These bits are set by the *busy*, *idle* and *finish* instructions. When the secondary cache is full, a line is chosen for replacement by first grouping them in reverse order of their usefulness (*empty*, *finish*, *idle* and *busy*) and then selecting the cache line from the least useful group which will generate the fewest coherence messages. This will probably cause lines which are not at their *home* controller to be replaced first, followed by the lines which have the least number of copies.

Since it is likely that there will be locality of reference within logical disk blocks, *busy* and *finish* instructions also allow important performance optimisations to be performed:

- Busy
  Each cache controller maintains a list of lines which are likely to be needed in the future, as defined by *busy* instructions. If a connection is about to be made to a cache controller, the list is first checked to see whether it is the *home* controller of any of the wanted lines. If so, they are read from the controller with the requested permissions, after first ensuring that there is room in the cache to accommodate them (by discarding existing lines if necessary).
- Finished
  Whenever a connection is made to a cache controller, a check is made to see whether it is the *home* controller of any cache lines which are marked as *finish*. If so, they are all written back to the cache controller and discarded from the cache.

When a cache line is replaced in its *home* secondary cache, its copy lists are lost. All copies of the line must, therefore, be invalidated before it can be replaced. To allow primary cache lines, which are virtually addressed, to be selectively invalidated, each

Status	Communication Between		
	Requesting Controller and Home Controller	Home Controller and Other Controllers	Home Controller and Requesting Controller
Want: read Others: nil Copy: yes	lists + data read	□	□
Want: write Others: nil Copy: yes	lists + data read	□	□
Want: write Others: nil Copy: no	lists	□	□
Want: read Others: read(s) Copy: yes	lists + data read	□	□
Want: read Others: write Copy: yes	lists	invalidate to read + data write	lists + data read
Want: read Others: read(s) Copy: no	lists	□	□
Want: write Others: read(s) Copy: yes	lists	invalidate to nil	lists + data read
Want: write Others: read(s) Copy: no	lists	invalidate to nil	lists
Want: write Others: write Copy: yes	lists	invalidate to nil + data write	lists + data read

Key

Want:	The permissions required for a line
Others:	The permissions currently possessed by other controllers
Copy:	Whether or not a copy of the data is required (it is not already cached)
Lists:	Examine and maintain copy lists
Data Read:	Load faulting line from *home* controller
Data Write:	If modified, write line back to *home* controller

Table 4.2: Coherence Mechanism

is tagged with its corresponding logical disk address. This allows secondary cache invalidations to be propagated up to the primary cache.

Due to their unusual access semantics, it is not possible to cache the contents of hardware objects.

- Primary Cache

  The first layer of the memory hierarchy is constructed from on–processor SRAM. It is expected that each primary cache will have around 512 Kbytes of storage, constructed from lines no larger then 256 bytes in length — the granularity of data sharing in the secondary cache.

  For reasons of performance, namely elimination of the TLB address translations, the primary cache is addressed using virtual addresses. This means that only data from the virtual address space is held in the primary cache. The cache employs a write–through policy to maintain data coherence with the secondary cache.

  It is expected that the primary cache will be fully associative, thus allowing a pseudo random replacement policy to be employed. This is important because virtual address space usage is not expected to be uniform; most data accesses will be in the first few lines of objects, which always start in 2^{40} byte boundaries. This makes direct mapped and set associative replacement schemes, which work best when virtual address use is fairly evenly distributed, inappropriate.

  As in the secondary cache, each line has two bits which indicate its usefulness. The replacement mechanism groups cache lines in reverse order of their usefulness (*empty*, *finish*, *idle* and *busy*) and then selects a line at random from the least useful group.

  In general, object data coherence is maintained by simply ensuring that lines are discarded from the primary cache when they are invalidated in the secondary cache. Process objects, however, are slightly unusual in that the processor's registers must also be kept coherent. This is accomplished by ensuring that the primary cache contains writable copies of each of a process objects's 13 lines before it is executed. The processor's registers are then loaded from them. If, however, any of these lines is requested by another cache controller, then execution of the process must be stopped and its registers written back before the appropriate cache line can be discarded.

  Protection is provided using the protection domain registers. Each time a process accesses the primary cache its protection domain registers are simultaneously examined to determine whether the accessed address lies within a mapped object. If so, the appropriate capability is validated against the correct one for the object. Each capability has a single bit tag which indicates whether it has already been validated. These tags mean that capabilities only have to be validated when they are first used. When a line in the secondary cache which contains an object descriptor is invalidated, the capability tags are cleared, forcing them to be revalidated when they are next used.

  The *last accessed* and *last modified* times in an object domain descriptor indicate when the object was last successfully mapped into a protection domain register with the appropriate capabilities. Although this is not semantically identical to the UNIX file times, which indicate when files were last accessed and modified, this mechanism provides similar information at a very low performance cost.

In this section the *Cherub* data caching, sharing, coherence and protection mechanisms have been discussed. The lower three levels in the memory hierarchy act as successive levels of cache for the logical disk address space. Data sharing and address translation are implemented in the secondary cache. Very important cache line coherence and replacement optimisations are made possible through the *busy, idle* and *finish* instructions. Protection is implemented in the primary cache.

4.5 Performance Evaluation of the Implementation Mechanisms

The previous two sections outlined algorithms for the implementation of the *Cherub* Object Space. In this section the latencies of these algorithms will be examined in order to determine the requirements of *Cherub's* underlying communications network.

Before the latency of the *Cherub* process life–cycle as a whole can examined, however, it is necessary to estimate the latency for accessing a cache line. Most data sharing in the *Cherub* life–cycle will take the form illustrated in figure 4.11. This involves five steps:

1. Cache controller A requires write access to line X.
2. It first makes room in its cache for the new line by invalidating line Y. This requires a connection to be made to C, Y's *home* controller, relinquishing access to the line and if it is dirty, writing it back.
3. Controller A can then make a connection to X's home controller, B, and request the new line.
4. However, it is likely that X is currently owned by another cache controller, say D. A connection is, therefore, made to D, invalidating and obtaining its copy of the line.
5. Finally, a connection is made to controller A and cache line X is transferred to it.

This process can be generalised to cover the prepaged transfer of a number of lines, if two reasonable simplifying assumptions are made:

- All of the data required will be somewhere in a second level cache; i.e. nothing has been paged out to the lower levels in the memory hierarchy.
- All of the cache lines in a given data block are owned by the same cache controller. This is very likely to be true.

If a network message requesting a cache line is 16 bytes in length, then the component latencies for accessing a cache line will be:

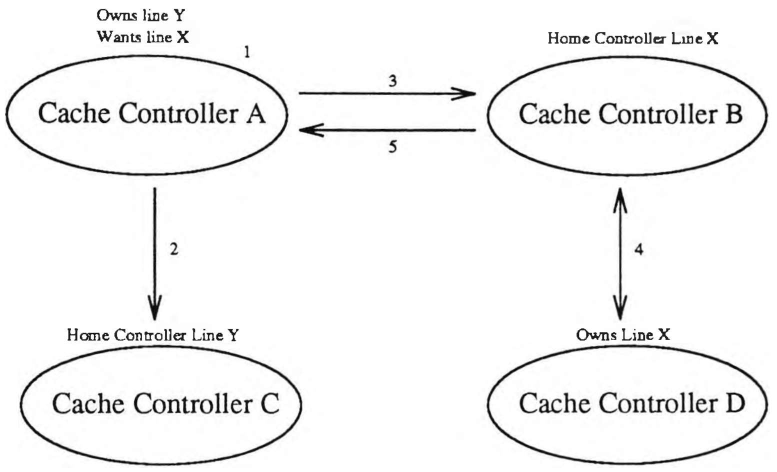


Figure 4.11: A Cache Line Read With Write–back and Invalidation

Stage	Operations	Latency
1	Cache lookup and miss on line X.	A
	Select a cache line, Y, for replacement.	A
2	Make connection to Y's home cache controller.	C
	Invalidate line Y.	$272T$ per dirty line
		$16T$ per clean line
	Read line into cache.	N/A (pipelined)
3	Make a connection to line X's home cache controller.	$C + 16T$
	Cache lookup on line X. Find current owner.	A
4	Make connection to current owning cache controller.	$C + 16T$
	Lookup and invalidate line X in cache.	A
	Transfer line back to home cache controller.	$272T$ per line transferred
5	Make connection to requesting cache controller.	C
	Transfer required line.	$272T$ per line transferred
	Read line into cache.	N/A (pipelined)

Where:

C = Time to make a network connection (network connection latency)
T = Time to transfer a byte across a network connection (related to network bandwidth)
A = Time to access a cache line
I = Time to execute one machine instruction (will be used later)

Obviously, optimisations can be performed. For example, where a cache line is known to be empty — when it has been removed from a free list, for instance — there is no need to transfer its contents across the network when obtaining permissions for it. However, the latency of a cache line access as stated — the summation of its component latencies — is:

$$4A + 4C + 272T \text{ per dirty line flushed } + 16T \text{ per clean line flushed}$$

$$+ 576T \text{ per new line transferred}$$

Assuming that 25% of lines flushed from the cache are dirty[12], this averages to:

$$4A + 4C + 80T \text{ per flushed line } + 576T \text{ per line transferred}$$

$$= 4A + 4C + 656T \text{ per line transferred}$$

It is now possible to estimate the latency of the *Cherub* process life–cycle using the list of tasks given in chapter three.

- **Create Process, Stack and Rendezvous Objects (Parent Cache Controller)**

  To create these objects, it is necessary to remove three object descriptors and eight blocks — four for the process object and two each for the stack and rendezvous objects — from the appropriate free lists. It is assumed that the cache lines from the object descriptors and blocks are originally owned by the parent cache controller; the validity of this assumption will be determined when the cleanup operation is discussed. The latency of creating the objects is, therefore:

Operation	Latency
Get three new object descriptors	$9A$
Get eight new blocks and zero fill	$41A$

- **Build environment on Stack Object (Parent Processor)**

  It is estimated that up to 500 instructions will be needed to initialise a process and perform *finish* instructions on the process object and its stack[13]:

Operation	Latency
Build environment and issue *finished* instructions	$500I$

[12]Experimental evidence for this is difficult to obtain since little analysis has been performed on SSAS systems. However, a rough estimate can be made from examining the UNIX System V buffer cache:

- File cache blocks — which include data from files, directories, inodes, pipes and program texts — are rarely dirty (5% of the time) and occupy around 65% of the cache [Bac88a].
- The remainder of the cache (35%) [Bac88a] contains data and stack pages from the virtual memory. If these pages are in an executing process's working set, it is likely that they will be dirty (100% of the time).

The expected proportion of dirty cache lines is, therefore:

$$0.05 \times 0.65 + 1.0 \times 0.35 = 0.38$$

UNIX System V's one Kbyte page size will, however, incur considerable false data sharing. It is, therefore, not unreasonable to assume that *Cherub's* 256 byte page size would result in a considerably lower fraction of dirty pages, say, 25%.

[13]Initialising the five registers in each of the 12 domain registers might reasonably take 200 instructions. Placing the function arguments and environmental data on the stack could take another 250. Finally, up to 50 *finish* instructions might have to be issued.

- **Place Process Object on Process Run–Queue (Parent Cache Controller)**

  The child process is placed on a random processor's run–queue:

Operation	Latency
1 line read, 2 line writes	$12A + 12C + 1,968T$

- **Remove Head of Run–queue (Child Cache Controller)**

  Assuming the new processor is idle, on average it will poll its run–queue after 50 instructions. (If the processor is busy, on average it will only poll after 6,000 instructions. This is intended to keep processor throughput high.) The processor removes the process from its run–queue prior to execution:

Operation	Latency
2 line reads, 1 line write	$12A + 12C + 1,968T + 50I$

- **Transfer Process Object (Child Cache Controller)**

  The new processor starts to execute the child process. It prepages in the child's process object:

Operation	Latency
Read object descriptor	$4A + 4C + 656T$
Get 13 lines from process object	$4A + 4C + 8,528T$

- **Transfer Stack Object (Child Cache Controller)**

  The child process prepages in its stack object:

Operation	Latency
Read object descriptor	$4A + 4C + 656T$
Get data block entry	$4A + 4C + 656T$
Get data block (4 lines)	$4A + 4C + 2,624T$

- **Synchronise (Child Cache Controller)**

  When the child process has terminated, it writes the address of its results to the rendezvous object. This unblocks the parent process which is reading from the rendezvous object:

Operation	Latency
Read object descriptor	$4A + 4C + 656T$
Read sleep–list	$12A + 12C + 1,968T$
Put parent process back on processor queue	$12A + 12C + 1,968T$

- **Return Results on Stack (Parent Cache Controller)**

  The parent process prepages in the stack object, which contains the child's results:

Operation	Latency
Read object descriptor	$4A + 4C + 656T$
Get data block entry	$4A + 4C + 656T$
Get data block (4 lines)	$4A + 4C + 2,624T$

- **Cleanup Process, Stack and Rendezvous Objects (Parent and Child Cache Controllers)**

  The destruction of the rendezvous and process objects is performed by the child process. Therefore, the latency of the cleanup operation should be hidden from the parent process. The cleanup will, however, generate extra communication which will add to network load.

  The stack object contains the child's results. The parent process will probably not destroy this object until it terminates, thus hiding the latency from its parent.

  The cache lines from the object descriptors and blocks freed in the cleanup operation will be owned by the terminating processor. They are placed back on its freelist, thus minimising the latency of creating new objects.

Therefore, from section 1.1.1, a given level of granularity g is optimal iff:

$$t_{overhead} \leq (g \cdot \ln 2)I$$

$$= 134A + 84C + 25,584T \leq (g \cdot \ln 2 - 550)I$$

Now assuming that:
g = 10,000 instructions;
I = 6 ns; and
A = 2 ns (32 ns to transfer a 256-bit line to primary cache)

then the performance of the underlying communications network must be such that approximately:

$$84C + 26,000T \leq 40,000\,\text{ns}$$

If it is assumed that the network traffic will be fairly evenly spread over time[14], the number of connections resulting from a program constructed from 100 pairs of child and parent tasks will be:

$$\text{Simultaneous connections} = \frac{(7,000 - 550)I - 50A}{(10,000 + 7,000)I} \times 100 \approx 40$$

From inspection, it can be seen that *Cherub* requires a very high performance communications network which scales well when heavily loaded; it must certainly have a connection latency below 500 ns and a bandwidth in excess of 650 Mbytes per second when loaded with 40 connections.

[14]It is assumed that perfect parallelisation cannot be achieved in most programs and so the various process life-cycles will drift out of synchronism with one another.

4.6 Conclusion

In this chapter an efficient implementation for the *Cherub* architecture was suggested, based on predictions about future hardware. It was decided that the Object Space should be implemented on disk storage for persistence, the memory hierarchy being used as successive layers of cache.

In a tradeoff between minimising false data sharing and table costs, it was decided that data sharing should be performed at the level of the secondary processor cache. As a result of this decision, the data coherence and address translation mechanisms must also exist at this level.

It was shown that important performance optimisations can be achieved through careful use of the *busy*, *idle* and *finish* instructions. These allow cache line prepaging to be performed, resulting in a significant reduction in the number of network connections which are necessary.

The latencies of the proposed mechanisms were estimated, showing that *Cherub* requires a very high performance communications network which scales well when heavily loaded. The design of a network which has these properties is developed in the next chapter.

Chapter 5

A Wafer–Scale Communications Network

5.1 Introduction

Now that the mechanisms for the implementation of Cherub have been outlined, a communications network must be designed which has the latency and bandwidth to support them:

- Latency is defined as the observed delay through a network component. The total latency of a message path is the sum of its component latencies. Latency is a function of network loading.

- Bandwidth is defined as the throughput of a network component. The available bandwidth of a message path is that of the narrowest component in the path.

In most systems overall latency is the most important network performance metric. This is the time taken from the start of data transmission at the source until its complete reception at the destination. Chapter four established that a typical 310 byte message must have an overall communication latency of around 480 ns in a network already loaded with 40 connections. This implies a minimum bandwidth of around 650 Mbytes per second.

This chapter suggests that wafer scale integration could be used to achieve the level of network performance required by *Cherub*. A suitable wafer network is designed and its performance is estimated by simulation.

5.2 Classifying Networks

The two most common ways of classifying communication networks are according to their topology and communication strategy:

5.2.1 Network Topologies

All communications networks can be thought of as a number of nodes joined in various topologies by data transmission channels. Total connectivity, supposedly the ideal situation where all the nodes in a system are connected to one another, is only possible for small networks. This is due to the complexity of the interconnections required, which is exponentially related to the number of nodes in the system.

The topology of any network with less than total connectivity will have profound effects on network loading, traffic density and locality within the communications system. The designers of parallel systems have experimented with numerous network topologies [Geo90].

Due to their ease of construction, the most popular types of topology employed are one and two dimensional networks, although networks with higher dimensions, such as hypercubes, or irregular topologies have also been shown to be practical in medium scale architectures.

- One dimensional networks

  One of the simplest methods of networking computers is to connect them all by a one dimensional network. A common type of one dimensional network is the multiprocessor bus. This uses one or more globally shared transmission channels to connect a number of processors. This type of network supports broadcast transmissions, any conflicts occurring on the buses being handled by arbitration logic. A bus system is illustrated in figure 5.1.

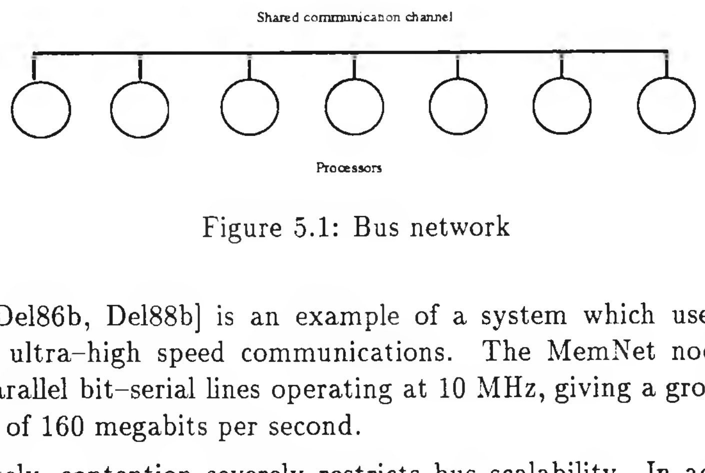


Figure 5.1: Bus network

MemNet [Del86b, Del88b] is an example of a system which uses multiple buses to provide ultra-high speed communications. The MemNet nodes communicate using 20 parallel bit-serial lines operating at 10 MHz, giving a gross aggregate data bandwidth of 160 megabits per second.

Unfortunately, contention severely restricts bus scalability. In addition the speed at which a bus can be run is often physically limited by its length. An alternative mechanism is to use a staged communications network consisting of a number of local communication channels. A staged network is illustrated in figure 5.2. The bandwidth of such networks increase with the number of nodes, but at the expense of increased latency.

The *latency* of a network topology is at least partially determined by its maximum communication path length, or diameter δ. For a staged network comprising n nodes this is:

$$\delta(Staged(n)) = n - 1$$

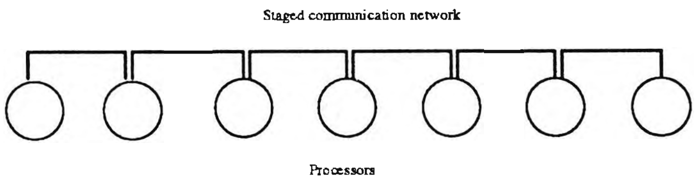


Figure 5.2: Staged network

The main disadvantage of this communication system is that it has a poor ratio of interconnection, τ, the ratio of nodes to links. Assuming that the network traffic is distributed evenly, which is clearly not always possible:

$$\tau(Staged(n)) = 1 - \frac{1}{n}$$

The diameter of a staged network can be halved by joining its ends to form a ring. A ring network is illustrated in figure 5.3. A ring containing n nodes will have a diameter:

$$\delta(Ring(n)) = \left|\frac{n}{2}\right|$$

and a ratio of interconnection:

$$\tau(Ring(n)) = 1$$

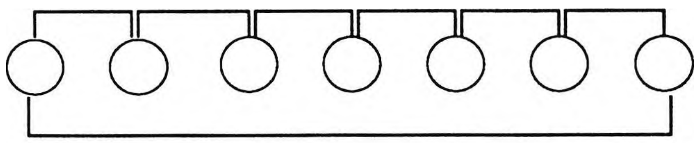


Figure 5.3: Ring network

The Kendall Square Research KSR1 architecture [Bur92] uses a ring network to provide scalable communication among 32 processors. In addition, up to 34 of these rings can be linked by a master ring to create a machine with 1,088 processors.

- Two dimensional networks

  To reduce the bandwidth limitations imposed by one dimensional communication networks, two dimensional communication networks have been explored. These typically take the form of 4–connected meshes of nodes. A mesh network is illustrated in figure 5.4.

  For the same number of nodes, two dimensional networks have smaller diameters than their one dimensional counterparts. A square mesh containing n nodes has a diameter of:

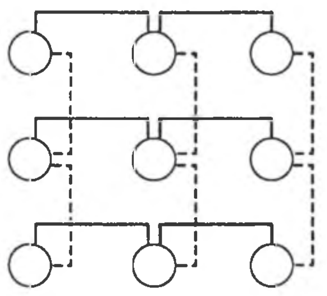


Figure 5.4: Mesh network

$$\delta(Mesh(n)) = \left|\sqrt{2n^2}\right|$$

Contention in two dimensional networks is also reduced due to the greater number of potential paths in the system. In a square mesh containing n nodes the ratio of interconnection is:

$$\tau(Mesh(n)) = 2\left(1 - \frac{1}{\sqrt{n}}\right)$$

By wrapping the opposite edges of the mesh to form a torus it is possible to reduce a mesh network's diameter further. A torus network is illustrated in figure 5.5. A square torus network containing n nodes will have a diameter:

$$\delta(Torus(n)) = 2\left|\frac{\sqrt{n}}{2}\right|$$

The ratio of interconnection in such a network is:

$$\tau(Torus(n)) = 2$$

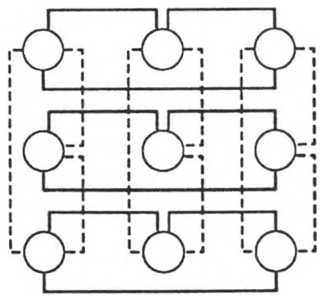


Figure 5.5: Torus network

The Topsy [Win89] computer uses such a torus communications network. Each processor node has two custom networking chips, the BIC [Win88] and NCU [Win87]. These allow each node to be connected to up to four others, thus making it possible to experiment with different network topologies.

5.2.2 Communication Strategy

Another common way of classifying networks is by communication strategy. Four strategies are commonly employed:

- Circuit switched routing

  In a circuit switched network an electronic circuit is created between the source and destination nodes. Once the circuit has been made, the message can be transmitted over it. When the transfer is complete, the circuit can be cleared. If during a circuit's construction a required communication channel is already in use, the complete circuit is cleared and the sender waits for a back-off period before retrying. This prevents deadlock.

  The main advantage of this scheme is that once the circuit has been established, the bandwidth is independent of network load. The scheme's main disadvantage is that the latency of establishing a circuit is highly dependent upon network load and is potentially unbounded. This can make short message based communication inefficient.

- Packet switched routing

  In a packet switched network a message is broken down into small packets at the source node. These are transmitted across the communication network and reassembled in order at the destination. Each packet carries its own routing information and is routed independently. This allows messages to be interleaved on the same communication channel. If a communication channel required by a packet is already in use, the packet is buffered until the channel becomes free. Thus, there is the potential for deadlock.

  The main advantage of this system is that the low latency of transmitting a single packet makes short message based communication efficient. Furthermore, as the network exhibits graceful loading characteristics, message latency is relatively predictable. The scheme's main disadvantage is that its bandwidth is not constant — the packet throughput depends upon the network load. In addition each node requires memory buffers to hold the message packets while they are being routed.

- Wormhole routing

  Wormhole routing is a variation on packet switched communication. In this scheme a message is broken down into packets which are routed between the source and destination nodes in a continuous stream. Only the head packet in a stream carries routing information and so, unlike true packet switched communication, packets from different streams cannot be interleaved on a single communication channel.

  If the head packet becomes blocked, all of the packets in the stream stop advancing, thus blocking the progress of other messages requiring the channels they occupy. As a result there is the potential for deadlock.

  The main advantage of this scheme is that once the head of a message stream has arrived at its destination, the bandwidth is independent of network load.

- Virtual cut–through routing

  This mechanism is similar to wormhole routing, except that when the head of a message stream becomes blocked, the packets in the message are temporarily buffered at that node, thus removing them from the network.

  This scheme has the bandwidth advantages of wormhole routing and exhibits better performance under load, resulting in lower latencies. However, each node requires enough memory buffers to be able to buffer complete message streams and, consequently, there is the potential for deadlock.

As previously established in section 4.5, the *Cherub* architecture requires a network which has an overall bandwidth of 650 Mbytes per second for 310 byte messages. It is asserted that this is an unprecedented level of performance, totally beyond the capabilities of conventional networks. Single mode fibre optics, one of the most modern communication mediums, has the bandwidth to match this requirement — Gbits per second — but compatible low latency switching elements are currently not available without resorting to fabrication technologies such as gallium–arsenide[1]. Instead, a new very fast and wide interconnection network is required. One way of achieving this is by employing wafer–scale integration.

5.3 Wafer–Scale Integration

Conventional very large–scale integration (VLSI) computer chips are typically manufactured by lithographing superimposed layers of metal and polysilicon onto circular wafers of silicon to form the circuits of discrete chips. The completed wafers are then cut into individual chips prior to packaging and testing. The maximum chip size is normally determined by factors such as the VLSI yield, the defect density, the allowable interconnection lengths and the dimensions of the available packaging.

Wafer–scale integration (WSI), the process by which wafers are packaged whole rather than as individual chips, is an interesting alternative method of creating electronic circuits. There are two main schools of thought in WSI: whole wafer integration and hybrid wafer integration.

- Whole wafer integration

  In this technique wafers are lithographed and packaged whole. This allows the very highest levels of integration to be employed, but is prone to crystal and processing defects.

- Hybrid wafer integration

  In this technique tested circuit dies are attached, often by flip–chip bonding [Gol83, Mil69, Bac88b], to a wafer substrate which contains an interconnection network.

[1] Whitcroft [Whi92] has proposed a silicon based switching circuit which uses the Fibre Channel standard [fib92]. He predicts that this should be able to perform a routing decision in 1 μs. Even this is too slow for *Cherub*.

Flip-chip bonding is illustrated in figure 5.6. The tiles and substrate are manufactured with matching pairs of electrical contact pads. Solder is applied to these and the tiles are placed on the substrate so that the pads make contact. The substrate is then heated, causing the solder to flow. Surface tension in the solder helps to correct any misalignment between the surfaces. The wafer is then allowed to cool.

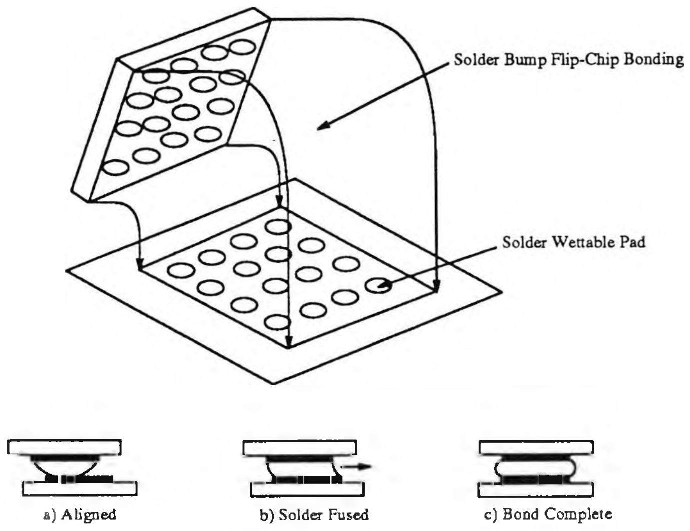


Figure 5.6: The Flip-chip Bonding Process

Flip-chip bonding also allows processors with the very large secondary caches required by *Cherub* to be constructed. Wilkinson [Wil91b] has suggested small processor dies with tested cache circuitry should be flip-chip bonded onto much larger fault tolerant cache dies. The high yield processor dies would test the lower yield cache dies and only use the working cache lines.

Hybrid manufacture has the advantage of very high yield and allows circuits which require different manufacturing techniques, such as high density memory and high performance processors, to be intermixed on a single wafer substrate. However, due to the relatively large size of solder-bumps, its data-paths are narrower then can be achieved with whole wafer integration and it requires expensive testing and bonding.

Wafer-scale integration has three main attractions when compared with similar functionality constructed from individual chips mounted on PCBs:

- Higher performance

  The main performance advantages wafer-scale integration has over conventional chip and PCB technologies are:

  - Higher speed (Decreased latency)

    Conventional VLSI technology uses chains of output transistors to drive the pins of chips. These prove to be slow. The capacitance of the solder bumps used in hybrid wafer integration is about 20 times lower than that of wire bonds and hence the speed of internal wafer communication can be considerably faster then conventional chip to chip communication — a factor 4 speed-up is not unusual. The circuit interconnections in whole wafer integration, made by aluminium alloy metalisation on silicon, are faster still.

- Increased wire density (Increased bandwidth)

  As Dally suggests [Dal87], VLSI chip technology is severely limited by the number of pins that can be placed on a chip. If the number is high, the package must be large to accommodate them. Consequently the chip is expensive and its PCB density is low. In addition, layout considerations make it hard to route large numbers of tracks away from a chip; expensive multiple layer PCBs are required. Therefore, data-paths in conventional computers are severely limited in width.

  This is not so much of a problem in whole-wafer integration devices as these can employ very dense tracking. Unfortunately, in hybrid wafers the size of the solder bumps limits the track density[2]. Even so, communication data-paths can be much wider then those in conventional systems.

For these two reasons wafer-scale communications networks have a significant bandwidth advantage over conventional chip and PCB based networks.

- Lower cost per function

  A chip's packaging is a major proportion of its cost of manufacture; a 400 pin grid-array ceramic package can cost as much as $50 [Hen90]. WSI offers much higher levels of integration on a single device. This reduces the volume of packaging required for a given amount of functionality.

  The cost per bit for semiconductor and disk memory decreases at very close to the rate at which density increases. The continued fall in cost of VLSI technology is dependent upon achieving ever-increasing levels of function integration and chip yield. There is, however, a hard limit to the level of integration possible using VLSI. This could eventually force manufactures to turn to wafer-scale integration in order to achieve further price reductions.

- Higher reliability

  Hardware reliability problems are often caused by defective connections, either in the form of the soldered joints connecting the chip pins to the printed circuit boards (PCBs), or the bond wires inside chips which connect the pins to the silicon. In effect, a system's mean time to failure is proportional to its number of pins. Aubusson [Aub91] reports that 40% of US avionics failures arise directly from solder contacts and PCBs. Assuming that a future processor might reasonably have as many as 400 pins, the processors alone in a *Cherub* system will have over 102,400 bond wires. Consequently, the system could have a significant reliability problem.

  Most of the interconnections in a wafer are made by aluminium alloy metalisation on silicon. This is inherently more reliable than printed circuit board (PCB) interconnections. In addition, successful WSI must have built-in fault-tolerance as the

[2]Typical solder bumps are approximately 125 μm in diameter and are spread 75 μm apart [Tew89].

yield of defect-free wafers is essentially zero. An additional increase in reliability may be provided by this.

WSI does have three considerable problems, however, which significantly affect its commercial viability:

- Wafer based products are difficult to design and manufacture

  This is mainly for three reasons:

  - The need for fault tolerance

    Although manufacturers treat chip yield statistics as proprietary information, it is generally accepted that a significant proportion of chips are defective[3]. Rough estimates show that VLSI defect density is inversely proportional to the square of the feature size. This is due to defects, which were previously too small in relation to the feature size to be significant, now being large enough to cause failures.

    Yield models have been developed from both theoretical and empirical studies of defect distribution. Wallmark [Wal60] applied basic Maxwell-Boltzmann statistics to develop a yield model. Although he noted that defects are probably correlated, his model assumed none. Hofstein and Heiman [Hof63] and more recently Cunningham [Cun90] have modelled random VLSI defects using the Poisson distribution.

    Eventually the use of Bolzmann and Poisson statistics for modelling defects was shown to be too pessimistic for larger chips [Moo70]. This is because defects are not placed uniformly on wafers, but tend to cluster, particularly towards the edge of the wafer. Stapper et al [Sta83] believe this to be due to aggregates of particles which have settled on the wafers during manufacture. However, Stapper [Sta81] has shown that, over many wafers, the Poisson distribution is valid.

    Although manufacturers may be able to afford to mass produce chips with quite low yields, the yield of defect-free wafers is essentially zero and will remain so for the foreseeable future. VLSI memory manufacturers improve chip yield by producing devices with extra rows or columns, as was first proposed by Tammaru et al in 1967 [Tam67]. Defective components are located by external test circuitry[4]. These are substituted for working components by a linking and fusing process using a laser. However, because of the difficulty of isolating circuits in a wafer, external functional testing techniques are difficult to apply to complete wafers. Successful WSI, therefore, probably requires built-in fault tolerance.

  - Long signal lengths

    It is difficult to operate large silicon circuits, such as wafers, synchronously at high clock speeds. This is due to the accumulated line rise and settle delays over

[3]If the yield is near to perfect, it can be reasoned that a higher density should be employed.

[4]Testing is often a complicated process [Van90] and internal test circuits could well consume one third of the total silicon area.

the wafer surface leading to problems such as clock skew. This problem can be overcome by clocking areas of the wafer independently and using asynchronous communications protocols [Mar89, Gin90]. Such circuits are, however, more difficult to design.

- Difficult to mix different technologies on a single wafer

  It is difficult to combine technologies requiring different fabrication processes (such as high density DRAM and high performance logic) or materials (such as silicon and gallium-arsenide) on a single wafer substrate. This can be overcome with flip-chip bonding techniques.

• Wafers are difficult to package

WSI products are difficult to package for a number of reasons:

- Wafers have a large area

  The large difference in the heat expansion qualities of silicon and the packaging leads to problems of wafer breakage. Heat-sinking a complete wafer is also more difficult. This problem can be solved by cooling wafers using liquids or blown air.

- Wafers require large numbers of pins

  A wafer device with a million gates could have as many as 4,000 signal pins[5] [Car86a]. Current pin bonding technology limits pin bonding sites to the periphery of the wafer as bond wires cannot be crossed. Routing large numbers of signal lines to the edge of a wafer is wasteful of silicon area. It also incurs appreciable signal delays. Furthermore, it is expensive to route large numbers of wires away from a wafer.

  This problem can be overcome by using fibre optic technology instead of conventional pins. Techniques have been developed which allow optical fibres to be bonded directly to the wafer surface [Pru86]. This allows fibre optic drivers to be placed at any point on the wafer. Gallium-arsenide based optical network technology currently supports Gbit per second data transfer rates. As optical bonding technology currently limits a fibre to a single emitter and receptor, optical fibres must be paired if bi-directional communication is to be supported.

- Power delivery and removal is difficult

  The line inductance in a large wafer sized power grid causes significant noise. This has to be suppressed by decoupling capacitors. The power grid also consumes an unacceptable proportion of the wafer. This can be solved by placing power and ground pins at regular intervals across the wafer's surface.

- Wafer stacking must be possible

  A wafer has a large PCB footprint. To achieve a reasonable packing density and to reduce the cost and reliability problems associated with large PCBs, it will be necessary to stack wafers on top of each other.

[5]Hughes' 3D Wafer Stack Cellular VLSI has 1,000 microspring bridge pressure contacts to neighbouring wafers. Mosaic System's Wafer-Scale Hybrid has 840 pins per wafer.

This can be achieved by creating vertical electric interconnections, vias, which pass through the wafer substrate, enabling stacked wafers to be electrically connected. Vias can be created using a process such as aluminium thermomigration [Cli76]. This directionally diffuses liquid aluminium though the silicon wafer. Microspring bridges [Gri84] can be used to connect the vias to the signal lines on the adjacent wafer. This arrangement is illustrated in figure 5.7.

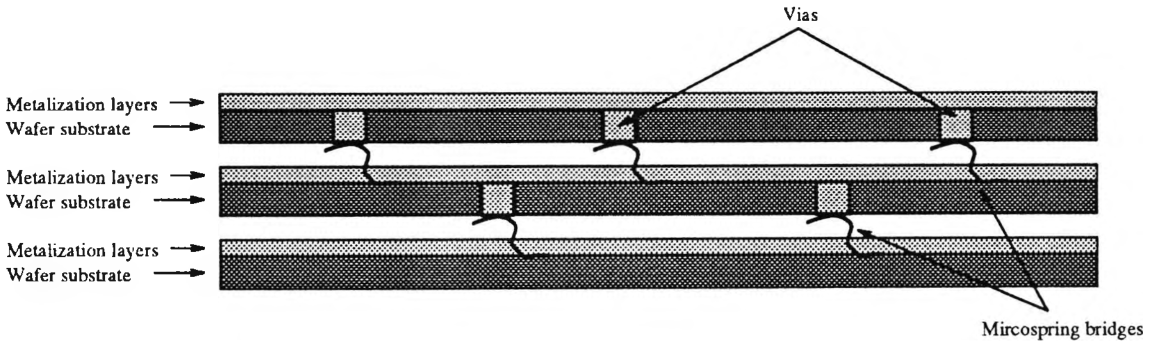


Figure 5.7: Stacking Wafers

Special wafer packages must be designed which stack. It is more difficult, however, to power and cool such an arrangement.

- Wafer-scale integration has a bad name

  There have been a number of expensive project failures in the history of wafer-scale integration. The most famous of these was Amdahl's company, Trilogy, [Gup88] which attempted to create a high performance IBM mainframe clone using WSI. To provide fault tolerance all circuits on the wafer were created in triplicate. This effectively reduced the usable wafer area by two thirds, which meant that a number of wafers had to be employed. These were connected by many bond wires, which led to insoluble unreliability problems. Trilogy crashed with the loss of $230 million.

  As a consequence of such failures, it is difficult to obtain funding for further research.

These disadvantages are not impossible to overcome and commercial wafer-scale products, particularly for memory, have started to appear. From the point of view of fabrication, memory is ideally suited to wafer-scale integration. This is because:

- The memory modules in a wafer are identical. This makes them easy to manufacture and allows global redundancy to be employed.
- The memory modules in a wafer can share a common bus, thus reducing the pin-out.
- Memory is more in demand than logic and can, therefore, be packaged in larger quantities at reduced cost.

Although Carlson and Neugebauer [Car86a] dismiss memory as a WSI candidate, Chesley [Che88b, Che87] has suggested a way of using wafer-scale memory to replace conventional DRAM chips:

- Chesley proposes a non-redundant approach to constructing wafers of DRAM using whole wafer integration. By not employing a redundancy scheme, Chesley uses all of the area of the wafer. This results in a low storage yield of around 50 per cent, but simplifies the manufacturing process.

  A list of usable wafer regions is used by the computer's virtual memory hardware to map virtual addresses to defect-free physical addresses on the wafer. When each wafer is initially tested for defects all perfect blocks are added to the region list. Damaged blocks are ignored. The region list is either loaded from disc, whenever the system is booted, or is reconstructed after testing the wafer. The latter scheme has the advantage of detecting subsequent failures.

As Chesley's scheme does not employ a defect control mechanism, it is as fast as conventional DRAM. However, its yield is low, making it relatively expensive. By employing a defect control mechanism, a wafer's storage yield can be substantially improved, but at a cost of increased latency. Such wafers are better suited as alternatives for magnetic disk storage devices.

As a storage medium, wafer-scale silicon storage has several advantages over conventional magnetic storage devices:

- it is faster;
- it can be accessed randomly while magnetic storage only allows pseudo random access to data;
- it is more reliable; and
- it has better handling properties.

It has the disadvantage, however, of being volatile. Anamartic's Wafer Stack product [Ano89, Cur89] has shown wafer-scale silicon storage to have considerable promise:

- Anamartic's Wafer Stack silicon storage device is one of the first commercial attempts to provide wafer-scale memory. The device uses six-inch wafers containing 202 one-Mbit DRAM and control logic tiles embedded in a communications network. External test circuitry is used to identify the working DRAM tiles and to calculate the longest possible Catt Spiral incorporating them which can be grown from a tile at the periphery of the wafer [Aub78]. This electrically configures adjacent working DRAM tiles to form a single long shift register along which data can pass. Tiles which cannot be included in the spiral are wasted. An example wafer is illustrated in figure 5.8. A map of the resulting spiral is stored in a PROM. This is used by an external controller to access the working DRAM tiles.

  On average, defects reduce a wafer's usable storage capacity to just over 20 Mbytes. By pairing relatively good and bad wafers, storage modules can be constructed with a minimum storage capacity of 40 Mbytes. It is possible to connect up to four storage modules to a single controller.

  Wafer Stack has two main disadvantages which reduce its commercial viability:

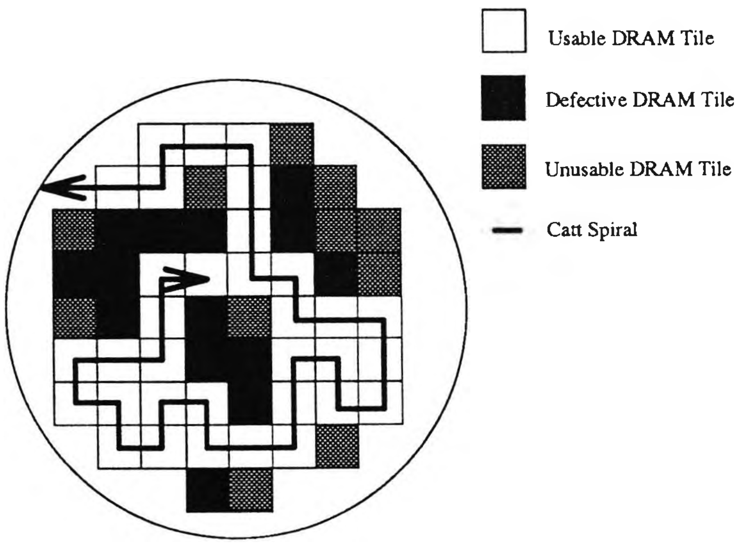


Figure 5.8: A Catt Spiral

– It is slow because it only employs a single 8-bit wide data-path and has a low performance processor acting as an intelligent interface. Consequently, it has a relatively high latency of around 200 μs and a low peak transfer bandwidth of 800 Kbytes per second.

– It is expensive because it does not employ state-of-the-art memory chips. In an attempt to overcome these restrictions prototype wafers employing four Mbit DRAM technology have also been developed. These can hold around 128 chips, giving a capacity of 64 Mbytes.

One of the most interesting aspects of the Wafer Stack design is that it employs standard DRAM tiles linked by a wafer-scale communications network. This network, however, is only one dimensional and is very narrow.

City University's COBWEB project [And90b] proposed using wafer-scale integration to create a parallel graph reduction architecture. It suggested that a wafer can be constructed which embeds conventional VLSI tiles in a four-connected packet-switched communications network. This could provide a very high performance communications network similar to that required by *Cherub*.

5.4 The COBWEB Wafer-Scale Architecture

The COBWEB and related projects [And90b, And89, Gul91] suggest WSI techniques for the construction of ultra high performance communications networks. These techniques are highly applicable to *Cherub* and, therefore, COBWEB will now be described in detail.

COBWEB is based on a set of six wafer-scale integration design rules formulated at City University:

- Whole wafer integration should be employed to reduce costs and maximise performance.
- Successful wafer-scale integrated devices should comprise a large number of independent payload blocks embedded in a communications network. This will result in high fault-tolerance.
- The communication and payload circuitry should be separated so that the communication architecture is general purpose and reusable. This will lead to faster product design and lower costs.
- The wafer should be kept as simple as possible. If performance restrictions allow, any complex processing should be done off the wafer. This improves the yield of usable components.
- To reduce production costs, the minimum amount of defect repair should take place. Only short circuits, which otherwise would be fatal, are patched prior to wafer packaging.
- Wafer testing should be accomplished by external circuitry which will inject test patterns into the edge of the wafer. This well-known testing strategy, called percolation [Wie82], minimises the amount of wafer consumed by test circuitry.

The structure of a COBWEB wafer is illustrated in figure 5.9. It consists of a regular mesh of cells comprising communication (CE) and payload (PE) pairs. Each CE is connected to its nearest neighbours by a set of shared registers, thus forming a four-connected communication network. Each CE is also connected to its PE, usually a processor and associated memory. By separating the communication and payload functions of a cell in this way, a high level of defect-tolerance may be obtained; a working CE can be used even if the associated PE is defective.

It is estimated that a eight inch wafer composed of one square cm cells would have a diameter of 20 cells and, being circular in shape, would contain 316 cells[6].

5.4.1 The Communications Element

One of the major benefits of the COBWEB architecture is that the useful silicon area is comparable to that of a wafer containing conventional VLSI tiles. This is because the communication network uses the space normally reserved for pads, test dies and scribe lines. This makes COBWEB as economical as conventional VLSI, if not more so.

The communications element (CE) consists of three 128-bit bi-directional communication channels, a 64-bit routing bus and a controller. Two of the communication channels combine with those of adjacent cells to form a four-connected communications mesh. The

[6]Calculated from the Hennessy and Patterson estimate for the number of dies on a wafer [Hen90]:

$$number\ dies \approx \frac{\pi \times (\frac{wafer\ diameter}{2})^2}{die\ area} - \frac{\pi \times wafer\ diameter}{\sqrt{2} \times die\ area} - test\ dies$$

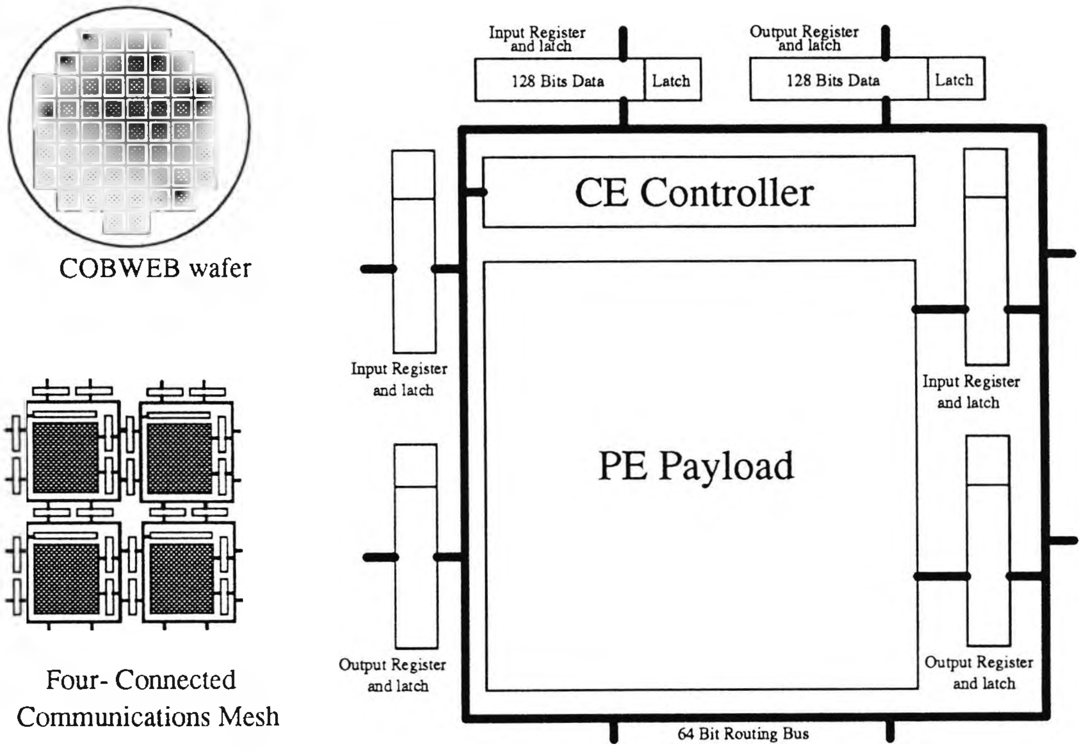


Figure 5.9: A COBWEB Wafer

third connects the CE to its payload. Conservative fabrication technology in the network increases its yield. Using 1.5 micron CMOS technology and one square cm cells, it is estimated that less than 25 per cent of the wafer will be devoted to CEs.

The communication channels comprise two 128-bit register and one-bit status latch pairs. The registers are used to hold input and output packets. The status latches signal when the registers are full. The registers and latches are shared between adjacent cells so that the output register and latch of one cell is the input register and latch of its neighbour.

COBWEB uses packet switched communication; it was thought that this was the simplest to implement in silicon and would, therefore, result in the highest CE yield. The CE controllers supervise the movement of 128-bit data packets in a series of cell-to-cell hops. Packets are transferred in two 64-bit operations. This reduces the CE bus size, a major yield hazard. Packet transfer can broken down into a series of steps:

- A packet arrives in the output register of the neighbouring CE. The neighbour sets the input latch to indicate the input register is full.
- The CE controller constantly scans each input latch in turn. When it finds a latch which is set, it gates the upper 64 bits of the packet, which contain the routing information, from the appropriate input register, onto the routing bus and into an address latch within the controller.
- The CE controller applies a routing algorithm to determine which output register

the packet is to be sent to. Simultaneously, the lower 64 bits of the packet are gated onto the routing bus.

- The controller waits until the appropriate output register is empty (the output latch is reset).
- The lower bits of the packet are gated into the bottom of the appropriate output register. The upper half of the packet is then gated into the top of the register.
- The input latch is reset to show that the input register is empty. The output latch is set to show that the output register is full.

The packet transfer speed is determined by the CE's clock rate; the CE's multiplexer can combine an input register poll and a route operation in a single clock cycle. As each CE has five input registers, any packet can be routed in 5 clock ticks, ignoring collisions. A futuristic 2 ns clock rate will, therefore, result in a 10 ns hop time.

5.4.2 The Payload Element

When a packet arrives at its destination, the CE controller loads it into the payload's input register and sets the input latch. When the payload produces output, it loads its output register and sets the output latch.

In COBWEB, there are two types of payload cell:

- Processor and memory cells (PMCs)

  Most of the payload cells in the wafer will be of this type. Each contains a processor and some local memory. The processors are microcoded to perform graph reduction. For fault tolerance, the memory is divided into 512 byte pages, each with its own associated control logic. The memory is accessed as a heap, with defective pages being omitted from the free list.

- Input and output cells (IOCs)

  COBWEB is unable to use fibre optic communications because whole wafer integration does not allow different technologies to be mixed on a single wafer.

  IOCs are used to interface the wafer to the pins of its packaging. They contain parallel to serial converters, for reducing the number of pins required, pads and drivers. Each IOC has 32 data, two signal and two power pins. Limitations in current pin bonding technology dictate that the IOCs must be located around the edge of the wafer.

5.4.3 Manufacturing and Packaging

COBWEB wafers would be fabricated by lithographing the two types of cell (PMCs and IOCs) onto a square grid using conventional wafer stepping. The wafers would be manufactured with an excess of IOCs to ensure that all of the pins can be accommodated.

Defective CEs may cause a wafer to have a number of disjoint cell networks. The largest network of working CEs is determined. If the size of this fails to exceed a threshold value, the wafer is discarded. Power is disconnected from the cells which contain short circuits[7] by electronically blowing fuses placed at the junctions of the power grid.

Serviceable wafers would be packaged, their pins being wired to working IOCs. IOCs in excess of the pin sets are ignored. Post-production testing is used to verify that the wafer pins function correctly. Failing wafers are discarded. This guarantees that every completed wafer has a full complement of working pins and a minimum level of functionality.

5.4.4 Packet Routing

COBWEB uses packet switched communication. When a new packet arrives at a CE that is not its destination, it is forwarded to one of its neighbours according to a routing algorithm. As packets must be routed around damaged areas on the wafer, routing will differ from wafer to wafer according to their defect distributions. The routing algorithm used in COBWEB determines its performance, cost and yield.

Routing is implemented using signposts. A signpost is a two-bit number which informs the CE where to route the packets for a particular destination. Each CE contains a small amount of memory which is used to hold arrays of signposts. Packets contain their destination addresses. These are used as indexes into the signpost arrays. Figure 5.10 illustrates signpost routing.

There are two main advantages of using signposts:

- Graceful degradation

  If a damaged node routes a packet incorrectly, there is still a good chance that it will eventually arrive at its destination.

- Dynamic routing

  Routes can be altered dynamically so that packets are routed around communication hot-spots or CEs that fail suddenly. Furthermore, the routing information on the wafer need not always be consistent; packets can become temporarily lost without harm as long as all the signposts are ultimately consistent.

In his PhD thesis, Anderson [And90a] suggests that a packet-switched wafer communications network should employ two routing algorithms side-by-side. The Default algorithm is fast and efficient, but can result in deadlocks. The Chain algorithm [Ros86] is slower, but will resolve deadlocks when they occur. Each CE holds two sets of signposts, one for each routing algorithm. In a 316 cell wafer each CE requires 1,264 bits to hold both routing arrays. These must be generated and downloaded to the CEs upon wafer initialisation.

- The Default routing algorithm

[7]Short circuits are detected by applying power to various test pads on the wafer and comparing the actual and expected power drains at those points. The circuits are powered at a fraction of their normal operating voltages to avoid causing additional damage.

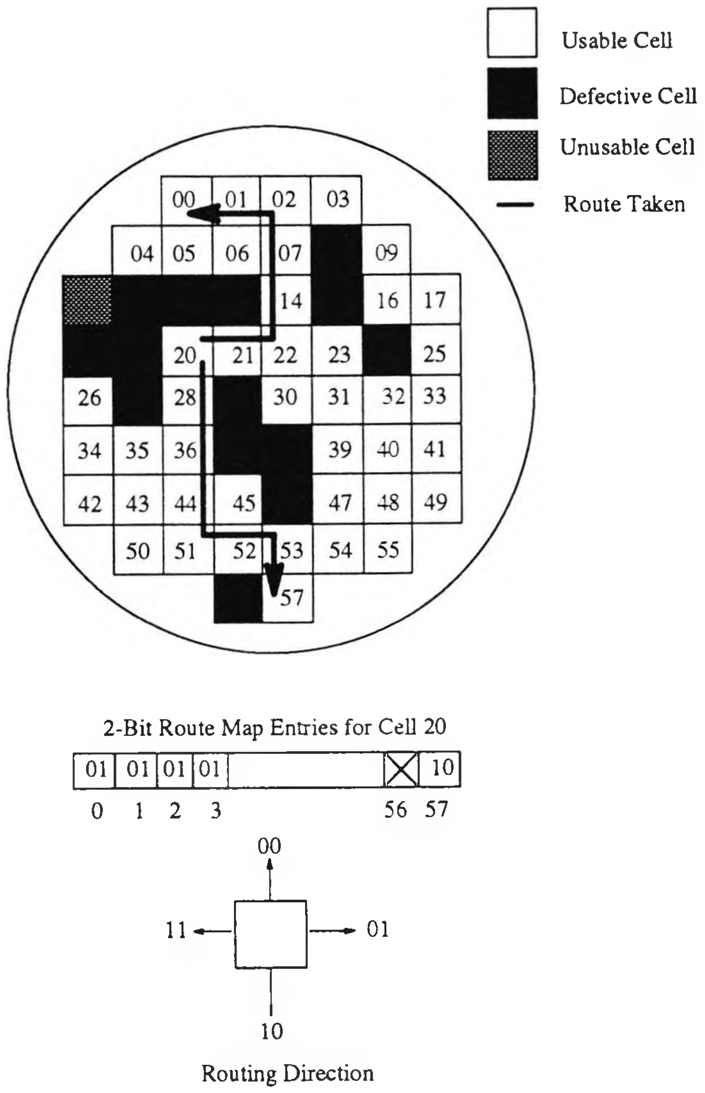


Figure 5.10: Using Signpost Routing to Avoid Wafer Defects

In the Default routing algorithm packets travel to their destination by the most direct route possible. This may involve routing packets around damaged regions of the wafer as illustrated in figure 5.10.

- The Chain routing algorithm

  The Default algorithm is not deadlock free. Simulation shows that it is prone to deadlock whenever the number of packets in flight approaches one per CE. Figure 5.11 shows how deadlocks can occur in a wafer.

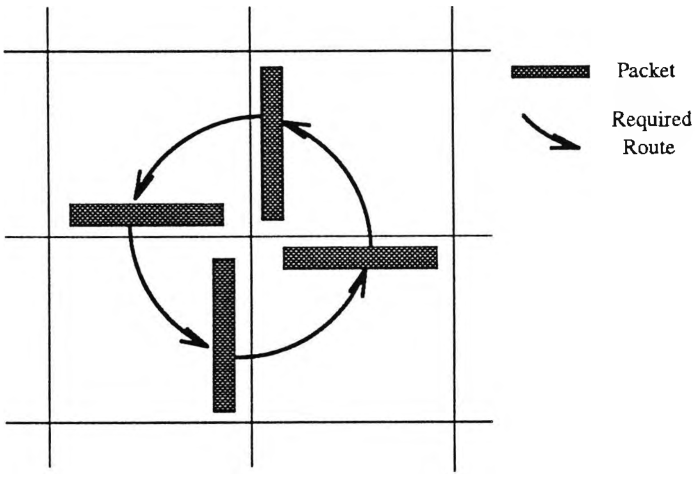


Figure 5.11: An Example Deadlock in a Wafer

  The communication network normally operates according to the Default algorithm, but when deadlock is suspected, it reverts to the Chain algorithm which guarantees to deliver all the packets in the wafer within a minimum number of hops, k, called the Chain Delay.

  When a CE detects that it is contributing to a deadlock situation (when a communication channel's countdown timer expires), it lowers a signal line which places the whole network into the Chain mode. The CE remains in this mode until it has waited for k hops to elapse. It then reverts to the Default mode and raises the signal line. While the wafer is in the Chain mode, its IOCs are prevented from accepting further packets.

  In the Chain mode, the communication network is configured into an endless loop. If there are n working CEs on a wafer, then a packet can be delivered to its destination in a maximum of n hops. To be safe, the Chain Delay is set to a value somewhat greater than n. This allows all of the CEs to empty their input registers.

One point not addressed by Anderson, however, is that in order to clear the deadlock, the Chain algorithm effectively re–orders the packets in the network. Therefore, whenever a wafer enters deadlock, the order in which packets are delivered can no longer be guaranteed. This has an important impact upon the communication protocols which must employed by the processors. In extreme cases this can severely reduce the usable communications bandwidth.

5.4.5 Yield And Harvest Predictions

A gate equivalence scheme has been used to compute the area of a CE using very conservative 1.5 micron technology. The values obtained are shown in table 5.1. The majority of the wafer area is consumed by memory and power rails.

Structure	Area ($\times 10^6 \mu m^2$)	Gate Equivalent
Input and Output Registers	1.7	5,000
Input and Output Register Controllers	0.2	400
Routing Logic	0.5	1,000
Control Logic	0.9	2,000
Routing Bus	16.0	-
Power Supply	6.3	-
Total	25.6	-

Table 5.1: Areas of the COBWEB Cell Components using 1.5 Micron Fabrication

Given that the CE has an area of $25.6 \times 10^6 \mu^2$, its yield is predicted by the negative binomial model[8] to be 0.75 using: known logic fault rates of 0.03 defects/mm^2; metal fault rates of 0.01 defects/mm^2; and clustering parameters of 0.75 to show a high degree of defect locality.

The harvest of a wafer is defined as the proportion of the total CEs which can be configured into a connected network attached to the wafer's pads. If a wafer has several disjoint networks, the harvest is said to be the largest. COBWEB's fault tolerant architecture ensures that, given a reasonable CE yield, almost all working PEs are harvestable.

It is estimated that a eight inch COBWEB wafer constructed using 1.5 micron technology would have a diameter of 20 cells and would contain 260 PCs and 56 IOCs, 28 of which would be connected to pins. Simulations of such wafers have shown their harvests contain an average of 191 PCs and 38 IOCs. Furthermore, 98 per cent of the wafers will have a network consisting of 28 or more IOCs connected to at least 160 working PCs. The graph in figure 5.12 illustrates the harvest distribution in 100 typical wafers.

The relationship between CE yield and harvest size in 50 simulated wafers is illustrated in figure 5.13. It should be noted that yields below 30 per cent produce very poor harvests due to the lack of connectivity.

5.4.6 COBWEB Performance

Simulations have been used to determine the average communication path length between the IOCs and PCs on wafers with the yields obtained from the defect models. The results

[8]The yield Y according to a particular defect type j is given by the equation:

$$Y_j = (1 + \frac{D_j A}{\alpha_j})^{-\alpha_j}$$

where A is the circuit area, D_j is the density of defect j and α_j is the clustering parameter for defect j.

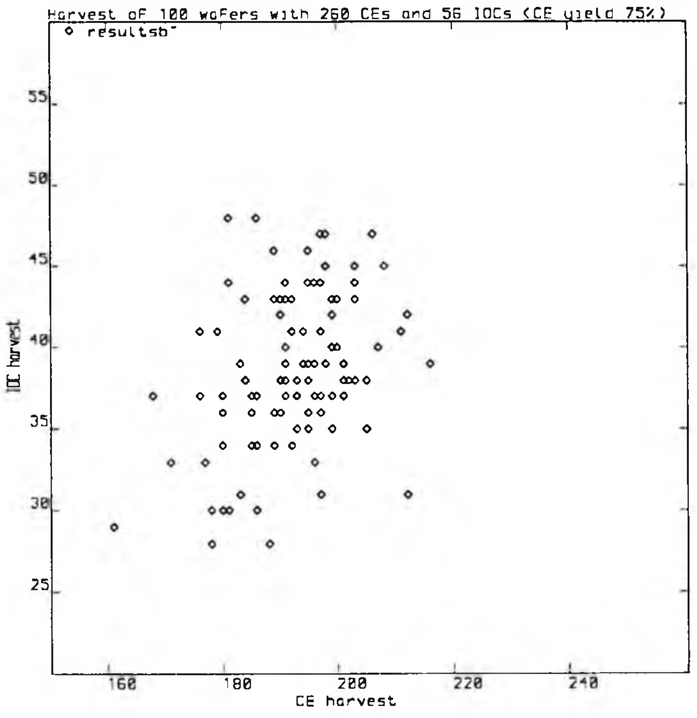


Figure 5.12: CE and IOC Harvest of 100 Wafers (260 CEs, 56 IOCs, CE yield 75%)

are illustrated in figure 5.14. The mean path length was found to be approximately 19. If each hop takes around 10 ns, the average latency of a packet will be around 190 ns.

The performance p of a packet routing can be expressed as the ratio between the length of the shortest possible path and that actually taken:

$$p = \frac{L_{ideal}}{L_{actual}}$$

Simulations have shown that performance deteriorates exponentially as the load on the network, the number of packets in flight at any one time, increases. This is illustrated in figure 5.15. However, under loads of less than 20 per cent, the performance is close to perfect.

5.4.7 Infeasibility of COBWEB

There were two main reasons why COBWEB machines were judged to be infeasible:

- To reduce costs and maximise performance it was intended that COBWEB wafers would be manufactured using whole wafer integration. Since it is hard to mix different technologies on a single wafer, fast processors and inexpensive DRAM could not be combined. Therefore, COBWEB would have to use static memory, which means that it suffers from a lack of storage capacity.

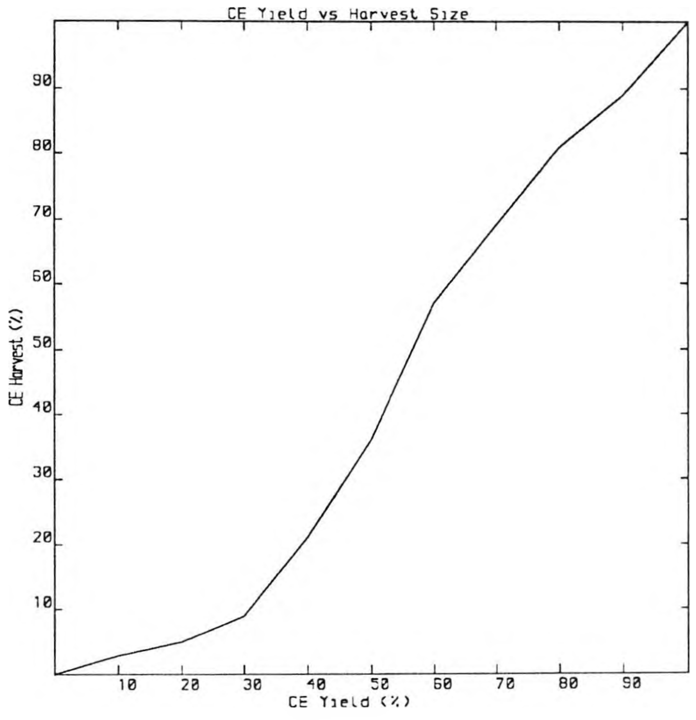


Figure 5.13: The Relationship Between CE Yield and Harvest in 50 Simulated Wafers

- COBWEB assumes pin mounting technology; fibre optics communication was either not envisaged or was discounted because of the whole wafer integration fabrication process. This results in a communication bottleneck because of the huge disparity between the internal bandwidth of the wafer and its I/O bandwidth.

The COBWEB project, however, did draw attention to the feasibility of creating an ultra high performance wafer–scale communications network. The technique is highly applicable to *Cherub*.

5.5 A Wafer–Scale Integrated Network For Cherub

The COBWEB project suggests how an ultra high performance communications network may be created using wafer-scale integration. Paper studies have shown that this architecture will provide a large degree of defect tolerance. However, the packet–switched communications network employed in COBWEB has several serious limitations:

- Interleaved packet delivery is possible. As a consequence, a complicated stateless communications protocol is required.
- The network can deadlock. When this happens the deadlock clearance algorithm results in unordered packet delivery, further complicating the communications pro-

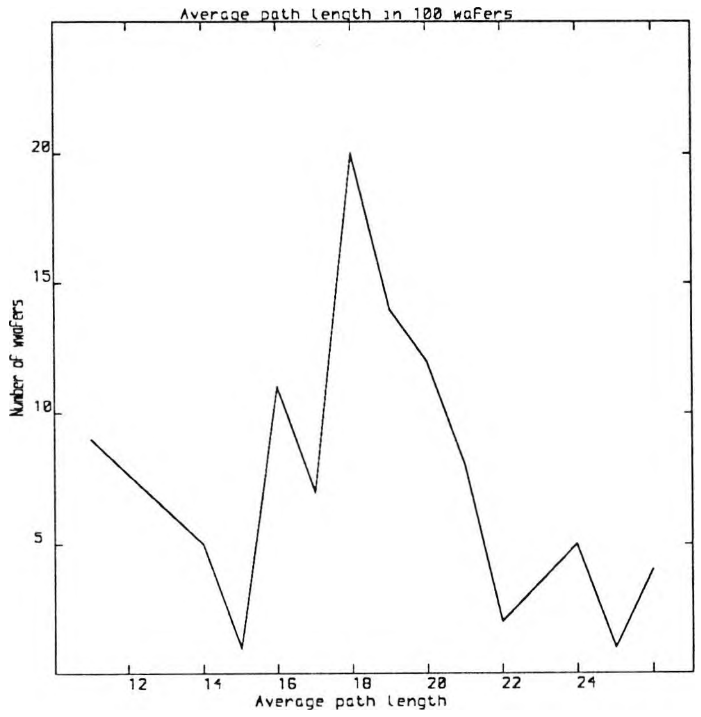


Figure 5.14: The Average Path Length in 100 Simulated Wafers

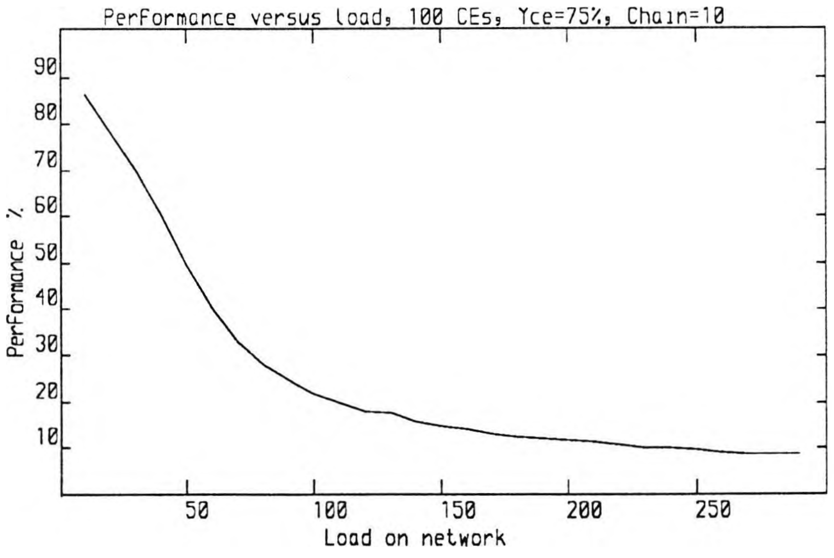


Figure 5.15: Simulated Network Performance Vs Load in 100-CE Wafers

tocol.

- The data–path is limited in width because a channel requires two separate register and bus pairs to support bi–directional communications.

- In order to route packets, each communications element scans its five input registers in turn. As a result, packets from a given source are only routed every fifth clock cycle. Therefore, very high clock rates must be employed if a high data bandwidth is to be achieved. This increases the cost of the circuitry involved.

Circuit–switching is an alternative communication mechanism which offers solutions to these problems and is easy to implement in silicon. This technology has been investigated in the MESHNET communications architecture [Win89]. This is a network built from Network Control Unit (NCU) [Win87] chips which perform circuit routing. Processor and memory pairs are connected to the network by Buffer Interface Controller (BIC) [Win88] chips. These relatively simple chips are responsible for performing circuit initiation and DMA data transfer.

The advantages of employing this communication scheme in *Cherub* include:

- It uses well–understood technology which has already been applied practically at City University.

- The communication protocol has been formally proved to be deadlock free by Whobery [Who88].

- When a circuit is established, exclusive communication with the CE is guaranteed during its lifetime. This guarantees atomicity and message delivery order.

- Once a circuit is established, it can be multiplexed, allowing low latency, high bandwidth two way communication.

The main disadvantage with circuit switching is that it requires complex, and hence silicon expensive, routing switches. This severely limits the width of the communications buses which can be employed on the wafer.

The proposed Cherub network will employ circuit switching because it is very easy to implement in silicon. It is asserted that a very wide circuit switched network can be constructed by using a hybrid wafer fabrication technique to bond tested processor, SRAM and optical fibre tiles onto a network wafer. The optical fibre tiles can be connected to mass storage devices, such as Wafer Disc[9] *and conventional disk drives, as illustrated in figure 5.16.*

It should be noted that the proposed *Cherub* wafer does not contain DRAM. As the processor's primary and secondary caches will be large enough to make main memory accesses relatively infrequent, the DRAM can be provided on a separate wafer. This helps minimise the size of the *Cherub* wafer, thus reducing the network diameter and minimising connection latency.

[9]One of the spinoffs from City University's COBWEB project was a proposal for a WSI based DRAM mass storage device called Wafer Disc [And89]. Logically, this is accessed as a very fast and reliable disk drive. However, it has two major differences: it is volatile; and it supports concurrent access via multiple optical fibre connectors.

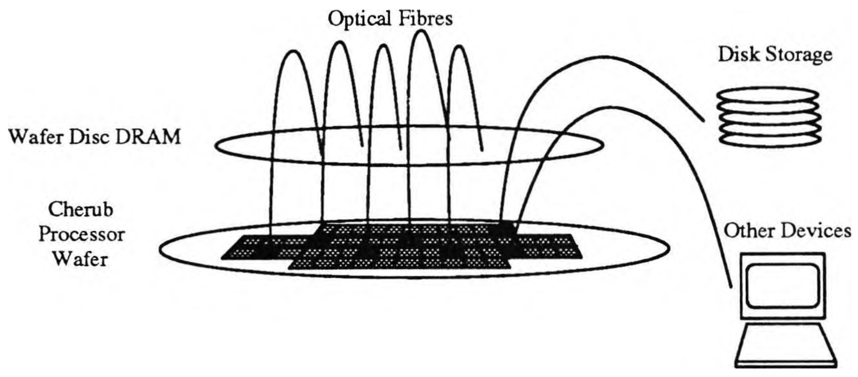


Figure 5.16: The Cherub Architecture

The proposed layout of a *Cherub* wafer is shown in figure 5.17. The raw wafer only contains NCUs and buses forming a communications network. Flip–chip bonding is used to mount tested processor, SRAM and fibre optic tiles, each with their own BIC, onto it. As the tiles are known to function correctly, less fault tolerance and post–fabrication testing is required than with COBWEB. The two layer wafer this forms, allows the construction of very wide — 256 bits is reasonable — network data–paths and switching circuits.

As with COBWEB, it is estimated that a eight inch wafer could reasonably hold around 316 one square cm tiles. Assuming a 75% communications network harvest, this gives 237 working tiles, which is enough to accommodate the required 200 processor and cache pairs, as well as a number of fibre optic tiles.

COBWEB employed a network with a mesh topology. The *Cherub* wafer contains a network with a torus–like topology, possibly requiring an extra layer of metal[10]. The interconnection scheme illustrated in figure 5.18 is used to balance the bus lengths. Although this results in an increased yield hazard, it substantially decreases the diameter of the network[11], thus increasing its performance. This is important with such a large network.

The estimated wafer surface area of the network components, using conservative 0.4 micron technology to increase yield, is shown in table 5.2. The width of the communication buses employed is limited by the dimensions of solder bumps; a processor connected to a 128–bit wide communications bus might reasonably have 225 signal lines (15×15), its solder bumps occupying 9 square mm of wafer surface. There is room to employ upto 256–bit wide communication buses in each direction. Due to a tradeoff between communication latency and bandwidth which will be explained later, it was decided to have two 128–bit communications buses in each direction. The nine resulting communication channels require a four bus crossbar switch.

The *Cherub* network was first proposed by the author in [Gul91].

[10] A maximum of three layers of metal can reasonably be used in a VLSI circuit. This is due to step coverage problems incurred in the multilayer metalisation process.

[11] The diameter of a torus network of width n nodes is $\frac{n}{2}$. The diameter of a mesh is $\frac{2n}{3}$. Thus a 20 node wide *Cherub* wafer has an diameter of 10, while a similarly sized mesh has one of 13.

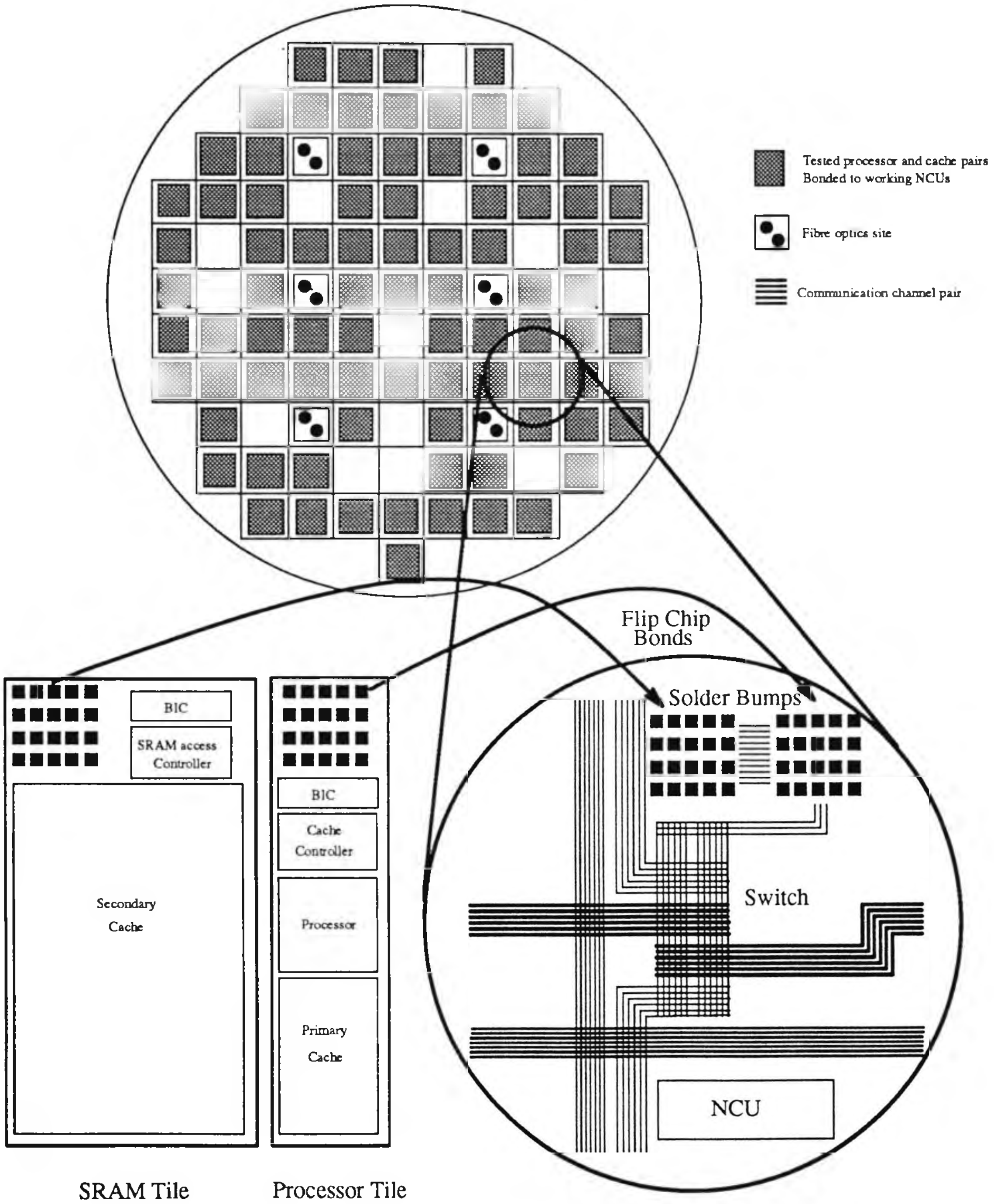


Figure 5.17: A Simplified Cherub Wafer

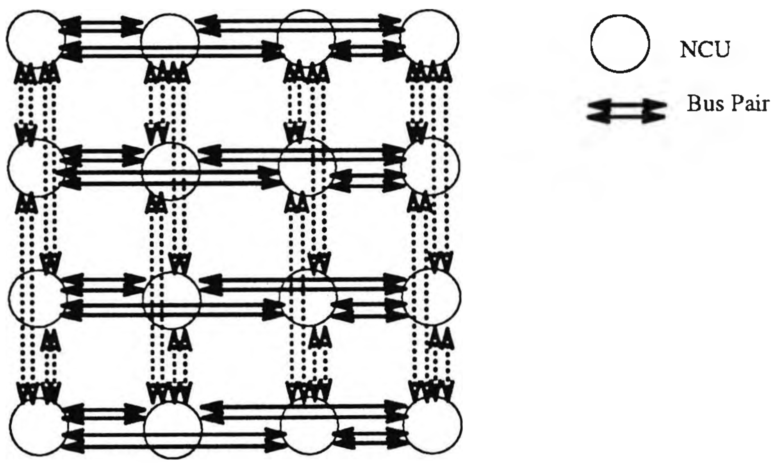


Figure 5.18: A Double Channel Cherub Network

Structure	Area ($\times 10^6 \mu m^2$)
2 × 128-bit Communication Channels	49.2
4-way × 128-bit switch	6.0
2 × 225 Solder Bumps	18.0
NCU	< 0.2
Total	< 73.4

Table 5.2: Areas of the Cherub Cell Components using 0.4 Micron Fabrication

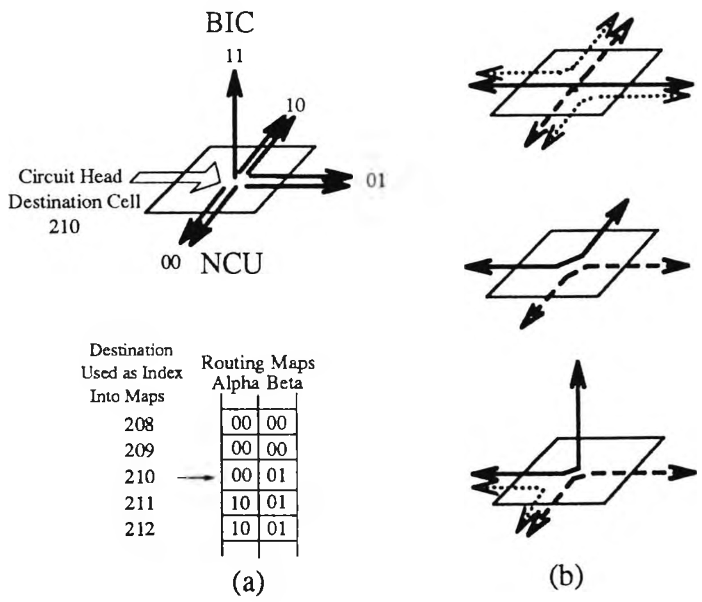
5.5.1 The Communication Element

The *Cherub* communication element differs significantly from that of COBWEB because it uses circuit, rather then packet, switching. The main component of the *Cherub* CE is a nine input, 128–bit wide, four–way crossbar switch. This allows up to four pairs of its nine inputs to be electronically connected at any one time. The switch is controlled by a NCU which makes the routing decisions.

Circuit–switched communication occurs in three stages: circuit construction, data transfer, and circuit collapse.

- During circuit construction an electronic link is established between the source and the destination BIC. The head of the circuit starts at the source BIC and contains the unique identification number of the destination node. Just as with COBWEB, at each hop this number is used as an index into routing maps held in SRAM. The two-bit result gives the direction, relative to that of travel, that the head must be routed in. The head of the circuit is passed to the NCU in this direction. This procedure is repeated until the circuit head reaches the destination BIC. This routing scheme is illustrated in figure 5.19a.

  In order to make an alternative routing decision when the head of a potential connection becomes blocked, two routing maps, α and β, are maintained for each destination. Thus, assuming a wafer has 256 destinations, each CE requires 1,024 bits of maps.



Destination Used as Index Into Maps	Routing Maps Alpha	Beta
208	00	00
209	00	00
210	00	01
211	10	01
212	10	01

Figure 5.19: Circuit–Switched Routing

Up to four circuits can be routed through each NCU as long as they use different channels. Several possible combinations are shown in figure 5.19b. If it is found that a circuit cannot be completed, it immediately collapses back to the source BIC, thereby freeing the allocated channels and avoiding deadlocks. The transmitting BIC then retries after waiting for a back-off period.

- Once a circuit has been established, data transfer can take place. The circuit appears to be an auto-simplex 128-bit wide parallel shift register. This is clocked by the source BIC at 100 MHz[12], giving a hop time of 10 ns and a raw data bandwidth of 1.5 Gbytes per second. The NUCs are clocked at 2 ns, allowing them to make a complete scan of their inputs in 10 ns.

- When the communication is complete, the source BIC dissolves the connection, which in turn causes circuit collapse. The collapse frees the allocated channels.

Employing twin 128-bit wide communication channels has the effect of sacrificing half of the network's bandwidth in order to improve its connection latency. A CE is allowed to use either of the channels travelling in a desired direction to route a circuit. This is illustrated in figure 5.20, in which node A communicates with node A′ and node B with B′ though two common channel pairs.

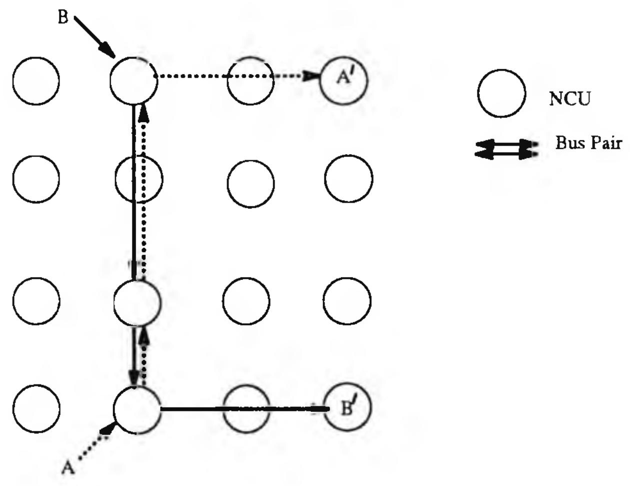


Figure 5.20: Routing in A Double Channel Network

One disadvantage of this double channel scheme is that a CE has to scan almost twice as many channels as on a COBWEB wafer. This doubles the hop time for a given clock rate. This problem can be overcome by scanning a channel pair on each clock cycle. The channels are prioritised so that if circuit heads appear on both channels simultaneously, the one on the highest priority channel is routed first. The second circuit is routed the next time the pair is scanned. A performance degradation is, therefore, only experienced after collisions.

When the wafer is booted, the wafer temporarily enters a configuration mode. Each NCU undergoes a self-test and generates a series of 128-bit test signatures. These are examined by an external test circuit which makes a connection to each NCU in turn. This is made possible by forcing NCUs which do not yet have routing tables to accept all incoming

[12]Slow enough to allow for the line rise and settle delays across the eight inch wafer.

connections. This allows the NCUs to be downloaded with temporary route maps, which enables them to route connections to their neighbours. When all of the NCUs have been examined, proper route maps are constructed which avoid the NCUs generating incorrect signatures. Once these have been downloaded the wafer can function normally.

5.5.2 The Payload Element

There are two possible types of payload element in the *Cherub* wafer:

- Processor and SRAM Tile Pairs

  Most of the *Cherub* wafer's surface is populated by processor and SRAM tile pairs. The processor tile consists of a high performance RISC processor, 512 Kbytes of primary cache, a cache controller and a network interface (BIC). The processors, having a 5 ns instruction cycle, are similar in performance to those described by Borg et al [Bor90]. The SRAM tiles contain 16 Mbytes of secondary cache.

- Input and Output Tiles

  These tiles are placed at regular intervals on the wafer. The payload contains bond sites for optical fibres and is used to interface the wafer to external devices. Each tile has two bond sites, allowing bi-directional communication. The tile also carries a network interface (BIC) and the high speed parallel to serial converters and drive circuitry required to transmit and receive data via the fibres.

5.6 Cherub Performance Simulation

Chapter four determined that, when loaded with 40 connections, the *Cherub* communications network must fulfill the following criterion:

$$84C + 26,000T \leq 40,000\,\text{ns}$$

Where:

$$\begin{aligned} C &= \text{Time (ns) to make a network connection (network connection latency); and} \\ T &= \text{Time (ns) to transfer a byte across a network connection.} \end{aligned}$$

The proposed wafer-scale communications network is able to transfer 128 bits in 10 ns. Therefore, for the 10,000 instruction level of granularity to be optimal on the hardware, the network connection latency, C, must be less then about 280 ns.

The performance of eight different circuit switched routing algorithms have been simulated. The results are illustrated in figures 5.21 and 5.22. The simulations assume a 316 node torus network with an average message length of 19 clocks and a constant back-off period of 6 ± 1 clocks[13]. A 100% network yield was assumed. The routing algorithms used in the simulation are explained in appendix C.

[13] Approximately one quarter the average message length: an estimate based on the assumption that, on average, a collision will occur when a connection is half constructed, being blocked by another connection which is half way through its transmission.

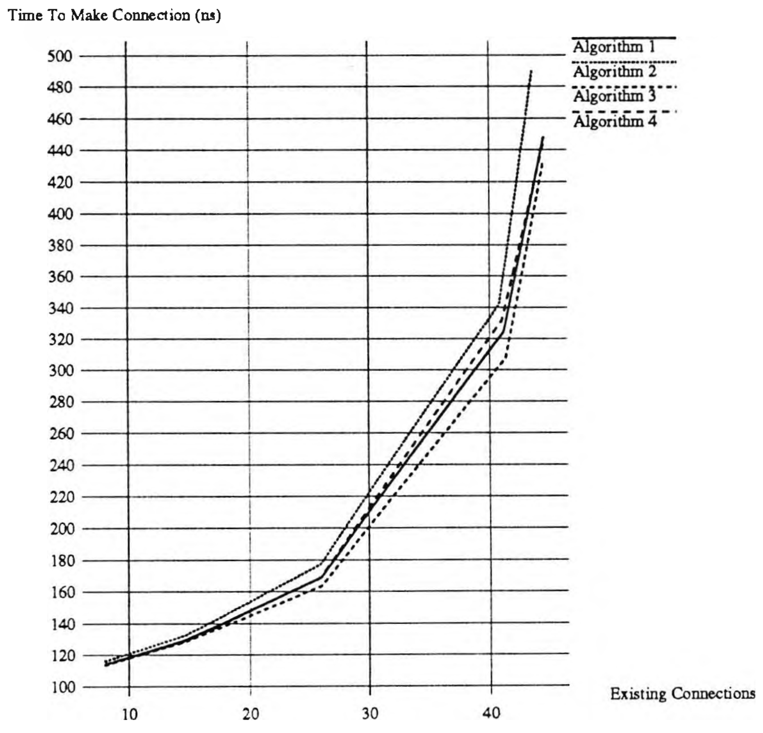


Figure 5.21: The Effect of Routing Algorithms 1-4 on the Time to Make a New Connection

The graphs show that only routing algorithm eight is able to achieve a connection latency below 280 ns in a network loaded with 40 connections. It should be noted that the shapes of the curves indicate that the communications network is being pushed to its limits, and is on the brink of thrashing. This is not unreasonable, however, as it is always desirable to employ the smallest level of granularity that the communications network is able to support.

Load simulations of routing algorithm eight on *Cherub* wafers with different communication network yields produced the results shown in figure 5.23. Once again, a constant back-off period of 6 ± 1 network clocks was employed upon collisions.

The graph shows that the yield of the communications network dramatically affects its performance. In general, the higher a network's communications yield, the lower its connection latency will be; communication bottlenecks occur where connections are routed around defects. Very high yields indeed are required, certainly above 90%, if *Cherub's* network is to achieve the desired latency. Due to the in-built redundancy of the proposed double bus system and the planned use of conservative 0.4 micron fabrication technology, it is expected that such a yield can be achieved.

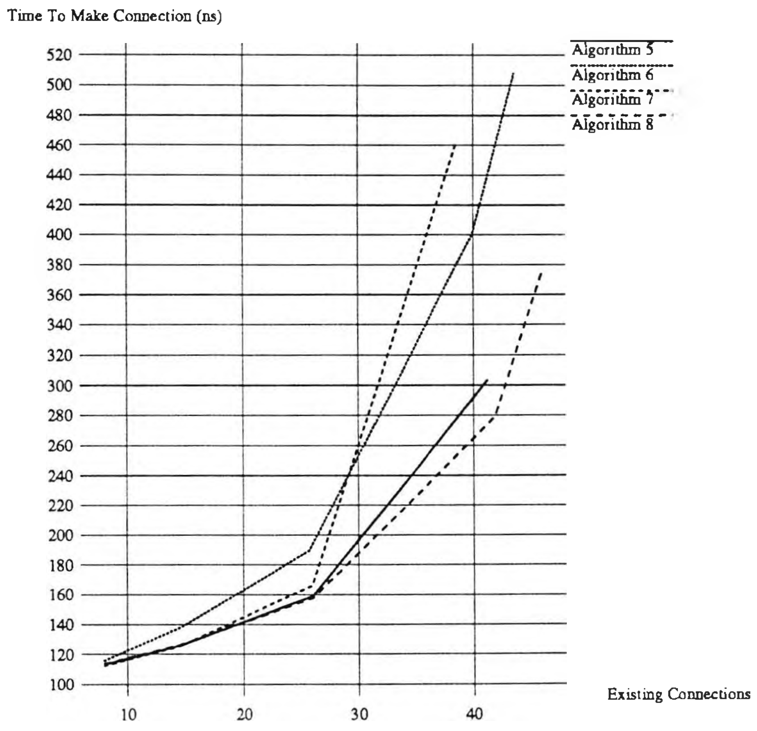


Figure 5.22: The Effect of Routing Algorithms 5-8 on the Time to Make a New Connection

5.7 Conclusion

This chapter examined the requirements of the *Cherub* communications network. It was asserted that the level of performance required could only be achieved through the use of techniques such as wafer scale integration.

The COBWEB architecture was examined. This allows processors to be combined with a packet-switched communications network though the use of whole wafer integration. The architecture achieves high performance, but suffers from the inability to combine state-of-the-art processor and memory technologies. In addition, packet-switching allows unordered and interspersed packet delivery.

It was decided that a large hybrid wafer-scale network would provide both the level of performance and the ability to combine the different VLSI technologies required. The wafer would employ a double channel circuit-switched communications network with a torus-like topology. This would have the benefits of a low connection latency, a high data bandwidth and simple communication semantics.

As the processor's primary and secondary caches would be large enough to make main memory accesses relatively infrequent, it was decided to provide DRAM on a separate wafer. This helps minimise the size of the *Cherub* wafer, thus reducing the network diameter and minimising connection latency.

The proposed network was simulated and found to be able to provide the level of perfor-

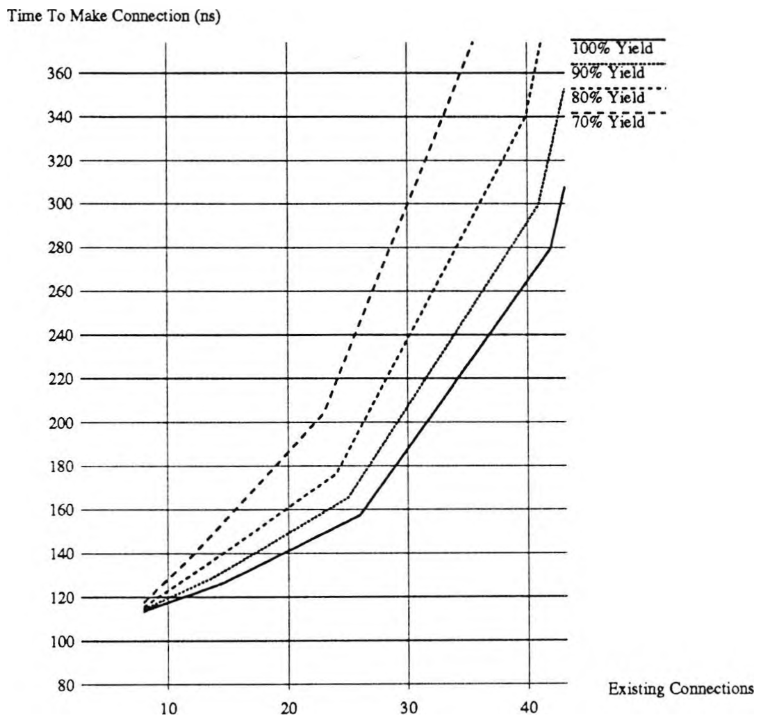


Figure 5.23: Effect of Network Yield on Connection Latency

mance required, given that a network yield in excess of 90% could be achieved. It was asserted that the fault tolerant design of *Cherub's* network would make this possible.

Chapter 6

Conclusion

6.1 The Thesis

Typically, most parallel architectures are either multiprocessors — which are able to support a few fine-grained (10^3 instructions) parallel tasks — or multicomputers — which can support hundreds of coarse-grained (10^5 instructions) ones; there is currently no middle ground.

The thesis of this dissertation asserted that there is a significant number of applications for which a new type of parallel architecture is desirable. Ideally, this will combine:

- a medium granularity (10^4 instructions) of processing;
- provision for up to several hundred parallel tasks; and
- the programmability of the shared variable parallel programming paradigm.

The dissertation described the design of the *Cherub* architecture, which has these properties, and demonstrated its usefulness with a large example, airborne early warning.

6.2 Proving the Thesis

Due to its scalability, it was decided to employ a multicomputer as the basis for the *Cherub* architecture. Shared variable programming paradigms are typically implemented on multicomputers using a mechanism called distributed shared memory. DSM potentially offers a scalable shared memory architecture, but as the message handling latency in multicomputers is high, DSM typically only supports coarse-grained parallelism effectively. The latency of the intertask communication and DSM mechanisms must, therefore, be reduced if a medium-grain of parallelism is to be supported.

Intertask communication latencies are composed of two significant components:

- Software latency
- Overall network latency (a combination of its latency and bandwidth)

The dissertation, therefore, suggested the combination of two approaches for reducing communication latency:

- **Implementing Intertask Communication Mechanisms in Hardware**

  A significant proportion of the intertask communication latency is due to the software used to implement the mechanisms. This can be reduced by implementing them in hardware. Unfortunately, in conventional operating systems these mechanisms are very complex and, hence, are difficult to implement in hardware. They must, therefore, be simplified.

  The dissertation suggested that the intertask communication mechanisms can be simplified by combining a hardware distributed shared memory (HDSM) with a single shared address space (SSAS). The dissertation showed how this combination allows the unification of many traditional operating system mechanisms.

  The dissertation described the design of an operating system which provides a single 64–bit address space, called the Object Space, which is shared by all of the processes. Objects are fixed size regions within the Object Space which are accessed like conventional memory. Objects have the following properties:

  - An object is named by an unique *global_name*, which is its start address within the Object Space.
  - Objects can have differing access semantics. The dissertation demonstrated that only six types of object (*memory*, *process*, *sleep-wakeup*, *semaphore*, *rendezvous* and *hardware*) need to be supported to provide the functionality of a traditional operating system. As a consequence of this unification, only three system calls (*create_object*, *destroy_object* and *object_info*) are needed to administer the Object Space.
  - Memory objects only support strong data coherence, because of its well understood programming semantics. However, as this coherence scheme can often perform unnecessary work, the architecture provides special instructions (*busy_read*, *busy_read_write*, *idle* and *finish*) which allow the programmer to give clues to the memory system about intended future data use. The thesis shows how careful use of these instructions allows the latency of the process life–cycle to be significantly reduced.
  - The dissertation suggested that objects can be protected by passwords called *capabilities*. Each object has three *capabilities* (*read*, *write* and *execute*). Each process has a number of *protection_domain_registers*. An object can only be accessed after its details — its *global_name* and the appropriate *capabilities* — have been loaded into one of these registers. It is thought that 64 *protection_domain_registers* per process will be sufficient for most applications.

The dissertation implemented the object and system call mechanisms on paper to show that they are simple enough to be constructed in hardware. This also allowed the performance required from the underlying network to be determined.

- **Improving the Latency and Bandwidth of the Underlying Network**

  Current state-of-the-art networks, typically fibre optics based, can provide high bandwidth communications, but appropriately low latency switching elements are not available. The dissertation, therefore, suggested constructing a network using wafer-scale integration. It is asserted that such a network will be able to combine low latency switching circuitry with wide, and hence high-bandwidth, data paths.

  The dissertation suggested that circuit switched routing be employed in the network, because of its high guaranteed bandwidth. A new technique is proposed for the construction of very wide circuit switched communications networks using hybrid wafer-scale integration. The main limitation in such a network is the physical constraint of the line rise time across the network.

  Simulations were used to show that, provided that the network yield exceeds 90%, a sufficiently low connection latency can be achieved. It was asserted that the fault tolerant design of *Cherub's* network would make this possible.

6.3 Contribution to Knowledge

This dissertation has made two significant contributions to computer science:

- It has designed a new HDSSASMA based operating system which is simple enough to be implemented in hardware, yet sufficiently comprehensive to provide all of the functionality of a conventional operating system like UNIX.
- It has proposed a new wafer-scale integration technique which allows the construction of very wide circuit switched networks.

6.4 Future Work

Two main ways forward can be envisaged:

- Implementing *Cherub's* Operating System

  The *Cherub* operating system could be constructed in hardware to prove that it is easy to implement. A *Cherub* cache controller can be prototyped in a reprogrammable device such a Xilinx programmable gate array (PGA) [Xil91]. A processor which can support an external secondary cache controller, such as the DEC Alpha [Dig92], is therefore required.

- Implementing *Cherub's* Communications Network

  Unfortunately, it is not realistic to expect to be able to implement the proposed wafer-scale integrated network. This needs the backing of a commercial sponsor with wafer fabrication facilities. However, fibre optics technology could be used to construct a network with the required communications bandwidth, if not the latency. The communication elements could be prototyped using Xilinx programmable gate arrays [Xil91] and connected by optical fibres with gallium–arsenide driving circuitry.

The network requirements derived in section 4.5 can be used to estimate the process granularity that can be supported by such a network. For example, if the fibre optic network has a bandwidth of 1–Gbit per second and a connection latency of 1 μs, then:

$$134A + 84C + 25,584T \leq (g \cdot \ln 2 - 550)I$$

Given that:
I = Time to execute an instruction = (say) 10 ns;
A = Time to access a cache line = (say) 5 ns;
C = Time to make a network connection = 1,000 ns; and
T = Time to transfer one byte = 8 ns ($\approx$ 1 bit per ns)

Rearranging this gives:

$$g \approx 42,000$$

Thus, such a network is able to support a process granularity of around 42,000 instructions.

6.5 Concluding Remarks

In one way, the designers of parallel computers have been fighting against the flow of technology, for the performance of processors has improved much faster then that of networks. With the growing popularity of fibre optics networks, communications technology is now approaching the limitations imposed by the speed of light. As a consequence, the optimal granularity of parallel processing can only be expected to increase with time.

In addressing a related problem, this dissertation has suggested a number of techniques which can reduce the overall latency of intertask communication. It is asserted that the use of such techniques will become commonplace as the limits of network technology are approached.

Bibliography

[Acc86] Accetta M. et al. "MACH: A New Kernel Foundation for UNIX Development". In *Proceedings of the Summer USENIX Technical Conference*, pages 64–75, June 1986.

[Ahu88] Ahuja S. et al. "Matching Language and Hardware for Parallel Computation in the Linda Machine". *IEEE Transactions on Computers. Vol. 37, No. 8*, pages 921–929, August 1988.

[And89] Anderson P. et al. "Wafer Disk: A New Systems Architecture Component". Technical Report, Department of Computer Science, City University, April 1989.

[And90a] Anderson P. *Computer Architecture for WSI.* PhD thesis, Department of Computer Science; City University, December 1990.

[And90b] Anderson P. et al. "The Feasibility of a General–purpose Parallel Computer using WSI". Technical Report, Computer Science Department; City University, 1990.

[And91a] Anderson M. "RAID 5 architecture provides economical failsafe disk storage". *EDN (USA)*, 36(12):141–143, 6 June 1991.

[And91b] Anderson P. et al. "Implementation of Paragon Specifications". Technical Report, City University, 1991.

[Ano89] Anonymous. "Whole stack, not wafer–thin". *Workstation*, page 8, December 1989.

[Arm86] Armand F. et al. "Towards a Distributed UNIX System – The Chorus Approach". In *Proceedings of the EUUG Autumn Conference*, pages 413–431, Autumn 1986.

[Aub78] Aubusson R. & Catt I. "Wafer–Scale Integration – A Fault–Tolerant Procedure". *IEEE Journal of Solid State Circuits; Vol. SC–13*, June 1978.

[Aub91] Aubusson R.C. "Wafer–Scale Integration – An International Perspective". In *Computing and Control Division Colloquium on Wafer Scale Integration*, pages 1/1—1/4, 1991.

[Bac86] Bach M.J. *The Design of the UNIX Operating System.* Prentice–Hall International Editions, 1986.

[Bac88a] Bach M.J. & Gomes R. "Measuring File System Activity in the UNIX System". In *Proceedings of the Spring EUUG Conference*, pages 43–52, 1988.

[Bac88b] Bache R.A.C. et al. "Bond design and alignment in flip chip solder bonding". In *Proceedings of 8th IEPS Conference*, 1988.

[Bar86] Barak A. & Paradise O.G. "MOS - A Load-Balancing UNIX". In *Proceedings of EUUG Autumn Conference*, pages 273–280, 1986.

[bbn86] "Butterfly Parallel Processor Overview". Technical Report, BBN Laboratories, June 1986.

[Bel88] Bell M. & Clarke B. "Research into Air Traffic Tools". Technical Report, Cambridge Consultants Ltd, February 15th 1988.

[Bel92] Bell G. "Ultracomputers: A Teraflop Before its Time". *Communications of the ACM*, 35(8):27–47, 1992.

[Ben90] Bennet J.K. et al. "Adaptive Software Cache Management for Distributed Shared Memory Architectures". *IEEE Ch2887 8/90/000/0125*, 1990.

[Ber88] Bernabeu-Auban et al. "Clouds - A Distributed, Object Based Operating System Architecture and Kernel Implementation". In *Proceedings of the EUUG Autumn Conference*, pages 25–37, October 1988.

[Ber91] Bernard G. "A Decentralised and Efficient Algorithm for Load Sharing in Networks of Workstations". In *Proceedings of EurOpen Spring Conference*, pages 139–148, 1991.

[Bis90] Bisiani R. & Ravishankar M. "PLUS: A Distributed Shared-Memory System". *17th Annual International Symposium On Computer Architecture*, 1990.

[Bol] Bolosky W.J. & Scott M.L. "A Trace-Based Comparison of Shared Memory Multiprocessors using Optimal Off-Line Analysis". Technical Report, Department of Computer Science; University of Rochester.

[Bor88] Borrmann L. et al. "Tuple Space Integrated into Modula-2: Implementation of the Linda Concept in a Hierarchical Multiprocessor. English translation from German". In *Proceedings 10. GI/ITG-Fachtagung Architektur und Betrieb von Rechensystemen; Paderborn;*, March 1988.

[Bor90] Borg A. et al. "Generation and Analysis of Very Long Address Traces". In *Proceedings of the 17th Annual International Symposium on Computer Architecture (Cat. No.90CH2887-8)*, pages 270–279, 1990.

[Bro82] Brookner E. "Developments in Digital Radar Processing". *Trends and Perspectives in Signal Processing*, pages 7–23, January 1982.

[Bug90] Bugge H.O. et al. "Trace-Driven Simulations For a Two-Level Cache Design of Open Bus Systems". In *Proceedings of the 17th Annual International Symposium on Computer Architecture (Cat. No.90CH2887-8)*, pages 250–259, 1990.

[Bur92] Burkhardt H. III. "Announcing the KSR1 Supercomputer". Press Release: comp.arch newsgroup, Feb 1992.

[Cam87] Campbell R. et al. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *Operating Systems Review*, 21(3):9–17, July 1987.

[Car86a] Carlson R.O. & Neugebauer C.A. "Future Trends in Wafer Scale Integration". In *Proceedings of the IEEE; Vol. 74, No. 12*, pages 1741–1752, December 1986.

[Car86b] Carriero N. & Gelernter D. "The S/Net's Linda Kernel". *ACM Transactions on Computer Systems, Vol. 4, No. 2*, pages 110–129, May 1986.

[Car86c] Carrington T. "Nimrod Lasted Despite Problems, Say the U.S. Proponents of AWACS". *The Wall Street Journal (WSJ861218-0045)*, 18th December 1986.

[Car89] Carriero N. & Gelernter D. "How to Write Parallel Programs: A Guide to the Perplexed". *ACM Computing Surveys; Vol. 21, No.3*, pages 323–357, September 1989.

[Cat89] Catt I. "The Kernel Logic Machine". *Electronics and Wireless World*, pages 254–295, March 1989.

[Cat91] Catt I. "From Spiral to Kernel". *IEE Computing and Control Division Colloquium on Wafer Scale Integration*, pages 5/1–5/3, Tuesday 28th May 1991.

[Cha] Chaves E.M. Jr. et al. "Kernel–Kernel Communication in a Shared–Memory Multiprocessor". Technical Report, Department of Computer Science; University of Rochester.

[Cha92a] Chase J.S. et al. "How to Use a 64–Bit Virtual Address Space". Technical Report 92–03–02, Department of Computer Science and Engineering; Univerisity of Washington, March 1992.

[Cha92b] Chase J.S. et al. "Lightweight Shared Objects in a 64–Bit Operating System". Technical Report 92–03–09, Department of Computer Science and Engineering; Univerisity of Washington, March 1992.

[Che87] Chesley G. "Addressable WSI: A non–redundant approach". *Computer Architecture News; Vol. 15, No. 1*, pages 73–80, March 1987.

[Che88a] Cheriton D.R. "The V Distributed System". *Communications of the ACM*, 31(3):314–333, March 1988.

[Che88b] Chesley G. "Comment on 'Future Trends in Wafer Scale Integration'". In *Proceedings of the IEEE; Vol. 76, No. 3*, pages 283–284, March 1988.

[Cli76] Cline H.E. & Anthony T.R. "Thermomigration of Aluminium–Rich Liquid Wires Through Silicon". *Journal of Applied Physics*, 47:2332–2336, 1976.

[Cor65] Corbato F.J. & Vyssotsky V.A. "Introduction and Overview of the MULTICS System". In *Proceedings of AFIPS Fall Joint Computer Conference*, pages 185–196, 1965.

[Cun90] Cunningham J.A. "The Use and Evaluation of Yield Models in Integrated Circuit Manufacturing". *IEEE Transactions On Semiconductor Manufacturing; Vol. 3, No. 2*, pages 60–71, May 1990.

[Cur89] Curran L. "Wafer–scale integration arrives in 'disk' form". *Electronic Design; Vol. 37, No. 22*, pages 51–54, October 1989.

[Dal87] Dally W.J. "A VLSI Architecture for Concurrent Data Structures". *Massachusetts Institute of Technology; Kluwer Academic Publishers*, pages 137–143, 1987.

[Dal89] Dally W. et al. *The J-Machine: A Fine-Grain Concurrent Computer*, pages 1147–1153. Elsevier, 1989.

[Das88] Dasgupta P. et al. "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work". In *International Conference on Distributed Computing Systems, IEEE*, 1988.

[Del69] Delong D.F. & Hofsletter E.M. "The Design of Clutter–Resistant Radar Waveform with Limited Dynamic Range". *IEEE Transactions on Information Theory; IT–15(3)*, pages 376–385, 1969.

[Del86a] Delp G. & Farber D. "MemNet: An Experiment on High–Speed Memory Mapped Network Interface". Technical Report 85–11–1R, University of Delaware; Computer Science Department, 1986.

[Del86b] Delp G. & Farber D. "MemNet: An Experiment on High–Speed Memory Mapped Network Interface". Technical Report 85–11–1R, Computer Science Department; University of Delaware, 1986.

[Del88a] Delp G. *The Architecture and Implementation of MemNet: A High Speed Shared Memory Computer Communication Network.* PhD thesis, University of Delaware; Computer Science Department, 1988.

[Del88b] Delp G. *The Architecture and Implementation of MemNet: A High Speed Shared Memory Computer Communication Network.* PhD thesis, University of Delaware; Computer Science Department, 1988.

[Die89] Diepers, H. "Key–parameters of vertical magnetic recording". In *Proceedings of VLSI and Computer Peripherals - VLSI and Microelectronic Applications in Intelligent Peripherals and their Interconnection Networks (Cat. No.89CH2704-5)*, pages 1/79–84, 1989.

[Dig92] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[Dij65] Dijkstra E.W. "Co-operating Sequential Processes". *Programming Languages; Genuys, F. (ed.). London: Academic Press*, 1965.

[Don91] Dongarra J.J & Duff I.S. "Advanced Architecture Computers". Technical Report, 1st Mathematics & Computer Science Division; Argonne National Laboratory, 1991.

[Dou89] Douglas F. "Experience with Process Migration in Sprite". In *Proceedings of USENIX Workshop on Distributed and Multiprocessor Systems*, pages 59–72, 1989.

[Eag84] Eager D.L. et al. "Dynamic Load Sharing in Homogeneous Distributed Systems". Technical Report, University of Washington, October 1984.

[Far85] Farina A. & Studer F.A. *Radar Data Processing ; Volume 1 – Introduction and Tracking.* Research Studies Press Ltd, 1985.

[Fel79] Feldman S. "Make – A Program for Maintaining Computer Programs". *Software – Practice and Experience*, April 1979.

[Fel89] Feldman S. & Brown C. "IGOR: A System for Program Debugging via Reversible Execution". In *ACM SIGPLAN/SIGOPS: Proceedings of the Workshop on Parallel and Distributed Debugging, May 1988*, volume 24, number 1, pages 112–123, January 1989.

[fib92] "Fibre Channel: Physical and Signaling Interface (FC–PH)". Technical Report, American National Standard for Information Systems, June 1992.

[Fis73] Fishman G.S. *Concepts and Methods in Discrete Event Digital Simulation.* John Wiley and Sons, Inc., 1973.

[Fle89a] Fleisch B.D. "Distributed Shared Memory in a Loosely Coupled Environment". *University of California, Los Angeles*, July 1989.

[Fle89b] Fleisch B.D & Popek G.J. "Mirage: A Coherent Distributed Shared Memory Design". *Operating Systems Review*, December 1989.

[Fly66] Flynn M.J. "Very High-Speed Computing Systems". *Proceedings of the IEEE*, 54:1901–1909, 1966.

[For89] Forin A. et al. "The Shared Memory Server". In *Proceedings of the Winter USENIX Technical Conference*, pages 229–243, 1989.

[Fow86] Fowler R.J. *Decentralised Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington; Department of Computer Science and Engineering, 1986.

[Gel89] Gelernter D. "Multiple Tuple Spaces In Linda". *Lecture Notes in Computer Science; Parallel Architecture and Languages; Europe*, pages 20–27, 1989.

[Geo90] Georgescu I. "Communication in Parallel Processing Systems". *Studies and Researches in Computers and Informatics; Vol. 1; No. 1*, March 1990.

[Gib89] Gibson G.A. et al. "Failure Correction Techniques For Large Disk Arrays". In *Proceedings of the 16th Annual International Symposium on Computer Architecture (Cat. No.89CH2705-2)*, pages 123–132, April 1989.

[Gin90] Ginosar R. & Mitchell N. "On the Potential of Asynchronous Pipelined Processors". Technical Report, VLSI Systems Research Group, Department of Computer Science, University of Utah, March 1990.

[Gna89] Gnadinger F.P. "High speed nonvolatile memories employing ferroelectric technology". In *Proceedings of VLSI and Computer Peripherals. VLSI and Microelectronic Applications in Intelligent Peripherals and their Interconnection Networks (Cat. No.89CH2704-5)*, pages 1/20–23, 1989.

[Gol83] Goldman L.S. & Totta P.A. "Area array solder connections for VLSI". *Solid State Technology*, 1983.

[Gri84] Grindberg J. et al. "A Cellular VLSI Architecture". *IEEE Computer*, pages 69–81, 1984.

[Gri91] Grimshaw A.S. & Vivas V.E. "FALCON: A Distributed Scheduler for MIMD Architectures". In *Symposium on Experiences with Distributed and Multiprocessors Systems*, pages 149–164, 1991.

[Gui82] Guillemont M. "The CHORUS Distributed Operating System: Design and Implementation". In *ACM International Symposium on Local Computer Networks*, pages 207–223, April 1982.

[Gul88] Gull A. "STIX: A Port of The MINIX Operating System to the Atari ST". Undergraduate Thesis, 1988.

[Gul89] Gull A. "Memory Management Hardware: Panacea or Pain". In *EUUG Spring Conference '89*, pages 217–221, April 1989.

[Gul91] Gull A. "Using A Wafer-Scale Component to Create an Efficient Distributed Shared Memory". In *EurOpen Autumn 1991 Conference Proceedings*, 1991.

[Gun90] Gunston B. *Avionics: The Story and Technology of Aviation Electronics.* Patrick Stephens Ltd, 1990.

[Gup88] Gupta U. "Who's News: Gene Amdahl Can't Stop Tilting at IBM". *Wall Street Journal*, 7th June 1988.

[Hag91a] Hagersten E. et al. "DDM — A Cache-Only Memory Architecture". Technical Report, SICS Research Report R91:19, November 1991.

[Hag91b] Hagersten E. et al. "A Performance Study of the DDM — A Cache-Only Memory Architecture". Technical Report, Swedish Institute of Computer Science; Report R91:17, Submitted to ISCA92, 1991.

[Hag92] Hagersten E. *Towards Scalable Cache Only Memory Architectures.* PhD thesis, The Royal Institute of Technology (KTH), Stockholm, Sweden, October 1992.

[Hen90] Hennessy J.L. & Patterson D.A. "Computer Architecture: A Quantitative Approach". *Morgan Kaufmann Publishers, Inc.*, Front sheet, 1990.

[Her88] Herrmann F. et al. "CHORUS, a New Technology for Building UNIX Systems". In *Proceedings of the EUUG Autumn Conference*, pages 1–18, Autumn 1988.

[Hes90] Hesse J. "Development trends in nonvolatile semiconductor memories". *Radio Fernsehen Elektron (East Germany)*, 39(8):495–498, 1990.

[Hew90] Hewlett–Packard Company. *PA–RISC 1.1 Architecture and Instruction Set: Reference Manual*, November 1990.

[Hil92] Hill M.D. et al. "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors". In *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, To appear (October 1992).

[hit90] "HT^2 Hitachi Technology Transfer". *Product Release Information; Hitachi Europe Ltd*, 1990.

[Hoa74] Hoare C.A.R. "Monitors, An Operating System Structuring Concept". *Communications of the ACM; Vol. 17*, pages 549–557, Oct 1974. Erratum in Communications of the ACM; Vol. 18; p. 95; Feb 1975.

[Hof63] Hofstein S. & Heiman F. "The Silicon Insulated–Gate Field–Effect Transistor". In *Proceedings of the IEEE 51(9)*, pages 1190–1202, September 1963.

[Hor] Horwat W. et al. "COSMOS: An Operating System for a Fine–Grain Concurrent Computer". Technical Report, Massachusetts Institute of Technology.

[Int86] Intel Corporation. *80386 Programmer's Reference Manual*, 1986.

[Jac89] Jacqmot C. & Milgrom E. "UNIX and Load Balancing: a Survey". In *Proceedings of the EUUG*, pages 1–15, Spring 1989.

[Jam86] James Q. Arnold. "Shared Libraries on UNIX System V". In *Proceedings of the USENIX Association*, 1986.

[Jef90] Jeffery B. "IBM's RS/6000–a strategic report". *Comput. Econ. Rep. (USA)*, 12(5):1–4, May 1990.

[Jul87] Jul E. et al. "Fine–Grained Mobility in the Emerald System". *Proceedings of the 11th ACM Symposium on Operating System Principles; Operating Systems Review*, 21(5):105–106, November 1987.

[Kil61] Kilburn et al. "The Manchester University Atlas Operating System Part 1: Internal Organization". *Computer Journal*, 4(3):222–225, October 1961.

[Kil62] Kilburn et al. "One–Level Storage System". *IEEE Transactions on Electronic Computers*, EC–11(2):223–235, April 1962.

[Kil85] Killian T.J. "Processes as Files". Technical Report, Bell Laboratories, March 1985.

[Kim86] Kim M.Y. "Synchronised Disk Interleaving". *IEEE Transactions On Computers*, C–35(11):978–988, November 1986.

[Kin90] King R.P. "Disk arm movement in anticipation of future requests". *ACM Transactions Computer Systems (USA)*, 8(3):214–229, August 1990.

[Kru89] Krueger S. "Are 32 Bits Enough?". *BYTE*, November 1989.

[Lah90] Lahti W. & McCarron D. "Store data in a flash (flash-memory ICs)". *BYTE (USA)*, 15(12):311-13, 315, 317, 318, November 1990.

[LeB89] LeBlanc T.J. et al. "Memory Management For Large-Scale NUMA Multiprocessors". Technical Report, Department of Computer Science; University of Rochester, March 1989.

[Lee89] Lee R.B. "Precision Architecture". *IEEE Computer*, pages 78-91, January 1989.

[Lef89] Leffler et. al. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[Len90] Lenoski D. et al. "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor". In *Proceedings of 17th International Symposium on Computer Architecture*, pages 148-159, 1990.

[Len92] Lenoski D. et al. "The Stanford Dash Multiprocessor". *Computer*, pages 63-79, March 1992.

[Li 86] Li K. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University; Department of Computer Science, 1986.

[Li 89] Li K. & Hudak P. "Memory Coherence In Shared Virtual Memory Systems". *ACM Transactions on Computer Systems*, 7(4):321-359, November 1989.

[Lin86] Lin F.C.H. & Keller R.M. "Gradient Model: A Demand-driven Load Balancing Scheme". In *Proceedings of International Conference on Distributed Computing Systems*, pages 329-336, 1986.

[Mad90] Madni, A.M. "Digitally Programmable Voltage-to-frequency Converter". In *1990 IEEE Aerospace Applications Conference Digest (Cat. No.90TH0223-8), IEEE*, pages 171-9, 1990.

[Mar] Marsh B.D. et al. "First-Class User-Level Threads". Technical Report, Department of Computer Science; University of Rochester.

[Mar86] Marcom J. "Britain Picks Boeing's Plane Over Nimrod - AWACS Order of $1.4 Billion Ends What Officials Calls 'sad story' of GEC craft". *The Wall Street Journal (WSJ861219-0105)*, 19th December 1986.

[Mar89] Martin A.J. et al. "The First Asynchronous Microprocessor: The Test Results". Technical Report, Department of Computer Science, California Institute of Technology, April 1989.

[Mas89] Massalin H. & Pu C. "Threads and Input/Output in the Synthesis Kernel". In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, volume 21, number 5, pages 191-201, December 1989.

[Mas91a] Mashey J.R. "64-bit Computing". *BYTE*, September 1991.

[Mas91b] Masuoka F. et al. "Review and prospects of non-volatile semiconductor memories". *IEICE Trans. (Japan)*, 74(4):868-74, 1991.

[Mel91] Mellor–Crummey J.M. & Scott M.L. "Algorithms for Scalable Synchronisation on Shared Memory Multiprocessors". *ACM Transactions on Computer Systems*, 9(1):21-65, February 1991.

[Mey88] Meyer A. "Direct Mapped Files". In *Proceedings of the Spring EUUG Conference*, pages 231–236, 1988.

[Mik83] Mike Hirst. *Airborne Early Warning: Design, development and operations.* Osprey Publishing Limited, 1983.

[Mil69] Miller L.F. "Controlled collapse reflow chip joining". *IBM J.Res.Develop*, May 1969.

[MIP] MIPS Computer Systems Inc. *User's Manual Hardware; RISC Microprocessors V_R4000 / V_R3600 MIPS RISC Architecture.*

[Mit89] Mitchell P. et al. "Latency the block to speed (disk drives)". *Computer Systems Europe (UK)*, 9(11):51, November 1989.

[Moa90] Moazzami R. et al. "A ferroelectric DRAM cell for high–density NVRAMs". *IEEE Electron Device Lett. (USA)*, 11(10):454–456, October 1990.

[Moo70] Moore G. "What Level of LSI is Best for You?". *Electronics 43(4)*, pages 126–130, February 16, 1970.

[Mul91] Mullender S.J. "Experiences with Amoeba". In *Proceedings of EurOpen Spring Conference*, pages 1–11, 1991.

[Ni 85] Ni L.M. et al. "A Distributed Drafting Algorithm for Load Balancing". *IEEE Transactions on Software Engineering*, SE–11(10):1153–1161, October 1985.

[Ols89] Olson T.M. "Disk Array Performance In A Random I/O Environment". *Computer Architecture News*, pages 71–77, September 1989.

[Ore78] Oren T.I. "Concepts for Advanced Computer Assisted Modelling". In *Proceedings of Symposium on Modelling and Simulation Methodology*, pages 29–55, 1978.

[Pat88] Patterson D.A. et al. "A Case For Redundant Arrays Of Inexpensive Disks (RAID)". In *Proceedings of ACM SIGMOD 88*, pages 109–116, June 1988.

[Pin89] Ping–Hui K. "Support of the ISO–9660/HSG CD–ROM File System in HP–UX". In *Proceedings of the USENIX Association*, pages 189–202, Summer 1989.

[Pru86] Prucnal P.R. et al. "Integrated Fibre–optic Coupler for very Large Scale Integration Interconnects". *Optics Letts*, 11:109–111, 1986.

[Prz90] Przybylski S. "The performance impact of block sizes and fetch strategies". In *Proceedings of the 17th Annual International Symposium on Computer Architecture (Cat. No.90CH2887-8)*, pages 160–169, 1990.

[Pu 88] Pu C. et al. "The Synthesis Kernel". *Computing Systems*, 1(1):11–27, Winter 1988.

[Red80] Redel D. et al. "Pilot: An Operating System for a Personal Computer". *Communications of the ACM*, 23(2), February 1980.

[Ric89] Richard Stevens W. "Heuristics for Disk Positioning in 4.3 BSD". *Computing Systems*, 2(3):251–274, Summer 1989.

[Rob77] Roberts J.B.G. et al. "A New Approach to Pulse Doppler Processing". In *Radar–77 International Conference*, October 1977.

[Rob92] Robertson et al. "Okapi at TREC". Technical Report, Department of Information Science; City University, December 1992.

[Ron90] Ronnau U. "Massive storage". *Personal Computing (West Germany)*, (4):18–20, 22, April 1990.

[Ros86] Roscoe A.W. & Dathi N. "The Pursuit of Deadlock Freedom". *PRG–57, Oxford University PRG*, 1986.

[Ros89] Rosenthal D.S.H. "More Haste, Less Speed". In *EUUG Spring Conference '89*, pages 123–130, April 1989.

[Sca87] Scanlan M.J.B. *Modern Radar Techniques.* Collins, 1987.

[Scoa] Scott M.L. et al. "Implementation Issues for the Psyche Multiprocessor Operating System". Technical Report, Department of Computer Science; University of Rochester.

[Scob] Scott M.L. et al. "Multi–Model Parallel Programming in Psyche". Technical Report, Department of Computer Science; University of Rochester.

[Sco89a] Scott M.L. et al. "A Multi–User, Multi–Language Open Operating System (extended abstract)". Technical Report, Department of Computer Science; University of Rochester, April 1989.

[Sco89b] Scott M.L. et al. "Evolution of an Operating System for Large–Scale Shared–Memory Multiprocessors". Technical Report, Department of Computer Science; University of Rochester, March 1989.

[Sel90] Seltzer M. et al. "Disk scheduling revisited". In *Proceedings of the Winter 1990 USENIX Conference, USENIX*, pages 313–324, Winter 1990.

[seq87] *Symmetry Technical Summary*, 1003–44447 Rev. A; edition, 1987.

[Sho88] Short R.T. & Levy H.M. "A Simulation Study of Two–Level Caches". In *Proceedings of the 15th Annual International Symposium on Computer Architecture (Cat. No.88CH2545–2)*, pages 81–88, 1988.

[Sin92] Singh J.P. et al. "SPLASH: Stanford Parallel Applications for Shared Memory". *Computer Architecture News*, 20(1):5–44, March 1992.

[Sle91] Sleat P.M. *A Static, Transaction Based Design Methodology for Hard Real-Time Systems.* PhD thesis, Department of Computer Science; City University, 1991.

[Sta81] Stapper C.H. "Comments on 'Some Considerations in the Formulation of IC Yield Statistics'". *Solid-State Electronics 24*, pages 127–132, February 1981.

[Sta83] Stapper C.H. et al. "Integrated Circuit Yield Statistics". In *Proceedings of the IEEE; Vol. 71, No. 4*, pages 453–470, April 1983.

[Sta84] Stankovic J.A. & Sidhu I.S. "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups". In *Fourth International Conference on Distributed Computing Systems*, pages 49–59, 1984.

[Sti92] Stiemerling T. "ANGEL Object Space and Virtual Memory Management". Technical Report, Computer Science Department; City University, 1992.

[Sun89] Sundaram G.S. "DRFMs: A Leap Forward For Electronic Warfare". *Int. Def. Rev. (Switzerland), suppl.*, pages 138–9, September 1989.

[Swi86] Swinehart D.C. et al. "A Structural View of the Cedar Programming Environment". *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[Tam67] Tammaru E. & Angell J.B. "Redundancy for LSI yield enhancement". *IEEE Journal Solid State Circuits; Vol. SC-2*, pages 172–182, December 1967.

[Tam90] Tam M.C. & Farber D. "CapNet – An Approach to Ultra High Speed Networks". In *IEEE International Conference on Communications ICC '90 Cat. No.90CH2829-0)*, pages 955–961, 1990.

[Tei84] Teitelman W. "A Tour Through Cedar". In *Proceedings of 7th International Conference on Software Engineering*, March 1984.

[Tev87a] Tevanian A. & Rashid R.F. "MACH: A Basis for Future UNIX Development". Technical Report, Department of Computer Science; Carnegie Mellon University, June 1987.

[Tev87b] Tevanian A. et al. "A Unix Interface for Shared Memory and Memory Mapped Files under Mach". Technical Report, Department of Computer Science; Carnegie–Mellon University, 1987.

[Tew89] Tewksbury S.K. *Wafer-Level Integrated Systems: Implementation Issues.* Kluwer Academic Publishers, 1989.

[Van90] Van De Goor A.J. & Verruijt C.A. "An Overview of Deterministic Functional RAM chip Testing". *ACM Computing Surveys; Vol. 22, No.1*, pages 5–33, March 1990.

[Wal60] Wallmark J. "Design Considerations for Integrated Electronic Devices". In *Proceedings of the IRE 48(3)*, pages 293–300, March 1960.

[Wan89] Wang W.H. et al. "Organization and Performance of a two–Level Virtual–Real Cache Hierarchy". In *Proceedings of the 16th Annual International Symposium on Computer Architecture (Cat. No.89CH2705-2)*, pages 140–148, 1989.

[Whi92] Whitcroft A. & Osmon P. "The CBIC: Architectural Support for Message Passing or Shared Memory?". In *Proceedings of the 8th UK Performance Engineering Workshop; Imperial College of Science, Technology and Medicine*, 21st – 22nd September 1992.

[Who88] Whobrey D. "A Communications Chip for Multiprocessors". Technical Report, Department of Computer Science, City University, June 1988.

[Wie82] Wierman J.C. "Percolation Theory". *Ann. Prob.*, 10:509–524, 1982.

[Wil91a] Wilkinson et al. "A Proposal for an Object Oriented Architecture". Technical Report, Computer Science Department; City University, 1991.

[Wil91b] Wilkinson T. "Combining Tested and Fault–tolerant Silicon for High Performance Microprocessors". Technical Report, Computer Science Department; City University, July 1991.

[Wil91c] Wilson P.R. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces in Standard Hardware". *ACM SIGARCH Computer Architecture News*, 19(4), June 1991.

[Wil92] Wilkinson et al. "ANGEL: A Proposed Multiprocessor Operating System". In *European Workshops on Parallel Computing 92*, March 1992.

[Wil93] Wilkinson T. *Implementing Fault Tolerance in a 64–bit Distributed Operating System.* PhD thesis, Computer Science Department; City University, To be published 1993.

[Win87] Winterbottom P. "NCU: Network Control Unit. Preliminary Data Sheet". Technical Report, Computer Science Department; City University, December 1987.

[Win88] Winterbottom P. "BIC: Buffer Interface Controller. Preliminary Data Sheet". Technical Report, Computer Science Department; City University, February 1988.

[Win89] Winterbottom P. & Osmon P. "Topsy: An Extensible UNIX Multicomputer". Technical Report, Computer Science Department; City University, 1989.

[Xil91] Xilinx Inc. *The XC4000 Data Book*, 1991.

[Yok89] Yokoyama S. et al. "A Contiguous High Performance File System". In *Proceedings of the EUUG*, pages 197–206, April 1989.

[Zen88] Zenith S.E. "The Simplicity of Linda". *Parallel Update; BCS Parallel Processing Specialist Group Newsletter*, pages 9–14, March 1988.

Appendix A

Glossary of Terms

- **Amdahl's Laws**

  Two frequently quoted conjectures by Gene Amdahl. The first conjecture states that the performance improvement to be gained from an enhancement is limited by the time it can be used:

  $$Speedup = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

  The second conjecture relates the performance of a computer to its memory size and I/O bandwidth:

  > *A balanced computer system needs about 1 Mbyte of main memory capacity and 1 Mbit per second I/O bandwidth per MIPS of CPU performance.*

  It is thought to represent a balanced system for general–purpose computing.

- **Broadcast Network**

  A communications network which allows a message to be transmitted to a number of destination nodes simultaneously. Such networks are inherently non–scalable.

- **Cache**

  A high speed memory which holds recently used data. Used to increase performance.

- **Cache Coherency**

  A mechanism to keep the contents of processor caches in agreement, so that they do not hold different values for the same memory location.

- **Cache–Only Memory Architecture (COMA)**

  A machine architecture which is based solely on a distributed shared memory.

- **Capability**

  The right to access a particular resource in a specified manner. In *Cherub* capabilities are implemented as passwords and are used to grant *read*, *write* and *execute* permissions to objects.

- **Circuit Switching**

  A communication routing scheme in which an electronic circuit is constructed between the source and destination nodes. Once the circuit has been made, message transfer can take place over it. When transfer is complete, the circuit is cleared.

- **Coarse–Grain Parallelism**

  A program in which the parallel processors communication and synchronise infrequently, normally only after hundreds of thousands of instructions (10^5).

- **Cherub**

  A proposed implementation of a hardware distributed single shared address space memory architecture.

- **Context Switch**

  The process by which a processor switches execution from one task to another. Usually performed when an executing process blocks, for example on a DSM page fault, or its execution time quantum expires.

- **Copy On Write (COW)**

  A technique, often employed in operating systems, which allows one or more virtual pages to be represented by a single physical page. When one of the virtual pages is modified, a copy of the physical page is made and the modification is made to it.

- **Data Granularity**

  The unit of data shared between processes. In a distributed shared memory this is usually a main memory page.

- **Distributed Memory Multicomputer**

  A type of parallel architecture in which each processor has its own private main memory which other processors cannot access. Typically these architectures employ non–broadcast communication networks and are, therefore, scalable to many hundreds of processors.

- **Distributed Shared Memory (DSM)**

  A mechanism which creates the illusion of a shared memory on a distributed multicomputer, albeit with a coarser granularity of data sharing.

- **Fine–Grain Parallelism**

  A program in which the parallel processors communicate and synchronise frequently, normally after a few thousand (10^3) instructions.

- **Flip Chip Bonding**

  A wafer–scale integration technique in which circuit dies are mounted upside–down on a silicon substrate, joined electrically with solder bumps.

- **Generation Scalability**

  Whether an architecture can be implemented in a new technology and, thus, take advantage of increased circuit and packing technologies as they become available.

- **Hardware Distributed Shared Memory (HDSM)**

  The implementation of a distributed shared memory in hardware in order to reduce intertask communication latencies.

- **Hardware Distributed Single Shared Address Space Memory Architecture (HDSSASMA)**

  The combination of a distributed shared memory and a single shared address space architecture. Intended to simplify intertask communication mechanisms, making their implementation in hardware easier.

- **Hot Spot**

  An area of data that must be accessed frequently by many processors in a multiprocessor system. When this occurs, the effective performance of the multiprocessor is severely degraded.

- **Hybrid Wafer Integration**

  A wafer–scale integration technique which uses flip–chip bonding to mount tested dies on the wafer surface.

- **Intertask Communication**

  The communication, both implicit and explicit, which occurs between tasks. Includes the communication needed to create, share data with and terminate a child process.

- **Medium–Grain Parallelism**

  A program in which the parallel processors communicate and synchronise after tens of thousands of instructions (10^4).

- **Message Passing Programming Paradigm**

  A parallel programming paradigm in which tasks communicate through primitives for the protected sending and receiving of message. The sequential arrival of messages inherently synchronises the actions of the tasks.

- **Multiple Instruction–stream, Multiple Data–stream (MIMD)**

  A class of parallel processors in which each processor executes a different program on different data from other processors.

- **MIPS**

  Millions of instructions per second. A measure of computer performance.

- **Multicomputer**

  See distributed memory multicomputer.

- **Multiprocessor**

  See shared memory multiprocessor.

- **Non–broadcast Network**

  A communication network which only allows a message to be transmitted to a single destination node. Such networks are inherently scalable.

- **Packet Switching**

  A communications routing scheme in which a message is broken down into small packets at the source node. These are transmitted across the communication network and reassembled in order at the destination.

- **Problem Scalability**

  Whether an application can implemented efficiently on an architecture with a given granularity.

- **Redundant Arrays of Inexpensive Disks (RAID)**

  The use of many small inexpensive disks rather then few large expensive ones. This provides higher performance (both latency and bandwidth) and reduces power consumption. With large numbers of disks, failure rate becomes a significant issue, making fault tolerance necessary.

- **Relaxed Coherence**

  A cache coherence mechanism designed for increased efficiency. A read operation does not necessarily return the most recently written value.

- **Single Shared Address Space (SSAS)**

  A memory model in which all tasks occupy a single common address space, rather then multiple privates ones.

- **Scalability**

  See generation scalability and problem scalability.

- **Shared Memory Multiprocessor**

  A type of parallel architecture with a global memory which is accessible to all of the processors. Typically these architectures employ broadcast communications networks and, therefore, their scalability is limited to a few tens of processors.

- **Shared Variable Programming Paradigm**

  A parallel programming paradigm in which tasks communicate and synchronise through the use of common data structures. Often implemented using a shared memory architecture.

- **Strong Coherence**

  A cache coherence mechanism in which a read operation on a memory location returns the most recently written value.

- **Synchronisation**

  The delaying of process execution while constraints on the ordering of actions are satisfied.

- **Thrashing**

  Very high paging activity which results in severe performance degradation. In general, a process is said to be *thrashing* when it is spending more time paging then executing.

- **Translation Look–aside Buffer (TLB)**

  A partially associative processor cache used in the translation of the virtual addresses used by processes into the physical addresses used to access the cache and main memory.

- **Wafer–Scale Integration (WSI)**

  A method of manufacturing electronic circuits in which wafers are packaged whole rather than as individual chips.

- **Whole Wafer Integration**

  A wafer–scale integration technique in which circuits are lithographed on whole wafers.

Appendix B

Airborne–Early Warning: An Application for Cherub

B.1 Introduction

This appendix describes airborne early warning (AEW): a non–trivial application naturally suited to a medium grain of processing. It demonstrates how the *Cherub* architecture can be used to provide an efficient and elegant solution.

B.2 The Problem

Modern aircraft attack by flying low and fast, escaping detection by ground–based radar stations through using the horizon as cover. To counter this threat, modern anti–aircraft defences employ airborne early warning (AEW) systems, high endurance aircraft equipped with powerful *look–down* radars, to fly at high altitude, thus minimising the masking effect of the horizon.

Producing radars which are able to look–down and distinguish enemy aircraft from ground and sea *clutter* is far from trivial, however. Providing enough computing power to process a noisy radar image, track an average of 400 targets and distinguish friend from foe, is beyond most computer architectures available today. Furthermore, the architecture must be compact as it, together with its power supply and cooling system, must fit into the body of a plane the size of a small airliner.

B.2.1 Background

Airborne early warning (AEW) was first employed during the American Civil War. Binocular equipped observers were raised in balloons thousands of feet above the battlefields to spy on enemy troop movements and to spot for artillery. Although a severely limited means of gathering intelligence, airborne spotters were soon employed by armies throughout the world.

During the First World War aircraft rapidly superseded balloons, and the romantic image of the dawn patrol was born. At first sun-up formations of biplanes would take off and circle the battlefield searching for enemy planes. Dogfights were rare however, for the visual range was short, making it easy for enemy aircraft to cross the lines and make sneak attacks unobserved.

When in 1936, Watson-Watt demonstrated long-range aircraft detection using radar (radio detection and ranging), the need for airborne early warning aircraft was immediately appreciated. Radar waves travel in approximately straight lines and, therefore, are unable to reach air-space hidden by the curvature of the earth. So, the lower an aircraft approaches a radar site, the nearer it can get before being detected, thus reducing the time available to intercept it. This phenomenon is illustrated in the graph in figure B.1.

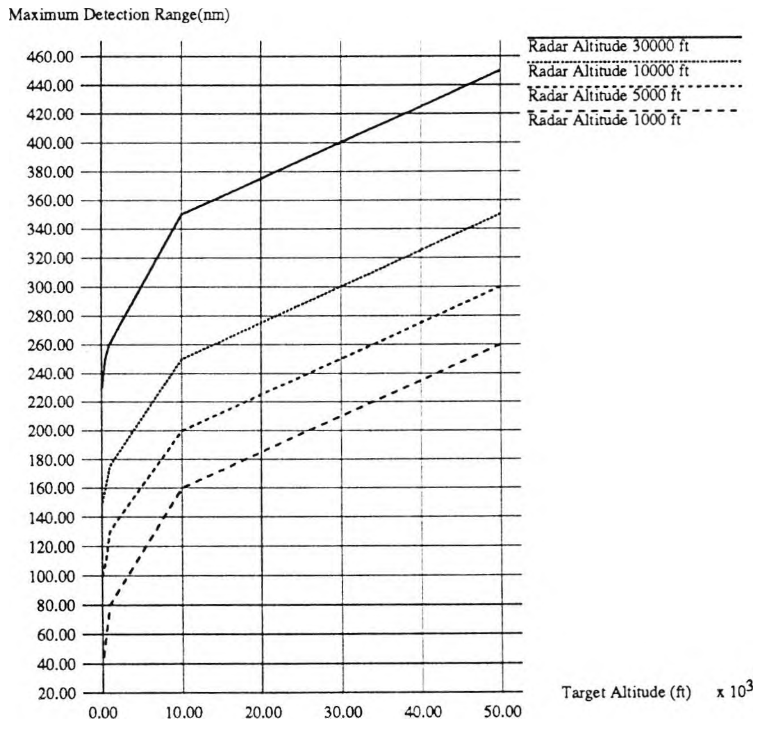


Figure B.1: The Relationship Between Radar Altitude And Maximum Detection Range

Radar technology improved rapidly. By the mid 1940s it became possible for aircraft to operate in an AEW role. The look-down capacity of these systems, however, was very limited; random reflections from the earth's surface produce thousands of false contacts, or *clutter*, which blot out returns from real targets. Hence early overland AEW was out of the question and overwater detection was only possible when the sea was calm. It is only with the advent of digital signal processing techniques that effective AEW has become possible.

Modern AEW aircraft are important targets for the enemy to attack. The loss of information should one be shot down would be catastrophic. To minimise the risk of attack, AEWs are deployed deep within friendly air-space. An aircraft flying a circular path 150 nm within its own air–space can reasonably protect a border 500 nm while minimising the risk to itself. This is illustrated in figure B.2.

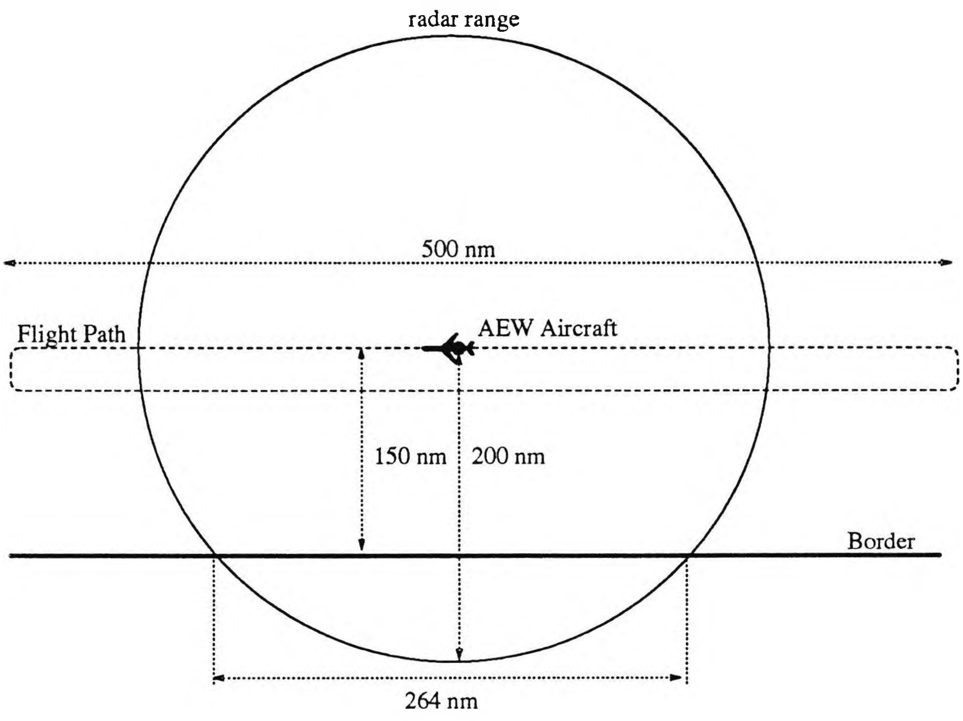


Figure B.2: The Air–Space Covered by an AEW Aircraft

B.2.2 The Load AEW Places On An Architecture

The load placed on a computer architecture by the AEW role is severe; a large number of complex algorithms need to be performed concurrently. Hirst [Mik83] identifies eight major tasks performed by computers in AEW aircraft:

- Radar Target Information Gathering, Storage and Processing
- IFF Target Information Gathering, Storage and Processing
- PDS Target Information Gathering, Storage and Processing
- Target Data Correlation
- Providing the Man–Machine Interface
- Data Sharing and Communication
- Flight Equipment Monitoring
- Navigation

This is by no means a comprehensive list, however, it gives some idea of the complexity of the task which must be accomplished in real-time. Only the first four tasks in the list are included in the following discussion as these incur most of the processing overheads. They will now be examined in greater detail.

B.2.3 Radar Target Information Gathering, Storage and Processing

Although radar is only one of the detection mechanisms used in AEW aircraft, it is both its most important and complex system. A simplified Pulse-Doppler radar system is shown in figure B.3.

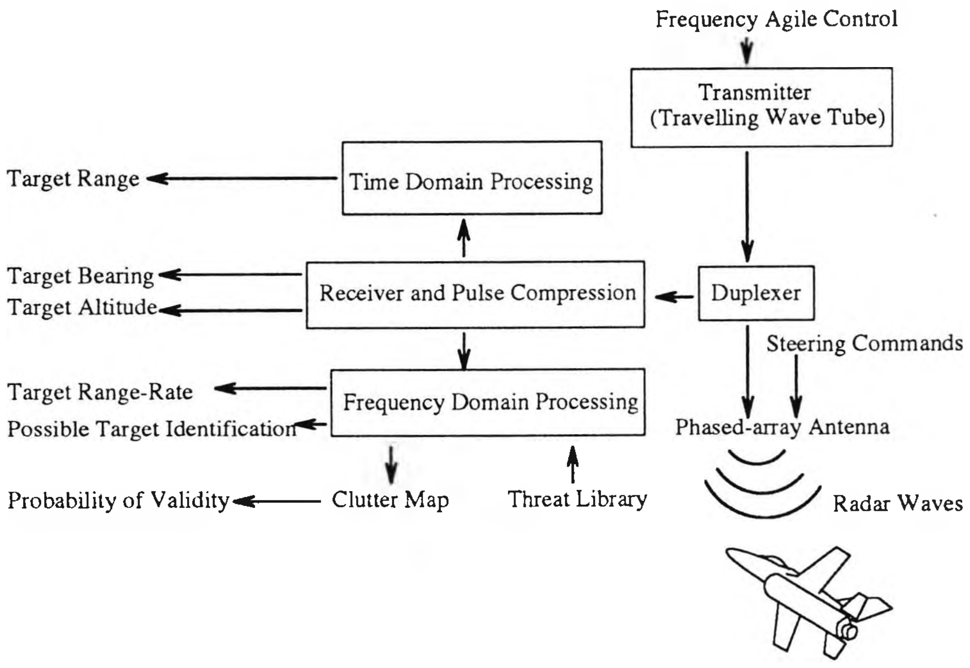


Figure B.3: A Simplified Pulse-Doppler Radar System

Radars have much in common with searchlights. A searchlight uses a mirror to emit directed visible electromagnetic radiation. Its operator searches for reflections off targets. A radar uses a directional antenna to emit non-visible electromagnetic radiation and collect returning reflections.

Search radars, the type used in AEW aircraft, typically emit pulses of electromagnetic radiation on a single frequency. Periodically, however, the radar will switch to a different operating frequency. This complicates the task of an enemy blinding, or *jamming*, the system by transmitting pulses of a similar wavelength. This is called *frequency agility*.

Radiation spreads out from a radar proportionally to the square of the distance. This effects both the strength of an initial radar pulse and that of a returning echo. Therefore, the maximum detection range of a radar is proportional to the fourth-root of the power it transmits. As the strength of returning signals are often very weak, random noise within the radar itself can look like false targets.

Most modern airborne radars use either phased-array or inverted-cassegrain antennas[1]. These allow the radar energy to be electronically formed into a number of individually steerable beams. This flexibility permits the radar to control the number of pulses it transmits towards a desired location. The chance that noise in the system will produce false targets is almost eliminated by the ability to transmit additional pulses in the direction of a new target until the radar has adequate information to make a reliable decision about its presence or absence.

The frequency spectrum around that of the radar's fundamental frequency is extracted[2] to produce digital data which can be processed to reveal further information [Bro82, Far85]:

- Antenna Azimuth and Elevation Analysis

  When a return from a target is detected, the antenna's azimuth shows its bearing and the antenna's elevation indicates, to some degree, its altitude.

- Time Domain Analysis

  By measuring the time taken for the pulse to reach a target and return, its range can be calculated.

- Frequency Domain Analysis

  The movement of a target relative to the radar will produce Doppler shifts in the frequency of the reflected pulses. This phenomenon can be used in three ways:

  - It allows a target's instantaneous speed to be calculated. This avoids having to obtain speed information over a number of radar scans and is, therefore, both faster and more reliable.
  - It allows the radar to look down on and distinguish moving targets from ground clutter. Objects on the ground will appear to move at the same speed as the aircraft. This allows the radar to discern true targets from clutter. Delong and Hofsletter [Del69] quote an example of a radar which, with one pulse, is able to distinguish a single target from one hundred clutter plots.

    In addition to ignoring ground clutter, it is necessary to discard returns from slow moving targets which are unlikely to be valid[3]. This process is called velocity gating. Too high a velocity gate, however, can cause slow moving aircraft, such as helicopters, to be missed.
  - Turbine blades and propellers have characteristic Doppler effects of their own. Some radars are able to use these to identify targets using libraries of known patterns[4].

[1]These antennae also reduce *sidelobes*, the transmission or reception of radar signals well off the centerline. Sidelobes waste power and can be exploited by enemy deceptive jammers.

[2]This is accomplished by first amplifying, filtering and mixing the analogue signal. It is then digitised, averaged with the previous pulse to cancel clutter and Fast Fourier Transforms (FFTs)[Rob77] are applied to extract the frequency spectrums. Research is currently being conducted on Digital Radio-Frequency Memories (DRFMs) [Sun89, Mad90]. These digitise the analogue signals directly, allowing their permanent storage for analysis.

[3]It is not unusual for American F-15 Eagle fighters stationed in Germany to accidentally lock-on to fast moving BMWs on the Autobahns.

[4]It is claimed that the powerful AWG-9 radar employed in the American F-14 aircraft is able to count the number of blades in a spinning jet turbine.

One of the major innovations in modern radar is the use of short wavelengths to enhance resolution. This greatly improves target identification, albeit at a much increased signal processing cost; dividing the wavelength by n increases the processing by n^2, as is shown in figure B.4.

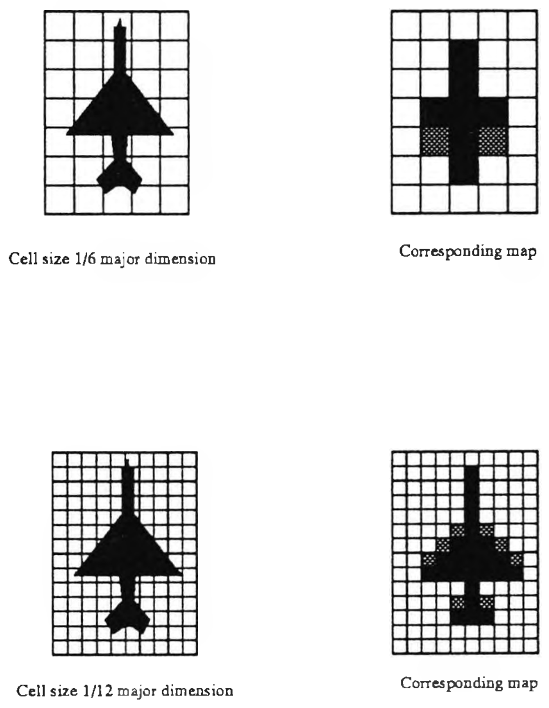


Figure B.4: How the Radar Resolution Effects Target Identification

Given the real-time constraints placed on a radar, it is necessary to carefully plan the activities it must perform. A prioritised task list is generated, from which the commands issued to the radar are scheduled. Tracking existing targets is regarded as a very cost effective task and is given a high priority. Scanning for new targets is treated as a background activity to be performed when the radar is otherwise idle.

B.2.4 Identification, Friend or Foe (IFF) Target Information Gathering, Storage and Processing

The greatest proportion of traffic detected by an AEW system at any one time will be friendly. To distinguish friendly targets easily, a system called Identification, Friend or Foe (IFF) is employed.

Most aircraft, whether civil or military, carry a IFF transceiver which listens for coded radio messages at a frequency of 1,090 Mhz. AEW aircraft, and ground-based tracking stations accompany their radar pulses with an IFF interrogation request message transmitted on this frequency. If an aircraft receives a correctly coded interrogation message, it immediately sends back a coded reply on 1,030 Mhz, giving its identity and height. This process is illustrated in figure B.5. The direction of the IFF antenna gives the bearing of the target and the time delay between the issue of the interrogation request and the arrival of the reply corresponds to the range.

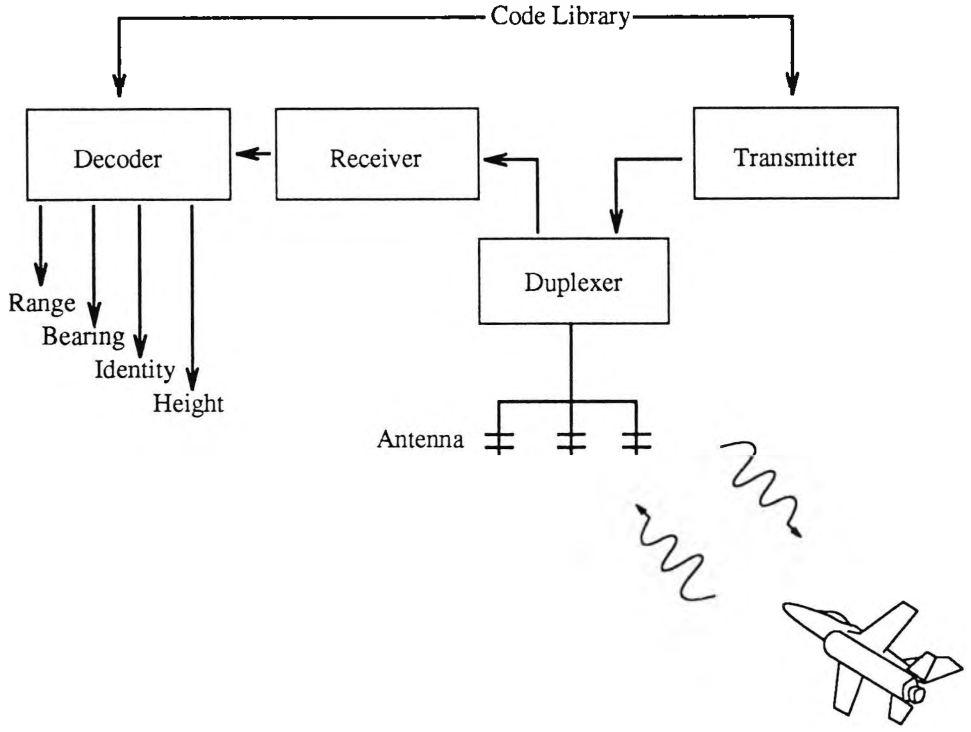


Figure B.5: The Identification, Friend or Foe (IFF) System

Civil air traffic use universal recognised codes, while military traffic codes are secret and are varied continuously to avoid forgery. It is therefore easy to recognise a friendly aircraft that is using the ascribed coding procedure.

Even though the IFF process is relatively simple, it incurs a considerable overhead. Typically, several hundred targets would be visible at any one time and they can be established or lost at a rate of 25 or so per minute [Bel88].

B.2.5 Passive Detection Systems (PDS) Target Information Gathering, Storage and Processing

Target identification is sometimes possible by examining the electromagnetic radiation it emits. A low-flying bomber, for example, will have a terrain-following radar and a fighter will have a search and targeting radar. These have distinct frequency patterns which are as recognizable as human voices.

AEW aircraft are equipped with a set of antennas which feed a broad-band radio receiver. Interesting signals are extracted from the noise using Fast Fourier Transforms. The resulting frequency spectrums are identified using an accurate library of known electronic signatures. Such a library will have several hundred entries. This process is illustrated in figure B.6.

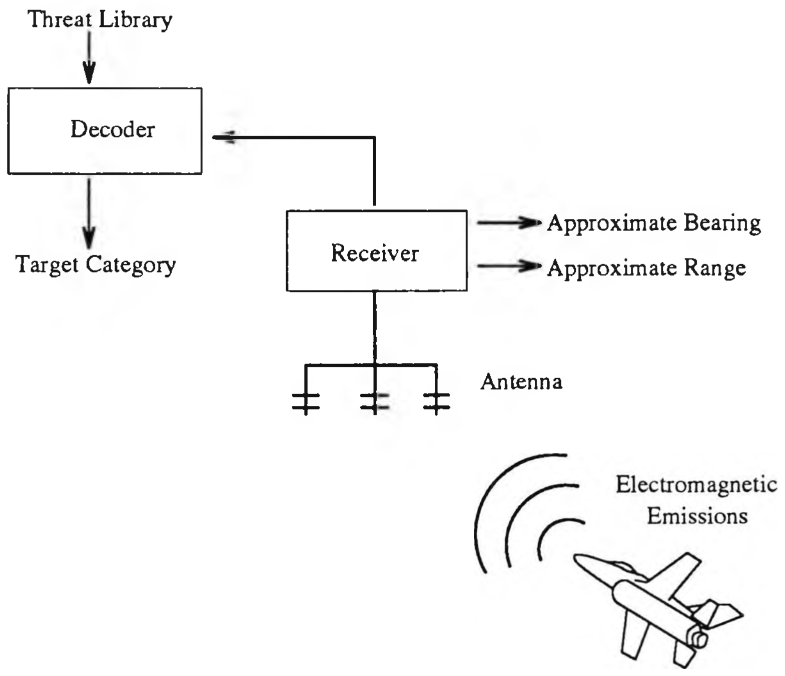


Figure B.6: The Passive Detection System (PDS)

Previously unknown signals must be recorded, or transmitted directly back to base, for analysis by signal experts. This process is called Electronic Intelligence (ELINT) and is invaluable for tasks such as locating enemy positions, intercepting classified transmissions and designing efficient jamming techniques.

B.2.6 Target Data Correlation

The radar, IFF and PDS systems must be correlated to ensure their most effective use. This involves the following tasks.

- Radar Command Scheduling

  The computer must schedule radar steering commands to perform searches within a certain time-frame. These commands must be issued to the radar hardware at the appropriate point in the antenna's scan.

- Target Tracking

  The positions of a target in successive scans of the radar form a track of its movements [Far85, Rob77, Sca87]. The greater the track of a target can be predicted, the more the radar is able to distinguish between targets and false plots.

- Identifying Targets

  Clues about a target's identify can be gained from each of the detection systems. The computer maintains databases of IFF codes, pulse Doppler patterns and PDS signatures which can be correlated to assign each track a probable identity. In a hostile jamming environment, however, it is possible that such clues may be unreliable, or even contradictory[5].

- Statistics Gathering

  It is necessary to regularly gather information about the status of the radar, IFF and PDS systems. This allows performance analysis and fault detection.

B.3 Current Computer Architectures Employed

A number of computer architectural solutions to the AEW problem have been proposed or tried:

B.3.1 Centralised Computing: Grumman E–2 Hawkeye

Originally commissioned by the US Navy in 1959, the Hawkeye is the most exported AEW in the world. The aircraft has been constantly updated and now carries a comprehensive radar, IFF and PDS detection suite.

As the small airframe can only accommodate three operators, the two linked L–304 computers must support automatic target tracking. Consequently, the Hawkeye is only able to operate over water, which requires less signal processing. In this role, however, it is highly successful[6] and can track over 600 targets.

B.3.2 Centralised Computing: Boeing E–3 Sentry (AWACS)

Initially designed in 1966, the American Boeing E-3 Sentry, often called the Airborne Warning and Control System (AWACS), is based on the airframe of a long–bodied DC–8 airliner. This is a relatively large aircraft and as such provides ample space for a radar and its associated electronics.

For simplicity, the designers of AWACS decided to employ a centralised computer, the IBM CC-1. This machine was capable of 0.75 MIPs and could track up to 100 equally distributed targets simultaneously. It was later discovered that it was necessary to track over 400 targets and the more powerful CC-2 computer was retrofitted to earlier aircraft. This machine achieves two MIPS. Radar signal processing is accomplished using banks of software controlled solid state filters.

The low powered computer system is unable to support automatic track initiation, target identification or PDS processing. As a consequence, nine operators are required to control the system. This is does not pose a significant problem, however, as the airframe is large enough accommodate them.

[5]During the Vietnam War, the Americans deliberately operated squadrons of F–4 Phantom fighters such that to radar and PDS they appeared to be slow, vulnerable bombers. It was hoped that these tactics would draw the elusive enemy fighters into aerial ambushes.

[6]The Hawkeye was recently used for catching drug smugglers off Florida. 45 aircraft, 7 vessels and 12,242 Kg of marijuana were captured.

B.3.3 Distributed Computing: British Aerospace Nimrod AEW.3

The British Nimrod AEW.3, designed in 1971, was based on the Comet Airliner. This had a much smaller airframe then the American AWACS and, consequently, space efficiency was always much more of a concern in this project. As the number of operators which could be accommodated was low, six as opposed to the nine in the American AWACS, it was intended that most of the extra workload be absorbed by the computer systems. The specification required greater functionality, such as auto-track initiation and identification, and a faster response time, then any other system of the time could demonstrate.

It was immediately apparent that a single central computer would not provide sufficient processing power, so Marconi Avionics chose to adopt a distributed processing solution. The main computer, the Integrated Data Processor (IDP), was developed from a GEC 4080M. This had a 1 Mbyte memory and was only fast enough to perform all the basic sensor correlation and track maintenance tasks. Other functions, such as digital signal processing and operator console control were performed by a combination of forty four other processors, linked to the IDP by a single communications bus. These contributed another 1.4 Mbytes of storage.

The Nimrod project was plagued by two basic design flaws:

- Even with an advanced architecture, the computer system was not able to perform adequately. It performed well over water, where its complex software made it better then AWACS, but did not have enough computing power to distinguish targets from clutter when flying over land [Mar86].
- The complexity of the distributed system compromised the software's reliability. During French comparison trials against AWACS in 1984, a number of computer failures were experienced.

By 1986 the British government had flown 20 Nimrod trials and only during one did the computers perform well enough to produce useful results. In December 1986 the British Prime Minister, Margaret Thatcher, terminated the Nimrod project, writing off the 1.4 billion already spent on it [Car86c].

B.3.4 Distributed Computing: Boeing E–8A System

The E–8A is intended as the successor to the successful E–3A AWACS. It has a highly advanced radar, the Grumman J–Stars (Joint Surveillance Target Attack Radar System) [Gun90] which has unprecedented precision, being able to detect targets both in the air and on the ground. This is combined with comprehensive IFF and PDS suites.

The detection systems produce so much data that the aircraft requires a distributed computer system of 27 processors, including one for each of the 15 operator consoles. The signal processor alone performs 625 million complex operations per second[7] (MCOPS). The E–8A's software has 600,000 lines of code.

[7]Faster then a CRAY–1 or about 4,000,000 Apple IIEs

The computers used in the system are so large that the E–8A's airframe has had to be based on that of a Boeing 707, a substantially larger aircraft then the AWACS. This makes the aircraft very expensive to buy and maintain.

B.3.5 Ivor Catt's Proposed Kernel Logic Machine

Ivor Catt [Cat89, Cat91] suggests using his Kernel Logic Machine architecture in an AEW role. This is an array of 1,000 by 1,000 processors on 100 wafers. Each processor would be assigned a unit square of ground–space so small that it would never be overloaded.

There appear to be three problems with this solution:

- The proposed solution is very wasteful of computing power; most of the processors in the array will be idle at any instant in time waiting for the radar to provide information.
- It is not clear how much time it would take to distribute the radar data to, or accept the radar steering instructions from, an array of one million computers. It is likely that this process would be a bottleneck in the system.
- The amount of power 100 wafers would consume and the Flourinert liquid–based cooling system required to dissipate the heat they would generate are likely to prohibit the installation of a Kernel Logic Machine in an aircraft. It is possible that this architecture would be better suited to ground–based control where space is not at a premium.

It is probable that the level of granularity chosen in the system is wrong. A smaller system, comprising a few wafers at most, may well be more cost effective in this role.

B.4 Summary

No computer architecture has yet been shown to be completely satisfactory for the AEW application. This is because:

- AEW software is complicated. It involves many closely cooperating tasks and requires large amounts of data sharing and communication. This implies that the simplified programming model of a shared memory multiprocessor architecture is desirable.
- AEW software must run in real–time and, therefore, requires a greater amount of processing power then a multiprocessor can typically offer. This implies the scalability of a distributed multicomputer is required.

B.5 Using the Cherub Architecture

The *Cherub* architecture provides a compromise which is useful for the AEW role. It offers a substantial increase in throughput over multiprocessors without sacrificing the simplicity of their shared variable parallel programming paradigm.

We avoid any formal analysis of the real–time aspects of the solution; instead a more intuitive approach is taken. Sleat considers the formal implications of a similar real–time example in his PhD thesis [Sle91].

B.5.1 A Data Flow Analysis of the Problem

The dataflow diagram shown in figure B.7 illustrates the interdependencies within the AEW system. This is meant to be a representative, rather then a complete, solution to the problem.

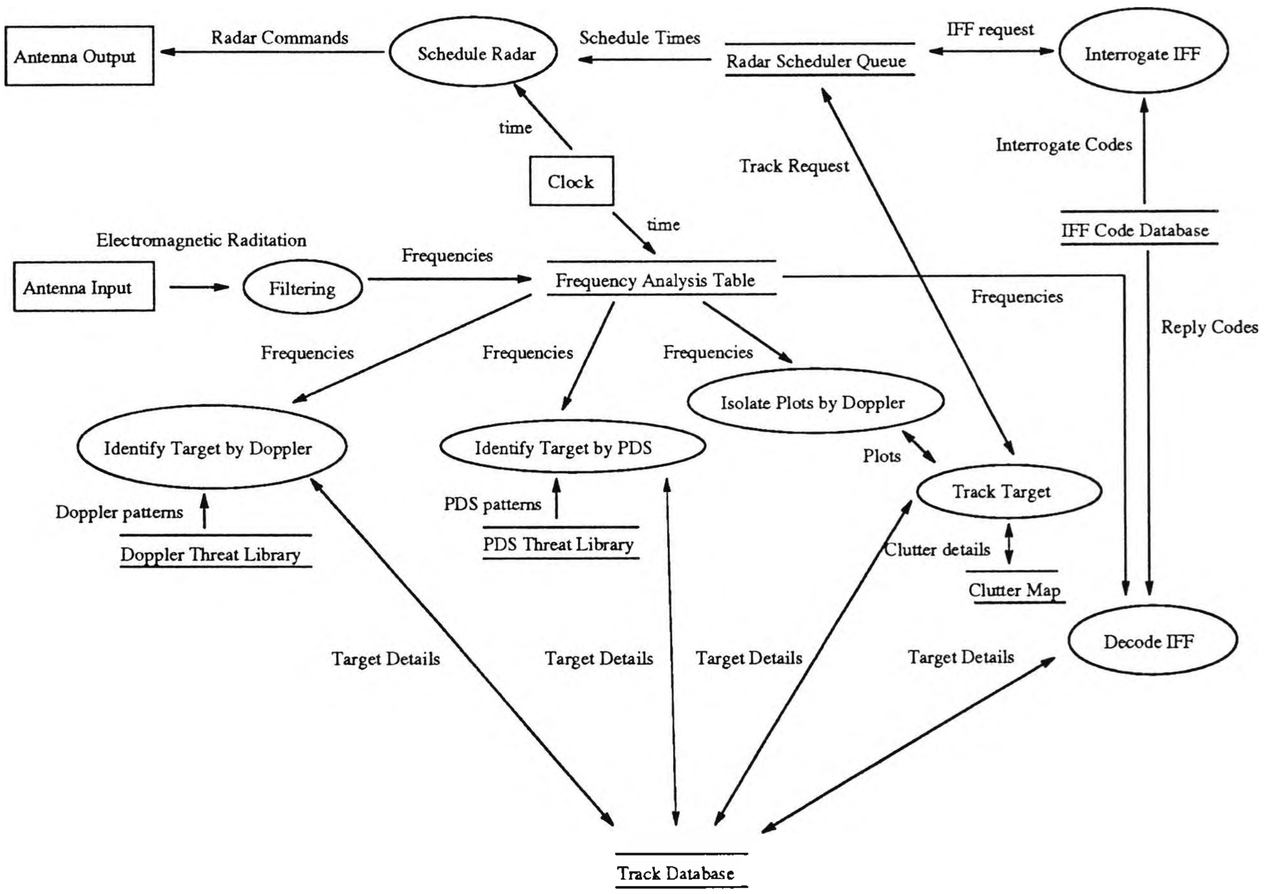


Figure B.7: A Data Flow Diagram of the AEW Detection System

The following data sources are required:

- **Antenna Input (I.ANT_IN)**

  This detects electromagnetic radiation which is being emitted in its direction of facing. There may be one or more of such sources in the system.

  Read by: T.RAD_FIL

- **Real–time Clock (I.R_T_CLK)**

  This data source provides timing information.

  Read by: D.FAT and T.SCH_RAD

The following data output is required:

- **Antenna Output (O.ANT_OUT)**

  This data output accepts the steering and frequency selection commands which control the operation of the radar system. The system may have one or more of these outputs.

  Written by: T.SCH_RAD

The following databases are required:

- **Track Database (D.TRK)**

  This is the second most important database in the system. This contains details about the targets which are currently being tracked. It includes their locations, altitudes, speeds, headings and any information which may be used to identify them, such as their IFF, PDS and pulse Doppler signatures. The contents of this database are constantly added to, updated or deleted.

  Read by: T.IFY_DOP, T.IFY_PDS, T.IFY_IFF and T.TRK_TAR

  Written by: T.IFY_DOP, T.IFY_PDS, T.IFY_IFF and T.TRK_TAR

- **Frequency Analysis Table (D.FAT)**

  This is the most important database in the system. It contains the filtered and processed input from the antenna source in the form of a frequency distribution plotted against time. This database is constantly updated with new information. The contents of the database are stored permanently for future ELINT analysis.

  Read by: T.IFY_DOP, T.IFY_PDS, T.IFY_IFF and T.ISO_PLT

  Written by: T.RAD_FIL and T.R_T_CLK

- **Doppler Threat Library (D.DTL)**

  This is a fixed library which contains pulse Doppler patterns for known friendly and hostile aircraft. Although this database will contain several thousand entries, it will be far from complete; little will be known about an enemy's most modern fighter aircraft.

  Read by: T.IFY_DOP

- **PDS Threat Library (D.PTL)**

  This fixed library contains details about the distinctive frequency patterns emitted by friendly and hostile radar systems. Like the Doppler Threat Library, it will contain several thousand entries, but the information will be incomplete and in many cases inaccurate.

  Read by: T.IFY_PDS

- **Clutter Map (D.CLT)**

  This database holds details about the level of radar clutter which has been found to exist in certain regions of air–space. This allows the probability that a radar return represents a valid target to be assessed. The database is constantly updated as new radar scans are made.

  Read by: T.TRK_TAR

  Written by: T.TRK_TAR

- **IFF Code Database (D.IFF)**

  This fixed database contains the IFF coding patterns which are used to identify aircraft. The codes are complex and are varied according to both the region of air–space and to the time of the day. Hence the database may have several hundred entries.

  Read by: T.IFY_IFF

- **Radar Scheduling Queue (D.SQU)**

  This database holds information which is used for scheduling the issue of commands to the radar. The queue contains antenna steering and frequency selection commands, along with the time that they are to be issued to the radar hardware.

  Read by: T.SCH_RAD

  Written by: T.TRK_TAR and T.INT_IFF

The following data transformers are required:

- **Schedule Radar (T.SCH_RAD)**

  This operator uses data from I.R_T_CLK and D.SQU to issue commands to O.ANT_OUT at the appropriate time.

  Reads: I.R_T_CLK and D.SQU

  Writes: O.ANT_OUT

- **Interrogate IFF (T.INT_IFF)**

  This operator uses data from D.IFF to schedule IFF interrogate requests in D.SQU.

  Reads: D.IFF, D.SQU

  Writes: D.SQU

- **Decode IFF (T.DEC_IFF)**

  This operator identifies IFF Reply messages in D.FAT. It uses D.IFF to decode these and update the track information stored in D.TRK.

  Reads: D.FAT, D.IFF and D.TRK

  Writes: D.TRK

- **Track Target (T.TRK_TAR)**

  This operator uses the information from T.ISO_PLT and D.CLT to update the track information stored in D.TRK. In addition, this operator updates the information stored in D.CLT and tracks the targets by scheduling steering commands in D.SQU.

  Reads: D.FAT, D.CLT, D.SQU and T.ISO_PLT

  Writes: D.CLT, D.TRK and D.SQU

- **Identify Target by PDS (T.IFY_PDS)**

  This operator uses data from D.PDS and D.FAT to update the identity information stored in D.TRK.

  Reads: D.PDS, D.FAT and D.TRK

  Writes: D.TRK

- **Identify Target by Doppler (T.IFY_DOP)**

  This operator uses data from D.DTL and D.FAT to update the identity information stored in D.TRK.

  Reads: D.DTL, D.FAT and D.TRK

  Writes: D.TRK

- **Radar Data Filtering (T.RAD_FIL)**

  This operator reads data from T.ANT_INP and performs a number of compression and signal analysis techniques upon it. The resulting continuous frequency distribution patterns are stored in D.FAT.

  This operator will be implemented using Digital Radio–Frequency Memories (DRFMs).

  Reads: T.ANT_INP

  Writes: D.FAT

- **Isolate Plots by Doppler (T.ISO_PLT)**

  This operator filters the frequency spectrum in D.FAT using Doppler analysis to isolate moving plots from background clutter. It updates D.CLT accordingly.

  Reads: D.CLT and D.FAT

  Writes: D.CLT and T.TRK_TAR

It can be seen that the system can be naturally decomposed into a large number of operators which read and write several large shared databases.

In the following analysis we will consider one operator, T.TRK_TAR, and one database, D.TRK in greater detail.

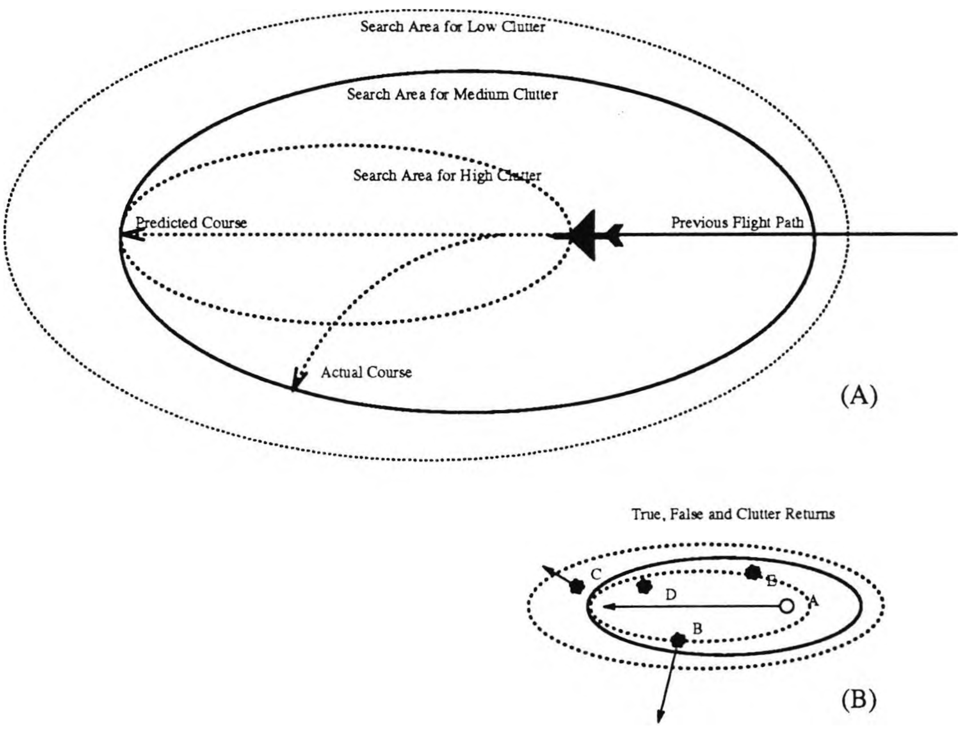

Search Area for Low Clutter
Search Area for Medium Clutter
Search Area for High Clutter
Predicted Course
Previous Flight Path
Actual Course
(A)
True, False and Clutter Returns
C
D
B
A
B
(B)

Figure B.8: Tracking a Target

B.5.2 T.TRK_TAR

As it is necessary to maximise a radar's power efficiency, it is desirable to follow the movements of the targets it has already detected so that they can be relocated with the minimum expenditure of energy. This is called *tracking* and is illustrated in figure B.8.

The life-time of a track can be divided into three phases:

- Track Initiation

  The radar detects a plot and associates a probability of validity with it according to the number of scans it appears in. If this exceeds a threshold value taken from the *clutter map*, a map showing the density of ground clutter previously detected in region of the plot, it is assume to be valid. If not, it is discarded.

- Track Continuation

  Once a target is known to exist, its position must be followed from one radar scan to the next. The radar predicts the new position of the target based on its current course and speed. It then calculates a search area, or *tracking dwell*, centered around this predicted position, based on the target's manoeuvrability and the amount of background clutter. The tracking dwell must be large enough to ensure that it includes the target's new position, but small enough to minimise clutter. The size of the dwell is altered dynamically according to D.CLT.

Once the location and size of the tracking dwell has been determined, steering commands can be issued to the radar to search it. When this has been performed, a best match algorithm, using both position and velocity, is applied to merge the new plots with the existing tracks. This gives:

- New plots which have appeared;
- Plots which correspond with existing tracks; and
- Existing tracks for which there are no corresponding plots.

Figure B.8b shows a number of false targets within the tracking dwell of track A's last position. Plot B is within the tracking dwell and has a reasonable direction and speed vector. Plot C is outside the tracking dwell and has an unreasonable speed vector for the distance it must have covered. Plots D and E are within the tracking dwell, but are stationary. B is, therefore, the only plot which could reasonably correspond to the original track.

Modern search radars employ high pulse rates; typically, a pulse is emitted every $100\mu s$ [Mik83]. This greatly reduces the size of tracking dwells because targets move less between successive pulses. This saves radar power and reduces the number of false targets in the dwells. However, it does mean that tracking must be performed both more frequently and quickly.

- Track Termination

  When there has been no successful correlation between a track and the radar plots for a sufficient number of scans, it is removed. This could indicate that the target has landed, flown beyond the range of the radar, or has been shot down.

As the radar must be aimed in real-time, tracking performance is a major factor in the design of AEW computer systems. If the radar emits pulses every $100\mu S$, to maximise its power effectiveness the computer must process all the plots found by the previous pulse in an average of $50\mu S$. As military aircraft travel in large numbers for mutual protection[8], this will require a considerable amount of processing.

To solve this problem we suggest the creation of one T.TRK_TAR process for each track entry in D.TRK. Each process would have a life-cycle analogous to that of a track:

- Initiation

  Track initiation is accomplished by the creation of a new *Cherub* process which is responsible for handling that track. This process performs the following actions:

  - Calculates a circular tracking dwell around the target based upon its current speed. Its direction is currently unknown.
  - Schedules a high priority command for the radar to search this dwell.
  - Performs a read on the appropriate address in a semaphore synchronisation object to block until the radar performs the search.

[8]During the 1982 Bekka Valley conflict flights of 12 or more Syrian aircraft were common.

  - When the search is complete, the process examines the resulting plots for one which is consistent with the previous plot.
  - If such a plot is consistently found in the number of scans dictated by D.CLT, it is deemed to be a track and the process performs track continuation. Otherwise, it is deemed to be noise and the process terminates.

  Once a plot is deemed to be worthy of tracking, it is classified according to its speed. If the speed is below some threshold it is deemed to be ground clutter, otherwise it is taken to be a valid target.

  Even if a plot is random noise which disappears on the subsequent scan of the radar, its tracking process will have executed enough instructions (at least 10,000), by scheduling the radar, synchronising and examining the results, to be cost effective.

- Continuation

  Continuation processes are prioritised so that tracks known to represent ground clutter execute first, thus easing the task of the processes representing moving tracks.

  Track continuation involves the following steps:

  - When the radar next searches the target's tracking dwell, the process examines the resulting plots for one which is consistent with the previous plot.
  - It then removes the plot from the set;
  - It updates its D.TRK record according to its new position and speed;
  - It calculates a new tracking dwell for the track;
  - It schedules a low priority command for the radar to scan this dwell; and
  - It performs a read on the appropriate address in a semaphore synchronisation segment to block until the radar performs the search.

  If a process cannot locate its plot after a prescribed number of radar scans it terminates.

  Once all of the track processes in a region have run, if there are any plots which have not yet been accounted for, new track processes are created and assigned to monitor them.

- Termination

  The process of track termination is consistent with the termination of a *Cherub* process.

It is not unreasonable to assume that the T.IFY_DOP, T.IFY_PDS, T.INT_IFF, T.IFY_IFF, tasks for a given track would be combined with its T.TRK_TAR process to form a single task which examines targets, T.EX_TAR. This merge enables the D.TRK and D.FAT tables to be shared more efficiently. Furthermore, the T.IFY_PDS and T.IFY_DOP operations require considerable off-line processing. Hence they are ideal background tasks for processors which are waiting for the radar to supply target information.

The number of targets that a radar will be tracking, and hence T.EX_TAR processes, can be large; AWACS was designed to cope with over 400 [Mik83]. However, due to the directional nature of the radar antenna, only a subset of these tracks, half say, will be processed simultaneously. This is consistent with horizontal parallelism and suggests that the 400 T.EX_TAR processes be spread over 200 *Cherub* processors.

B.5.3 D.TRK

The D.TRK database contains one record for each track which has been detected. During periods of heavy air-space usage it could hold 500 or more records.

The database is shared for read and write access by six data operators; the T.TRK_TAR operators add, modify or remove records, while T.IFY_DOP, T.IFY_PDS and T.IFY_IFF only modify them. It is necessary to maintain the strict coherence of the records at all times.

The records in the database can be divided into three categories:

- Stationary Tracks

  These are plots which are known to exist, but which Doppler analysis shows are not moving at a high speed relative to the aircraft. These are assumed to be ground clutter. They are stored to allow their easy identification in future scans.

- Tentative Tracks

  These are new plots which are in a state of track initiation; they have not been observed long enough to determine whether they really exist or not.

- Firm Tracks

  These are existing plots which have been categorised as being worthy of tracks. The movements of such tracks are relatively predictable.

A record will be added to the track table whenever a radar search detects a new object and will be removed after it is not found in a number of radar scans.

Studer and Farina [Far85] suggest a typical track record will include information such as:

- Track Identification Code

  The code by which the operator and computer refer to the track.

- Quality Measure

  The probability that the track exists. As a tentative track appears in a number of radar scans, this probability increases until it reaches some threshold value. At this point the track is deemed to be firm.

- Filter Variables

  These are used in setting the signal filters for the target. They take into account factors such as jamming levels and prevailing radar conditions.

- Time of Last Update

  This is the time the track record was last updated. This is used for radar scheduling.

- Track Status

  This contains details such as whether the track is new, tentative, firm, or terminating.

- Information about the track's last six or more positions, representing over a minute of tracking.

  - Track Speed Vector

    This includes the track's current speed and acceleration.
  - Track Direction Vector

    This includes the track's current direction of travel and rate of turn.
  - Track Position

    This includes the track's azimuth, range and altitude.

- Possible Track Identity

  Information gained from Doppler, IFF and PDS which may be of use when classifying or identifying the track.

We estimate that such a record could be contained in a 256 byte page with little wasted space.

The database used in this example is not large, although it would be if a ground surveillance radar was used, but it is accessed in real-time. This makes false data sharing or expensive communication overheads, common with architectures with larger page sizes, unacceptable.

It is asserted that the shared variable programming paradigm presented by *Cherub* is one of the most natural ways of implementing such a shared database. The database fulfills the criteria for being well suited to *Cherub*:

- It is shared for read and write;
- Strict data coherence must be maintained;
- Its records are suited to a 256 byte page; and
- It must be accessed in real time and so false data sharing is unacceptable.

Record locking can be implemented efficiently by using a semaphore synchronisation segment. Each record is assigned a semaphore which processes have to gain before being allowed to modify it.

B.5.4 Process Scheduling

Considering the substantial amount of processing which must be performed by the T.EX_TAR operators, it is reasonable to allocate one processor per two targets which are to be processed simultaneously. If a maximum of 400 targets is assumed, 200 processors will be required.

Although the resulting workload on the processors is relatively light, it is not unreasonable to assume that as the capabilities of radar increase, so will the demand for additional data processing power.

The T.SCH_RAD operator, more then any other, must be performed quickly. Consequently, it must be assigned its own processor.

B.6 Conclusion

In this appendix we have selected airborne early warning (AEW) as an example of a real application which would test the *Cherub* architecture. We have demonstrated how it can be efficiently decomposed into large number of tasks which operate on shared databases and have properties which make them especially suited to Cherub.

The Cherub architecture is able to provide the very high data processing capacity required by the AEW role while maintaining a natural data sharing mechanism which simplifies the structure of the software.

Appendix C

The Network Simulations

Although, due to the difficulties in constructing accurate models, simulations are generally not well thought of[1], they remain one of the only ways of predicting and analysing the performance of complex systems such as *Cherub's* proposed WSI–based communications network.

Chapter four showed that, when loaded with 40 connections, the *Cherub* communications network must fulfill the following criterion:

$$84C + 26,000T \leq 40,000 \text{ ns}$$

Where:

C	=	Time (ns) to make a network connection (network connection latency); and
T	=	Time (ns) to transfer a byte across a network connection.

In chapter five it was stated that the proposed wafer–scale communications network will be able to transfer 128 bits of data every 10 ns. Therefore, the network's connection latency, C, must be less than about 280 ns if an optimal granularity of 10,000 instructions is to be achieved.

This appendix describes the simulations which have been performed to understand whether the *Cherub's* proposed WSI network will be able to achieve this connection latency.

C.1 The Simulator

A 1,000 line C program was written to simulate an eight–inch circular wafer containing a communications network composed of one square cm communication element (CE) tiles. A two dimensional array of structures was used to represent the CEs' communication busses. These busses could be configured into either a mesh or a torus network topology,

[1]Fishman [Fis73] has identified 12 types of data misinterpretations common in simulations and Ören [Ore78] lists 9 main categories and 44 subcategories of tests for the assessment of acceptability of simulation models, programs and data.

although only the latter was really of interest. The incomplete tile sites at the edge of the simulated wafer were not used, giving 316 usable CEs in a perfect wafer.

Circuit switched communication was simulated by allocating busses in a path between a source and a destination CE chosen at random. Once the simulated circuit was made, it was kept open for 19 network clock periods[2], the average length of a connection as determined in section 4.5. The circuit was then cleared, freeing the allocated busses.

If during a circuit's construction a required communication bus was already in use, the complete circuit was cleared and the source CE waited for a back–off period before retrying. This prevented deadlock. The simulation was able to support various network back–off schemes:

1. The connection failed; it did not retry.
2. The connection retried after a constant delay. (After n network clocks.)
3. The connection retried after linearly increasing delay. (After $n, 2n, 3n, 4n \ldots$ network clocks.)
4. The connection retried after exponentially increasing delay. (After $n, 2n, 4n, 8n \ldots$ network clocks.)

Experiments showed that, generally, the constant delay back–off scheme results in the lowest connection latency in networks containing up to 40 connections. A constant delay back–off period of 6 ± 1 clocks[3] was therefore used in all of the experiments.

The average number of connection in the network was determined by varying the *issue chance* — the probability that an idle CE would attempt to make a new connection on each network clock tick. In all the experiments various *issue chances* were tried, although those that produce network loads of about 40 connections were of most interest.

In all of the experiments the networks were run for 10,000 simulated network clock periods (100 μs). The networks were first allowed a period of 1,000 network clock ticks to 'warm up' (10 μs), thus ensuring that they were not devoid of connections at the start of the simulation.

Two sets of experiments were performed:

1. An investigation into the performance of different routing algorithms. This was used to find the best possible circuit switched routing algorithm.
2. An investigation into the effect of different network yields on the routing algorithm found to perform best in the previous experiment.

These are discussed in detail in the following two sections.

[2]The simulations were independent of the actual network clock rate employed. However, it is assumed that a 10 ns clock will be employed.

[3]This is approximately one quarter the average connection length: an estimate based on the assumption that, on average, a collision will occur when a connection is half constructed, being blocked by another connection which is half way through its transmission. The random variation helps to prevent live–locks, where a number of connections continually collide because they are using the same back–off steps.

C.2 Investigating Different Network Routing Algorithms

The first experiment examined the performance of several routing algorithms under different network loads. Eight routing algorithms have been devised:

1. Normal MESHNET Routing

   At each hop the head of a connection will first attempt to travel north or south in order to reduce the relative vertical distance between it and its destination, according to the route held in the α map. If there is no vertical separation, then it will attempt to decrease the horizontal separation by travelling east or west. If the route in the α map is blocked, the one in the β map is attempted. If this is also blocked the connection is immediately dissolved.

2. Offset Straight Routes

   One of the problems with scheme one is that if either the vertical or horizontal separation between the source and the destination is initially zero, an alternative routing decision cannot be made upon a collision.

   This scheme attempts to overcome this problem by forcing the head of a connection to take a single hop away from its destination if either the vertical or horizontal separation is initially zero. Unfortunately, this requires a third routing map, which is only used on the initial hop of a connection.

   The simulations have shown that this routing scheme is slightly inferior to number one; the greater routing choice provided by the initial sideways hop does not outweight the extra hops it incurs.

3. Back-Up One Step On Collisions

   Another limitation of scheme one is that once the head of a connection becomes blocked both horizontally and vertically, it simply gives up.

   In this scheme, once the head of a connection becomes completely blocked, the head takes one hop (at most) back and tries the other route (if any) it could have made.

   The simulations have shown that this scheme's extra attempts at making a connection are very cost effective while the network is under light and moderate loads (60 connections), but not so when under heavy load, where they just add to network contention.

4. Zig-Zag Routing

   In scheme one, the head of a connection was always first routed vertically and then horizontally. That results in indirect, L-shaped, connection paths.

   In this scheme, the head of a connection is routed so as to minimise the greatest relative distance, either horizontally or vertically, to the destination. This results in zig-zag shaped connection paths.

   The simulations show that this scheme performs slightly worse then number one. This is because, in general, the number of paths blocked by a circuit is minimised when the head makes as few direction changes as possible. This is clearly not the case with zig-zag movement.

5. Recursive Routing

   This scheme is similar to number three except that the head of a connection can takes as many steps backward as necessary, until all possible routes have been exhausted.

   The simulations have shown that this scheme's excessive attempts at making a connection are not cost effective except when the network is under very light loads (20 connections).

6. Fixed Routing

   This scheme uses a fixed routing path; if a collision occurs, the connection is immediately dissolved, rather then attempting another route. This has the advantage of only requiring a single routing map.

7. Wait Before Backing–Up

   One of the main disadvantages of scheme three is that once the head of a connection becomes blocked, it is immediately backed–up one step — in effect, moving it away from its destination.

   This scheme attempts to overcome this by, instead of immediately backing–up, waiting for up to half a message length for a route to clear.

8. Permanently Change Direction of Travel On Collision

   Another disadvantage of scheme one is that, although the head of a connection will temporarily use map β to avoid a collision, it will always revert to using map α again. On collisions, this results in zig–zag connection paths, which scheme four has shown to be bad.

   This scheme is similar to number one except that once the head of a connection has used a given map for routing, it will continue to use it until another collision occurs. This, hopefully, will reduce zig–zagging.

Each routing algorithm was simulated a number of times with different *issue chances*, thus varying the average number of connections in the network. To simplify the analysis of the results a 100% network yield was assumed.

C.2.1 Results

The performance of the eight circuit switched routing algorithms is illustrated in figures C.1 and C.2. The graphs show that only routing algorithm eight is able to achieve a connection latency below 280 ns in a network loaded with 40 connections. It should be noted that the shapes of the curves indicate that the communications network was being pushed to its limits and was on the brink of thrashing. This is not unreasonable, however, as it is always desirable to employ the smallest level of granularity that the communications network is able to support.

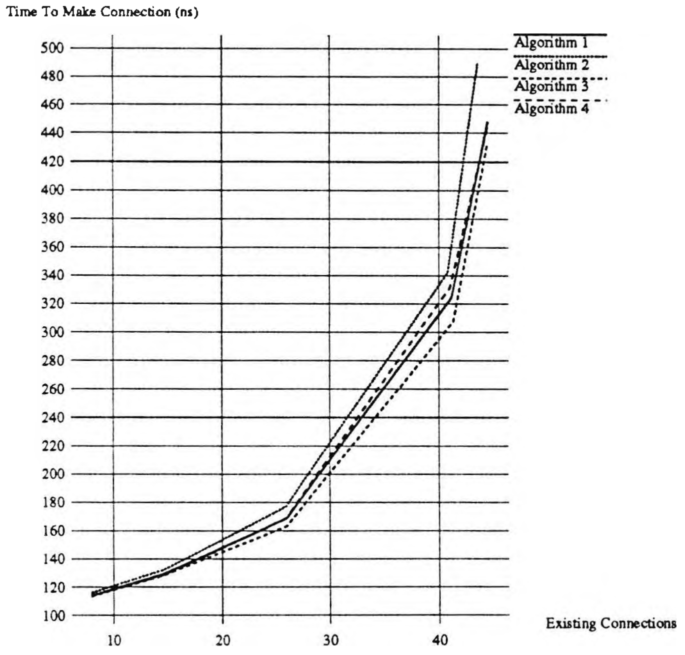


Figure C.1: The Effect of Routing Algorithms 1-4 on the Time to Make a New Connection

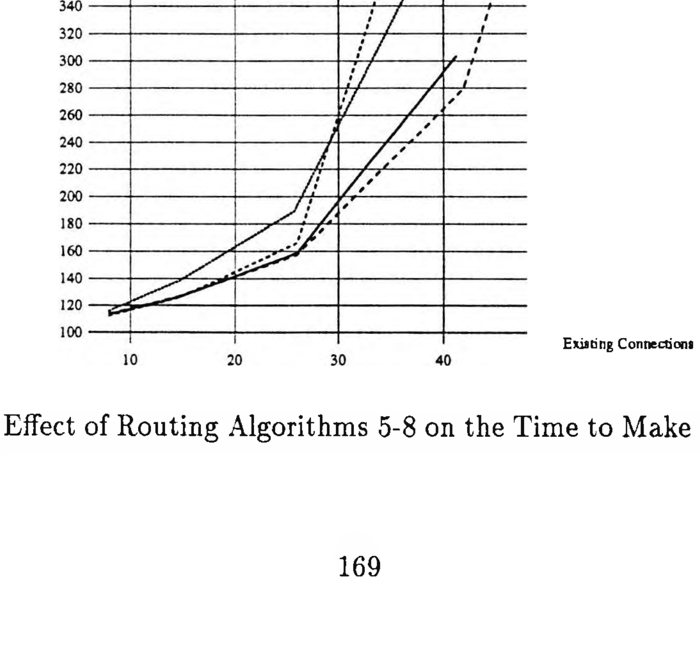


Figure C.2: The Effect of Routing Algorithms 5-8 on the Time to Make a New Connection

C.3 Investigating the Effects of Different Network Yields

Having determined that routing algorithm eight is highly effective in a defect-free network, it was necessary to investigate its performance in an imperfect network. The simulation was altered so that a proportion of the CEs were randomly assumed to be defective[4]. The simulation then constructed routing maps for the wafer which avoided the defects. This was achieved using a recursive 'flood-fill' algorithm which examined all possible routes between a source and destination CE. As an optimisation, a route was only followed until either it reached its destination, or it equaled the length of the shortest route so far found.

C.3.1 Results

Two experiments were performed:

1. **The Effect of Network Yield Upon the Average Communication Path Length**

   In this experiment the *issue chance* was kept constant (at 0.1) while the yield of the network was varied. Networks with mesh and torus topologies were simulated for comparison.

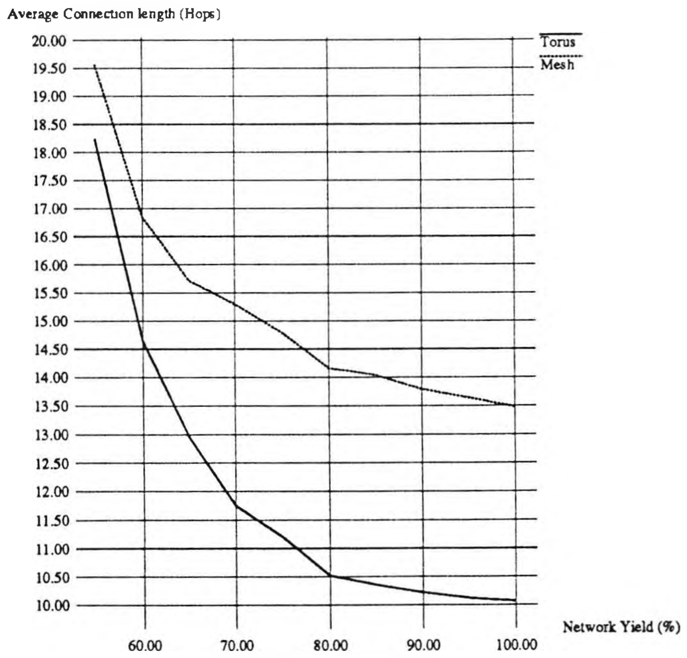


Figure C.3: Effect of Network Yield on Average Connection Path Length

The results of the experiment are shown in figure C.3. It can be seen that when the network yield is high, the average path length in the torus network is much shorter

[4]It was assumed that a single defect would render a CE completely inoperative. In reality this will not be the case, as most defects will occur in the replicated communication busses.

then that in the mesh. However, as the network yield falls, the difference between the two types of networks becomes less marked.

2. **The Performance of Routing Algorithm Eight In Imperfect Networks**

   In this experiment, routing algorithm eight was simulated in networks with different CE yields. The number of connections in the networks was determined by varying the *issue chance.*

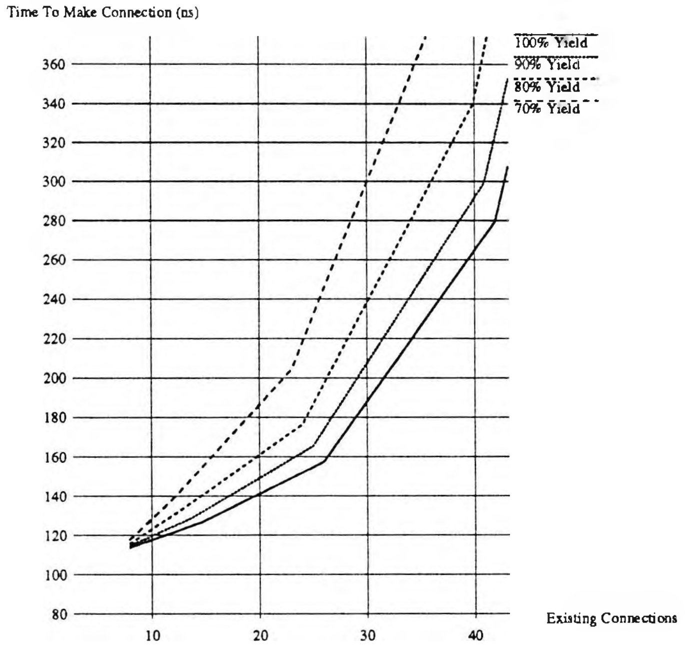


Figure C.4: Effect of Network Yield on Connection Latency

The results of the experiment are shown in figure C.4. The graph shows that the yield of the communications network dramatically affects its performance. In general, the higher a network's communications yield, the lower its connection latency will be; communication bottlenecks occur where connections are routed around defects. Very high yields indeed are required, certainly above 90%, if *Cherub's* network is to achieve the desired latency. Due to the in-built redundancy of the proposed double bus system and the planned use of conservative 0.4 micron fabrication technology, it is expected that such a yield can be achieved.

C.4 Conclusion

The proposed network was simulated, showing that it is able to provide the level of performance required given that a network yield in excess of 90% can be achieved. It was asserted that the fault tolerant design of *Cherub's* network will make this possible.