Computer Architecture for Wafer Scale Integration

Paul Anderson
Department of Computer Science
City University

December 1990

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.

Contents

Acknowledgements

This work was carried out in the Department of Computer Science at City University from 1986 until 1990, firstly in my capacity as a Research Assistant on the COBWEB project, and then as a lecturer in the department.

This work has been influenced by many discussions with colleagues and friends throughout the years. The greatest single influence has been that of my supervisor, Professor Peter Osmon. I wish to convey my heartfelt thanks to Peter for his help, support, and inspiration and for fostering and encouraging a lively research environment.

In addition, I would like to convey my thanks to the following, each of whom has been especially helpful.

Steve Ashcroft
David Bolton
Simon Croft
Stephen Dedalus
Hugh Glaser
Chris Hankin
Paul Kelly
Malcolm Shute
David Till
Phil Winterbottom

My wife, Randi Kepecs deserves special recognition. As well as providing some of the original encouragement for starting this work, she lent her support and inspired confidence, especially during the darker periods of writing up. Finally, I would like to thank Noel and Phyll Anderson for their unwavering help and support for the last twenty seven years.

Declaration

Abstract

This thesis addresses the problem of specifying, designing and implementing parallel computer architectures based on wafer scale integration (WSI). The requirements and constraints of WSI are considered and the class of computer architecture that is most suited to the technology is identified. This takes the form of a regular array of similar processors connected by a general purpose communications network. The communications function of the array is considered separately from the processing function.

Three routing algorithms for regular two dimensional arrays of processors are proposed. These are specified, and their properties are analysed. The performance of each of these is measured by simulation under varying conditions.

The problem of specifying and designing the processors is addressed next. A functional language engine is chosen as the target architecture. The processor specified and designed is a parallel graph reduction machine (named Cobweb) that uses directors as the instruction set. The programs executed on the machine are compiled from strictness analysed Hope$^+$ via FLIC to a director and parallelism annotated directed acyclic program graph.

A specification of a single processor, using a novel object-oriented parallel graph rewrite notation (named Paragon) is given. A methodology for translating Paragon specifications into a hardware design is given. This methodology is applied to the Cobweb specification. The resulting design is seen to be inefficient, so the specification is transformed, whilst retaining its semantics, to make it more efficient, and the translation process applied again. The resulting design has been simulated and some of the results from the simulator are shown.

The COBWEB specification is expanded to a multiprocessor one. Some of the problems in producing a specification for this type of machine are discussed. This specification is used to produce a design.

The results from a simulation of the multiprocessor COBWEB along with the results from the communications network chapter are used to predict the performance of a multiprocessor WSI graph reduction machine.

The thesis ends with a discussion of the merits and problems of specification and evaluation of this type of computer architecture. The communications architecture is found to be especially suitable for WSI; the specification and design tools are found to be sufficiently powerful, although limited in their scope. Finally the conclusion is drawn, with caveats, that WSI is a suitable technology for parallel graph reduction.

Chapter 0

Introduction

The recurring dilemma in the study of the design of computer architecture is whether raw speed is preferable to programmability. In traditional systems, this has been a simple tradeoff. However, as alternative programming paradigms and implementation technologies emerge, it is becoming more reasonable to expect speed *as well as* programmability. This thesis reports on an investigation into one way of achieving this goal. We attempt this by coupling two very different areas of computer science and engineering: Wafer Scale Integration (WSI) for the increased performance; and parallel graph reduction as an implementation technique for a class of languages with highly desirable features. In one line, the question that this thesis addresses is: "Is graph reduction feasible on a wafer?".

Inevitably, many other issues are raised as this question is addressed. Two of these become particularly prominent in this work. These are communications for WSI, and the formal specification and derivation of hardware.

0.1 Wafer Scale Integration

Since the advent of the integrated circuit, the method of manufacture has remained fairly constant. A large number of individual circuits are etched using photolithographic techniques onto a large slice of ultra-pure silicon known as a wafer. The wafer is then diced into "chips", and each is tested individually. Unfortunately, because of defects originally in the silicon crystal, or introduced at the time of manufacture, many of the circuits will be faulty and will not work. After testing, the working circuits are packaged and delivered, and the non-working ones are discarded. The principle idea behind WSI is that the entire wafer is packaged into a working product and sold.

The motivation behind moving to wafer scale integration can be summarised as follows. Basically WSI is

- cheaper
- faster
- smaller
- more reliable

It is cheaper because the dominant costs in all phases of production are reduced. It is faster because slow off-chip communications are reduced. It is smaller because more components can be packed into a smaller space. It is more reliable, because there are fewer unreliable off-chip connections. These are compelling reasons, but the reason that WSI has not taken the world by storm is because of the scale and the difficulty of the one dominating problem. This is that the device must be guaranteed to work in the presence of the inevitable defects, ie it must be fault-tolerant.

In the past, this problem has proved so intractable that despite the obvious benefits of WSI, designers have opted to stick with tried and tested VLSI technology rather than attempt ambitious solutions to the fault-tolerance problem. As they have demanded increasing levels of integration and speed from their systems they have remained happy with their choice, because each new generation of VLSI component has consistently managed to deliver more performance over the previous. However, time is running out for VLSI. It is becoming much harder to extract yet more speed, and to make circuits even smaller.

What would a WSI device look like? Relatively vast areas of silicon are available, yet we know that a lot of circuits will be unusable because of faults. To get round this many circuits will be replicated, so that if one is faulty, another can be used instead. The replication may be at many levels, for example from extra rows and columns in a block of memory, to entire microprocessor blocks being replicated across the silicon. It is the latter type of system that we are most interested in.

At the same time WSI is offering substantial performance advantages over VLSI. The most dramatic of these is the vast increase in communications bandwidth between system components on the wafer. VLSI systems are limited by their off chip communications which tend to be slow, power hungry, unreliable, and limited by their physical size. However communications between blocks on a wafer do not have any of these problems. Very wide parallel data paths are fast, cheap and reliable.

Given such a system, where there are a large number of identical processors that can communicate with each other, we can separate the processor and communication function and consider each in turn. We call the processor-communication pair a *node.*

0.2 Wafer Scale Communications

Communications on a wafer are not straightforward. The nature of the technology is such that long connections must be avoided, so the natural solution is one where each node can communicate with its physical neighbours only. Communications then take place as a number of steps from neighbour to neighbour. Communications between non-neighbouring nodes must then be via a set of intervening nodes. Of course, if a node is faulty, then it must be avoided. A suitable analogy is of an explorer in the dark equipped with a weakly powered lamp. The explorer needs to get from A to B as quickly as possible, and cannot afford to spend much time deciding where to go next. Unfortunately the way is made dangerous by patches of quicksand randomly placed in her way. The lamp allows her to see only a yard in front, but happily this is the length of her stride, so she can always avoid unwittingly stepping into the quicksand. However, this does not stop her from getting into a situation where she is totally surrounded by quicksand except for the way she came in. We provide three ways of allowing the explorer to complete the journey safely and in good time. The first is equivalent to planting a homing beacon on the destination, so that she can see the direction in which she should be going, and how far away she is. For the second, we provide a set of instructions at the outset in the form of "two steps forward, turn right, one step forward....". For the third, we provide a roadsign at each position that indicates "this way to your destination $\boxed{\rightarrow}$ ".

Each method has its advantages and disadvantages. For example, method two will get her to the destination in fewer steps than method one, but if even one of the instructions in method two is wrong, then our explorer will end up anywhere but at the correct location.

Each of these methods has been designed as a communications algorithm for WSI, and we will discuss and evaluate each in turn.

0.3 Graph Reduction

Since the invention of the electronic digital computer, the dominating architecture has been the "von Neumann". This is the class of computer designs that consist of a single processor plus some memory, where the program that the processor runs is stored in the memory. The processor executes one instruction at a time, after which it computes where the next instruction to be executed is located. This class of computer was designed for the efficient execution of a particular class of languages known as the sequential imperative languages. Over the years designs have become highly optimised for this task, and present day von Neumann computers can now execute sequential imperative programs fast and efficiently.

However, at the same time large programs written in the sequential imperative paradigm have become increasingly unwieldy. The problems of managing large programs with complicated relationships of components are immense. It is difficult to state that the programs match their specification, and they are expensive to maintain. This has led to the "software crisis" — the realisation that programming large systems is difficult.

Many solutions have been proposed to deal with this problem. The one that is of interest to this thesis is the recognition that the underlying programming paradigm — the imperative sequential one is fundamentally unsuited to the requirements of constructing large software systems. There many other paradigms — the object oriented paradigm is one that has received most attention recently. However, the one we deal with in this thesis is the functional paradigm. In this paradigm, programs are expressed as sets of functions mapping input to output. As the functions are pure mathematical entities, the task of reasoning about programs is made much easier than with the imperative paradigm.

The paradigm has a hidden bonus — functional programs have the property that expression evaluation is guaranteed to be side effect free. This means that as a given expression will always evaluate to the same result, it does not matter when it is evaluated as long as it is safe to evaluate it in the first place. Many expressions can thus be evaluated concurrently. It is much easier to exploit hidden parallelism in functional languages than it is in the imperative paradigm.

Graph reduction is a technique for the execution of functional languages. In this thesis we present a formal specification for a graph reduction architecture and translate it into a hardware design. We prototype the design using a simulator, and we use the results from the simulator to predict the performance of a graph reducer.

0.4 Formal Specification of Hardware

As the level of integration of hardware has increased, hardware systems have become much more complex. In the past it has been acceptable to assert that a hardware design is correct simply because it has been tested thoroughly. However we have long been in the position where it is simply impractical to test hardware designs completely because the number of test input/output combinations has grown combinatorially with the complexity of the design. At the same time, users of such systems are placing increasing confidence in them, and are understandably becoming decreasingly tolerant of design faults. When a safety critical application has a component whose reliability cannot be guaranteed, then the usability of the entire system is brought into question.

A proposed solution to this problem is the use of formal methods for hardware design. This way, designers hope to exclude design faults by deriving a design from a specification using rigorous mathematical techniques. Faults can then be proved absent rather than tested for presence.

There are various notations for formally specifying hardware. Unfortunately most of these are at a rather low level and express requirements in terms of physical hardware blocks with a specified topology and certain timing characteristics. As these are at such a low level, they make for fairly easy translation into hardware designs, and it is not hard to show that the design conforms to the specification.

However, there are higher levels at which we wish to express our requirements. For example if we require a system that behaves dynamically, one that grows and shrinks in size and capacity as demands are made of it, then instead of specifying it at a level which emulates the dynamicism, we would like a notation that allows us to express such behaviour directly. In this thesis we use such a notation. It allows us to leave the implementation of its behaviour to the system. Unfortunately this makes the process of producing a hardware design much more difficult. Eventually the design must be in terms of a physical hardware system, and as these systems cannot be physically dynamic such behaviour must be emulated.

We provide a route from this very high level requirements specification to a low level hardware design in terms of a number of logic circuits connected by wires.

0.5 Contribution

The main contribution of this thesis is threefold. Firstly, designs and performance results for a series of novel communications architectures for WSI. This work builds on previous work in [KS86]. Secondly, a formal methodology for transforming very high level specifications into hardware designs. This builds on the hardware specification language work of [BHK90]. Finally, some insight into the properties and problems of wafer scale graph reduction. This expands on some of the work done on the two Alvey COBWEB projects [HOS85, AHK+87, ABH+89].

0.6 Structure

The structure of this thesis is as follows. In chapter 1 we introduce the main subject areas of this thesis — wafer scale integration (WSI) and parallel architectures. We give the motivation for WSI and identify its major strengths and weaknesses. We introduce parallel architectures, and identify

the requirement that they be specified rigorously. We identify the type of architecture most suited to WSI. In chapter 2 we look at communication architectures for WSI and identify one which is especially good. In chapter 3 we give a full specification for a graph reduction architecture at a very high level, and propose a methodology for transforming such specifications directly into hardware. We apply our methodology to our specification and produce a prototype in the form of a simulator. In chapter 4 we estimate the performance of a parallel graph reducer on a wafer. Chapter 5 presents the conclusions, which in brief are that WSI is a suitable technology for parallel graph reduction.

Chapter 1

WSI and Parallel Architectures

The purpose of this chapter is to review the fundamental issues in the study of the two main subjects addressed by this thesis. The first of these is Wafer Scale Integration. The second is parallel computer architecture.

1.1 Review of WSI

Since the invention of the integrated circuit, the trend has been towards further levels of integration. From single transistors on a chip, we have seen an evolution to gates on a chip, to simple system functions, and on to entire microprocessors.

From early in the history of the technology the method of manufacture has been to etch many devices on a single wafer of silicon, dice the wafer into individual devices, test each device, discard the non-functional ones and package and deliver those that work. The idea behind WSI is that entire systems are fabricated on a single slice of silicon, and it is this slice that is packaged and delivered to a user.

There is a growing interest in WSI as reflected in the growing body of literature. [ST86, Lea87] report on two conferences on WSI. Later conferences have been held, but proceedings have not yet been published. [Tew89] is the definitive guide to the problems of implementing wafer scale systems. His introduction provides a readable overview of the subject. Two major projects in Europe and the UK have been investigating WSI. The Alvey project 073 is approaching completion. Its goal was to produce two technology demonstrators, one a non-regular signal processor, the other a regular SIMD image processing module. The ESPRIT 824 programme, started in 1986 aims to produce three technology demonstrators: a large RAM, a systolic array, and a highly fault tolerant microprocessor.

There are two major approaches to WSI: the monolithic and the hybrid (or "jellybean"). The monolithic is where the device is constructed on a single slice of silicon. The hybrid approach is where diced chips are bonded directly onto the surface of the wafer. The wafer can then carry communications or memory. The scope of this work is monolithic WSI only.

The discussion will outline the issues involved in WSI, concentrating on those that are of relevance to the parallel computer architect. Physical implementation issues are beyond the scope of this thesis, although work in these areas will be referred to in many places.

1.1.1 Why WSI?

As mentioned earlier, the reasons why WSI is a better technology than VLSI are as follows:

- It is faster
- It is more reliable
- It is smaller
- It is cheaper
- It is the natural successor to VLSI

In this section we will explain why the technology has these properties in the context of the differences between a WSI implementation of a system, and a VLSI version of the same system with the same functionality. That is, say we construct a system from VLSI from a number of discrete packages mounted on a PCB, connected to each other using PCB tracks. If we construct a functionally identical system in WSI, with each component written directly onto the silicon, and connected using on-wafer connections, then what are the properties of the WSI system compared to the VLSI implementation.

We will then discuss some of the problems associated with the implementation of wafer scale devices.

Technology Issues

WSI is in fact a fairly old idea, but has met with spectacular failures in the past, notably and infamously with the Trilogy project [Pel83]. This is not because the technology is fundamentally unsound, but is evidence of the difficulty of solving the related problems in the context of a rapidly expanding and vigorous VLSI industry.

Although WSI has been unsuccessful in the past, considerable progress has been, and is being made in addressing the problems that WSI raises.

Much of this progress has arisen as the result of research into increasing integration for VLSI. WSI can thus be considered not as a radical departure from the conventional, but as another step, though in a different direction, in the evolution of integrated circuits.

VLSI has evolved with spectacular speed and has consistently delivered major performance and integration improvements from generation to generation. It has done so while many of the enabling underlying factors remain constant. However the evidence is that these underlying factors are approaching their fundamental limit.

Here we expand on the reasons VLSI is approaching its fundamental limit.

Feature Size. We have seen the continual shrinkage in the physical size of VLSI structures. The evidence is that the laws of classical digital electronics will not hold as devices get much smaller.

Feature sizes have been getting smaller at an exponential rate since the introduction of VLSI [Tew89]. State of the art commercial processes can now deliver chips with sub-micron devices. However the lower limit is expected to be about 0.25μm.

As the feature size gets smaller, laws governing the behaviour of the devices break down. The statistical laws that normally apply simply cannot be relied upon when the number of charge carriers in a device gets small. Devices begin to behave non-deterministically, displaying behaviour that varies around an average.

As dimensions shrink, they begin to approach the physical dimensions fundamental to the classical analysis of electronic devices. Quantum effects come into play and also introduce non deterministic effects. Although there might be a place and an application for such devices, they are quite different from conventional electronic devices.

Integration We have seen increasing levels of integration. This has resulted from the decreasing feature size, but also from improvements in component density and packing efficiency.

Packing efficiency measures the fraction of silicon actually used for devices. Silicon is unusable due to the requirement for minimum separations between devices. The packing efficiency has been increasing, leading to a greater density of components on the silicon, but it will be difficult to increase packing density much further.

Chip Size Of particular interest to the study of WSI is the projected increase in chip size. Chip edge sizes have been increasing exponentially since the introduction of the technology [Tew89]. However as the chip size increases, it becomes increasingly difficult to make chips that are free of fabrication defects.

The manufacture of VLSI devices has until recently relied on the presumption of perfection. If a device has a fault, no matter how small the defect that caused the fault, the entire device is discarded. This is becoming increasingly unreasonable. The system designer must provide for the component to function effectively in the presence of faults.

At the same time it becomes more and more difficult to test these chips as access to components located centrally through the peripheral connections is not easy.

We can see from the above that VLSI has almost exhausted every avenue available for increasing the number of components on a chip. The only avenue left is the one that increases the size of the chip, and which deals directly with new the issues implied. Thus there is a trend which will inevitably lead the study of VLSI towards WSI.

Speed, Reliability and Size

As well as offering increased levels of integration, the technology of WSI offers an increase in component connectivity which has implications on the speed of the system, its reliability, and its size. The number of pins on a package, and indeed in an entire system is a crucial limiting factor on VLSI systems for the following reasons:

- Connections from chip packages to PCBs are the most unreliable component of a system.
- Driving pins involves transforming the on-chip voltage and current levels to off chip levels, a process that consumes both time and power.
- VLSI packages are bulky compared with the chip itself.

These three reasons impose several limits on VLSI systems. First, the chip pin-out must be kept low so as to enhance the reliability. At the same time, the total number of connections in the system must be kept down, for the same reasons of reliability, and because of the physical space occupied by inter-chip connections. Second, the speed at which the chips are driven must be kept artificially low because they are physically distant from each other (compared with internal chip distances), and because driving pads at a high frequency incurs a high overhead in terms of power consumption.

WSI avoids the above problems because connections between circuits on a wafer are reliable, fast, and cheap to drive. Because the circuits are packed much closer together, they can be driven at a much higher clock rate, although for reasons given in section 1.1.2, it is not desirable for clocks to be distributed across the entire wafer. The greatest boon to the designer is

that the level of connectivity between on-wafer circuits is much increased. Extremely wide parallel data paths are easily achievable at levels VLSI can never attain.

Economic Issues

The key question here is if a system designed from WSI components will be cheaper than an equivalent system designed from VLSI components. The cost of such a system can be broken down into five areas[Sum86]:

Die cost includes the cost for processing the silicon, dicing the wafer into chips, and testing using probes.

Component cost consists of packaging the device, and doing a final component test.

Board cost consists the cost of the PCB the component is attached to, mounting the component on the PCB, and a test of the PCB.

System Hardware cost will include the cost of connecting the PCBs together, cooling, supplying power, and the cabinet, and top level interface.

Ownership cost includes the cost of maintaining the system throughout its lifetime. The factors that influence this cost are the system's reliability and "diagnosibility", and the cost of spares and services.

We consider each of the above in turn, and focus on the cost differences between a WSI system and a VLSI system.

Die cost The cost of a die is proportional to the number of working components that can be yielded from the wafer. Because we throw away all the non-working VLSI chips, the WSI approach is more cost-effective because we can (nearly) always yield a working device. With a WSI system, no dicing is done, and because testing is usually left until later, the amount of probe testing will be minimal.

Component cost The cost of a package for an integrated circuit is proportional to the number of pins. With a WSI system, only one device needs to be packaged. With a VLSI system the packaging cost is replicated for each component. Although the package for one WSI device will be more expensive than one VLSI package, the total cost will be less simply because the total number of pins will be less.

Board cost The board cost is dependent on its sophistication, and the number of boards. A WSI system wins on both counts as higher integration leads to simpler boards and less of them. The cost of testing a PCB once the components have been mounted is proportional to the number of integrated circuit pins attached to the board. Again, the WSI system has the edge.

System Hardware Cost The cost of board interconnection, the cooling system, the power supplies, and the cabinet is directly proportional to the number of boards. As the increase in integration leads to fewer boards, the cost for a WSI system is less than for a VLSI system.

Ownership cost The cost of ownership includes the cost of services such as space, cooling, and power. As a WSI system will be physically smaller because it contains fewer boards, it will cost less in respect of space occupied. However the situation regarding cooling, and thus power consumption is less clear. The total cost of ownership is dominated by the maintenance costs for the system, and is thus proportional to the reliability of the system. The reliability of any integrated circuit is inversely proportional to the number of pins, as the connections to the PCB are by far the most unreliable component of any such system. So the WSI system wins again.

It is clear from this discussion that a WSI system offers substantial cost benefits over an equivalent VLSI system.

1.1.2 Implementation Issues

The principal difficulty in the implementation of wafer scale systems is the avoidance of the inevitable faults. The problem is approached at a two levels: the circuit level, and the architectural level. Additional difficulties include testing, electrical design issues such as power distribution, and physical design issues such as packaging and cooling. Here we summarise these issues.

Reconfiguration and Redundancy

Given a circuit with a defect, there are three basic methods of working around that defect:

1. Physical repair including laser "zapping" to either cut connections and thus bypass the fault, or to add connections and enable the use of spare components.

2. Electronic switching using programmable switches.

3. Functional avoidance whereby a faulty circuit is simply ignored.

Each of the above methods requires some redundancy to be present. For the faulty block that is configured out, there needs to be a spare present to be configured in as a replacement. In some applications (for example memories) it may be possible to use the spares as well. Redundancy is a double edged sword however, as the discussion on yields will show.

Architectural Structures

A WSI architecture will typically consist of a number of circuits connected together using an interconnect. These can be characterised as having a *regular* or *irregular* structure. There will need to be some type of architectural reconfiguration mechanism to allow faulty units to be disconnected and replaced, or simply avoided. Standard reconfiguration techniques are explained in [NSS89].

Regular arrays The wafer scale device might take the form of a large number of identical or similar circuits replicated in a regular manner across the wafer. This might be so for several reasons. The application area might demand a regular architecture. For example systolic arrays demand an array of identical processors operating in lock-step. However, this has the requirement that the architecture must be made to look regular even though faults may have disrupted the physical layout.

Another reason for regularity is that the application might demand a large number of identical units, irrespective of their topology. In this case the application problem would be mapped on to a number of these units. In this case the physical layout is unimportant.

This class of WSI device requires some sort of interconnect structure exists to connect the elements together. Again there are several classes of interconnect, for example an architecture might only allow a circuit to communicate to its nearest neighbour, or the circuits might be connected in a tree fashion. Again the choice is very much application dependent.

Irregular structures A wafer scale device may simply be an extension of a VLSI type device where the entire silicon area embodies one function, albeit much more integrated. For example an entire CPU for a mainframe computer implemented in random logic. Faults are much less forgiving in this type of scheme — a small set of well placed faults can make the entire wafer unusable. This type of structure requires a much more sophisticated level of fault tolerance, and at several levels.

Defects, Failures and Yield Models

A *defect* is defined as a fault introduced at the time of manufacture. A *failure* is defined as a fault introduced after manufacture, typically appearing as the device is begin used in service. Defects are caused by imperfections in the manufacturing process. A wafer is manufactured from an ingot of silicon which has undergone extensive purification. However the process is not perfect, impurities are left behind and faults develop in the lattice structure of the silicon crystal. As the wafer is processed it is subjected to further stresses, such as impurities introduced by its chemical treatment, and thermal expansion and cooling. The lithography process whereby the wafer is etched may also be imperfect. Thus every wafer that is produced will have at least some faults on it. These faults are not distributed randomly. Analysis has shown that they are more likely to occur towards the edge of the wafer.

The *yield* Y of an integrated circuit manufacturing process is the fraction of working circuits to the total number manufactured. The principal problem with manufacturing large chips including wafers is that the yield decreases exponentially (or nearly so) with the area of the chip. Thus a whole wafer chip, with no fault tolerance has an infinitesmally small chance of working.

Early yield models were applied to VLSI processes with some success. The simplest model uses Poisson statistics and relates Y to the defect density D of a process and the area of the chip as follows:

$$Y = e^{-DA}$$

where D is measured in terms of the number of defects per unit area, and is a constant for the entire wafer. However this model is inappropriate for WSI in several respects. Firstly it takes no account of the fact that defects tend to cluster together, and that different types of defects have different tendencies to cluster. Secondly it does not address the fact that the defect density varies over the surface of the wafer, with the purest section around the center, and defects tending towards the edge.

More sophisticated models are sensitive to this clustering. [HS88] give the *generalised negative binomial* model. The yield of a circuit is dependent on the contributions of different types of defects. Each defect j has a coefficient α_j that models its tendency to cluster. Higher values of α indicate lower clustering, with unity indicating no tendency to cluster. The yield of a circuit that has area prone to fault j, A_j depending on a fault type j is given by

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

and the yield of the entire cell Y_C where there are m types of fault is given:

$$Y_C = \prod_{j=1}^{m} (1 + \frac{D_j A_j}{\alpha_j})^{-\alpha_j}$$

Whichever model is used, it can be seen that the yield of a circuit is strongly dependent on its area. This has several important implications:

- A single large structure within a circuit may cause that circuit to yield unacceptably. This large structure is known as a *yield hazard.* For example long connections such as busses in WSI circuits are known to be significant yield hazards.

- The addition of fault tolerant circuits may have a deleterious effect on the yield. The extra area occupied may cause the entire cell to yield at a lower level than before. The net effect might be fewer working cells than if fault tolerance had not been attempted at all.

Recognition of the radial distribution of defects is important for WSI, as a designer can choose to place critical circuits near the center of the wafer, or the more robust circuitry towards the edge [HS88]. This is modelled using the generalised negative binomial model by making D a function of the distance from the center of the wafer.

Testing

As fault tolerance is of prime importance to WSI, testing of all circuits is necessary so that the non functional ones can be configured out. Testing can be performed either internally, or externally. Internal testing techniques allow the circuits to test themselves or each other. A common technique is to wire in a test circuit that generates a "signature" only if all the components of the circuit are working correctly. The correct signature can be generated during the design phase using simulation and can be hard-wired into the testing circuitry for comparison. This technique is known as *signature analysis.*

External techniques usually take the form of generating test vectors, applying them to the inputs, and checking if the outputs are correct. The number of test vectors required to rigorously test a circuit increases rapidly with the number of inputs, so the number of test vectors for a WSI device is expected to be unreasonably large. It is therefore necessary for test circuitry to be built into the device so as to minimise the amount of external testing.

One technique for testing WSI systems is to have a configuration phase when the device is powered up. This will consist of initiating internal test sequences and constructing an external view of the wafer that indicates where the faulty areas lie. This map can be used for several purposes, for example to switch in spare circuitry, or configuring the communications so that the faulty areas can be avoided.

Electrical and Physical Issues

Electrical issues include the distribution of power and signal. Signal distribution is particularly difficult as clock skew is an unwanted effect associated with distributing signals along long wires. A global clock is needed when all the circuits on the wafer need to operate synchronously, such as in a systolic array to give a regular example, or in a large processing unit. However more liberal architectures will operate asynchronously, so the signal distribution problem can be avoided.

Power distribution has not been identified as a major problem by any of the workers in the field [WL87], although it is important to be able to isolate power shorts. The major physical implementation issue is packaging which comprises mounting, and heat dissipation. [ML86] shows that WSI packaging is not simply an natural extension of VLSI packaging, assuming SIMD processors. [Pit87] proposes some methods of cooling WSI devices. The amount of heat generated is very application dependent. For example the Anamartic wafer memory product generates so little heat that it does not even need cooling fins.

1.2 Parallel Architectures

Since the invention of the electronic digital computer the dominant design has been the "von Neumann" architecture. This architecture is characterised by a single processor executing a program which stored in a linear memory along with the data required for running the program. The model was designed for, and is particularly suited to running a class of programs known as sequential imperative programs.

However it is becoming evident that the premises on which the architecture is based, ie: "imperative" and "sequential" are limited. Imperative languages have been blamed for the "software crisis" — a phenomenon whereby the dominant cost in the production of most software is maintenance. At the same time, as increasing performance is demanded from computer systems, the single processor model is being being abandoned in favour of the multi processor.

The multiprocessor model also demands a different model of computation. Here there is a mismatch between the nature of the imperative paradigm, and this new model of computation. That is, it is difficult to program imperatively so as to exploit parallelism. Again a promising solution to the problem of efficiently exploiting parallel architectures is a different programming paradigm, where parallelism does not need to be made explicit.

More recently, as alternative programming paradigms have come to the fore, it has been realised that the von Neumann architecture has some fun-

damental problems executing such programs efficiently. The cause that has been identified is the memory bottleneck. This is a term that is used to describe the single narrow access point to the memory, and the mismatch between the speed of the processor and the relative sluggishness of the memory.

1.2.1 Specifying Parallel Architectures

If we are to implement a parallel architecture, we first need to be able to specify it. Also, once we have a specification, we need to be able to transform that specification into a design. Specification is a hierarchical process. We start from requirements and move through stages of adding increasing detail and constraints until we arrive at a design that is capable of being implemented. As we progress through the stages we would like to be able to assert with confidence that certain properties remain invariant

For example the stages in specifying an parallel architecture might be as follows [BHK90]:

1. High level requirements statement, known as the *logical model.*

2. The systems architecture, which specifies logical processes and communications between these processes.

3. The processor architecture, where the logical processes are mapped onto virtual processors.

4. The physical architecture, where the virtual processors are mapped onto physical processors with associated memory and communications medium.

The top level requirements statement should be in a language close to the problem domain. However the process of getting down to the physical level is not easy. Some of the problems are elucidated in [BHK90].

The problem of formally specifying hardware has been studied extensively. Gordon at Cambridge has proposed a specification system based on higher order logic [Gor86]. Inmos have specified a hardware floating point unit for the Transputer [MK87]. The specification was originally written in Z, and transformed to OCCAM and then compiled into silicon.

However the specification of hardware systems that consist of a number of concurrently active agents, and especially those that communicate asynchronously and are dynamic, poses special problems. Several teams have proposed languages in which to specify this kind of architecture. DACTL [GKS87] and Lean [BvEG+87] are languages based on term graph rewriting. FP2 is a language based on term algebras for transition systems [SJ89].

AADL is an axiomatic specification language [DD89], with behavioural specifications expressed in an extension of CSP.

1.3 Summary

In the light of the technological requirements and constraints, we can recommend a particular kind of parallel computer architecture as well-suited to WSI.

This takes the form of a regular array of similar processors connected to a general purpose communications network. The communications architecture consists of a number of communication processors which communicate with their nearest neighbours only. Each processor operates asynchronously. Fault tolerance is handled mainly at the architectural configuration level, and each processor has some built in test circuitry. A configuration phase at power up time will arrange for the network and the processors to be tested both internally and externally by a controller. This controller will contain information about the status of the wafer, eg the location of working processors.

We choose a regular array because this is easier to yield than a non-regular architecture. We choose a general purpose communications network so as not to pin down the nature of the processor at too early a stage. So as to minimise yield hazards caused by large structures such as long busses, we permit nearest neighbour communications only. Because of the problem of distributing clock across the wafer, we will allow the processors to operate asynchronously relative to each other.

As the processor will replicated across the wafer, and because we cannot guarantee a regular topology, the processor most suited to this kind of architecture will be one that can operate concurrently with others, and which does not require a regular topology, or synchronous communications. The processor that we choose will be specified formally using a notation that allows us to express requirements at a very high level.

Chapter 2

Communications for WSI

In chapter 1 we concluded that the type of architecture most suited to WSI is one that is a regular array of similar processors connected by a general purpose communications architecture. In this chapter we present several communications architectures that match these requirements, and are independent of any specific processing element.

To aid in the analysis of these architectures we first introduce some terminology and metrics, and identify the requirements of a WSI communications architecture over and above those of a more general purpose communications architecture. We proceed with a review of other communications architectures for WSI. Finally we present three new designs. The designs are presented in historical order of conception, rather than in order of merit. For each design we say how it meets the requirements and estimate its performance in terms of the metrics defined in section 2.1.

In several sections we present results that have been calculated using a simulator. These results are generated by creating models of typical wafers at random, and measuring their properties. The location of faulty CEs on these synthetic wafers accurately reflects the radial distribution of defects found in real wafers [HS88].

2.1 Terminology, Metrics & Requirements

2.1.1 Terminology & Metrics

As explained earlier, the class of WSI computer we are studying here consists of a regular array of processors connected by a communications network. Each processor is connected to the communications network, and the processor-connection pair is called a *node*. A node can be thought of as a communications element (CE) connected a processing element (PE). Ideally, the CE and PE are independent of each other, though this is not always the

case in implementation.

Two types of network are of interest — *packet-switched* or *store-and-forward*, and *circuit switched*. Processors communicate *messages* across the network. In a packet-switched network, units of communication are called *packets* which for this design will be units of 128 bits. Messages are then sent as a series of one or more packets. A communications step in a packet-switched system consists of the complete transfer of a packet from one node to another. This is known as a *hop*. In order to complete a communication from a source node to a destination node the packet must perform a number of hops via a series of intervening nodes. In a circuit-switched network, messages are broken up and sent in pieces from a source to a destination over a path that is fixed and held open for the duration of the communication.

In order to be able to compare different communications networks we need to be able to describe these networks in terms of a number of network terms and metrics. These are introduced here. Some of these are specific to WSI.

Connectivity is defined as the maximum number of nodes any given node can be directly connected to. For example, a mesh type architecture, where each node is connected to nearest four neighbours only, has a connectivity of four.

Routing algorithm. A *communications network* I is a directed graph $I = G(C, N)$ where N represents the set of nodes and C represents the set of communication channels or links between nodes. A particular channel is denoted c_i, and an individual node is denoted n_i. Routing functions determine where to route individual packets. Routing functions are denoted by a type, eg: $\mathcal{R} : N \times N \mapsto C$, that is a function that takes the current node and the destination node, and produces the channel on which to forward the packet. A routing function or algorithm is said to be *correct* if it correctly routes a packet from its source to its destination in a finite time.

Deadlock freeness. One of the fundamental requirements of any routing algorithm is that it be deadlock free. There are various ways of ensuring that any particular algorithm is deadlock free, and these are reviewed for different networks in the context of WSI in section 2.3. Deadlock occurs when a set of processes is blocked because each is waiting for the exclusive use of a resource which is held by one of the other processes in the set, and where no process can back off.

Deadlock occurs in packet switched communication networks when there is a cycle of dependencies in the packet buffer dependency graph. Figure 2.1 shows an example of how deadlock might occur in this manner. Deadlock

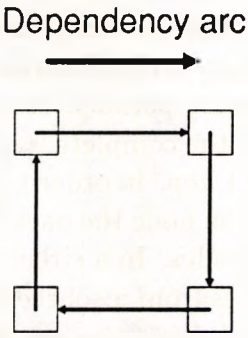


Figure 2.1: Deadlock in a dependency graph

avoidance strategies in general involve either breaking the dependency graph when it forms, or ensuring that it can never form in the first place.

CE/PE binding. A CE and a PE are logically independent blocks. In principle a PE may be connected to a number of CEs. However a PE will not in general be usable if its CE, or set of CEs do not work. A CE on the other hand will generally be usable if the PE is faulty. A PE is tightly bound to its CE (and vice versa) if the functionality of each is dependent on the other. The elements are loosely bound if the CE can work independently of the functionality of the PE. Obviously it is more advantageous in terms of functionality if the elements are loosely bound, and can work independently. Some architectures however are such that they are tightly bound.

Harvest/Sacrifice. A given CE/PE design will have a yield denoted Y_{CE} and Y_{PE}. Some CEs will be in strategically important positions on the wafer such that if these do not work, then they will cause other nodes to be unreachable, that is, the only communications to them is broken. The Harvest H is defined as the proportion of working nodes that can actually be used. H is of course sensitive to Y_{CE}. Usually there is a certain value of Y_{CE} at which H becomes unacceptably low. This is known as the *Yield cutoff point* Y_{Cutoff}.

If the elements are loosely bound then some PEs will not be usable because their CEs will not be working. These PEs are said to be *sacrificed.* Also, some nodes may be working, but unusable because there are no communication paths to them. These are also said to be sacrificed.

Latency l is defined as the amount of time taken between dispatch of a message by a sender, and its complete receipt at the destination. A given network will have an *average latency* $\bar{l}$ and a *maximum latency* $l_{\max}$. The average latency is defined as the average time taken to communicate a message between a source and a destination chosen at random. The maximum latency is defined as the time taken to communicate between the two furthest removed nodes.

Performance. The performance of a network can be defined fairly loosely as how it behaves in terms of metrics such as latency as certain parameters are varied. Parameters of interest include yield, and load. For example if the load rises, this can lead to increased congestion, and the latency might increase.

Overhead is defined as the amount of information in addition to the payload, a message must carry in order to complete successfully. This usually includes the destination address.

2.1.2 Requirements

The requirements of a communications network for WSI are a superset of the requirements of a general purpose communications network, that all messages must be guaranteed to get to their destination within a finite time. A corollary of this is that the network must be deadlock free.

WSI introduces some other requirements. The most important of these is that the communications node must be sufficiently small and simple to allow it to yield well. That is: $Y_{\mathrm{CE}} > Y_{\mathrm{Cutoff}}$. A second requirement is that a WSI communications architecture must be fault tolerant. That is it must be able to guarantee delivery of messages in the presence of faulty nodes. This may involve a configuration phase either when the WSI device is fabricated, or at power on time when CEs are tested and, if necessary, informed of the state of their neighbours.

2.2 Review

There have been a number of studies of communications networks for WSI. This section reviews two of interest.

In the context of regular WSI communications networks, figure 2.2 shows the convention for illustrating the functionality of nodes.

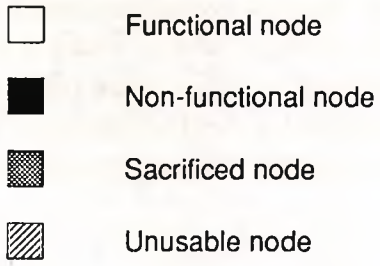


Figure 2.2: Diagramming convention

2.2.1 Catt's Spiral

Catt's Spiral was one of the original communications networks for WSI. It was proposed as a method for configuring working processors in a highly regular array [AC78]. It has had some success. The Catt spiral has been used in several designs for WSI machines. Cobweb-1 [Shu83, Kar87] used the spiral as its communications network. Anamartic Ltd. manufacture a wafer memory device using the Catt spiral [Cur89].

The spiral is basically a string of nodes linked together, usually starting from an node close to the edge of the wafer. Each node on the spiral is identified by an address which is relative to the start of the spiral. To communicate with another node the packets are sent along the spiral in the direction dictated by the difference between the current node address and the destination node address. In the context of the spiral, this is known as *serial* communication. Alternatively packets can be transmitted *radially*, that is a packet can go from an outer shell of the spiral to an inner one in one hop. At power up time the wafer controller initiates a test and configuration sequence which arranges for the nodes to be tested and working ones configured into the spiral.

The spiral is grown from a single node close to the edge of the wafer. This means that a number of external devices wishing to access the wafer must contend for access to this port. Any communication to the outside world must also go via this port.

Figure 2.3 shows an example four-connected wafer configured as a spiral. There are no dud CEs on this example wafer.

Figure 2.4 shows a typical node n_i with its connections to the nodes in the "forward", "backward" and "in" and "out" directions.

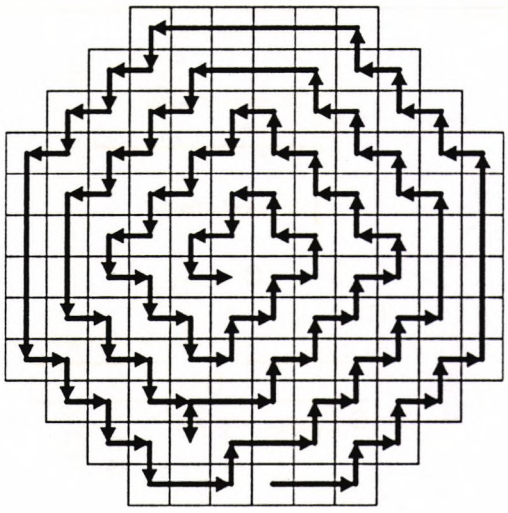


Figure 2.3: A wafer configured as a Catt spiral

Characteristics

Connectivity The standard Catt spiral with no radial routing has a connectivity of two. However with radial routing, the connectivity goes up to four. The spiral can also been proposed for wafers with connectivity six and eight [Shu83], although because eight does not tessellate, some tricky electronics are involved.

Routing algorithm The best thing that can be said about the spiral is that the routing algorithm is so simple that it is extremely easy to implement in hardware and as a result it is very fast. For the simple non-radial case each node has a forward connection and a reverse connection to its neighbours in the spiral. Each node has an address such that nodes near the start of the spiral have low addresses and those towards the end have high addresses. The routing algorithm is then as follows: given the address of the current node and the address of the destination node it returns the channel on which to forward the packet.

$$\mathcal{R}_{\text{Catt}} : N \times N \mapsto C$$
$$\mathcal{R}_{\text{Catt}}(n_i, n_d) = \begin{cases} c_{i,\text{forward}} & \text{if} \quad i < d \\ c_{i,\text{reverse}} & \text{if} \quad i > d \\ c_{i,\text{home}} & \text{if} \quad i = d \end{cases} \tag{2.1}$$

When there are radial connections the routing algorithm is more complex. As well as having links to the forward and reverse directions, the node must also have links to, and know the address of the nodes towards the center and

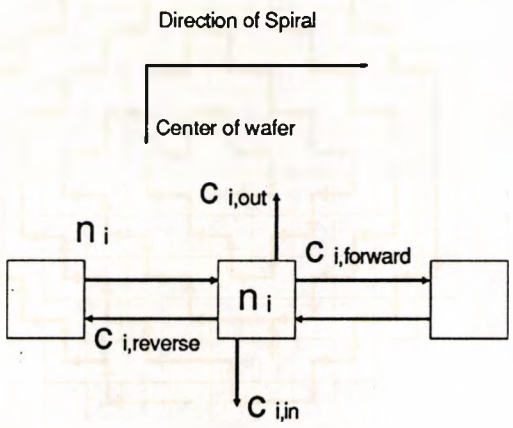


Figure 2.4: A single node in a Catt spiral

towards the edge of the wafer. These new directions are called "in" and "out" respectively. Of course some nodes will not have links in these directions if they are either on a corner of the spiral, or are next to a dud node. To describe this algorithm we need to define the function $link : C \mapsto N$ which takes a channel and returns the node that is connected to that channel (if one exists). This new routing algorithm is as follows:

$$\mathcal{R}_{\text{Catt}} : N \times N \mapsto C$$

$$\mathcal{R}_{\text{Catt}}(n_i, n_d) = \begin{cases} c_{i,\text{home}} & \text{if} \quad i = d \\ c_{i,\text{in}} & \text{if} \quad i < d \wedge x \leq d \\ c_{i,\text{forward}} & \text{if} \quad i < d \wedge x > d \\ c_{i,\text{out}} & \text{if} \quad i > d \wedge y \geq d \\ c_{i,\text{reverse}} & \text{if} \quad i > d \wedge y < d \end{cases} \tag{2.2}$$

where
$n_x = \text{link}(c_{i,\text{in}})$
$n_y = \text{link}(c_{i,\text{out}})$

Deadlock The Catt spiral is naturally deadlock free. This is because a packet is routed either in the directions forward and in or the directions out and reverse. Routing is essentially unidirectional. The cycle of dependencies can thus never form.

Harvest & Sacrifice One of the main problems with the Catt spiral is the fact that in its simplest form, not all nodes can be configured for use, even

though they are working perfectly and they have perfectly working neighbours. This is because there are situations where the spiral advances into what are termed "blind alleys" — positions from which it cannot continue. It must then backtrack and sacrifice the nodes along this section. Figure 2.5 shows an example of a wafer on which several nodes are sacrificed because they are in a blind alley some nodes are sacrificed because the spiral cannot use them.

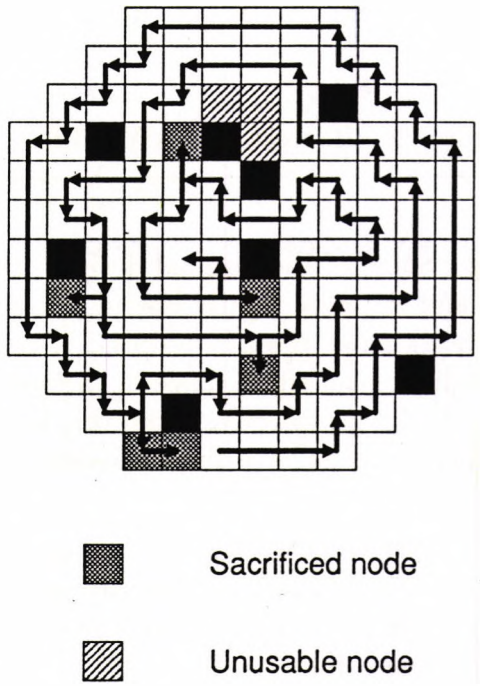


Figure 2.5: Catt spiral sacrificing several nodes that are in "blind alleys", and ignoring others that are unconfigurable.

This means that the harvest of working nodes is far from perfect.

Another problem with this is that Y_{Cutoff} is fairly high. So in order for it to work, the designer of the CE must ensure that Y_{CE} is high. Figure 2.6 shows the results of the simulation of the harvest given by the Catt spiral against Y_{CE}. It can be seen from this that the harvest is far from ideal. Here we see the communications architecture dictating which PEs are usable. The purpose of the CE is to *serve* the processors, not limit them.

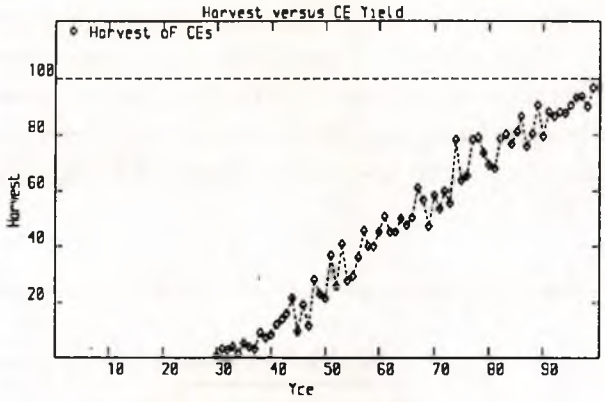


Figure 2.6: Harvest versus Yield for Catt spiral

CE/PE binding Some of the original designs using the Catt spiral were such that the CE and PE were tightly bound [Shu83]. This was because the PE address was the same as the CE address. If the PE did not work, then the CE could have no logical significance in the spiral, therefore it would be sacrificed, despite working perfectly. However, later COBWEB designs and the Anamartic wafer memory have the PE loosely bound to the CE.

Latency With the simple Catt spiral with no radial connections, if there are x nodes configured into the spiral, then the average latency is proportional to half the length of the spiral: $\overline{l} \propto \frac{x}{2}$.

With radial connections the latency is proportional to the dimension of the wafer: $\overline{l} \propto \sqrt{x}$. The Catt spiral has a fairly high average latency compared to the average physical distance between nodes. Consider the example in figure 2.7. The figure to the right shows path of the spiral, and to the left the route taken by a packet. The packet must go all the way round the wafer before it can get to its destination even though it is physically fairly close. This is true even for a wafer with radial connections.

Performance The Catt spiral is essentially a serial configuration mechanism. Although there may be many packets in flight at once in the machine, they must all be travelling along the spiral, or along radial connections, with the majority travelling along the length of the spiral. Queues inevitably build up along the spiral and lead to congestion. The performance does not respond well to increasing congestion[AO88].

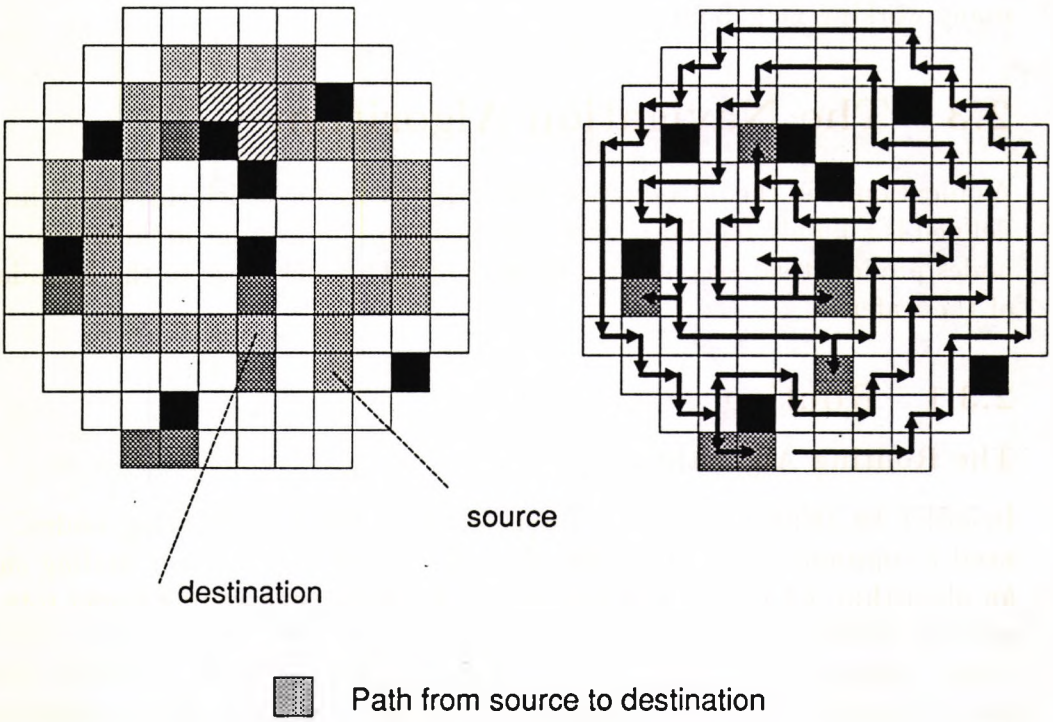


Figure 2.7: An unnecessarily long route taken by a packet following the spiral

2.2.2 Other networks

Tree networks

Some teams have identified tree networks as the basis of possible communications architectures. Lea at Brunel University has developed the WASP architecture for image processing based on string processors [JHL90] which is based on a tree architecture. Brunel WASP is a radically different architecture from the main subject of this thesis in that it is SIMD, and thus operates synchronously, and requires a fairly regular physical topology. This topology is achieved by linking a number of processors into "branches" around a central "trunk". The branches then link together to form a "string". Thus it is not subject to all of the terms and metrics described in section 2.1.

Although the team have had some success with the approach [JHL90], the communications network is not recommended for all applications. This is because a badly placed fault in the structure of the communications network can have disastrous consequences. For example, if a node close to the root of the tree fails, this can potentially cut off all the nodes on the other side of

the branch, and thus waste many good nodes even though they might have many working neighbours.

2.3 The Navigation Algorithm

As mentioned in the introduction the architecture for which this algorithm is defined is a highly regular packet-switched communications architecture. All nodes are identical except for a number of I/O nodes close to the boundary of the wafer.

2.3.1 Routing

The Routing Algorithm

In order to achieve communications between non-neighbouring nodes, we need a communication algorithm. To take a first step towards finding such an algorithm, let us imagine a perfect wafer where everything works totally reliably. Each node on this wafer is fully connected to its four nearest neighbours. We can think of this wafer as a two-dimensional mesh. Each node has a location in the mesh which can be written as the cartesian co-ordinates of that node in the mesh relative to some origin. Each node has a set of channels linking it to its neighbours. These can be envisaged as a set of directions $D = \{north, east, south, west, home\}$ as in figure 2.8.

A simple communications algorithm can be devised based on this addressing system. The routing function is shown below. This function maps the current node and the destination node onto the channel on which to forward the packet. It is a simple matter to prove this correct. For each hop, the packet is routed to a node closer to its destination. As long as the path is not blocked, the packet is guaranteed to be delivered.

$$\mathcal{R}_{\text{Nav}} : N \times N \mapsto C$$

$$\mathcal{R}_{\text{Nav}}(n_{x,y}, n_{xd,yd}) = \begin{cases} c_{x,y,\text{home}} & \text{if} \quad (x,y) = (xd,yd) \\ c_{x,y,\text{north}} & \text{if} \quad y > yd \\ c_{x,y,\text{south}} & \text{if} \quad y < yd \\ c_{x,y,\text{east}} & \text{if} \quad x > xd \\ c_{x,y,\text{west}} & \text{if} \quad x < xd \end{cases} \tag{2.3}$$

However, things are not so simple on a real wafer. The perfect mesh assumed for this routing algorithm is inevitably disrupted by defects. Any routing algorithm for WSI must take account of these defective areas and cause the packet to avoid them. Of course these areas can be arbitrarily complex. Figure 2.9 shows a particularly nasty wafer.

In a wafer that does form a perfect mesh, the routing algorithm above is sufficient. Indeed, there will be regions of a non-perfect wafer where this

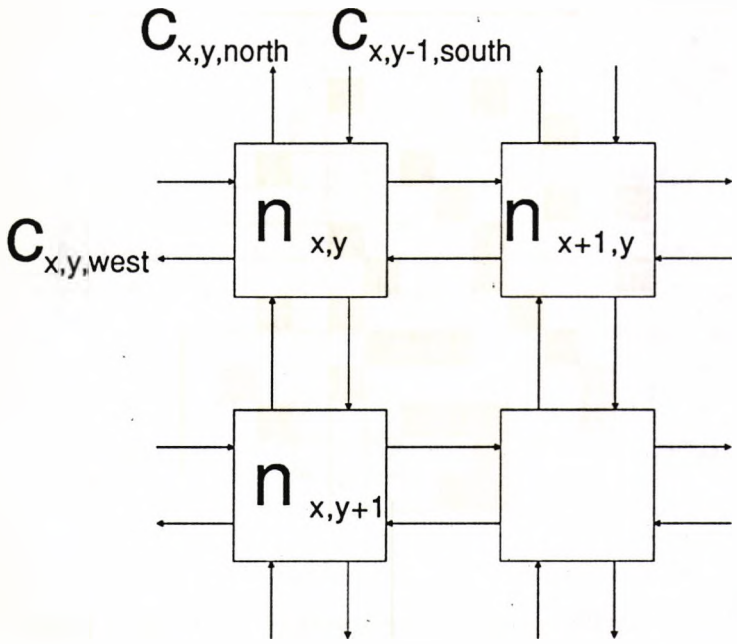


Figure 2.8: Channels to neighbours as compass points

routing algorithm will work. However, when there are defective areas, this naive routing algorithm leads packets into areas from where they cannot progress closer to their destination. The packet cannot backtrack, because the routing algorithm would take it right back to where it got stuck, leading to a livelock situation.

The strategy we adopt is to let a packet know when it is blocked from further progress and to let it then take evasive action. To implement this, the packet must operate in two modes. We call these blocked and unblocked modes. This algorithm has some origins in the cartesian routing algorithm mentioned in [KS86]. However that algorithm requires CEs to be sacrificed using convex wrapping so that "concavities" in areas of faulty CEs can be filled.

The routing algorithm for when the packet is unblocked is simple and is based on the previous routing algorithm with a minor modification. The modification is based on the observation that at some points in the mesh, the packet can move in more than one direction to get closer to its destination. For example if a packet is on the south-western diagonal, it can go either north or east. The modification to the routing algorithm is as follows: If the packet cannot move in the optimal direction, it will be routed in one of the less optimal directions. This relies on any particular channel knowing whether or not it is connected to a working CE. The predicate $dud : C \mapsto \{T, F\}$ indicates if a particular channel is connected to a non-working CE.

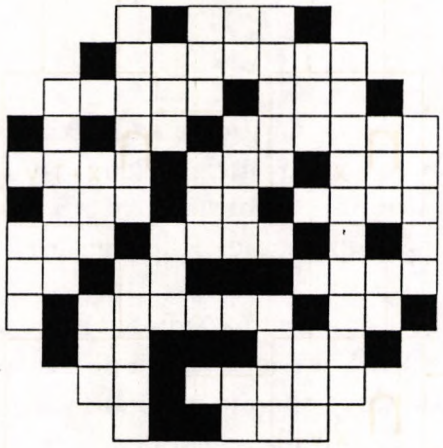


Figure 2.9: A pathological arrangement of faults on a wafer

The formal definition of the routing algorithm with the modification is given below. It is easy to prove that this routing algorithm is correct. Again, each step routes the packet closer to its destination.

$$\begin{aligned}
&\mathcal{R}_{\text{Nav}} : N \times N \mapsto C \\
&\mathcal{R}_{\text{Nav}}(n_{x,y}, n_{xd,yd}) = \begin{cases} c_{x,y,\text{home}} & \text{if} \quad (x,y) = (xd,yd) \\ c_{x,y,\text{d}} & \text{if} \quad \neg dud(d) \end{cases} \\
&\text{where} \\
&d = \begin{cases} \text{north} & \text{if} \quad y < yd \\ \text{south} & \text{if} \quad y > yd \\ \text{west} & \text{if} \quad x < xd \\ \text{east} & \text{if} \quad x > xd \end{cases}
\end{aligned} \tag{2.4}$$

As defined here, this is non-deterministic. If a packet can go in one of two directions, say north *and* east, then either can be chosen. However, the hardware to implement such a routing algorithm would be deterministic and would choose one direction in preference to the other.

If the packet cannot move in any direction that would take it closer to its destination, then it becomes blocked. At the time it becomes blocked the packet is effectively facing a wall of dud nodes. The purpose of the routing algorithm is to move the packet into a position where it can become unblocked and thus resume moving closer to its destination.

The condition that the packet will become unblocked when it is closer to its destination than it was when it became blocked, is essential if we are to prove that the routing algorithm works. If we can prove that when the

packet is blocked the routing algorithm manoeuvres the packet closer to its destination then we can be sure that the routing algorithm is correct.

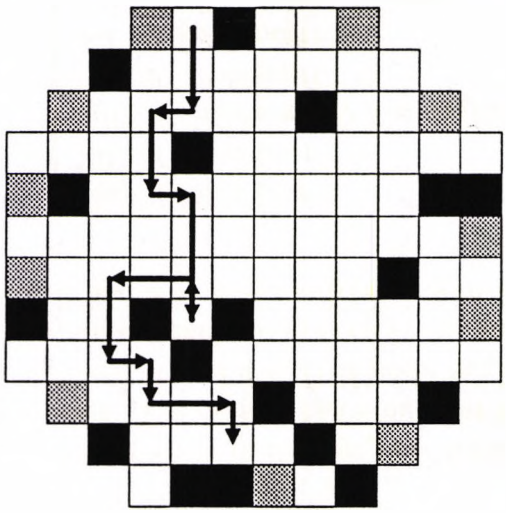


Figure 2.10: The routing algorithm in full action

In fact, getting into such a position is quite easy. All the packet has to do is to follow the edge of the area of duds. This is guaranteed to bring it closer to its destination. Figure 2.10 illustrates the action of the algorithm in the blocked phase.

There is an important optimisation that can be made to the algorithm. Consider the action of the algorithm when it encounters the physical edge of the wafer. One of the options open to it is to crawl round the edge as if the edge were just another area of duds. This is obviously non-optimal. The optimisation that can be made is for the packet to recognise that it has hit the physical edge and to simply go back the way it came. Figure 2.11 shows this optimisation.

Obviously this routing algorithm requires the packet to carry some state. which can change as the packet is routed towards its destination. For example the mode needs to be changed as the packet becomes blocked and unblocked.

This requires the routing algorithm to be of type

$$\mathcal{R} : P \times N \times C \mapsto P \times C$$

where P is the set of packet states. The "state" that the packet needs to carry is denoted $p_{s,b,h}$ where each subscript denotes

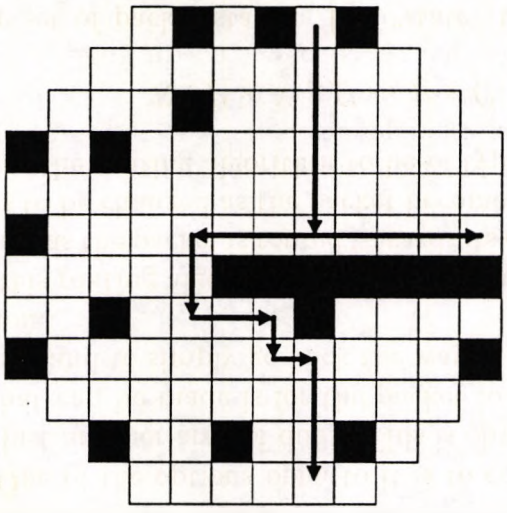


Figure 2.11: Packets "bouncing" off the edge of the wafer

- A displacement field $0 < s < maxs$ where $maxs$ is a constant whose value is greater than the distance between the two furthest removed nodes on the wafer
- A blocked bit $b \in B = \{T, F\}$
- A handedness bit $h \in H = \{L, R\}$

The routing function can now be given in its complete form, but first some auxiliary definitions are needed. A node $n_{x,y}$ has a number of channels $c_{x,y,d}$ which connect to each neighbour where $d \in D$ If node $n_{x,y}$ receives a communication from node $n_{x',y'}$, on channel $c_{x',y',d}$ then (x, y) and (x', y') can be related by $x = x' + modx(d)$ and $y = y' + mody(d)$ where $modx$ and $mody$ are functions:

$$
\begin{aligned}
modx(d) &= \begin{cases} 1 & \text{if } d = east \\ -1 & \text{if } d = west \\ 0 & \text{if } d = north \vee d = south \end{cases} \\
mody(d) &= \begin{cases} 1 & \text{if } d = south \\ -1 & \text{if } d = north \\ 0 & \text{if } d = east \vee d = west \end{cases}
\end{aligned}
\tag{2.5}
$$

Functions over directions are needed: $opp : D \mapsto D$ which maps any direction onto its opposite direction (eg: $opp(north) = south$); and $inc : D \times H \mapsto D$ which maps a direction, and the state of the handedness bit onto the direction clockwise, or anti-clockwise depending on the state of the handedness bit.

For example, $inc(east, L) = south$ and $inc(east, R) = north$ Also, several more predicates over channels need to be defined. These look at the state of the CE that the channel is connected to: *good*, *dud*, and *edge* all of type $C \mapsto \{T, F\}$. Finally, the negation operator $\neg$ is defined over H such that it delivers the opposite handedness: $\neg L = R$ and $\neg R = L$.

So the routing function can now be defined

$$\mathcal{R}_{\text{Nav}} : P \times N \times C \mapsto P \times C$$

$$\mathcal{R}_{\text{Nav}}(p_{s,b,h}, n_{xd,yd}, c_{x',y',in}) = \begin{cases} (p_{\text{maxs},F,h}, c_{x,y,d}) & \text{if} \quad closer \wedge good(c_{x,y,d}) \\ (p_{s',T,h'}, c_{x,y,d'}) & \text{if} \quad \neg closer \vee \neg good(c_{x,y,d}) \end{cases}$$

where

$$\begin{aligned} x &= x' + modx(in) \\ y &= y' + mody(in) \\ in' &= opp(in) \\ closer &= |\, x - xd \,| + |\, y - yd \,| < s \\ d &= \begin{cases} prim & \text{if} \quad good(c_{x,y,\text{prim}}) \\ alt & \text{if} \quad \neg good(c_{x,y,\text{prim}}) \end{cases} \\ s' &= \begin{cases} |\, x - xd \,| + |\, y - yd \,| & \text{if} \quad \neg b \\ s & \text{if} \quad b \end{cases} \\ (d', h') &= \begin{cases} f(prim, h) & \text{if} \quad \neg b \\ f(inc(in', h), h) & \text{if} \quad b \end{cases} \\ f(d, h) &= \begin{cases} (d, h) & \text{if} \quad good(c_{x,y,d}) \\ f(inc(d, \neg h), \neg h) & \text{if} \quad edge(c_{x,y,d}) \\ f(inc(d, h), h) & \text{if} \quad dud(c_{x,y,d}) \end{cases} \end{aligned} \tag{2.6}$$

The first clause of $\mathcal{R}_{\text{Nav}}$ deals with the normal unblocked case and the case where the packet is ready to leave blocked mode. The second clause deals with when the packet is blocked, or if it is just about to become blocked. The choice of direction is quite tricky, and is made trickier by its depending on the direction of the channel on which the packet arrives at the node *and* whether or not the packet was already blocked. The direction and the handedness when blocked are chosen by the auxiliary function f. This function implements a seek in the compass directions for the first good node available. Handedness is flipped if the packet is at the edge, implementing the "bounce" optimisation.

When the packet is unblocked, the algorithm chooses one of the directions *prim* or *alt* representing the primary direction and the alternative. These can be a simple table lookup with the keys being the $x - xd$ and $y - yd$. The table is as follows, with $x - xd$ on the horizontal and $y - yd$ on the vertical.

	$-ve$	0	$+ve$
$-ve$	*west,south*	*south*	*east,south*
0	*west*	*home*	*east*
$+ve$	*west,north*	*north*	*east,north*

When there are two directions, either can be the *primary* or the *alternative*. If there is only one, this must be the *primary*.

Deadlock Avoidance

As it stands, the routing algorithm is not deadlock free. Simulation shows that it is in fact fairly prone to deadlock at medium loads, ie: when the number of packets in flight approaches one per CE, the network will deadlock very quickly. This is clearly unacceptable, and we must devise a way round the problem. This section looks at some techniques for avoiding or breaking deadlock, and their applicability to this architecture.

Deadlock Avoidance Buffers

One solution to the deadlock problem is to have a buffer associated with every CE. The CE can guess that it is contributing to a deadlock situation when it has failed to forward a packet on a link after a certain number of cycles, and can then buffer the offending packet. This effectively breaks the dependency cycle.

This strategy does not completely eliminate the chance of deadlock occurring, but simply makes it less likely depending on the size of the buffer. The designer of the CE can decide on an acceptable probability of deadlock and can choose an appropriate buffer size. Unfortunately, for any realistic probability, the buffer needs to be unacceptably large for a WSI design. Simulation shows that if the buffer is allowed to grow to the necessary size to eliminate deadlock, then it takes up more space than is available for the CE to yield at an acceptable level. Therefore this method can not be used for WSI.

Structured Buffer Pool. A second solution to the deadlock avoidance is to create a structured buffer. When packets can not be forwarded on their appropriate link, they are inserted in this buffer. The difference between this method and the simple deadlock avoidance buffer mentioned above is that packets are assigned an order as they are inserted in the buffer, and are released from the buffer in this order. By this method, a network can be proven to be deadlock free, as long as it is large enough. [BBG87] give a suitable algorithm.

Unfortunately again, the addition of a buffer makes the CE design too large for WSI standards, and we must reject this design.

Virtual Channels. Virtual channels are a novel solution to the problem of solving the deadlock problem in communications networks [DS87]. The method works as follows: Each CE has a number of virtual channels along

which packets can travel. For each direction, there is an partial order between the channels such that packets can move to lower channels, but no packet on a lower channel can move to a higher one. The channels are arranged so that there is a route to every node by either staying on the current channel, or via a series of channels in descending order. There is always a lowest channel on which packets cannot be blocked by packets on higher channels. Using this mechanism, the dependency graph is guaranteed to be acyclic and so deadlock can never arise.

Virtual channels are ideal for regular networks. For example, a torus network needs only two virtual channels [DS87]. In terms of hardware requirements, each virtual channel requires one physical queue even if it is only one packet long. Unfortunately, the WSI network and we are studying is far from regular. It is not possible to prevent deadlock on such a network on a wafer using only two virtual channels, and more than two would make the design of the CE too large.

Chaining. So far, the deadlock avoidance strategies considered have been rejected because their hardware implementation would take up too much space. A solution appropriate to WSI should take up very little space. This solution is based on [RD86], and is as follows: The communications algorithm operates in a normal mode, but when deadlock is suspected it reverts to an algorithm which may take longer to deliver the packet, but which guarantees to deliver it within a specified number of hops k which is a predetermined constant for a particular network. When in this mode, the CE must not accept any new packets from the PE. After the time for k hops has elapsed, the CE can revert to its normal mode. k is known as the *chain delay.*

To implement this we need to choose a simpler routing algorithm. The one we have chosen connects every working CE in the network into a logical "chain". A packet is guaranteed to be delivered if it simply follows the chain until its reaches its destination. Figure 2.12 shows a wafer that has been configured as a chain. Every connectable node can be configured into a chain in this way. The algorithm for connecting the nodes into a chain is similar to the Catt spiral [AC78].

If there are n working CEs on the wafer, then every packet can be delivered to its destination in a maximum of n hops.

When a CE detects that it is contributing to a deadlock situation it first signals to its neighbours that it is going into this mode. It then forwards all the packets in turn on its input registers on one of the directions dictated by the chain. The CE remains in this mode and continues to forward all packets it receives until it has waited for the k hop times to elapse. It will then return to normal mode.

A CE goes into chain mode if it detects either of two conditions:

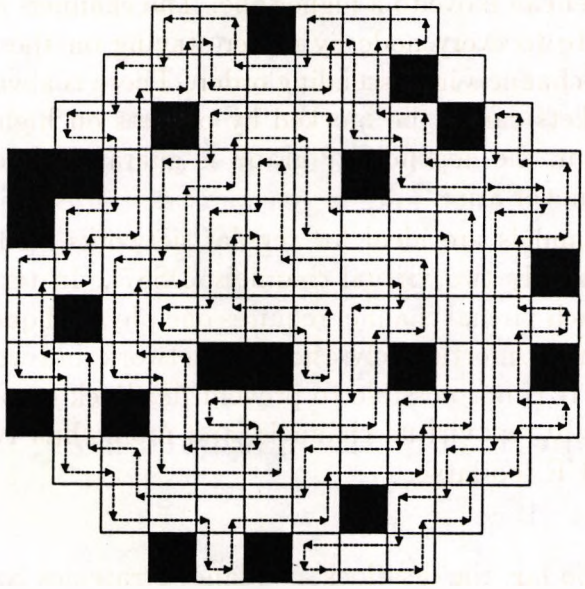


Figure 2.12: A wafer configured into a "chain"

- It has been trying to forward packets for the last h hops and has not succeeded. h is known as the *chain constant.*

- It receives a chain signal from a neighbour.

In each case it counts down from the initial value of the chain delay k, and when it reaches zero it reverts to normal mode. The signal to go into chain mode propagates from the original CE through to its neighbours, and on to their neighbours, and thus to all CEs that are connected in a wave fashion. If two distant CEs go into chain mode at the same time, the chain "waves" will run into each other. If one chain is older than the other, then the chain delay caused by the newest can be added onto the chain delay which already exists.

The value of the chain delay is a crucial design decision. If there are n CEs in the chain, then in the absence of congestion it takes n cycles to clear the network. To be safe, the chain delay should be somewhat greater than n, at a level that allows every CE to empty all its input registers. If the chain delay runs out before all the packets have been delivered to their destination, then as long as the packets are all marked as being unblocked, they will continue to their destination as though they had just been generated. The worst that

could happen in this case is that the network could go into a livelock situation where the same packets constantly cause congestion and thus chaining, and are constantly routed back to where they came from by the CE operating in chain mode. However, even this can be avoided by dynamically changing the value of the chain delay. Going into chain mode would then have the effect of simply redistributing the packets through the network seemingly at random.

This strategy is provably deadlock free, as long as k is large enough. It is fairly harmless if k is not quite large enough to route all packets in the given time as explained above. The amount of silicon needed to implement this strategy in hardware is small, so this will be the deadlock avoidance strategy adopted.

2.3.2 Properties of the navigation algorithm

Harvest & Sacrifice

Unlike the previous communications architectures, the navigation algorithm allows for all working PEs *that can be connected* to be used. Figure 2.13 shows the harvest for a wafer with 100 nodes. Y_{Cutoff} comes only when no I/O registers can be connected to any other working nodes. For the more interesting yield figures, ie between 70 and 90%, this scheme allows for almost all working PEs to be used.

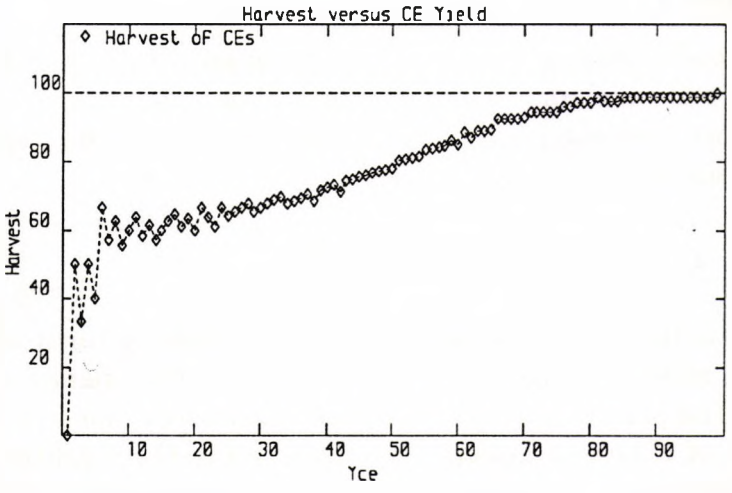


Figure 2.13: Harvest versus Yield for the navigation algorithm

Latency

Figure 2.14 shows how the average and maximum latencies vary with Y_{CE} for a wafer of 100 nodes. The unit of latency on the graph is one hop, the time taken to make a routing decision and transfer a packet to a neighbour.

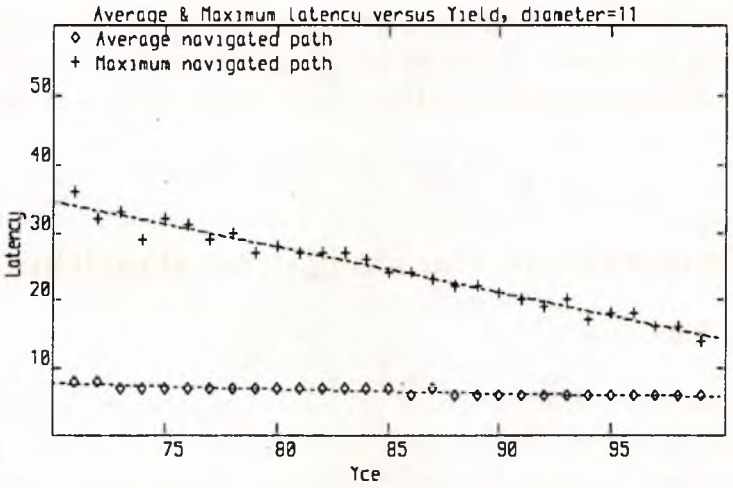


Figure 2.14: Average and Maximum latencies versus Yield

Performance

In order to model the performance of such a communications architecture an event driven software simulator has been written.

The performance of an individual routing from n_i to n_j on a specific wafer under some set of conditions is defined as

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

where L_{ideal} is the latency of routing a packet from n_i to n_j on a perfect wafer in the absence of congestion, and L_{actual} is the latency under those conditions. The performance of a wafer as a whole is defined as the average performance of a large number of routings, while the conditions are held constant. Measurement of the performance under differing conditions allows us to choose certain design parameters, and to predict results.

What are the design parameters that we can vary? One of the most important is the chain constant h. If this is too low then most routings will be via the chain, and will take a relatively long time to complete. If it is too high, then the network will take an inordinately long time to recover from

high congestion and the performance will suffer again. There must be an optimal value for the chain constant.

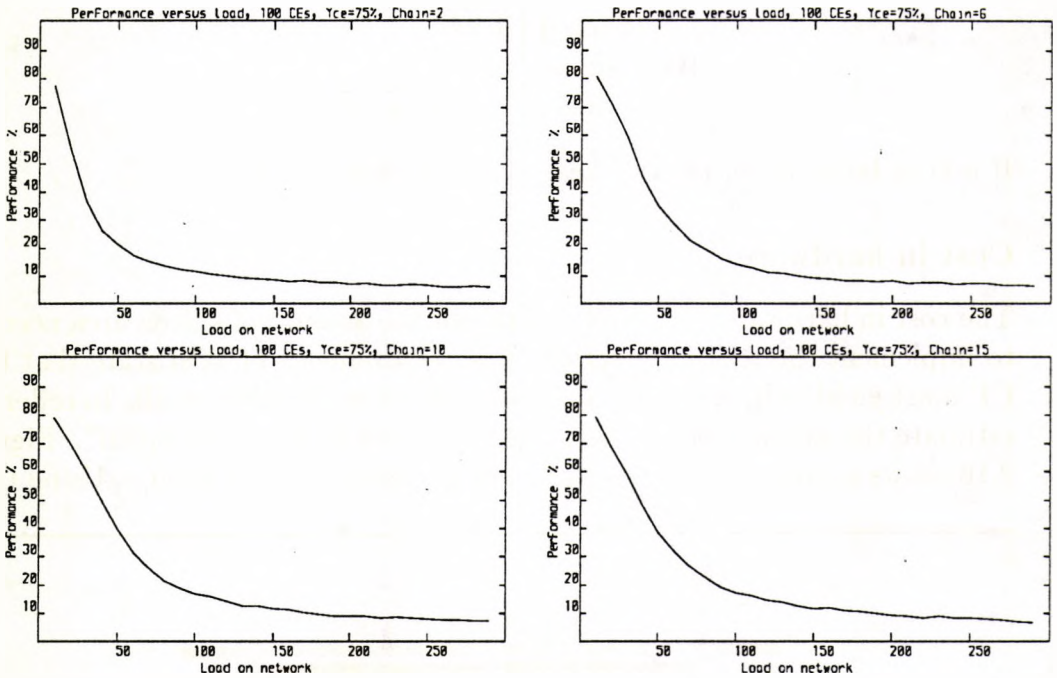


Figure 2.15: Performance versus load for increasing chain constants

The most important relationship that can be measured is the relationship between the performance and the load. The load is defined as the number of packets "in-flight". Each CE can contain a maximum of four packets in its input buffers. The maximum number of packets in a given network is equal to the number of connectable input buffers. A CE's input buffer is not connectable if it is up against a dud CE, or the boundary of the wafer. A packet is in-flight during the time between its dispatch from the source PE and its receipt at the destination PE. The results of simulation are quite surprising. Figure 2.15 shows the results on performance for various values of the chain constant. The performance levels out very early, prompting the choice of 10 for the chain constant.

Overhead

The packet overhead associated with administering a routing algorithm is defined as the amount of information additional to the payload that a packet has to carry to ensure delivery. The following table shows the overhead per packet for a wafer with a diameter of n nodes.

Overhead	number of bits required
Address field	$2\lceil \log_2 n \rceil$
Displacement field	$\lceil \log_2 2n \rceil$
Blocked	1
Handedness	1
Total	$3 + 3\lceil \log_2 n \rceil$

If n is as large as 16, the overhead is 15 bits per packet.

Cost in hardware

The cost in hardware is defined in terms of the amount of silicon area needed to implement the routing function. The requirements of WSI state that the CE must yield well, and it will only do so if it is sufficiently small. In order to estimate the amount of silicon needed, we need to do a "floorplan". Figure 2.16 shows a proposed floorplan of the CE taken from [AKW90]. Using this

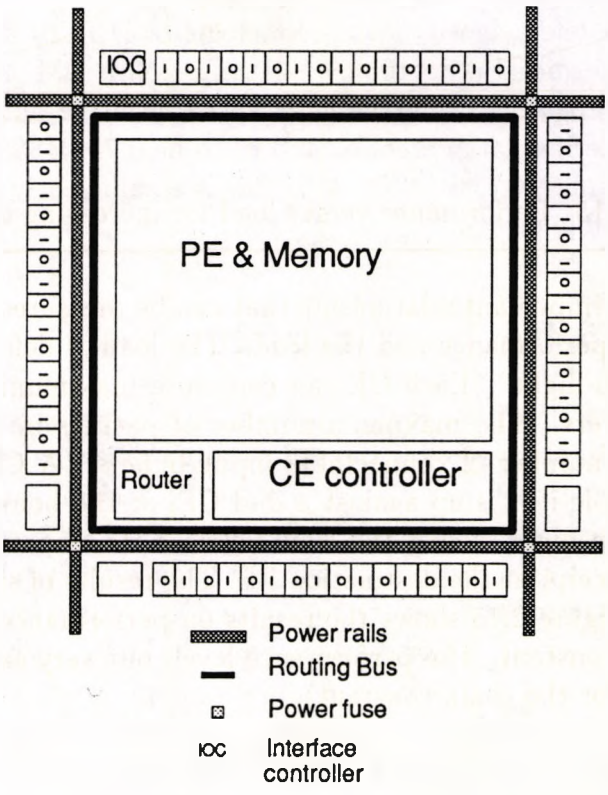


Figure 2.16: Proposed floorplan for a node

design, the area of the node is bounded by the length of its edges. edges.

We are assuming 128 bit wide communications between nodes. For a 6 inch wafer, we can lay out approximately 100 nodes of this size. The operation of the CE is explained in detail in [AKW90]. Here we outline its function. When a full input is detected, the CE controller clocks the first half of the packet into the router. As the routing decision is being made, the second half is clocked in, and by the time it settles, the decision will have been made, and the controller can present the packet on one of the output ports, or to the PE. The sizes of the blocks are calculated by using a gate equivalence scheme. Assuming 1.5μm CMOS, we have

Structure	Area/μ^2	Gate equiv
IO registers	1.7×10^6	5000
IO controllers	187×10^3	400
Router	468×10^3	1000
CE Controller	936×10^3	2000
Routing bus	16×10^6	0
Power supply	6.3×10^6	0

Assuming the defect density for random logic D_c is $3\times10^{-8}/\mu^2$, and the defect density for the metal is $1\times10^{-8}/\mu^2$. Using 1.5μm CMOS and assuming an area of $98.6\times10^6\mu m^2$, using the generalised negative binomial yield model, with the clustering parameters α_c and α_i to be 0.75 [HS88], we estimate Y_{CE} to be approximately 75% [AKW90]. This is a sufficiently high yield to make the CE a feasible design.

2.4 The Paths Algorithm

The navigation algorithm $\mathcal{R}_{\text{Nav}}$ is deterministic. A packet from a source to a destination will always follow the same route. This suggests a new approach to routing. Instead of making the same set of decisions for the packet every time it is routed, why not make them once. A network address can then be defined as a list of directions in which to route the packet. These can be determined externally for every source and every destination on the wafer, and loaded into some memory associated with the PE. When the PE wants to send a message to a particular node, it simply prepends the list of directions to the packets comprising that message. The routing algorithm then becomes trivially simple. The communications node can simply read the head of the packet's direction list and send it in that direction.

Clearly the directions need to be generated. Any communication network is a graph, however, and there exist algorithms that will find the shortest path between any two nodes in the graph. This is however not precisely what we need as we shall see later.

2.4.1 Routing

The Routing Algorithm

The routing algorithm is trivially simple, and is stated here. L is the set of path lists. The following operations are defined on a path l_i:

- A predicate $empty : L \mapsto \{T, F\}$ which indicates if the list is empty.
- $hd : L \times N \mapsto C$ which takes the list and a node and returns the channel on which to forward the packet.
- $tl : L \mapsto L$ returns the remainder of the list after its head has been removed.

$$\mathcal{R}_{\text{path}} : L \times N \mapsto L \times C$$
$$\mathcal{R}_{\text{path}}(l_i, n_j) = \begin{cases} (l_i, c_{j,\text{home}}) & \text{if } empty(l_i) \\ (tl(l_i), hd(l_i, n_j)) & \text{if } \neg empty(l_i) \end{cases} \quad (2.7)$$

Deadlock

This scheme can use the same deadlock avoidance strategy as the navigation scheme.

2.4.2 Properties

Harvest & Sacrifice

This network uses essentially the same topology as the navigation algorithm, so the harvest and sacrifice results are identical.

Latency

The major advantage of this scheme over the navigation scheme is that the latency is decreased substantially. This is because we have taken care to generate paths that are shorter than those generated by $\mathcal{R}_{\text{Nav}}$. In addition, the time taken to make a routing decision will be less because all that is required is for the chosen direction to be read from the head of the packet.

Figure 2.17 shows the maximum and average latencies for a wafer of diameter 11 versus decreasing Y_{CE}. The average latency remains fairly constant as the yield decreases, and the maximum latency increases only slightly.

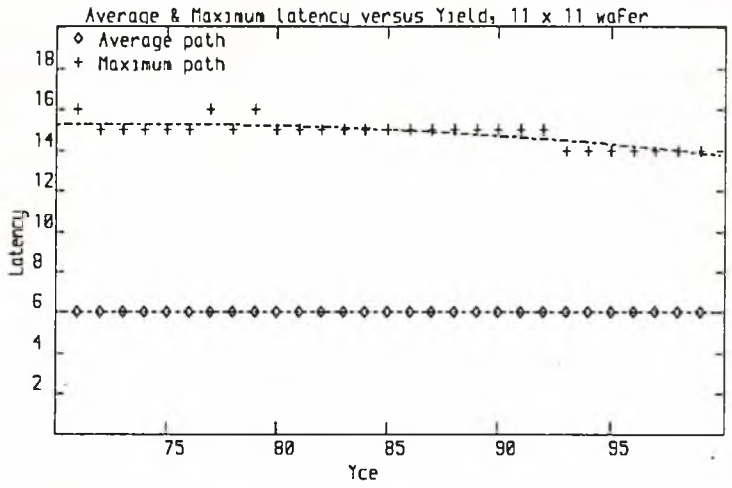


Figure 2.17: Maximum and average latencies in hops using the paths algorithm

Performance

The performance of a machine using the paths algorithm has been measured in the same way as the performance for the navigation algorithm. The same metric definitions apply here too. Figure 2.18 shows the performance for a set of values of the chain constant. It can be seen from the graph that the performance is better than the navigation algorithm, in that it tolerates load better. Note that these performance graphs are modulo the average latencies, so the scale of the increase in performance is not immediately apparent. That is, to find the average latency at a given load, the performance should be read from the graph and multiplied by the average latency from graph 2.4.2. The effect of the chain constant seems to be less than with $\mathcal{R}_{\text{Nav}}$. The performance peaks fairly early, and only starts to drop slowly when the chain constant reaches 30.

Overhead

A path is simply a list of directions on how to get from one node to another. For example to get from node S to node D in figure 2.19 the path would read

[east,east,east,north,east,east,east,south,east,east,east,
north,east,east,east,south,res]

There are some other considerations that we need to take into account before paths are generated. These have to do with the amount of space needed to

store the paths in the CE, and the overhead in terms of bits per packet. There are four possible directions in which a packet may be sent from any particular node: north, south, east and west. These can be encoded as two bits each. The length of the list also needs to be encoded. This can be encoded as a count field which is to be decremented every time the head is removed. The overhead per packet is defined as follows: For a particular wafer, there will be some maximum number of hops that can be taken. Let this be denoted M.

Field	no. of bits
count	$\lceil \log_2 M \rceil$
directions	$2M$

giving a total overhead of $2M + \lceil \log_2 M \rceil$ bits per packet. M is about 16 for a wafer with diameter 11, this results in 37 bits.

37 bits per packet is too large an overhead per packet to be acceptable, so we must devise a method to compress the paths information into a smaller space. Run length encoding of paths is one way of compressing the address, and there are (at least) two ways of doing this.

Method 1 The first two bits in the packet give the first direction to go in. Each following bit instructs the router to either route the packet in the same direction, or not. If not, then the following two bits indicate the next direction. The number of changes of direction is given by c'

Field	no. of bits
count	$\lceil \log_2 M \rceil$
directions	$3 + 3c' + (M - 1 - c')$

For a wafer of diameter 11, with a Y_{CE} of 75% the longest path is 16 hops long. This involves about five changes of direction, so the overhead for this is 28 bits. If the maximum number of changes of direction is eight, then the overhead is 34 bits.

Method 2 This method involves thinking of the move to the PE as just another direction. With this method we count the number of changes of direction, and have a count field for that. The first two bits give the initial direction. If the following bit is a 1 then keep going in the same direction. If not, then the bit after that indicates whether to go to the left, or to the right of the previous direction. When this is found, the count is decremented. However when the count is found to be zero, then this indicates that the packet should be routed immediately to the PE.

Field	no. of bits
count	$\lceil \log_2 c' \rceil$
directions	$2 + (M - 1)$

For a wafer of diameter 11, the longest path is about 16, and allowing c' to be a safe 8, the overhead is calculated to be 21 bits. This method has the smallest overhead, so we choose this.

Path Generation

These results have several consequences for the generation of paths. The most important of these is that whatever route is generated, it must have as few changes of direction in it as possible. $\mathcal{R}_{\text{Nav}}$ can be recast as an algorithm that generates a list of directions. This happens to have the property that it sends packets on paths that have few changes of direction. That is a packet will be sent in a straight line rather than change direction. So we can use this as a basis for generating the paths.

Because the program that generates paths has a more global view of the wafer and can afford to be a lot more 'intelligent', it can make a number of optimisations to the paths as they are generated.

These optimisations are as follows.

1. If a packet has to backtrack for any reason, then the section of the path that the packet backtracks over can simply be deleted completely from the path. This situation arises when a packet encounters the physical edge of the wafer and changes its handedness.

2. A packet from a source to a destination may take a different path depending on its handedness. The optimiser can simply choose the shortest of these.

3. A packet from a source to a destination may take a longer way round than a packet going in the reverse direction, i.e. from destination to source. The optimiser can choose the shortest of these.

4. For each node in the path from source to destination, if there is a shorter path from that node to any other node on the path, then the optimizer can replace part of that path with the shorter section.

Figures 2.20, 2.21, 2.22 and 2.23 shows how these optimisations shorten the paths. Obviously some of the optimisations overlap. The results reported in section 2.4.2 are results for paths found after repeatedly applying all these optimisations to the paths generated by the navigation algorithm

Cost in hardware

The cost in hardware of the paths communications algorithm is much lower than that for the navigation algorithm. This is because the hardware involved is much simpler. If we adopt the same design as in section 2.3.2, the only thing that changes is the size of the router. The router is estimated to occupy approximately 200 gate equivalents. As the hardware cost is dominated by the area occupied by the IO registers and the routing bus, the Y_{CE} is not affected much by the decrease in complexity of the routing algorithm, and the estimated yield figure is 75%.

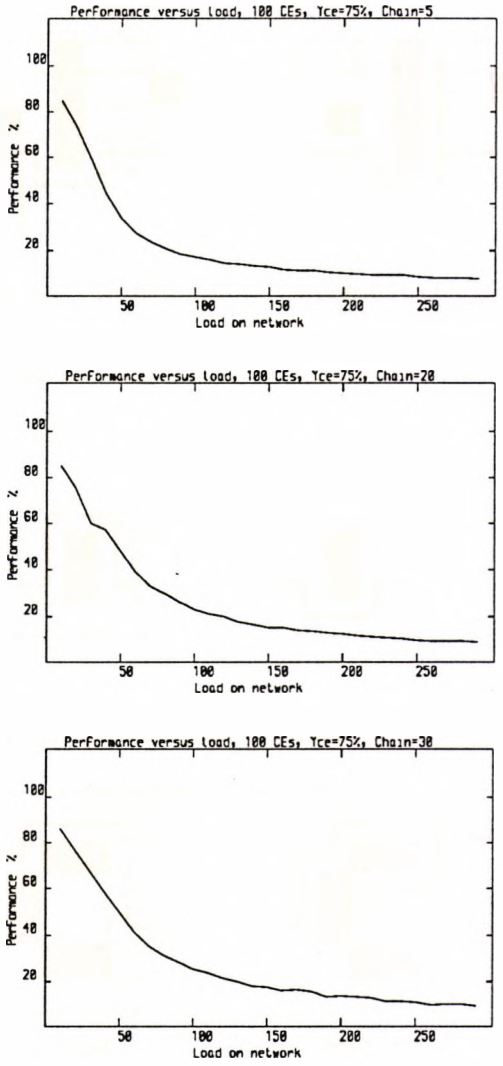


Figure 2.18: Performance versus load for increasing chain constants

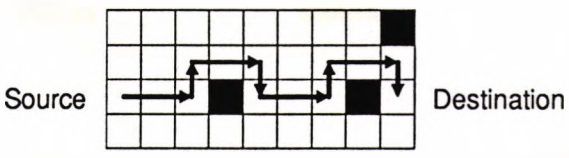


Figure 2.19: An example path from a source to a destination

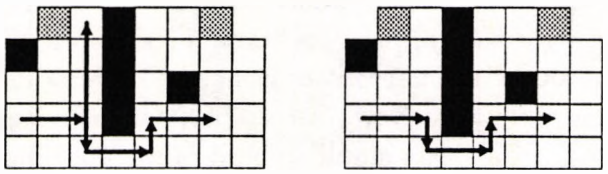


Figure 2.20: Optimisation one: Eliminate backtracking

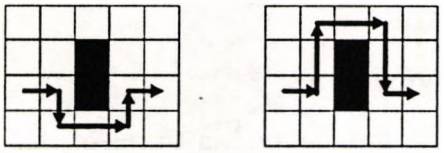


Figure 2.21: Optimisation two: Choose handedness with shortest path

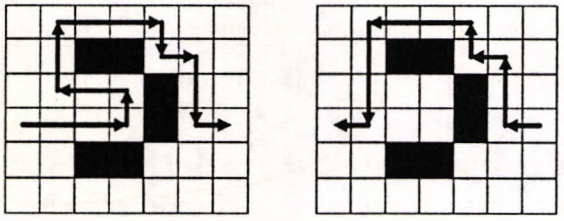


Figure 2.22: Optimisation three: Choose shortest of the forward and reverse

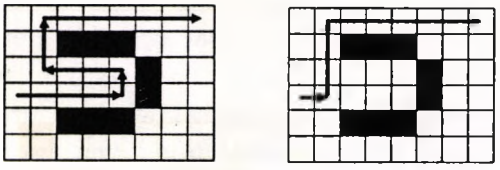


Figure 2.23: Optimisation four: Choose shortest paths to other nodes in the path

2.5 The Signpost Algorithm

The signpost algorithm is based on the observation that the paths generated for the paths algorithm remain constant for the wafer. Instead of having a packet carry around the path, it only needs to carry around the address of the destination processor. Now, instead of having a stateless communications architecture, each CE has a small amount of memory indexed by the address of the processor. The memory contains the direction in which to route the packet next. This set of directions is known as the "signpost". The paths can be generated beforehand in the same way as they were generated for the paths algorithm, and loaded into each CE at system initialisation time.

As the paths are the same, many of the properties of this algorithm are identical to $\mathcal{R}_{\text{path}}$.

2.5.1 Routing

The Routing Algorithm

The routing algorithm is defined in terms of the signpost. Each CE i has a signpost set S_i. The function operates on the destination address, the current address, and the signpost, and is thus trivially defined as follows:

$$\begin{aligned} &\mathcal{R}_{\text{SP}} : N \times N \times S \mapsto C \\ &\mathcal{R}_{\text{SP}}(n_i, n_j, s_j) = c_{j,d} \\ &\text{where} \\ &d = \begin{cases} \mathit{home} & \text{if} \quad i = j \\ s_{j,n} & \text{if} \quad i \neq j \end{cases} \end{aligned} \tag{2.8}$$

The routing algorithm delivers the packet to its destination as long as the signposts have been set up correctly. Therefore the burden of proof of correctness is upon the path generator. As we have shown that the path generator generates correct paths, we can assert that $\mathcal{R}_{\text{SP}}$ is correct.

This algorithm is more powerful than it looks. It offers two features that neither of the previous two offered. These are graceful degradation, and dynamic routing. If a routing fails with either of $\mathcal{R}_{\text{Nav}}$ or $\mathcal{R}_{\text{path}}$, we have a problem. With $\mathcal{R}_{\text{Nav}}$ there is potential for recovery but only if the packet is not in blocked mode. If it is in blocked mode, then on average, it will never get to its destination. With $\mathcal{R}_{\text{path}}$, if a direction is wrong, then because the paths are context sensitive, the packet will end up anywhere but its correct destination. With $\mathcal{R}_{\text{SP}}$, if a direction is given wrongly by a faulty CE, then as long as the packet is not returned to the faulty CE, then it will still get to its destination.

The second advantage is with dynamic routing. In both $\mathcal{R}_{\text{Nav}}$ and $\mathcal{R}_{\text{path}}$, the route from source to destination is fixed. With $\mathcal{R}_{\text{SP}}$, the route is controlled by the contents of the signpost at each CE. As this can be changed,

there is the potential for routes to be changed even as packets are being routed. We might want to use this so as to avoid heavily used "highways" on the wafer so as to minimise congestion. There is even no need for instantaneous consistency in the directions. A packet can get temporarily "lost" without harm as long as all the signposts eventually become consistent. This also has implications for graceful degradation. If a CE fails suddenly, then the controller can route packets round it without halting the machine, although detecting the failure is another story.

Deadlock Avoidance

We can use the same deadlock avoidance scheme as we did for $\mathcal{R}_{\text{path}}$.

2.5.2 Properties

Harvest & Sacrifice

The results for harvest and sacrifice are identical to $\mathcal{R}_{\text{path}}$.

Latency

The latency results are identical to $\mathcal{R}_{\text{path}}$.

Performance

The performance results are also identical to $\mathcal{R}_{\text{path}}$.

Overhead

The overhead for this algorithm is extremely low. Each packet only needs to carry the identifier of the destination processor. For a wafer of n nodes, there are $\sqrt{n}$ nodes in each dimension, so we have the overhead as follows:

Field	no. of bits
destination address	$2 \times \lceil \frac{\log_2 n}{2} \rceil$

For a wafer with 16 nodes in each dimension, this is an overhead of only 8 bits.

As the overhead is constant, we no longer have the constraint of needing to keep the number of changes of direction down. We can thus choose a path generation algorithm freely.

Cost in Hardware

Again we use the same floorplan as in section 2.3.2. This algorithm is also fairly easy to implement in hardware. It will consist of some memory to store the signpost, plus a small amount of addressing logic. The total amount of memory required for the signpost is given by the number of processors times the number of bits required to store one direction. If the total number of nodes on the wafer is 100, and there are four directions then this amounts to $100 \times 2 = 200$ bits. Using static RAM to store the directions, we estimate this area to be 670 gate equivalents. This gives us a Y_{CE} of approximately 75%.

2.6 Summary

We have shown introduced and studied three communications algorithms for regular arrays on WSI. We have measured the performance by simulation of these, and estimated their size, and therefore their yield. It is not immediately obvious which algorithm is best for a general purpose WSI product. Each has its good and bad points.

All the designs for CEs yield at acceptable levels, however, as we have seen, the area of the router is "in the noise", so the yield is a property of the CE design rather than the communications algorithm.

If a routing algorithm is to be chosen for a particular application, the best depends very much on the requirements of that application, and in particular on the number of nodes on the wafer.

The navigation algorithm has a higher latency than the others. For the design shown it would not be chosen, but for a different application, where the nodes are much smaller it might win over the other two. This is because the properties of both $\mathcal{R}_{\text{Path}}$ and $\mathcal{R}_{\text{SP}}$ are a function of the number of nodes on the wafer. If we increase the number of nodes n on the wafer then the dominating factor in the overhead for $\mathcal{R}_{\text{Nav}}$ is $3\mathcal{O}(\log_2 n)$, for $\mathcal{R}_{\text{SP}}$ it is $\mathcal{O}(\log_2 n)$ and for $\mathcal{R}_{\text{Path}}$ it is $\mathcal{O}(\sqrt{n})$. However the dominating factor in the space for the router is $\mathcal{O}(\log_2 n)$ for $\mathcal{R}_{\text{SP}}$ and $\mathcal{R}_{\text{Nav}}$, but less than this for $\mathcal{R}_{\text{Path}}$.

Another factor to be taken into consideration is whether dynamically changing routing, or some degree of graceful degradation is required. If these are required the only suitable algorithm is $\mathcal{R}_{\text{SP}}$.

If the number of nodes is very large, then $\mathcal{R}_{\text{Nav}}$ wins overall in terms of low overhead and high yield, despite its greater latency. If the number of nodes is small, then $\mathcal{R}_{\text{SP}}$ would win because of low latency compared to $\mathcal{R}_{\text{Nav}}$ and the dynamic routing capability compared to $\mathcal{R}_{\text{Path}}$. Somewhere between these two and in applications where the size of the router is critical, then $\mathcal{R}_{\text{Path}}$would be the preferred choice.

So we have a variety of routing algorithms within a common design for a CE. For this design $\mathcal{R}_{\mathrm{SP}}$ is the preferred because of its low latency with respect to $\mathcal{R}_{\mathrm{Nav}}$ and its low overhead with respect to $\mathcal{R}_{\mathrm{Path}}$. A bonus is the potential for dynamic routing and graceful degradation.

Chapter 3

A Graph Reduction Engine

One paradigm that promises to help solve some of the problems of the software crisis, and some of the problems of exploiting parallelism is the functional paradigm. The implementation of functional languages requires special techniques not especially suited to conventional computer architectures. Some alternatives to the von Neumann have been proposed which are better suited to executing functional languages.

Basically there are two feasible methods for the implementation of functional languages. These are graph reduction, and dataflow. This chapter reviews computer architectures that have been devised to execute functional languages efficiently, and introduces COBWEB — the parallel graph reduction architecture that is the subject of this thesis.

An abstract machine for graph reduction can be specified at many levels. The highest level at which we specify COBWEB is as a term rewrite system (TRS). Using such a system we can make fairly strong assertions about a machine. However while this is a considerable bonus this level tells us nothing about what the lower level functions of the machine might be. In order to complete a design for a machine from this specification we must proceed by transforming the high level specification into a low level design.

We proceed in phases starting with the original specification. The input to each phase will be a specification. The output from each phase will be a specification which is at a lower level (ie more detailed) than the input one. Each phase will thus consist of a process that adds constraints to the input specification and translates it into a specification at a lower level. For each phase we need to show that the output specification is correct with respect to the input specification.

In this chapter we demonstrate the translation process at each phase of the design process. The final output from this process is a design for an abstract machine expressed in terms of a topology and a number of blocks whose operation is described in terms of a low level imperative language. This design is then translated into a working program which we use to determine

some architectural parameters, and to eventually decide if a graph reduction engine using WSI is in fact feasible.

The chapter is structured as follows: We start with a brief review of dataflow architectures for functional languages. Section 3.1.2 will present a review of other work in the area of graph reduction architectures. To set the following sections in context, section 3.2 will provide an overview of the COBWEB system, and explain the route from a high level program written in a functional language to a representation of that program running on the machine. Two specification languages are used to specify COBWEB. The languages, one a term rewrite system (TRS), the second an object based TRS with message passing are introduced in section 3.3. This section goes on to introduce the specifications themselves. Section 3.4 shows how to translate a specification written in Paragon into a hardware design. Section 3.5 forms the body of this chapter. This chapter discusses the transformation of the high level specifications from section 3.3 into a low level design. Finally section 3.6 discusses an implementation of the final design from section 3.5.

3.1 Implementation Techniques for Functional Languages

As mentioned earlier, the two modern techniques for the implementation of functional languages are dataflow and graph reduction. In this section we review parallel computer architectures that are designed to execute functional languages based on both techniques. A more general overview of the field can be found in [Veg84].

3.1.1 Dataflow

Although dataflow is a feasible technique for the implementation of functional languages, it is beyond the scope of this thesis. Parallel dataflow computers for functional languages have been studied extensively. Arvind and his team at MIT have designed the Tagged Token Dataflow Architecture [AN87], and MONSOON [PC90]. Gurd and his colleagues have designed the Manchester Dataflow machine [GKW85]. At Southampton University and Imperial College Hugh Glaser leads the FAST project, which aims to have an implementation of the m'Tuki abstract machine running on a network of Transputers [Gla90].

3.1.2 Graph Reduction Architectures

Although graph reduction is a standard technique there are many abstract machine designs. The basic principle is that the functional program is rep-

resented as a graph. Applying transformation rules to the graph is known as reduction. A reduction rule can only be applied to an expression which is reducible — this is known as a *redex*. Repeated reduction of the graph will eventually deliver the result of the program. Throughout this document we discuss evaluating to *weak head normal form* (whnf) [Pey87]. This effectively means that evaluation proceeds until there are no top-level redexes.

The standard mechanisms for the implementation of graph reduction are:

- Combinators [Tur79]
- Supercombinators [Hug84]
- Directors [KS81]

A set of combinators correspond to the instruction set of a simple imperative computer, for example move and copy. Supercombinators can be described as an instruction set that has been invented to suit the program being executed. Directors can be thought of as annotations to the graph that indicate where a parameter is needed. Directors can be shown to be equivalent to combinators.

Concurrency in graph reduction machines is exploited by observing that many parts of the graph can be evaluated to normal form concurrently. Which parts of the graph this can be applied to can be elicited using strictness analysis, or by annotations added by the programmer.

Parallel machines can be *tightly coupled* or where the processors share some global memory and any local memory is a cache, or *loosely coupled* where the processors have local memory only. A machine is *neighbour coupled* if memory access to close neighbours is quicker than memory access to other processors.

Load balancing is achieved in tightly coupled machines by allowing idle processors to evaluate reducible parts of the graph at will. The situation with loosely coupled machines is more complex as the cost of exporting a graph to be evaluated by another processor may be greater than waiting for it to be evaluated locally.

The mapping of program code onto processor memories is another issue. The goal is to reduce communication by maintaining locality of access as much as possible. As machines get more tightly coupled, the problem of mapping a program onto a set of processors gets more difficult.

Parallel machines in general can be categorised by their grain size. This is defined by the size of a task, where a task is the minimum amount of work a processor can do in parallel with another task. A fine grain machine is one where a task might consist of a basic operation such as a copy, or an add. A medium grain machine is one where a task might consist of a small number of basic operations, perhaps at the function level. A large grain machine is one where the amount of work in a task is large, usually

at the program level. Combinator machines are typically fine grain, whereas supercombinator machines are medium grain.

GRIP

GRIP [PCS89] is a high performance graph reduction machine based on the Spineless Tagless G-machine, a supercombinator based abstract machine. It consists of a number of conventional CPUs attached to a number of memories (IMUs) which are managed by a novel intelligent controller and a packet switched bus. Each IMU contains a fixed segment of the global heap. Each processor in the machine contains some local memory, known as the local heap, in which new graph nodes are created, and which acts as a cache for the global heap. The machine can be said to be "programmably"-coupled as the level of coupling is dependent on the programming of the IMU. Load is distributed automatically by having idle processors poll IMUs for redexes.

Alice & FLAGSHIP

At Imperial College and at ICL the ALICE machine has been designed and a prototype has been built. Alice is a tightly coupled medium grain packet based reduction machine. It consists of a number of processors each of which has access to a packet pool. Reduction proceeds by processors taking reducible expressions from the pool, reducing them, and returning the result plus any new packets generated to the pool. This process proceeds until the packet that represents the result of the program has been reduced to normal form. Load is distributed automatically by having idle processors scan the pool for redexes.

A prototype machine has been built from Transputers connected by a delta network.

FLAGSHIP, formerly a collaborative venture involving Imperial College, Manchester University and ICL, is a descendant of the ALICE project [Kir89]. It is a system designed for declarative programming and so supports a larger computational model than graph reduction for functional programming. The heart of the FLAGSHIP system is an ALICE type graph reducer. The load balancing system is more sophisticated, and is controlled by an intelligent network. Each processor propagates some measure of how busy it is to the network, which can route tasks from processors which are busy to those which are idle.

The HDG-machine

The HDG machine is an abstract machine designed at GEC for the execution of lazy functional languages using graph reduction [LB90, Bur89b]. The machine has been designed to exploit evaluation transformers. These are a

way of expressing strictness in data structures, and functions which operate on data structures as well as functions which operate on basic values. It is based on the Spineless G-machine. The HD in the name indicates that it is highly distributed. That is to say that the program graph is distributed throughout the memory of the machine. This project is still in progress, and the current literature does not specify a particular load distribution scheme. A realisation of the abstract machine on a network of transputers is underway.

MaRS

MaRS is a graph reduction multiprocessor being developed by a team at the Centre d'Etudes et de Recherches de Toulouse in France [CCC+89]. A programming language named "MaRS_LISP" has been developed which includes constructs to control parallelism explicitly. The instruction set is indexed combinators which have a slightly larger grain than Turner style combinators. There are several types of processor, the most important being the reduction processors and the memory processors. It is a tightly coupled machine with processors connected using an Omega network. The network processors measure and control the spread of tasks throughout the machine.

ALFALFA

Paul Hudak at Yale leads a team at Yale University [GH86]. Their architecture is an implementation of distributed graph reduction on the iPSC, a loosely coupled MIMD architecture. The source language is one of Alfl or ParAlfl, the former being a functional language in the style of SASL, and containing no parallelism constructs; and the former being a language that permits the programmer to express parallelism. The grain of computation is the *serial combinator*, which are larger than supercombinators, and which contain no concurrent substructure. This enables parts of the program to be evaluated using the conventional stack, rather than on the heap. Work is distributed by *diffusion scheduling* whereby the program graph is distributed throughout the machine based on workload and locality.

PAM

PAM is a parallel abstract machine being developed mainly at RWTH Aachen in Germany [LKID89]. Their approach combines the work of Hudak on serial combinators, and Burn on evaluation transformers. Parallelism is thus detected automatically by the compiler. It is a medium grain architecture. Redexes waiting to be evaluated are stored in the communications processor, which can decide to export them to other processors if work is requested. Each processor has some local cache, so it is shared memory architecture.

An implementation of the abstract machine has been implemented on a network of transputers.

3.2 Overview of COBWEB

COBWEB is a parallel graph reduction architecture for functional languages. This section gives a brief overview of the philosophy and operation of COBWEB. Much of the material introduced here will be expanded upon in later sections. The purpose of this section is to explain how a program in a functional language is run on the machine.

The way COBWEB runs programs is as follows. The functional language Hope$^+$ [Per88] (with pure lazy semantics) is compiled to the intermediate code FLIC using the program *hfc* [Hun90]. A program in FLIC consists of a set of definitions in the enriched λ-calculus [PJ89]. One of these definitions has the name MAIN. The result of evaluating the body of this definition is the result of the program. *Hfc* produces a FLIC program with strictness annotations in the form of evaluators and evaluation transformers [Bur89a]. This program is then compiled into a director graph using techniques from [Pey87, BHK88]. The director graph is in the form of triples — each representing a node in the graph and consisting of a string of directors (including strictness annotation), and the left and right subgraphs.

The graph is loaded into the machine and the result is requested. Execution proceeds by transforming the graph into weak head normal form.

3.2.1 Hope$^+$ $\rightarrow$ FLIC

Parallelism in a functional program can be detected using techniques known as *abstract interpretation*, or *projection analysis* [Wad87, AH87, Bur87]. One of the most important properties of a program that can be revealed by these techniques is the strictness of functions. A function f is *strict* in its argument iff:

$$f\ \bot = \bot$$

where $\bot$(*bottom*) represents the undefined expression. If it is known that a function is strict in its argument, then it is safe to evaluate the argument in parallel with the body of the function. This is the only source of parallelism exploited in COBWEB.

Recently abstract interpretation has been used to reveal strictness of data structures in functional programs [Bur89a]. Given a data structure and a set of functions that operate on that data structure, this technique can reveal how much "evaluation" of the data structure that function requires. For example consider a list of 2-tuples. The function *length* returns the length of the list. When applied to an argument it only needs to know how many

elements are in the top level list, and needs no evaluation of the tuples. However, the function *first* which returns the list containing the first value of the 2-tuple, needs to evaluate the structure of each tuple in the list. *First* is said to do more evaluation than *length.*

Expressions and functions in a program can be annotated with information indicating the amount of evaluation to be done. This type of annotation is known as an *evaluator.* There are four evaluators:

- ξ_0 indicates no evaluation
- ξ_1 means that the expression can be evaluated safely to whnf
- ξ_2 the spine of the list can be evaluated
- ξ_3 the spine of the list can be evaluated and all the elements of the list can be evaluated to whnf

These evaluators have an order:

$$\xi_3 > \xi_2 > \xi_1 > \xi_0$$

where the $>$ indicates "does more evaluation than". In our example, the function *first* will be annotated with evaluator ξ_3 and *length* will be annotated with evaluator ξ_2.

In addition there are evaluation transformers, which map evaluators for an expression onto safe evaluators for its sub-expression. However these are beyond the scope of this study.

The program *hfc* [Hun90] compiles programs written in Hope+ to FLIC with strictness annotations of the form defined in [Bur87].

3.2.2 FLIC → COBWEB

FLIC programs consist of a set of definitions. For example, consider the following program in Miranda.

```
triple x = 3 * x
twice f = f . f
```

along with the application twice triple 7.

Ignoring for now the strictness annotations, this program will compile into the following FLIC program:

```
MAIN (twice triple 7)
twice (λf λt (f (f t)))
triple (λx (* 3 x))
```

This program can be represented as a graph, as figure 3.1 shows.

The evaluation technique chosen for COBWEB is *directors* [KS81]. A director is an annotation on a graph node that indicates which subgraph of that node an argument is needed in. A director simply defines a transformation on the graph. Arguments to functions can be envisaged as being sent down the graph following the directions indicated by the directors. At the leaves of the graph are "boxes" which the argument will eventually slot into. The process of transforming a program graph into a director graph is performed by abstracting out all the λ expressions. Figure 3.2 shows the program as a director graph.

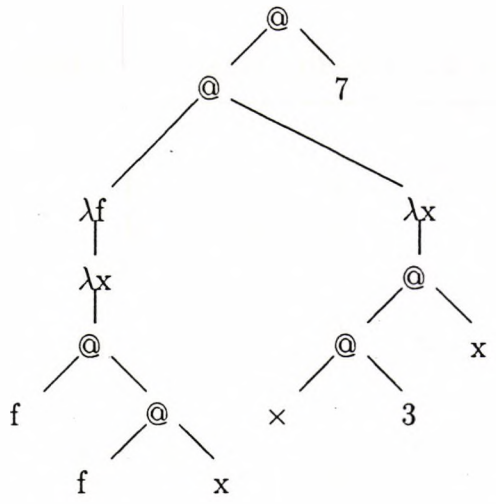


Figure 3.1: The program graph for twice triple 7

A COBWEB program consists of a set of "triples". Each triple represents a node in the graph, and consists of the list of annotations plus the left and right subgraphs of that node. Boxes are represented by the identity combinator I. Directors are effectively the "machine-code" of COBWEB corresponding to actions such as copying and moving data in a more traditional machine.

In addition to the directors, COBWEB uses some built-in functions and data constructors/selectors mostly taken from the standard FLIC set.

For example, the above program compiles to the following COBWEB program.

```
MAIN:      []      MAIN_2     7
MAIN_2:    []      twice      triple
twice:     [^\]    I          twice_2
```

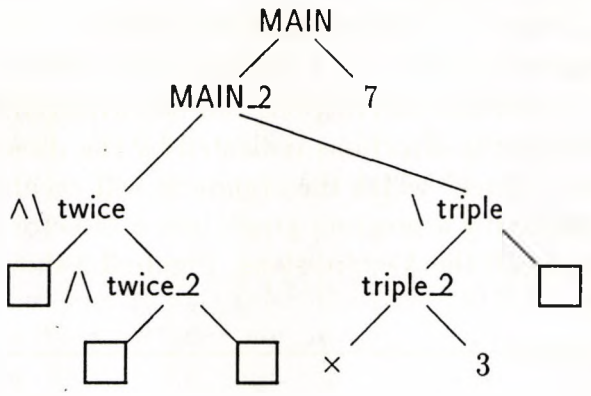


Figure 3.2: The director graph for twice triple 7

twice_2:	[/\]	I	I
triple:	[\]	triple_2	I
triple_2:	[]	×	3
MAIN ?			

Each line represents a triple, or the name of a node to be reduced. The result of this program will be the result of evaluating the node named MAIN.

Parallelism

COBWEB has two parallelism primitives. These represent context sensitive strictness and context free strictness. The context free strictness operator is the dyadic combinator P, whose semantics is that it evaluates both its operands in parallel. This operator is used in an application fx when it has been deduced that f always needs the value of x. The context sensitive parallelism primitive is an annotation written # which resides in the director list.

These are fairly primitive parallelism operators, and take no account of data strictness. The FLIC to COBWEB compiler compiles evaluators to expressions involving P and #.

If a function f is labelled as having an evaluator $\geq \xi_1$ on an argument x, then this is compiled to the expression P f x. If a graph node is annotated with the evaluator $\geq \xi_1$ then the context sensitive parallelism annotation is added to the list of annotations.

3.3 Specification of COBWEB

COBWEB is a parallel graph reduction architecture. We can describe the machine at its instruction level using a term or graph rewrite system. This system would describe the basic transformations to the graph. However, a description of this form is at a fairly high level. It can only give us an abstract view of how concurrency is exploited and how work is distributed among processors.

We wish to specify the operation of the machine at a lower level, and in particular we wish to describe the program graph and the mechanism by which the graph is reduced in more concrete terms.

This section introduces two specifications of COBWEB. The first is a traditional term-rewrite specification of the basic operations of the machine, and the second is a specification in Paragon of the machine at a lower level.

3.3.1 COBWEB as a Term Rewriting System

COBWEB can be described using a term rewriting system. A program in the term rewriting system consists of an expression to be reduced. An expression $\langle e \rangle$ has the syntax

$$
\begin{array}{lcl}
\langle e \rangle & ::= & \langle e \rangle\ \langle e \rangle \\
& | & (\langle e \rangle) \\
& | & [\langle d \rangle]\langle e \rangle\langle e \rangle \\
& | & \langle p \rangle \\
 & & \\
\langle d \rangle & ::= & \langle d' \rangle\langle d \rangle \\
& | & \epsilon \\
\langle d' \rangle & ::= & \wedge \mid - \mid / \mid \backslash \mid \#
\end{array}
$$

where $\langle p \rangle$ represents a primitive operator, combinator, or constant value for example $+$, I, the numerical constants.

For example the program in section 3.2.2 can be represented by the term:

$$(([\wedge\backslash] \mathrm{I}\ ([/\backslash] \mathrm{I}\ \mathrm{I}))\ ([\backslash]\ (\times\ 3)\ \mathrm{I})\ 7)$$

Redexes and Directors

The next redex in a term rewriting system is implicit: as an expression is reduced the leftmost outermost reducible expression is the next one to be reduced.

The rewrite rules for directors are very simple. For example the "left" director is defined as follows:

$$([//::d]\ f\ a)\ x \rightarrow [d]\ (f\ x)\ a$$
$$([//::d]\ \mathsf{I}\ a)\ x \rightarrow [d]\ x\ a$$

Appendix A gives a complete set of the term rewrite rules for the directors.

Primitives

The original COBWEB description in [BHK88] contained rules for a few of the primitives of the machine, for example the strict basic dyadic operator +, and the parallel combinator P, as well as some of the directors. However, the paper does not define a full set of primitives. A real specification for a machine needs a full set of primitives for operations such as data construction/selection, as well as a full set of arithmetic and logic operators..

The choice of primitives is itself a problem. This reflects the traditional debate between choosing a large set, and losing speed in the instruction cycle and choosing a small set and implementing the more complicated operations in terms of these. The balance that was struck was to choose almost the same set of primitives as are provided in the compiler target language FLIC. These include conditionals, data constructors/selectors, a wide choice of integer and floating point operators, and a few pragmatic primitives for sequencing, strictness and program termination. These are specified using a term-rewriting system in the FLIC report [PJ89], and are reproduced in appendix A.

The main difference between FLIC and COBWEB primitives is that some of the COBWEB primitives are indexed with integers. For example the selection primitive K-n-i is defined in FLIC so that it takes two integer arguments followed by a number of following arguments depending on the value of one of the integers:

$$\mathsf{K}\ n\ i\ x_0 \ldots x_n \rightarrow x_i$$

In COBWEB all of these types of primitive are indexed so that K-n and K-n-i represent a family of primitives for a small set of values of n and i. COBWEB only allows primitives without indexes. The presence of the indexes is a syntactic requirement of FLIC, so the transformation from non-indexed to indexed primitives is performed when FLIC is compiled to COBWEB-code.

Selection There are two selection primitives: K-n and K-n-i. The former is a weaker form that rewrites into the latter when applied to an argument as follows:

$$\text{K-}n\ i \rightarrow \text{K-}n\text{-}i$$

The full form is, as implied above:

$$\text{K-}n\text{-}i\ x_0\ \ldots\ x_{n-1} \quad \rightarrow \quad x_i$$

Data Constructors/Selectors FLIC provides several data structure manipulation primitives. PACK, SEL, UNPACK and its strict version UNPACK!, and finally CASE and TAG. Disregarding typing issues, the rest of the FLIC primitives can all be written in terms of these.

Data structures are written as a tag, followed by the data. This is denoted $\langle d \mid x_0, \ldots, x_n \rangle$, where d is the tag, and $x_i, 0 \leq i < n$ is the ith data component of the structure.

Packing PACK is the primitive for data construction. Given a number of arguments and a tag, it packs these up into a structure.

$$\begin{array}{lcl} \text{PACK-}n\ d & \rightarrow & \text{PACK-}n\text{-}d \\ \text{PACK-}n\text{-}d\ x_0\ \ldots\ x_{n-1} & \rightarrow & \langle d | x_0, \ldots, x_{n-1} \rangle \end{array}$$

Selection from structures Selection of one element from a data structure has two forms. SEL-n takes one argument to become SEL-n-i which in turn expects one argument which is expected to be in the form $\langle d \mid x_0, \ldots, x_{n-1} \rangle$. SEL-$n$-$i$ is strict, so it is defined as follows for $n \geq 0, i < n$:

$$\begin{array}{lcl} \text{SEL-}n\ i & \rightarrow & \text{SEL-}n\text{-}i \\ \text{SEL-}n\text{-}i\ \bot & \rightarrow & \bot \\ \text{SEL-}n\text{-}i\ \langle d | x_0, \ldots, x_{n-1} \rangle & \rightarrow & x_i \end{array}$$

Unpacking structures There are two primitives for unpacking: one strict version and the other non-strict. These are defined for COBWEB as follows, $n \geq 1$:

$$\begin{array}{lcl} \text{UNPACK!-}n\ f\ \bot & \rightarrow & \bot \\ \text{UNPACK!-}n\ f\ \langle d | x_0, \ldots, x_{n-1} \rangle & \rightarrow & f\ x_0\ \ldots\ x_{n-1} \\ \text{UNPACK-}n\ f\ e & \rightarrow & f\ (\text{SEL-}n\text{-}0\ e)\ \ldots\ (\text{SEL-}n\text{-}(n-1)\ e) \end{array}$$

Case analysis and tag extraction CASE-r does a case analysis on a list of arguments and a data structure. CASE-r is defined for $r \geq 1, d < r$ as

$$\begin{array}{lcl} \text{CASE-}r\ e_0\ \ldots\ e_{r-1}\ \bot & \rightarrow & \bot \\ \text{CASE-}r\ e_0\ \ldots\ e_{r-1}\ \langle d | x \rangle & \rightarrow & e_d \end{array}$$

Finally TAG is defined. This simply returns the tag of its argument, which must have been evaluated to a data structure.

$$\begin{array}{lcl} \mathsf{TAG}\ \bot & \rightarrow & \bot \\ \mathsf{TAG}\ \langle d|x\rangle & \rightarrow & d \end{array}$$

Booleans COBWEB has the boolean values TRUE and FALSE and the boolean operators IF, NOT, OR, AND and XOR.

However, not all the boolean operators are strict in both arguments. For example AND and OR are defined:

$$\begin{array}{lcl} \mathsf{OR}\ \bot\ x & \rightarrow & \bot \\ \mathsf{OR}\ \mathsf{TRUE}\ x & \rightarrow & \mathsf{TRUE} \\ \mathsf{OR}\ \mathsf{FALSE}\ x & \rightarrow & x \\ & & \\ \mathsf{AND}\ \bot\ x & \rightarrow & \bot \\ \mathsf{AND}\ \mathsf{FALSE}\ x & \rightarrow & \mathsf{FALSE} \\ \mathsf{AND}\ \mathsf{TRUE}\ x & \rightarrow & x \end{array}$$

Comparision and Numerical operators Several polymorphic comparison operators are defined: POLY=, POLY!=, POLY>, POLY<, POLY<= and POLY>=. These are all strict in both arguments. For example the polymorphic comparison operator POLY= is defined

$$\begin{array}{lcl} \mathsf{POLY}=\ \bot & \rightarrow & \bot \\ \mathsf{POLY}=\ a\ b & \rightarrow & \underline{a = b} \\ \mathsf{POLY}=\ \langle d|x_0,\ldots,x_n\rangle\ \langle d|x_0,\ldots,x_n\rangle & \rightarrow & \mathsf{TRUE} \\ \mathsf{POLY}=\ \langle d|x\rangle\ \langle d'|y\rangle & \rightarrow & \mathsf{FALSE} \end{array}$$

COBWEB arithmetic is with integers and floating point values. These are all strict in their arguments. The full set for integer arithmetic is given in the appendix. For a generic strict dyadic arithmetic operator f, and a generic strict monadic operator g we can define them as follows:

$$\begin{array}{lcl} f\ \bot\ b & \rightarrow & \bot \\ f\ a\ \bot & \rightarrow & \bot \\ f\ a\ b & \rightarrow & \underline{f\ a\ b} \\ & & \\ g\ \bot & \rightarrow & \bot \\ g\ a & \rightarrow & \underline{g\ a} \end{array}$$

where the underlining indicates the actual result of applying the operator.

Lists Lists can be defined using the structure manipulation primitives defined above as shown in the FLIC report.

Sequencing, Strictness & Termination Three primitives are defined. SEQ forces its first argument to be evaluated before returning the value of the second. STRICT is applied to a function and an argument, and forces the argument to be evaluated before it is passed to the function. ABORT is used for program termination. Any attempt to evaluate it results in an error.

$$\begin{array}{lcl} \mathsf{SEQ}\ \bot\ b & \rightarrow & \bot \\ \mathsf{SEQ}\ a\ b & \rightarrow & b \\ \\ \mathsf{STRICT}\ f\ \bot & \rightarrow & \bot \\ \mathsf{STRICT}\ f\ x & \rightarrow & f\ x \\ \\ \mathsf{ABORT} & \rightarrow & \bot \end{array}$$

Parallelism

The two parallelism constructs represent context free and context sensitive parallelism. The context free parallelism operator is P and is defined:

$$\mathsf{P}\ a\ b \rightarrow a\ b$$

The intention is that both a and b are evaluated in parallel, however note that this term rewriting system does not describe this.

The context sensitive parallelism construct is # which resides in the list of directors:

$$([\# :: d]\ a\ b) = ([d]\ a\ b)$$

Again the intention is that a and b are evaluated in parallel.

The example

Having defined the operation of the machine we can now see it in action. The program defined earlier, twice triple 7 can be reduced to normal form using the rules introduced above. The complete set of rules is reproduced in appendix A.

(([∧\] I ([/\] I I)) ([\] (× 3) I) 7)
([\] ([\] (× 3) I) (([/\] I I) ([\] (× 3) I)) 7)
(([\] (× 3) I) (([/\] I I) ([\] (× 3) I)) 7))
((× 3) (([/\] I I) ([\] (× 3) I)) 7))
((× 3) ([\] ([\] (× 3) I) I) 7))
((× 3) (([\] (× 3) I) 7))
((× 3) ((× 3) 7))
((× 3) 21)
63

3.3.2 COBWEB in Paragon

The description of COBWEB in the term rewriting system is at too high a level to be of immediate use to the designer. Before a design is attempted the designer needs to know more about the following:

1. How to select the next redex. This needs to be made explicit. The abstract machine cannot afford to look for the leftmost outermost expression as is implied by the term rewriting system. The next redex to be reduced can always be derived from the current one being reduced.

2. How to distribute concurrency. That is how to execute the two arms of a P operator or a # annotation, and how to distribute the work throughout the machine.

The designer could then proceed with a design, but a description that addressed just these issues still leaves the designer with some problems. How is the program represented? When has an expression been reduced to normal form?

The original COBWEB specification is described in [BHK88] using a notation named "Paragon". Paragon is an message passing object based term rewrite system. It is an experimental language and as such its syntax has not yet settled. This thesis is written using the latest version. The specification of COBWEB has highlighted some problems in the initial definition of the language. Here we provide two extensions to the language to handle variable sized left and right hand sides. This section gives a short explanation of the language and the original COBWEB specification. The extensions to the language are introduced as needed.

A Paragon specification consists of a number of class definitions Each class has a number of instance variables, and a set of methods. The driving force in Paragon is message passing. A Paragon specification for a method consists of a number of rules. A typical rule has the following structure

$\mathcal{S}$ given m(x) when
$\mathcal{G}$
$\rightarrow$ $\mathcal{S}'$
then
$\mathcal{C}$
where
$\mathcal{B}$

Each rule has two sides, the lhs followed by the $\rightarrow$ symbol (read "rewrites to") followed by the rhs. A rule basically says that when objects which have state $\mathcal{S}$ receive the message m with arguments x, then if the conditions specified in the guard $\mathcal{G}$ apply, this object will be rewritten into state $\mathcal{S}'$, and the communications $\mathcal{C}$ are generated. A list of bindings of expressions to names is provided in $\mathcal{B}$.

The message m carries an optional list of arguments. The arguments can be patterns, in which case the rule only matches when the pattern matches the incoming message. $\mathcal{G}$ consists of a set of pattern matching equations and boolean expressions. The pattern matching equations can match on any of the objects named in $\mathcal{S}$ or x. A underscore in a pattern matching equation is a "don't care".

The rhs includes a list of message sending actions. These messages can be synchronous or asynchronous denoted by the symbols ! and !! respectively. The actions are composed in sequence using the ; operator, or in parallel using the || operator.

Each rule can be given a name eg: [o×4].

Figure 3.3 shows the partial syntax for a Paragon rule.

$\mathcal{B}$ introduces a list of bindings of expressions to names. We allow pure functions to be defined here, but only if they run in constant space, ie iterative or tail recursive and no dynamicism.

Expressions can contain references to **self** which indicates the object receiving the message, or **nil** which indicates the null object. New objects are created using the expression **new**(classname, initialstate). In addition to the rules for messages a class may contain *spontaneous* rules. These have no **given** clause and are thus applied whenever the left hand side matches.

There are two types of object in Paragon: **class** objects, which are objects that can receive messages; and **data** objects which cannot receive messages, but can only be created or matched. These are written in the same way. All types are sums of products of types or basic types. These are written in the style of Miranda, except that the notation $\langle , \rangle$ is used to denote the anonymous data constructor when a type consists of just one product.

⟨rule⟩ → ⟨lhs⟩ $\rightrightarrows$ ⟨rhs⟩
⟨lhs⟩ → ⟨state⟩ [given ⟨message⟩] [when ⟨guard⟩]
⟨message⟩ → ⟨name⟩ [(⟨state⟩ { , ⟨state⟩ }$_0$)]
⟨guard⟩ → ⟨name⟩ = ⟨state⟩
| ⟨predicate⟩
| ⟨guard⟩ ∧ ⟨guard⟩
| ⟨guard⟩ ∨ ⟨guard⟩
| (⟨guard⟩)

⟨rhs⟩ → ⟨state⟩ [then ⟨tasks⟩] [where ⟨bindings⟩]

⟨tasks⟩ → ⟨name⟩ ! ⟨outgoing⟩ *synchronous*
| ⟨name⟩ !! ⟨outgoing⟩ *asynchronous*
| ⟨tasks⟩ || ⟨tasks⟩ *parallel composition*
| ⟨tasks⟩ ; ⟨tasks⟩ *sequential composition*
| (⟨tasks⟩)

⟨outgoing⟩ → ⟨name⟩ [(⟨expr⟩ { , ⟨expr⟩ }$_0$)]

Figure 3.3: A partial BNF for Paragon rules

Packets and Agents

The original specification for COBWEB described two classes in the machine: packets and agents. A program in COBWEB is represented by a number of *packets* forming the program graph which is distributed throughout the machine. Execution of this program takes place by repeated transformation of this graph into a normal form. The class *agent* represents the objects that perform the transformations on the packets. These transformations are known as *reductions.*

Packets have instance variables representing their state. The state of a packet is its left and right subgraphs, a list of directors, a flag indicating if it is in normal form, a flag indicating if it is currently being reduced by an agent, and finally a list of agents that are waiting for this packet to be reduced to normal form. The packet class in the Paragon description is defined:

```
class packet ::= ⟨rator,rand,string,innf,act,list agent⟩
data rator ::= packet | basic-value
data rand ::= packet | basic-value
data innf ::= Nf | Notnf
```

data act ::= Active | Inactive

where a basic-value can be a built-in operator, or a literal constant.

Agents reduce packets to normal form. As an agent is reducing a packet, it can be suspended as it waits for the result of a reduction of another packet, typically one or both of its subgraphs. The agent goes to sleep until it receives a "wakeup" message from a packet that has been reduced to normal form. A packet has two instance variables: the identifier of the packet that it is reducing, and a number that indicates the number of "wakeup" messages it needs to receive before it can continue. The agent class is defined as follows:

class agent ::= ⟨packet,integer⟩

Packets respond to three messages:

rewrite indicates that the packet is to be rewritten. The arguments to this message are the new structure of the packet.

need indicates that the packet is needed by an agent.

fire indicates that the packet is to be evaluated to normal form.

Agents respond to two messages:

reduce indicates that the agent is to reduce the packet which is an argument to the message.

wakeup indicates that the agent can resume reducing a packet.

For example, consider the strict basic operator × (multiply). Both arguments to × need to be integers. If any are packets, then they need to be fully evaluated to integers. In this case the agent sends a need message to the packets that have to be evaluated further, and it sleeps until these have been reduced to normal form.

This is modelled in Paragon by an agent receiving a reduce message. The argument to this message is the packet to be reduced. The Paragon for this operator is firstly for the case in which both arguments are integers and then for the case in which both are packets.

[o×1]
⟨_,0⟩ **given** reduce(p_1) **when**

```
        p1 = ⟨p2,n,nil,_,_,_⟩ ∧
        p2 = ⟨×,m,nil,_,_,_⟩ with
        is_integer(m) ∧ is_integer(n)
        → self
            then
            p1 ! rewrite(m×n,nil,nil,Nf)


[o×4]
⟨_,0⟩   given reduce(p1) when
        p1 = ⟨p2,p4,nil,_,_,_⟩ ∧
        p2 = ⟨×,p3,nil,_,_,_⟩
        → ⟨p1,2⟩
            then
            p3 !! need(self) ||
            p4 !! need(self)
```

The first rule rewrites the target packet so that it contains the result of multiplying m by n. The agent is no longer needed to reduce this packet, and so its state is unchanged.

The second rule changes the state of the agent so that it is suspended waiting for the packets p_3 and p_4 to be reduced to normal form. These packets are sent need messages.

Redexes and Directors. The process of finding the next reducible expression is made explicit in the Paragon description. The Paragon rule is as follows:

```
[ot]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,_,nil,_,_,_⟩ ∧
        P2 = ⟨_,_,_,Notnf,_,_⟩
        → ⟨P1,1⟩
            then
            P2 !! need(self)
```

This rule corresponds to looking down the spine of the program graph for a redex. The agent receives a message asking it to reduce the packet p_1. If the director list at this packet is empty, and the left subpacket is not in normal

form, then the agent sends p_2 a need message, and it goes to sleep until it receives one wakeup message. This wakeup message will come when p_2 is reduced to normal form.

Primitives. In the term rewriting system, we defined several primitives which take variable numbers of arguments. As defined in [BHK88] the language only allows us to define rules in terms of a constant number of matches on the lhs. For example, recall the definition of the selection primitive K-n-i:

$$\text{K-}n\text{-}i\ x_0 \ldots x_{n-1} \quad \rightarrow \quad x_i$$

This is can be defined in Paragon, as the following illustrates:

```
[K-n-i]
⟨_,0⟩   given reduce(p_n) when
        p_n = ⟨p_{n-1},x_{n-1},nil,_,_⟩ ∧
        p_{n-1} = ⟨p_{n-2},x_{n-2},nil,_,_⟩ ∧
              ⋮
        p_0 = ⟨K-n-i,x_0,nil,_,_,_⟩
        → ⟨_,_⟩
            then
            p_n ! rewrite(x_i,nil,nil,notnf) ;
            self !! reduce(p_n)
```

This involves a slight extension to the language in that it must now allow a variable number of matches on the lhs. Of course we could have completely defined the operation of K-n-i by writing n rules, one for each value of n, which would have been tedious and unnecessary.

Data Constructors/Selectors A similar problem arises when we come to specify the operation of some of the data constructors/selectors. PACK-n-d is treated almost exactly like SEL-n-i above. However the main problem arises this time from the rule requiring a variable number of packets on the rhs, and is present in the data structure unpacking primitives UNPACK-n, and UNPACK!-n.

The solution for the non-strict version is given here, as this is the most complicated of the two unpacking primitives.

[UNPACK-n]

```
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,e,nil,_,_,_⟩ ∧
        P2 = ⟨UNPACK-n,f,nil,_,_,_⟩ ∧
        n ≥ 2
        → ⟨P1,0⟩
            then
            P1 ! rewrite(pn−2,sn−1,nil,notnf) ;
            self !! reduce(P1)
            where
            sj = new(packet,⟨SEL-n-j,e,nil,notnf,inactive,nil⟩)
            p0 = new(packet,⟨f,s0,nil,notnf,inactive,nil⟩)
            pj = new(packet,⟨pj−1,sj,nil,notnf,inactive,nil⟩)
```

Unfortunately the $n = 1$ clause does not fit in to the general pattern and needs to be stated separately.

```
[UNPACK-1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,e,nil,_,_,_⟩ ∧
        P2 = ⟨UNPACK-1,f,nil,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(f,p,nil,notnf) ;
            self !! reduce(P1)
            where
            p = new(packet,⟨SEL-1-0,e,nil,notnf,inactive,nil⟩)
```

Again, we have to add slightly to the language. In this case, we use the where clause to allow us to introduce new packets using a schema for new definitions. The variable n is bound to a value when the rule is matched. We rewrite P_1 to a packet that contains packets s_{n-1} and p_{n-2}. We then provide a generic definition for the set of packets p_j and s_j. As the definition for p_j is recursive, we provide the base case definition for p_0.

Booleans It is worthwhile looking at the Paragon for the boolean operator OR to demonstrate the strictness of the operator.

```
[bool1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
```

```
        P2 = ⟨OR,x,nil,_,_,_⟩ ∧
        is_packet(x)
        → ⟨P1,1⟩
            then
            x !! need(self)

[or1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
        P2 = ⟨OR,FALSE,nil,_,_,_⟩
        → ⟨_,_⟩
            then
            P1 ! rewrite(y,nil,nil,notnf) ;
            self !! reduce(P1)

[or1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
        P2 = ⟨OR,TRUE,nil,_,_,_⟩
        → ⟨_,_⟩
            then
            P1 ! rewrite(TRUE,nil,nil,nf)
```

There are three rules. The first indicates that the operator is strict in its first argument. If it has not been evaluated, then it is sent a need, and the agent waits for a response indicating it has finished. The second rule applies when the first argument has been evaluated to FALSE. In this case the result is the result of the second argument. The agent continues to reduce the second argument to normal form. The final rule applies when the first argument to OR has been evaluated to TRUE. In this case the result is TRUE and no further evaluation needs to be done.

Comparison and Numerical operators The comparison and numerical operators are all defined in terms of a generic, as they are all strict in both their arguments. See the appendix for details.

Sequencing, Strictness and Termination Of the three operators SEQ, STRICT and ABORT, SEQ is the most interesting. It is defined to evaluate the first argument, then evaluate and return the second. There is no reason why this needs to be done sequentially, as long as the first argument has been fully evaluated before the second is returned. The Paragon for this might be:

```
[SEQ1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg2,nil,_,_,_⟩ ∧
        P2 = ⟨SEQ,arg1,nil,_,_,_⟩ ∧
        → ⟨P1,2⟩
          then
          arg1 !! need(self) ||
          arg2 !! need(self)

[SEQ2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg2,nil,_,_,_⟩ ∧
        P2 = ⟨SEQ,arg1,nil,_,_,_⟩ ∧
        arg1 = ⟨_,_,_,nf,_,_⟩
        → ⟨_,0⟩
          then
          P1 ! rewrite(arg2,nil,nil,notnf) ;
          self !! reduce(P1);
```

The first equation applies when arg1 has not been fully evaluated. We can safely spawn the evaluation of arg2 in parallel with that of arg1. The second equation applies when arg1 has been fully evaluated to normal form. It is only then that we can continue the evaluation of arg2.

STRICT can also evaluate its arguments in parallel. This time however, we must wait for the complete evaluation of the second argument before returning the result of applying the first argument as a function to the second. It can be defined in a similar way to SEQ and is included in appendix B.

ABORT handles exceptions. The graph is rewritten so that the ABORT operator propagates all the way to the top level.

```
[ABORT]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨ABORT,x,_,_,_,_⟩ ∧
        x ≠ nil
        → self
          then
          P1 ! rewrite(ABORT,nil,nil,nf)
```

The Interface

The specification of COBWEB given here introduces a new class named *interface*. This class deals with loading the program graph into the machine and starting it running. This is defined in terms of a number of "built-in" objects and methods which define the Paragon system interface with the outside world.

A Paragon specification has two objects already defined. These are named *i* and *o* for input and output. The input object that can send messages to objects within the system. Any object within the system can send the message *output* to the output object. Input and output are meant to define raw communication with the outside world.

The *interface* class for COBWEB is defined in terms of these objects. The function of this class is to load packets in, and arrange for them to be executed by an agent. When the result is known, then the interface class will be responsible for communicating the result to the outside world.

This class might be defined as follows.

```
class interface ::= ⟨packet⟩

⟨_⟩     given run(p)
        → ⟨p⟩
          then
          p !! fire

⟨p⟩     when
        p = ⟨_,_,_,nf,_,_⟩
        → self
          then
          o !! output(p)
```

The only instance variable known to the interface is a packet identifier. When the interface is given a packet to evaluate, then the interface fires the packet. It then waits until the packet has evaluated to normal form, whereupon it sends this packet to the outside world.

The system will be started with an instance of the class *interface*, and will be told that the input object *o* will be able to communicate with this instance.

In practice, this class will be more much complicated than this. For example it will arrange for errors to be reported, and for the complete result to be output if it is a data structure. We explain it here to introduce the concept of input and output in a Paragon system.

3.4 Translating Paragon to hardware

The previous section gave a specification for COBWEB in Paragon. This section will show how to translate any Paragon specification into a hardware design. The resulting general purpose design will be seen to be restrictive in several senses, so we suggest a design methodology that will allow us to avoid these restrictions.

3.4.1 The Target Description

Before addressing the problem of how to translate Paragon into hardware, we need to identify what we are to translate the specification to.

Hardware systems consist of physical circuits connected by a physical communications medium. This is the kind of design we wish to produce. This requires us to produce two parts to the design, the first is the topology of the blocks, and the second is a specification of the operation of those blocks.

The topology can be presented as a directed graph, where the nodes are the blocks, and the arcs are the connections. The direction of the arc specifies the direction of communications. The operation of the blocks can be specified in some target language.

As a language for describing hardware, the target language should be a hardware description language (HDL) for example ELLA [Ell86], or OCCAM [MK87] that allows compilation to silicon. Silicon compilation techniques are described in [Gaj88].

However Paragon is of course suitable for specifying systems other than hardware, for example we might want to describe a simple program running on a microprocessor in an embedded system. In this case the target language would need to be a program in the machine code of the microprocessor, or a program in a high level language that will compile to machine code for that microprocessor.

Even if we are specifying hardware with Paragon, we might want to first compile the Paragon to a program in a high level language so that we can simulate it, and maybe fine-tune some parameters. Given these requirements, the target language chosen is a procedural imperative language with some message passing primitives. It is restricted in the sense that it must be static, that is without dynamic storage management, or non-tail recursion. In this respect it is similar to OCCAM2, but with an algol-like syntax and data structures.

A program in this language consists of a set of procedure definitions plus a description of the initial state of the system. This describes objects plus their connectivity in the same way as the main body of an OCCAM program.

There are three message passing primitives. *Read* from a connection, *write* to a connection, and *acknowledge* that a message has been received. Messages are written and composed using the same syntax as in Paragon, except that only synchronous message passing is allowed, and that messages can be replied to. For example we can write

```
a := o ! m(x)
```

meaning send message m with argument list x to object o and assign the result to the variable a. In addition, the language has a reply x construct which sends the value x back to the caller.

3.4.2 The translation process

The fundamental constructs of Paragon are objects and messages. Objects have a state, and in a design this state must be stored somewhere. Objects also have methods, and there must be some mechanism for executing these methods after a message has been received. A simple first attempt at a hardware solution is to have a block of logic with memory that can store the state of the object, and execute the methods. We shall call these *object processors*. If there are a number of static objects in the system, then we can have number of static object processors in our design, each of which corresponds to an object in our Paragon specification. However in any system of reasonable complexity, we inevitably need to create new objects dynamically, and as we cannot create physical circuits dynamically, we can only use this solution when there are only static objects.

Messages can be synchronous or asynchronous. A possible design for a message passing medium is a physical synchronous connection joining two object processors. However this will only be feasible if all the messages are synchronous.

So the simplest Paragon system is one that is both static and synchronous. We shall see that it is fairly easy to translate a specification of this type into a hardware design. If we can translate a more general specification into one that is synchronous and static, then this will provide a route towards a general design methodology.

We can proceed by classifying Paragon specifications in increasing order of complexity.

1. A system with a fixed set of objects, and synchronous communications only.

2. A static system as above but with asynchronous communications as well as synchronous.

3. A dynamic system with synchronous and asynchronous communications.

First we shall concentrate on producing a specification of the topology of the system. Second we shall show how the specification of the operation of the blocks is produced.

Before dealing with these systems, we introduce some terminology.

A *specification* (C, O) is a tuple consisting of a set of classes, and a set of objects. The set of objects is known as the *configuration*, and comprises the objects that the system will start out with. A given class c is defined in terms of its structure, and its methods. We denote the set of all instances of a class c by $I(c)$.

There are two types of communication in Paragon. An object o that communicates synchronously with a set of objects O is said to be *synchronous in* O. This is written $S(o) = O$. Similarly an object o that communicates asynchronously with a set of objects O is said to be *asynchronous in* O, and is written $A(o) = O$. Finally, an object o that can create objects which are instances of the classes in C is said to be *dynamic in* C, and is denoted $D(o) = C$. We can define S and A over classes using the same criteria, but note that if a class C is synchronous or asynchronous in a set of classes C', that does not imply that an object of class C, say o, is necessarily synchronous in all objects, or indeed any objects of class C'. For example, an object may be synchronous or asynchronous in another object depending on the value of one its instance variables, or the value of one of the messages it receives. We can potentially analyse our specification and find that these circumstances can never arise. An object or a class can only be dynamic in a class however, as it is meaningless to assert that an object is dynamic in another object.

A *message* is a data construction appearing in a **given** clause. The message is identified by the constructor appearing outermost in this construction. Such constructors must appear only there or as the object of the "!" and "!!" message transmission operators. The *signature* of a class is the set of messages to which it may respond—that is, the set of constructors appearing in the class's rules' **given** clauses.

A message is *total* if, for the class to which it applies, it can always be consumed. That is, the rules admitting the message must cover all the possible values of the target object's class. A message is *partial* (i.e. not total) if pattern matching after message receipt can fail, requiring the message to be retained for re-matching whenever the target object's state changes.

3.4.3 A static synchronous system

The simple system is the static synchronous system, and as such is fairly easy to translate into a hardware design.

Consider a specification (C, O) which is static and synchronous with a configuration O. As the total number of objects is constant, we can create an object processor for each object in O. This might seem reminiscent of a CSP description, but there is one crucial difference: in Paragon, the address of an object can be communicated in a message, or indeed stored as an instance variable. So the set of other objects that a given object can communicate with is given by the union of all the domains of all the object variables known to that object.

For example consider a Paragon system $(\mathcal{C}, \mathcal{O})$ with four objects $\mathcal{O}_0 \ldots \mathcal{O}_3$, involving three classes $\mathcal{C}_0 \ldots \mathcal{C}_2$ with the following rules: $\mathcal{O}_0$ is an instance of $\mathcal{C}_0$, $\mathcal{O}_1$ is an instance of $\mathcal{C}_1$ and both $\mathcal{O}_2$ and $\mathcal{O}_3$ are instances of $\mathcal{C}_2$. The classes have the following partial specification. The structures of the packets have been omitted for clarity.

```
class C0 ::= ⟨⟩
⟨⟩      given m(x)
        → ⟨⟩
            then
            O1 ! z(x)

class C1 ::= ⟨⟩
⟨⟩      given n(c,x)
        → ⟨⟩
            then
            c ! y(x)

class C2 ::= ⟨⟩
⟨⟩      given o(x)
        → ⟨⟩
            then
            O1 ! n(self,x)
```

The first object $\mathcal{O}_0$ communicates only simple messages, and only ever to $\mathcal{O}_1$. So $\mathcal{O}_0$ needs to be connected only to $\mathcal{O}_1$, and only in one direction, as no other object can send a message to $\mathcal{O}_0$. $\mathcal{O}_1$ however can receive a message that contains an object address c. The objects that send this message to $\mathcal{O}_1$ are $\mathcal{O}_2$ and $\mathcal{O}_3$, and we can see that the only addresses that they communicate are the addresses of themselves. $\mathcal{O}_1$ must therefore be connected bidirectionally to both $\mathcal{O}_2$ and $\mathcal{O}_3$ Figure 3.4 shows the topology of this system.

In any system there may be some hidden communication between objects. For example, consider the following Paragon specification.

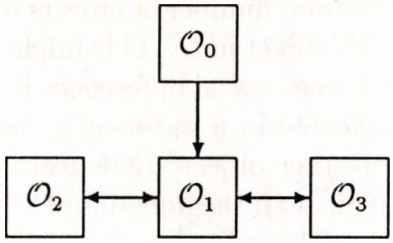


Figure 3.4: The topology of objects $\mathcal{O}_0$ to $\mathcal{O}_3$

```
class a ::= ⟨integer,b⟩
class b ::= ⟨integer,integer⟩

⟨B,c⟩   given m(x)
        when
        c = ⟨0,0⟩
        → ⟨B+1,c⟩
          then
          c ! n(x)
```

m is a message sent to objects of class *a*. This class contains an identifier for objects of class *b*. We attempt to pattern match against the structure of the object that this identifies. This requires a communication with the object. As we shall see in section 3.4.6 these communications can be made explicit.

Now the general case is when every object can potentially communicate to every other object in the system. This implies that they must be all connected directly. The obvious way of doing this is in hardware is by connecting all the object processors together using a synchronous interconnection network such as a synchronous bus, though we then lose some concurrency as object processors contend for its use. We can now see that it is a fairly easy matter to design a hardware system that will implement a Paragon specification that is static and synchronous.

3.4.4 A static asynchronous system

The second class of Paragon specifications has static objects, but there are asynchronous communications as well as synchronous. Given that we now know how to design a hardware system for the static synchronous system,

then if we can transform our specification into one where all the communications are synchronous, then we can design a hardware system for this class of specifications as well. In practice, this is easy. We can do this by creating a new object called the *task pool.* The task pool is written $T(C)$ where C is a set of classes. $T(C)$ is a task pool that can handle messages from all the classes in C. The function of the task pool object is to buffer asynchronous communications. We can then transform all the asynchronous communications to ones that communicate to the task pool. The task pool is made up of a list of tasks, where a task is defined as a destination, a message and a list of arguments. The task pool may be defined as follows.

```
class task-pool = list task
data task = ⟨destination,message,args⟩

⟨M⟩     given add(d,m,x)
        → ⟨M++⟨d,m,x⟩⟩

⟨⟨d,m,x⟩::M⟩
        → ⟨M⟩
          then
          d ! m(x)
```

Now all asynchronous communications, for example

```
a !! m(x)
```

can translated into

```
t ! add(a,m,x)
```

where t is an instance of task-pool. We need an instance of task-pool for every object that is asynchronous in any other object. Each instance of task-pool associated with an object o must be shared with (ie connected to) every object in which o is asynchronous.

Although the task pool is specified as being a list of tasks, for the system to remain static, this list must have an upper bound. This inevitably constrains the specification. As the methods for the task pool are synchronous, the whole system is now static and synchronous, so we can design a hardware system using the same technique as before. For example, consider a system consisting of four objects $\mathcal{O}_0 \ldots \mathcal{O}_3$ with the following properties:

$$
\begin{array}{lcl}
\mathrm{S}(\mathcal{O}_0) & = & \emptyset \\
\mathrm{A}(\mathcal{O}_0) & = & \{\mathcal{O}_1\} \\
\mathrm{S}(\mathcal{O}_1) & = & \{\mathcal{O}_3\} \\
\mathrm{A}(\mathcal{O}_1) & = & \{\mathcal{O}_2, \mathcal{O}_3\} \\
\mathrm{S}(\mathcal{O}_2, \mathcal{O}_0) & = & \emptyset \\
\mathrm{A}(\mathcal{O}_2, \mathcal{O}_3) & = & \{\mathcal{O}_1\}
\end{array}
$$

This system will have a topology as shown in 3.5 Notice that the functions

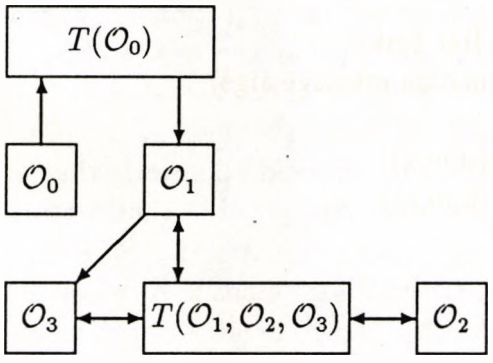


Figure 3.5: The topology of a system of four objects.

S and A dictate the direction of message flow in the design.

In general, where all objects are both synchronous and asynchronous in all others. we can design a system as follows. If we have a system $(\mathcal{C}, \mathcal{O})$ with n objects in $\mathcal{O}$ numbered from 0 to $n-1$, then we can connect the objects, and the task pool up to a bus as in figure 3.6.

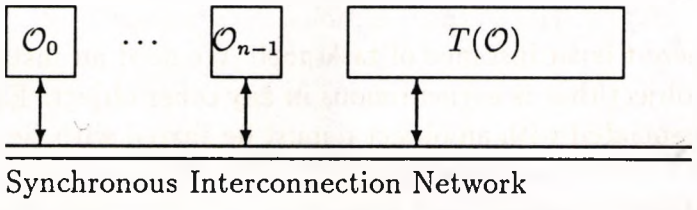


Figure 3.6: A general topology for a system from a static synchronous & asynchronous Paragon specification

3.4.5 A Dynamic system

We can now turn our attention to the problem of designing a system that is dynamic. The immediate problem is that we cannot simply create new physical object processors in hardware to model the new objects in our specification. What we need to do is to transform the system somehow to make it static. One solution is to say that all classes that are known to be dynamic will use a *heap* to store instances of the class. Each heap will be specific to its class. Individual objects will then be identified by their place in the heap. New objects will be created by allocating space from the heap. If we have in our configuration a class c that is dynamic, then we must replace this class with two new objects in our configuration: $H(c)$ which represents the heap for objects of class c, and $P(c)$ known as the *class processor* for c which will implement in hardware the methods for class c.

All objects in our configuration that are dynamic in c must now be connected synchronously to $H(c)$, and all objects that are synchronous or asynchronous in c must now be connected in that manner to $P(c)$.

The heap can be described as a static Paragon object, and will receive messages such as *new*, to allocate space to objects, *free* to release it again, and *read* and *write* to provide access. As the heap will be bounded, there is a need to reclaim space so the heap must be garbage collected. We can define the operation of the heap in Paragon, but it is better to define in the target language, as certain messages need to be replied to.

The class processor $P(c)$ is a static object to which all messages sent to objects of class c are delivered. Objects of class c will now be identified by their address in $H(c)$, so the operation of $P(c)$ when it receives a message will be to read the body of the object from $H(c)$, execute the method, and write the object back if it has been rewritten.

The main issue that arises from transforming a configuration that is dynamic into one that is static but uses a heap is the loss of concurrency among objects of the same class as they are now multiplexed in time over $P(c)$. However there is no reason why there should not be many instances of $P(c)$ each with access to $H(c)$. This is a choice the designer must make based on knowledge of the critical components of the system. Analysis of a specification can also tell us if we can create multiple instances of $H(c)$. If two object processors o and o' are dynamic in a class c, we can potentially find out from static analysis of the specification if $H(c)$ and $P(c)$ need to be shared physically between the o and o'. For example if o and o' create instances of class c for their personal use, and never communicate them directly or indirectly in a message, then we know we can create an instance of $H(c)$ and $P(c)$ for each object o and o'.

The most general system is one in which all the classes are dynamic in all others and themselves. The topology of a completely general system is

shown in 3.7.

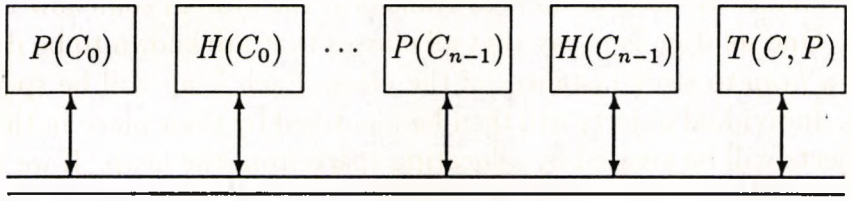


Figure 3.7: The topology of a completely general dynamic system

Now the task-pool T is itself an object processor, and the task-pool class can also be seen as a dynamic class. Therefore we can create a heap $H(T)$ and a processor $P(T)$ for the set of task pools in the same way as we did for the other dynamic classes.

We now have a clear separation of functions among object processors. For each class we have a processor, and a heap. Although the heap is an object processor itself, the methods for all the heaps are essentially the same, and we can therefore unify all the heaps into one greater heap $H(C,T)$.

We can make a further generalisation by observing that we can create a unified class processor $P(C,T)$ that will execute methods for all the classes, including the task pool. If we can express the methods for all these classes as a stored program, then we can express all the class processors as a single processor which takes its instructions from a heap. A processor of this type would be written $P(P(C,T))$, and its heap $H(P(C,T))$. This method heap can then be merged with the object heap, and what we are left with is a general processor $P(P(C,T))$, plus some memory $H(P(C,T),C,T)$.

It is interesting to derive what looks suspiciously like a conventional multitasking von Neumann uniprocessor from a general purpose Paragon specification. However, we do not always need to go down this route to a design, and can use analytical techniques on individual specifications to derive more concurrent designs.

3.4.6 The design of object processors

As defined above, object processors are blocks of logic. So far, we have concentrated on producing a static synchronous Paragon specification from a dynamic general one. This gives us a fixed topology for the system. Here we

will discuss how we might translate the methods for the classes into specifications for the operation of the object processors.

One way of doing this is to transform all the original Paragon methods into ones that use the heap and the task pool, and then translate these methods into our target language. This proves to be fairly awkward because Paragon has no mechanism for a "call-and-reply" type of construct, which is necessary when performing operations such as requesting space from the heap. So instead of transforming our specification into one that is static and synchronous, we can translate the methods directly into a procedure in our target language.

We start by giving a transformation scheme for a static synchronous system, and proceed by saying how we would add to it to handle asynchronous communications and dynamic classes.

We can describe a general rule $\mathcal{R}_j$ as follows:

$\mathcal{S}_j$ **given** m(x)
when $\mathcal{G}_j$
$\rightarrow$ $\mathcal{S}'_j$
then $\mathcal{C}_j$
where $\mathcal{B}_j$

A rule $\mathcal{R}_j$, $0 \leq j < k$ has a list of guards $\mathcal{G}_j$ that operate on the arguments passed by the message, and the variables bound by $\mathcal{S}_j$. If these evaluate to true, then the object which matches $\mathcal{S}_j$ is rewritten to $\mathcal{S}'_j$ and the communications $\mathcal{C}_j$ are generated. A list of bindings $\mathcal{B}_j$ to names is provided.

In addition there may be a number of spontaneous rules for each class. If there are l of these rules, they take the following form for $0 \leq j < l$

$\mathcal{T}_j$ **when** $\mathcal{G}'_j$
$\rightarrow$ $\mathcal{T}'_j$
then $\mathcal{C}'_j$
where $\mathcal{B}'_j$

We can produce a specification for an object processor given these rules. This will listen on its inputs for messages, read them, and execute the methods for these messages. It will also take care of executing the spontaneous rules. This is described as an infinitely looping procedure defined in terms of its connections.

```
def object-processor(connections) =
        while true do
                if there is a message pending then
                        if the spontaneous rule applies to the target then
                                apply the spontaneous rule
                        fi;
                        execute the method for that message;
                        if the method has succeeded then acknowledge sender fi
                else    attempt to apply the spontaneous rule to the target
                fi
        od
enddef
```

A message is pending if the sender has presented some data on a connection. The message remains pending until it has been acknowledged by the receiver. As there are a number of connections, the object processor will arbitrate among the connections by checking whether a message is pending on each one in turn.

Before we can define a translation function we need to address a problem that arises with these spontaneous rules. Imagine a "stopwatch" class with the following definition.

```
class stopwatch ::= ⟨integer,status⟩
type status ::= On | Off

⟨n,on⟩   →       ⟨n+1,On⟩
⟨_,_⟩    given reset →     ⟨0,Off⟩
⟨n,_⟩    given start →     ⟨n,On⟩
⟨n,off⟩  given stop →      ⟨n,Off⟩
⟨n,_⟩    given read(c)
         → ⟨n,_⟩
           then
           c !! reply(n)
```

This defines an object that behaves like a stopwatch and can be reset, started, stopped and read by a user. This definition has one spontaneous rule. We expect this rule to be applied spontaneously. If the class is dynamic then we are storing objects of this class in a heap. One option is for the class processor to cycle through every object in the heap and attempt to apply this rewrite rule to each object in turn. This could be done when the processor is idle waiting for a message. However if we are using the object processor heavily, we do not want to lock out the spontaneous rewrite indefinitely.

A partial solution is to transform the rule so that the rewrite takes place when a special message has been received. For example the new transformed definition would read:

```
class stopwatch ::= ⟨integer,status⟩
type status ::= on | off

⟨n,on⟩  given spontaneous
        → ⟨n+1,on⟩
          then
          self !! spontaneous

⟨_,_⟩   given reset
        → ⟨0,off⟩

⟨n,_⟩   given start
        → ⟨n,on⟩
          then
          self !! spontaneous

⟨n,off⟩ given stop
        →⟨n,off⟩

⟨n,_⟩   given read(c)
        → ⟨n,_⟩
          then
          c !! reply(n) ||
          self !! spontaneous
```

That is, every time an object is rewritten into a form where the spontaneous rewrite might apply, we send the *spontaneous* message to that object. When an object is created it will be sent this message.

Unfortunately, one of these rules might apply to an object even if it has not been rewritten itself. Consider for example the following example for a dynamic class:

```
class A ::= ⟨a,a⟩
type a ::= integer | A

⟨x,y⟩   when
        x = ⟨0,0⟩
```

```
        →⟨y,y⟩
```

The spontaneous rewrite applies to an object, which contains other object identifiers. Now the rewrite only applies when one of the sub objects has a certain structure, the top level object may never be rewritten, yet the rewrite might apply.

There are two alternative solutions to the problem: We can get the class processor to cycle through all the objects in the heap when it is idling, or we can send every object an *spontaneous* message every time they receive any message not just when they are rewritten into a form where the rewrite applies. These amount to the same thing, but the latter is the cleaner of the two, and for now this is the solution we will adopt.

Our example will thus need to be transformed into

```
class stopwatch ::= (integer,status)
type status ::= on | off

⟨n,on⟩  given spontaneous
        → ⟨n+1,on⟩
          then
          self !! spontaneous

⟨_,_⟩   given reset
        → ⟨0,off⟩
          then
          self !! spontaneous

⟨n,_⟩   given start
        → ⟨n,on⟩
          then
          self !! spontaneous

⟨n,off⟩ given stop
        → ⟨n,off⟩
          then
          self !! spontaneous

⟨n,_⟩   given read(c)
        → ⟨n,_⟩
          then
          c !! reply(n) ||
```

```
self !! spontaneous
```

We can sketch a set of translation functions that will translate the set of rules into a procedure in an imperative language that will execute the method for that message.

TM[[Paragon method]] which maps a complete set of Paragon rules for a given message onto a procedure in our target language.

TO[[Object]] will map an object on the lhs of a Paragon rule onto a set of statements that will both test if the current object matches and bind names in the pattern match to values (if necessary). If the match is successful, a flag success is set.

TB[[Where binding]] maps a set of Paragon where bindings onto a set of similar binding statements for variables in the target language.

TC[[Communication]] will map the set of communications onto a list of procedure calls that carry out these communications.

TG[[Guard]] will map the set of Paragon guards and required **where** bindings onto a logical expression in the imperative language. These guards will operate not only on the variables bound by the message, but also on the variables bound by **TO**, the required subset of those bound by **TB** and **self**.

TS[[Object]] will map an object onto an expression in the target language representing the structure of that object.

The body of each method is defined as a procedure using an informal pseudocode. For example **TM**[[$\mathcal{R}$]] is defined in figure 3.8. The procedure has the same name as the message being received and is defined in terms of self which indicates the object receiving the message. We can refine and optimise this procedure using standard techniques such as removing common subexpressions, and omitting statements that can never be reached, such as the FAIL which is not needed if the method is total.

Another issue with the specification is that all rules must be mutually exclusive. This is a side effect of there being no matching order. For the sake of proving the correctness of the specification, this is a bonus, but as a specification for a physical machine that might be implemented, this might be construed as a slight deficiency. The onus of specifying the order of matching is left to the designer. An arbitrary order is not going to be the most efficient. The designer must therefore inspect the rules and give to them an order which will be most efficient.

We can now take a further look at the definition of the translation functions.

```
def m(self, x_0, ..., x_{n-1}) =
        TO[[S_0]];
        if success and TG[[G_0]] (x_0,...,x_{n-1}, self)
        then    TB[[B_0]];
                self := TS[[S'_0]];
                TC[[C_0]]
        else    TO[[S_1]];
                if success and TG[[G_1]] (x_0,...,x_{n-1}, self)
                then ...
                   :
                        else    TO[[S_{k-1}]];
                                if success ∧ TG[[G_{k-1}]](x_0,...,x_{n-1}, self)
                                then    TB[[B_{k-1}]];
                                        self := TS[[S'_{k-1}]];
                                        TC[[C_{k-1}]]
                                else
                                        FAIL
                                fi
                   :
                fi
        fi
enddef
```

Figure 3.8: Translation function $\mathbf{TM}[[\mathcal{R}]]$

TC The translation of the communications is fairly easy. If the communication is synchronous and the target is a static object, then we can simply write the communication as a synchronous procedure call. As communications are composed in the same way as in Paragon, we can have:

$$\begin{aligned} \mathbf{TC}[[\mathsf{a\ !\ m(x)}]] &= \mathsf{a\ !\ m(x)} \\ \mathbf{TC}[[\mathsf{x} \parallel \mathsf{y}]] &= \mathbf{TC}[[\mathsf{x}]] \parallel \mathbf{TC}[[\mathsf{y}]] \end{aligned}$$

when a is a static object.

If it is asynchronous, we need to express the communication in terms of the task pool for that object. That is:

$$\mathbf{TC}[[\mathsf{a\ !!\ m(x)}]] = \mathsf{T\ !\ add(a,m,x)}$$

when a is a static object, and where T is the task pool that has been assigned to objects of the class that a belongs to.

If the target is in a dynamic class then we need to send the communication to the class processor.

$$TC[[a\ !\ m(x)]] = P\ !\ m(a,x)$$

when a is an object in a dynamic class, and where P is the class processor for that class.

The final case is when the target is in a dynamic class and the communication is asynchronous. In this case we must send the message to the task pool for that class.

$$TC[[a\ !!\ m(x)]] = T(P)\ !\ add(P,m,(a,x))$$

that is, the target is the task pool for the class processor for a.

TB. The binding function is the easiest to define. Given that we are using an imperative language, we can say that we have a number of variables to hold the values of the bindings. However, if the rhs of the binding is a new expression then this requires that we communicate with the heap associated with the class that is being allocated. This will take the form of a synchronous communication of the message *new* to the appropriate heap, and a wait until it returns.

TO. This function is fairly difficult to define, as it needs to do three things. It needs to test if the object on the rhs matches, and if so, it needs to bind some values to names, and set a flag success if the match was successful. This can be done in two stages — first check if the object matches the structure, and then bind the names. This part of the procedure would benefit from optimisation.

TG. The translation of the guards is also fairly difficult. The function generated can be thought of as delivering true or false, and having the side effect of binding some values to names. Techniques used for compiling pattern matches for functional languages [Pey87] may be used here.

TS. This function simply returns an expression representing the structure of its argument, and as such is fairly easy to define.

3.4.7 A general purpose methodology

The sections above suggest a general purpose methodology for translating a specification in Paragon into a hardware design. This consists of the following phases

1. Write down the system equations. That is for each object and class say what objects or classes it is synchronous, asynchronous or dynamic in.

2. Remove all dynamic classes by creating heaps and class processors for these classes. All classes and objects that were dynamic in these classes now become synchronous or asynchronous in these new objects depending on an analysis of the methods. Rewrite the system equations.

3. Remove all asynchronous objects by creating task pools. All objects that were asynchronous in other objects now become synchronous in the task pool allocated to those objects. The system equations should now be in terms of the synchronous function S only. This completely defines the minimum topology for the hardware design.

4. For each static object create an object processor, and specify the operation of each object by applying **TR** to each method.

Backtracking will possibly be necessary at any of these stages. For example if any of the block definitions prove too complex, then we might want to simplify some of the earlier rules.

This provides an algorithmic route from a specification to a design. In the next section we shall see that we can derive some heuristics in order to tune our design more closely to requirements not made explicit.

3.5 Design of COBWEB

In this section we shall apply our design methodology to the specification of COBWEB. We shall see that the specification as it stands is too complicated to make an efficient machine, so we shall return to the specification to make some amendments.

3.5.1 Design of the COBWEB class topology

There are three classes in the specification of COBWEB: Agent ($\mathcal{A}$), Packet ($\mathcal{P}$), and Interface ($\mathcal{I}$). The specification is $(\{\mathcal{A},\mathcal{P},\mathcal{I}\},\{i,o\})$. By analysing these classes by hand we come up with the following equations.

$$\begin{array}{lll} S(\mathcal{A}) = \{\mathcal{P}\} & A(\mathcal{A}) = \{\mathcal{A},\mathcal{P}\} & D(\mathcal{A}) = \{\mathcal{A},\mathcal{P}\} \\ S(\mathcal{P}) = \emptyset & A(\mathcal{P}) = \{\mathcal{A}\} & D(\mathcal{P}) = \{\mathcal{A}\} \\ S(\mathcal{I}) = \{o\} & A(\mathcal{I}) = \{\mathcal{P}\} & D(\mathcal{I}) = \{\mathcal{P}\} \end{array}$$

Following the methodology we remove dynamic classes by creating heaps, and class processors. The dynamic classes are $\mathcal{P}$ and $\mathcal{A}$. For $\mathcal{A}$ we need $H(\mathcal{A})$ and $P(\mathcal{A})$, for $\mathcal{P}$ we need $H(\mathcal{P})$ and $P(\mathcal{P})$. We shall merge the heaps into one $H(\mathcal{P},\mathcal{A})$.

Our set of objects becomes

$$\{H(\mathcal{A},\mathcal{P}),\ P(\mathcal{P}),\ P(\mathcal{A}),\ i,\ o\ \}$$

and the new set of system equations is

$$\begin{array}{lcl}
S(P(\mathcal{A})) & = & \{H(\mathcal{A},\mathcal{P})\} \\
A(P(\mathcal{A})) & = & \{H(\mathcal{A},\mathcal{P}), P(\mathcal{P})\} \\
S(P(\mathcal{P})) & = & \{H(\mathcal{A},\mathcal{P})\} \\
A(P(\mathcal{P})) & = & \{H(\mathcal{A},\mathcal{P}), P(\mathcal{A})\} \\
S(H(\mathcal{A},\mathcal{P})) & = & \{P(\mathcal{A}), P(\mathcal{P})\} \\
A(H(\mathcal{A},\mathcal{P})) & = & \{P(\mathcal{A}), P(\mathcal{P})\} \\
A(\mathcal{I}) & = & \{o, H(\mathcal{P})\} \\
A(i) & = & \{\mathcal{I}\}
\end{array}$$

To deal with the asynchronous message passing we need to create a task pool. This gives us the new object $T(\mathcal{A},\mathcal{P},\mathcal{I})$, and the system equations now read

$$\begin{array}{lcl}
S(P(\mathcal{A})) & = & \{H(\mathcal{A},\mathcal{P}), P(\mathcal{P})\, T(\mathcal{A},\mathcal{P},\mathcal{I})\} \\
S(P(\mathcal{P})) & = & \{H(\mathcal{A},\mathcal{P}), P(\mathcal{A}), T(\mathcal{A},\mathcal{P},\mathcal{I})\} \\
S(H(\mathcal{A},\mathcal{P})) & = & \{P(\mathcal{A}), P(\mathcal{P}), T(\mathcal{A},\mathcal{P},\mathcal{I})\} \\
S(T(\mathcal{A},\mathcal{P},\mathcal{I})) & = & \{P(\mathcal{A}), P(\mathcal{P}), P(\mathcal{I})\} \\
S(\mathcal{I}) & = & \{o, H(\mathcal{P},\mathcal{A}), T(\mathcal{A},\mathcal{P},\mathcal{I})\} \\
S(i) & = & \{\mathcal{I}, T(\mathcal{A},\mathcal{P},\mathcal{I})\}
\end{array}$$

This defines the minimum necessary topology for COBWEB. We can implement this topology using a bus as in figure 3.9. This introduces slightly more generality than specified. For example i can now communicate with $H(\mathcal{P},\mathcal{A})$, but this does no harm.

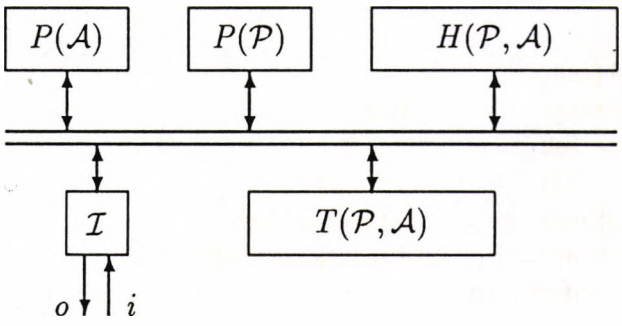


Figure 3.9: The topology of class processors for COBWEB.

3.5.2 Methods for the Object Processors

Now let us attempt to produce methods for the object processors. The object processors are:

$P(\mathcal{A})$	The processor for the agent class.
$P(\mathcal{P})$	The processor for the packet class.
$H(\mathcal{P},\mathcal{A})$	The heap for packets and agents.
$T(\mathcal{I},\mathcal{P},\mathcal{A})$	The task pool for the interface, packets, and agents.
i	The input from the outside world of class $\mathcal{I}$.

The following procedures have been designed by applying **TM** informally. As such each procedure has been refined quite substantially from the original output from the **TM** function. **TM**[[*rewrite*]] is shown to illustrate the result of simply applying the function blindly.

The interface

The interface has a structure that consists of one instance variable. This is an identifier of a packet, and as one of the rules for the interface consists of looking at the structure of this packet, then we need to look at the heap.

```
struct interface =
        packet
endstruct
```

The class processor is defined as follows:

```
def Interface(self) =
        if message = run(p) then
                self := p;
                T(I,P,A) ! add(fire,p)
        else if message = spontaneous then
                nf := H(P,A) ! read(packet.nf);
                if nf then
                        o ! output(p)
                fi
        fi
        T(I,P,A) ! add(self,spontaneous)
enddef
```

In this definition, the parameter self refers to the state of the processor. The parameter a refers to the heap identifier for the object that is receiving the message. The packet processor itself has no state — all the packet state is held in the heap. We pass the packet identifier as a parameter to the auxiliary procedures. These will refer to this as self to indicate we are working with the state of a packet, as opposed to the state of the processor.

```
def Packet-processor(self,a) =
        if message = rewrite(op,arg,ann,form) then
                packet-rewrite(a,op,arg,ann,form);
        else if message = need(agent) then
                packet-need(a,agent)
        else if message = fire then
                packet-fire(a)
        fi
enddef
```

Rewrite is a procedure that can benefit substantially from refinement. **TR**[[*rewrite*]] will produce the following procedure. This is shown in a structured english form rather than as a syntactically correct program.

```
def packet-rewrite(self,op,arg,ann,form) =
        if (ann = notnf) then
                H(P,A) ! write(self,(op,arg,ann,notnf,
                                self.activity, selfagentlist))
        else if (ann = nf) then
                H(P,A) ! write(self,(op,arg,ann,nf,inactive,
                                self.agentlist));
                wakeup every agent in self.agentlist
        else    ERROR
        fi
enddef
```

This can be refined into the following procedure.

```
def packet-rewrite(self,op,arg,ann,form) =
        H(I,P,A) ! write(self.op,op);
        H(I,P,A) ! write(self.arg,arg);
        H(I,P,A) ! write(self.ann,ann);
        H(I,P,A) ! write(self.nf,form);
        if form = nf then
                a := H(I,P,A) ! read(self.agentlist)
                while al ≠ nil do
                        a := H(P,A) ! read(hd(al));
                        T(I,P,A) ! add(a,wakeup);
                        al := tl(al)
                od
        fi
enddef
```

Need is defined as follows. As fire has a similar method, we do not it here.

```
def packet-need(self,agent) =
        packet := H(P,A) ! read(self);
        if packet is active then
                append(packet.agentlist,agent)
        else if packet is in nf then
                T(I,P,A) ! add(agent,wakeup)
        else
                append(packet.agentlist,agent);
                new-agent := H(I,P,A) ! new(Agent);
                T(I,P,A) ! add(new-agent,reduce,self)
        fi;
        H(I,P,A) ! write(self,packet)
enddef
```

The Agent processor

Spontaneous rewrites As explained in section 3.4.6 there is a problem with spontaneous rewrites. We described a solution for the general case, but with the COBWEB specification we find that there is an easier solution. The two rules of interest are **[ou1]** and **[ou2]**. The presence of these rules means that we need to make a check continuously to see if the conditions for applying them exist as in section 3.4.6. This might seem a large overhead, but in practice we narrow down the number of checks by observing that we only need to apply these rules when we need a particular section of the graph. Of course there is no point checking if a spontaneous rewrite might apply if

that part of the graph is no longer going to be used, so the check is made only when it is known that the packet is known to be needed for evaluation.

The top level is defined as follows:

```
def Agent-processor(self) =
        if message = wakeup then
                agent-wakeup(self)
        else if message = reduce(p) then
                agent-reduce(self,p)
        else    ERROR
        fi
enddef
```

Wakeup can be defined easily.

```
def agent-wakeup(self)
        agent := H(P,A) ! read(self);
        if agent.count ≥ 2 then
                agent.count := agent.count - 1
        else
                T(I,P,A) ! add(self, reduce, self.packet)
        fi;
        H(P,A) ! write(self,agent)
enddef
```

Reduce. Defining the procedure for reduce is more difficult. The first attempt at producing methods for this message runs into some difficulty. This is because some of the rules are defined in terms of a variable number of packets, ie some of the operators have a variable *reach*. This is a problem because it implies that we never know an upper bound on how far to look down the spine of the graph.

We need to backtrack into our specification somehow in order to simplify things so that this situation does not occur. The design can proceed by transforming some of the high level rules in the paragon specification into simpler rules. This will yield a set of rules that will be closer to how the machine will be implemented. There is a need to show that the semantics of the original rule is preserved.

Reducing arities

It is the set of rules that are defined in terms of a variable number of packets on the lhs that are particularly difficult to implement, so the first design

decision we make is to specify the machine so that it never needs to look at more than two packets in any one step before applying a rule. That is we make all operators *short reach.* This implies that all the primitives are defined as having an arity less than or equal to two.

For example, the select K-n-i primitive can be defined as an arity two primitive as follows using the TRS:

$$\begin{array}{lcl} \text{K-1-0 } a & \rightarrow & a \\ \text{K-}n\text{-0 } a\ x & \rightarrow & \text{K-}(n-1)\text{-0 } a \\ \text{K-}n\text{-}i\ a\ x & \rightarrow & \text{K-}(n-1)\text{-}(i-1)\ x \end{array}$$

It is a simple matter to prove this is equivalent to the original specification for K.

The new Paragon specification for these is shown in appendix section B.2. Note that these transformations do not change any of the system equations we derived earlier, so we do not have to backtrack our design all the way to the beginning.

The same problem exists with several of the primitives. IF takes three arguments. The rule can be rewritten as follows in terms of K-n-i:

$$\begin{array}{lcl} \text{IF } \bot\ a\ b & \rightarrow & \bot \\ \text{IF TRUE } a\ b & \rightarrow & \text{K-2-0 } a\ b \\ \text{IF FALSE } a\ b & \rightarrow & \text{K-2-1 } a\ b \end{array}$$

In addition, as the above forms of K-n-i will be frequently used, we can provide two specialised rules which are optimised for these forms:

$$\begin{array}{lcl} \text{K-2-1 } a\ b & \rightarrow & b \\ \text{K-2-0 } a\ b & \rightarrow & a \end{array}$$

The data constructor/selector primitives PACK and CASE in their general forms take a variable number of arguments. Recall the definition of PACK

$$\begin{array}{lcl} \text{PACK-}n\ d & \rightarrow & \text{PACK-}n\text{-}d \\ \text{PACK-}n\text{-}d\ x_0 \ldots x_{n-1} & \rightarrow & \langle d|x_0, \ldots, x_{n-1}\rangle \end{array}$$

We can transform PACK into an arity two primitive by introducing the new primitive STRUCT-n-i. The new rules for PACK are:

$$\begin{array}{lcl} \text{PACK-}n\ d & \rightarrow & \text{PACK-}n\text{-}d \\ \text{PACK-}n\text{-}d\ a & \rightarrow & \text{STRUCT-}n\text{-0 } \langle d|a\rangle \\ \text{STRUCT-}n\text{-}(n-1)\ \langle d|x_0,\ldots x_{n-1}\rangle\ a & \rightarrow & \langle d|x_0, \ldots, x_{n-1}, a\rangle \\ \text{STRUCT-}n\text{-}i\ \langle d|x\rangle\ a & \rightarrow & \text{STRUCT-}n\text{-}(i+1)\ \langle d|x,\ldots,x_n, a\rangle \end{array}$$

STRUCT-n-i is a primitive operator that takes two arguments. The first is a structure type with a tag and a body. The second argument is an element to be inserted into the structure. STRUCT-n-i is indexed with n and i where n represents the number of elements expected in the structure, i represents the number of elements that the structure so far contains. If $n = i$ then the structure is full. Again it is a simple matter to prove that this is equivalent to the original definition.

Recall the definition of CASE:

$$\text{CASE-}r\ e_0 \ldots e_{r-1}\ \langle d|x\rangle \quad \rightarrow \quad e_d$$

For COBWEB we need to use an intermediate, DCASE-r.

$$\begin{array}{lcl}
\text{CASE-}r\ a & \rightarrow & \text{DCASE-}r\ \langle _|a\rangle \\
\text{DCASE-0}\ \langle _|x\rangle\ \langle d|y\rangle & \rightarrow & x_d \\
\text{DCASE-}n\ \langle _|x_0,\ldots,x_k\rangle\ a & \rightarrow & \text{DCASE-}(n-1)\ \langle _|x_0,\ldots,x_k,a\rangle
\end{array}$$

That is, CASE-r uses DCASE-n to package up its r arguments in a structure. When all have been packaged, DCASE-0 indexes into this structure using the tag from its second argument which will also be a structure. The tag of the first structure is unused.

Importantly, we observe that the transformation on the methods for the reduce method does not invalidate any of the procedures that we have already defined. We can now continue with a definition for the reduce procedure.

Reduce As this is by far the most complicated of the procedures, figure 3.10 is a structured description of the operation of the procedure. The agent operates on a packet. The square brackets enclose the Paragon rule from the transformed specification that may apply. This code references additional procedures, for example execute-basic-operator which will include the methods for monadic operators and rewriting to normal form.

```
def agent-reduce(self,packet) =
        [ol]; [ou1]; [ou2];
        if annotation starts with # then
                [#]
        else    case packet.rator of
                        Y : [oY]
                        OPERATOR:
                                execute basic operator
                        PRIMITIVE:
                                execute primitive
                        CONSTANT:
                                [onf]
                        PACKET:
                                [ou1]; [onf];
                                if not in nf then
                                        [ot]
                                else
                                        if there are directors then
                                                [directors]
                                        else
                                                [dyop] or [2ary primitives]
                                        fi
                                fi
                        otherwise: error
                endcase
        fi
enddef
```

Figure 3.10: The pseudocode for the method reduce

3.6 Results of Implementation

A working simulation of the COBWEB abstract machine based on the design of section 3.5 has been developed. This has enabled us to validate that the design works, and predict the performance of a single processor COBWEB. This section reports some results of running small programs on the simulator.

For example, consider the following small program in Miranda:

```
f x y  = (x * y) - (x + y)
—
? f 10 20
```

This compiles into the following director code:

```
f:          [#^^]       f_2         f_5
f_2:        [#\\]       INT-        f_3
f_3:        [#/\]       f_4         I
f_4:        [#\]        INT*        I
f_5:        [#/\]       f_6         I
f_6:        [#\]        INT+        I
MAIN:       []          MAIN_2      20
MAIN_2:     []          P           MAIN_3
MAIN_3:     []          MAIN_4      10
MAIN_4:     []          P           f
MAIN ?
```

When we run this on our machine, forty-eight Paragon rules from the specification of appendix B.2 are applied to evaluate the expression. Figure 3.11 shows a graph to indicate the relative frequencies of rules applied, and the number of the step at which each was applied.

In addition, we can trace the number of reducible expressions as the program runs. Figure 3.12 is a diagram of this activity. The horizontal axis represents time in terms of reductions performed, and the vertical represents the total number of reducible expressions in the task pool at that time.

Of course the trace is simple for this program, and the number of redexes never exceeds two. The number of redexes increases when a fire or need message is sent to a packet. Note that this happens when the # annotation is executed at steps five and sixteen; and when a strict basic operator is executed at step twenty-three. The task pool shrinks when an packet that has been the target of a fire reaches normal form.

The standard functional programming benchmark is nfib. In Miranda this is defined as follows

[#]	:[5,6,15,16,36,37]
[dyop1]	:[44,45,48]
[dyop4]	:[23]
[o/1]	:[26,27]
[oP]	:[1,3]
[o\1]	:[19,21]
[o\2]	:[30,31,40,41]
[o∧1]	:[9,11]
[onf]	:[7,8,10,17,18,20,22,28,29,38,39,42,43]
[ot]	:[2,4,12,13,14,24,25,32,33,34,35]
[ou1]	:[47]
[ou2]	:[46]

Figure 3.11: Relative frequencies of rules applied to the graph

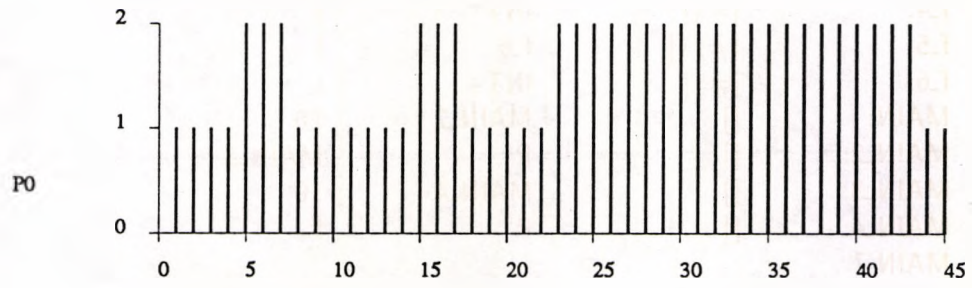


Figure 3.12: A trace of reducible expressions

```
nfib n  = 1, n < 2
        = 1 + (nfib (n−1)) + (nfib (n−2)), otherwise
```

The special property of nfib is that it delivers the number of function instantiations it has evaluated in calculating the result. In addition to the code for the function, is an application of the function to an argument.

For example the execution of nfib to the argument 4 has a rule frequency table as shown in figure 3.13. The execution profile for this program is shown in figure 3.14.

```
#           :[20,21,30,41,44,46,52,55,60,62,114,115,141,144,153,156,163,296,299,305,308,313]
IF0         :[36,129,131,273,275,277,279,374,376]
IF1         :[26,96,97,220,221,222,223,347,348]
K-2-0       :[280,281,282,377,378]
K-2-1       :[37,132,133,283]
PACK-n-i    :[8,75,183,326]
SEL-n-i     :[4,10,70,77,176,185,321,328]
UNPACK-n    :[1,6,73,181,324],
dyop1       :[34,122,123,125,127,260,261,262,263,265,267,269,271,287,293,367,368,370,372,381,
             385,387,390,392,395]
dyop2       :[106,107,240,241,242,243,357,358]
dyop3       :[67,171,173,318]
dyop4       :[61,164,165,314]
o/1         :[17,27,88,89,98,99,108,109,204,205,206,207,224,225,226,227,244,245,246,247,339,340,349,
             350,359,360]
o/2         :[2,42,57,142,159,297,310]
oY          :[5,72,179,323]
o\1         :[7,13,24,51,53,56,64,65,69,74,82,92,93,152,154,157,158,167,168,169,170,
             175,177,182,194,212,213,214,215,304,306,309,316,317,320,325,333,343,344]
o\2         :[32,102,103,118,119,232,233,234,235,252,253,254,255,353,354,363,364]
o^1         :[15,40,45,84,85,140,145,146,196,197,198,199,295,300,335,336]
onf         :[14,22,23,25,31,33,43,47,49,54,58,59,63,66,83,94,95,104,105,116,117,120,121,143,147,155,
             160,161,162,166,195,216,217,218,219,236,237,238,239,256,257,258,259,284,285,288,298,301,
             307,311,312,315,334,345,346,355,356,365,366,379,382]
ot          :[3,12,16,18,19,28,29,39,48,50,68,71,80,81,86,87,90,91,100,101,110,111,112,113,138,139,148,
             149,150,151,172,174,178,180,190,191,192,193,200,201,202,203,208,209,210,211,228,229,230,
             231,248,249,250,251,294,302,303,319,322,331,332,337,338,341,342,351,352,361,362]
ou1         :[11,38,78,79,135,137,186,187,188,189,290,329,330]
ou2         :[9,35,76,124,126,128,130,134,136,184,264,266,268,270,272,274,276,278,286,289,291,292,327,
             369,371,373,375,380,383,384,386,388,389,391,393,394]
```

Figure 3.13: Rule frequencies for nfib 4

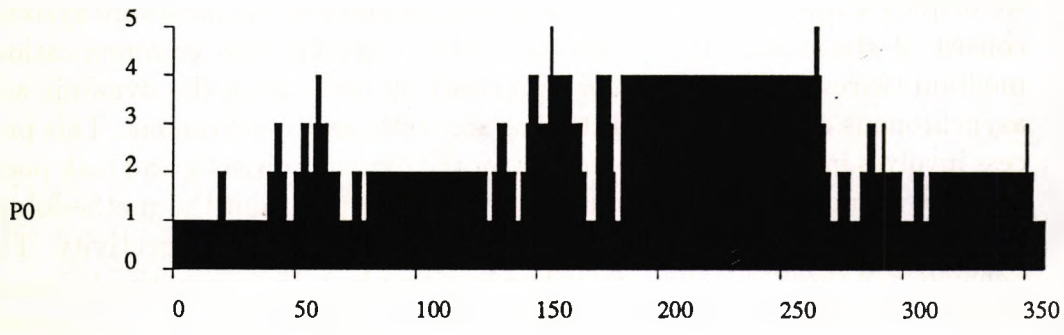


Figure 3.14: Profile of nfib 4

3.7 Summary

In this chapter we introduced and defined COBWEB — a computer architecture for the execution of functional languages.

We began by identifying the various techniques for the implementation of functional languages. For historical reasons, we concentrated on one of these techniques — graph reduction. We reviewed contemporary parallel graph reduction architectures.

We then described the operation of COBWEB. COBWEB is a machine that executes programs in the form of director graphs. The director graph is generated from FLIC which has been generated from programs written in Hope$^+$. The Hope$^+$ to FLIC compiler performs strictness analysis and produces annotations on the FLIC output in the form of evaluators and evaluation transformers. The FLIC to COBWEB compiler translates these annotations into parallelism operators and directors.

We identified the need to specify computer architectures formally. COBWEB can be defined formally using a term rewrite system. We saw that this method did not allow us to express some of the lower level details of how we wish the machine to operate. We introduced Paragon as an object based term rewriting system with message passing to specify COBWEB at a lower level. In the course of specifying COBWEB we found that the language in its original form was not suitable for specifying some aspects of the machine so we introduced some minor extensions. Paragon is found to be an expressive hardware specification language. Unlike other HDLs it can describe dynamic systems cooperating using asynchronous message passing. We found that this is precisely the level that is appropriate for the high level specifation of graph reduction architectures.

Given that we wish to produce hardware designs from our specification we defined a route from Paragon to hardware designs. As hardware systems consist of static logic blocks connected by a synchronous communications medium (wires), the methodology proceeds by translating the dynamic and asynchronous objects into objects that are static and synchronous. This process involves introducing heap storage for the dynamic objects, and task pools to allow asynchronous communications. The output from the methodology consists of a number of objects and a description of their connectivity. The behaviour of these objects is described in a simple static imperative language similar to a conventional HDL. The connectivity of the blocks is expressed as a directed graph where the arcs are synchronous connections, and the direction of the arcs indicate the direction of message flow. Applied to a generalised Paragon description, the methodology produces a description of a system that closely resembles a multi tasking von Neumann uniprocessor. However, we noted that static analysis techniques can be used to produce better designs for less general specifications.

We applied the methodology to our specification of COBWEB and found that the resulting design was inefficient in some respects due to the long reach of some of the built in operators. We transformed our original specification to reduce the reach of all operators and applied the methodology again. The new design was satisfactory so we used the design to produce a prototype in the form of a simulator.

We have compiled several programs written in Hope$^+$ into COBWEB code and have executed them on the simulator. We have shown the results as profiles of operations on the program graph, and as redexes available for execution.

Chapter 4

A Parallel WSI Cobweb

In chapter 2 we introduced a communications architecture for WSI. In chapter 3 we introduced a graph reduction architecture. In this chapter we bring these threads together by designing a parallel graph reduction architecture for WSI.

We proceed as follows. In section 4.1 we show how we can construct a model for measuring the performance of a fairly general purpose loosely coupled multiprocessor. In section 4.2 we expand our specification of chapter 3 to include support for multiprocessors. We apply the design methodology of the previous chapter to this specification and produce a simulator as a prototype. In section 4.3 we present the results of this simulation. Finally in section 4.4 we bring the results together to produce estimates of the performance of a parallel graph reduction architecture for WSI.

4.1 A performance model for WSI multiprocessors

In [AKW90] we introduce a performance model for loosely coupled WSI multiprocessors. We measure the performance as the total number of memory accesses per second. The machine is modelled as a memory hierarchy. Each processor has some local memory, and is connected to every other processor in the system by a network. A processor can access the memory associated with another processor by communicating a message to it, and waiting for a reply.

Programs in this system are modelled by a single parameter, M, which represents the percentage of non-local memory accesses. Two schemes are modelled, characterised by whether their memory accesses are synchronous or asynchronous. In the suspension scheme, the processor must wait idly until the result of an access is returned. With the multiplexing scheme, the processor schedules another program for running while it waits for the result

to be returned.

The Performance Model

We model the performance in terms of the following parameters.

t_l local memory access time.

t_{hop} time taken to transfer one packet between CEs.

N the number of working PEs.

L the loading factor, that is the percentage of working CEs that have a working PE.

$C(x)$ is the congestion function defined in terms of the number of messages in the network per CE. This measures the ratio of hops taken to the ideal minimum number of hops. This can be read straight from the performance graphs in chapter 2.

p is the average path length. Again, this can be read from the graphs of chapter 2.

ϕ is the traffic level — the total number of messages in the network at once. The number of packets per PE is ϕ/N, the number per CE is $L\phi/N$.

M is the miss rate — the number of memory accesses that are non local.

The PE is modelled by τ, the time taken to access non-local memory. It is the sum of three components:

t_{tx} is the time taken to transmit

t_{rx} is the time taken to receive a message

t_{cs} is the time taken to restart the process upon receipt of a message.

The reply will become available after a latency. The average latency T can be calculated in terms of the above as follows.

$$T = 2pt_{hop}C(L\phi/N)$$

This assumes a random non-local target.

The total average non-local memory access time $t_g$is thus given

$$\begin{aligned} t_g &= T + \tau \\ &= 2pt_{hop}C(L\phi/N) + t_{tx} + t_{rx} \end{aligned}$$

We model two schemes:

The suspension scheme. Under this scheme, all memory accesses are synchronous. When a process attempts a non-local memory access, the CPU must wait until the value has been returned. This means that the maximum number of packets in the network will never exceed the number of working processors, ie $\phi \leq N$. The performance in terms of the number of memory accesses per second is given by

$$r_M = \frac{N}{Mt_g + (1-M)t_l}$$

The multiplexing scheme. Under the multiplexing scheme we have a number of processes per PE. When one process requests a non-local read, it is stopped, and the CPU can schedule another to run until the result comes back. The non-local access delay is now increased by the context switch time t_{cs}. We have a number of processes ready to run scheduled in a cycle. This can increase congestion because each process may have a message pending in the network, so we put an upper limit V on the number of messages a PE can have current in the network. We can think of each process in the cycle as a loop making $1/M$ local memory accesses followed by one non-local access. If there are enough processors in the cycle then the PE need never be idly waiting for a message to return. This is the *latency concealment condition*, and it can be expressed

$$V(\tau + \frac{1}{M}t_l) > T$$

To increase latency tolerance, we increase V, but this also increases the congestion in the network, and we may lose any benefit. If we can satisfy the condition, then the overall performance is independent of T, so we have

$$t_g = \tau = t_{tx} + t_{rx} + t_{cs}$$

and the overall performance is given by

$$r_M = \frac{N}{M\tau + (1-M)t_l}$$

Assuming latency concealment, then the multiplexing scheme wins over the suspension scheme when

$$\frac{N}{M(t_{tx} + t_{rx} + t_{cs}) + (1-M)t_l} > \frac{N}{M(2pt_{hop}C(L) + t_{tx} + t_{rx}) + (1-M)t_l}$$

or,

$$t_{cs} < 2pt_{hop}C(L)$$

ie when the context switch time is less than the average round trip time.

Some real numbers The results from the performance model when some numbers are fed in are reported in [AKW90]. These are worth repeating here. One figure that should be explained is the yield of the PE. We assume a fairly small (eg Transputer sized without the floating point) PE with some memory. The memory is arranged in blocks of 512 bytes. For a PE to work, we need the first of these blocks to work. We can use all of the other blocks that we can yield. The maximum amount of memory available to one PE is 25 kilobytes.

$t_{\mathrm{l}} = 70$ ns

$t_{\mathrm{tx}} = 70$ ns

$t_{\mathrm{rx}} = 70$ ns

$t_{\mathrm{cs}} = 420$ ns

$t_{\mathrm{hop}} = 100$ ns

$p = 6$ hops. This is read from the graph for the average latency for the $\mathcal{R}_{\mathrm{Path}}$ in chapter 2

$L = Y_{\mathrm{PE}} = 62.7\%$

$N = Y_{\mathrm{PE}} \times Y_{\mathrm{CE}} \times N_{fab} \approx 47$ PEs.

$M = 2\%$

For the suspension scheme, the result is approximately 370 million memory accesses per second. For the multiplexing scheme, as latency concealment is easily satisfied, the result is approximately 590 million memory accesses per second.

The only problem is the total amount of memory available — approximately 1.5 megabytes per wafer. As explained in [AKW90] this could be improved by use of a high density custom memory process.

4.2 Specification of a multiprocessor Cobweb

The conclusion of the previous section is that multitasking within processors is essential. The specification of the previous chapter does not consider multitasking because each agent needs synchronous access to the global heap. If we are to do multitasking then we need to rewrite the specification. It is not enough to just replicate this design a number of times.

We proceed by explicitly describing the role of processors as localities where objects are processed. The key point is that every object in the system is to be "associated" with a processor. Although the specification does not need to say this explicitly, this effectively means that each object is stored in a heap local to a particular processor. The program graph is now assumed to be distributed throughout the machine, with each packet residing in the heap associated with a particular processor. Similarly agents will be associated with particular processors.

We then need to specify how we deal with agents that need to access packets that are not on the same processor. We do this using a form of remote copy. When an agent decides it needs to know about a packet that is on a remote processor, then it sends a message to that remote processor asking it for the values it needs. While it waits for the result, it goes to sleep. In the interim another agent can be scheduled for execution by the processor. Access to remote packets is now asynchronous, and we thus have achieved multitasking.

Now that we have a collection of processors, we can arrange to have a number of them evaluating different parts of the graph concurrently. We do this by transmitting need and fire messages across the network, and by exporting work to processors when necessary. Note that no load balancing is attempted, just load distribution.

We specify this by labelling each packet and agent with the processor identifier on which it resides.

We shall now discuss the new specification in terms of the new classes and modified methods.

4.2.1 The new classes

We introduce a new class to deal with "remote" packets. The previous packet definition will now be known as a local packet. A remote packet is identified by the processor it resides in plus the local packet identifier. Any packets' left and/or right subpackets can be situated on remote processors. We have our new class definitions as follows:

```
class rpacket ::= ⟨processor,packet⟩
class packet ::= ⟨rator,rand,ann,activity,innf,list agent⟩
data rator ::= basic-value | rpacket
data rand ::= basic-value | rpacket
```

We must also distinguish local agents from remote agents. We do this by modifying our agent class so that it includes the name of the processor on which the agent resides.

class agent ::= ⟨processor,rpacket,integer⟩

The processors themselves are actually stateless.

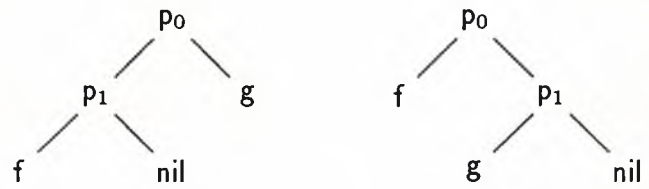
class processor ::= ⟨⟩

We shall consider the methods for each class in turn.

4.2.2 Packets

Spontaneous transformations

There is a problem with spontaneous transformations on packets. For example, consider the following two graph fragments to which a spontaneous transformation applies.



In both cases the graph rewrites to f applied to g using a spontaneous rule. However consider the case when p_1 is not on the same processor as p_0. The rewrite can only be applied when the body of p_1 is known. We have the situation where a spontaneous rewrite implies knowing the body of a non-local packet.

We solve this problem by asking the processor associated with p_1 to say whether the spontaneous rewrite applies. If it does, then the remote processor can return the value of the appropriate packet. The rules for the first spontaneous rewrite are as follows.

[spontaneous-local]
⟨x,lp⟩ when
lp = ⟨p_0,p_1,ann,flag,_,_⟩ ∧
p_0 = ⟨x,⟨f,nil,_,_,_,_⟩⟩
→ ⟨x,⟨f,p_1,ann,flag,_,_⟩⟩

[spontaneous-remote]

```
⟨x,lp⟩  when
        lp = ⟨p0,p1,_,_,_,_⟩ ∧
        p0 = ⟨y,rp⟩ ∧
        x ≠ y
        → ⟨x,lp⟩
            then
            y !! p-spontaneous(self,rp)
```

The first rule applies when a spontaneous rewrite applies to a packet that is known locally. It is rewritten to eliminate the nil. The second applies when the packet has a rator that is not local. The rule might apply, so a message is send to the processor on which it resides. See section 4.2.4 for the specification for p-spontaneous.

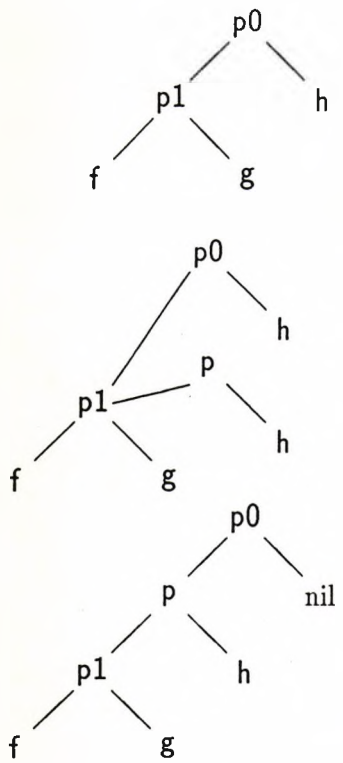
Remote Packets

We first deal with remote packets. These are a tuple consisting of the processor name, and the packet body. Remote packets respond to two messages: *need* and *fire*. When a packet is needed or fired, and it is not active or in normal form, we create an agent to reduce it. Until we actually create the agent, we have the freedom to choose where it will be situated. It is here that we deal with load distribution, by asking a different processor to reduce the packet. There is little point exporting the agent to another processor if everything necessary for the agent to perform a reduction is local to the current processor, so we only export an agent when we know that some of the information needed is definitely on a remote processor.

For example consider the situation where the expression f g h receives a fire or need message. The graph is shown in the first part of figure 4.1 Packet p0 is local to processor 0, and packet p1 is local to processor 1. We do this in three steps. The first step is to create an agent newagent to be responsible for reducing this packet, and rewrite packet p0 so that it is marked as being active. In addition if the message received is need then the agent argument is appended to p0's pending list.

The second step is to send a message p-balance to the processor on which p1 resides. This message takes four arguments. The first three arguments are the operator, operand and the annotations of p0. The fourth argument is newagent. When this has been received by processor 1, the processor creates a new packet p identical in contents to p0, but residing on a different processor. The pending list of p is set to contain newagent. An agent then is created to reduce the packet p.

The third step is to inform p0, which is now semantically equivalent to p that it is being evaluated elsewhere. The processor sends a message p-return-balance with arguments newagent and p to the original processor. Upon



Packet p0 on processor 0 receives a fire message. To be reduced it needs p1 which is on processor 1. It is marked as active and balance(p1,h,ann,newagent) is sent to processor 1

Processor 1 receives the balance message, and creates p and an agent to reduce p. p0 is unchanged.

Processor 0 receives the return-balance message and rewrites p0

Figure 4.1: The three stages involved in exporting agents to remote processors.

receipt of this message, packet p0 is rewritten so that it has operator p, and operand and annotations that are nil.

The Paragon for the fire message is as follows: The first two rules are largely unchanged.

```
⟨_,lp⟩   given fire
         when
         lp = ⟨_,_,_,_,Active,q⟩
            → self

⟨_,lp⟩   given fire
         when
         lp = ⟨_,_,_,Nf,Inactive,q⟩
         → self
```

Rule [**remote-fire**] sends a message to a remote processor in the manner described above.

```
[local-fire]
⟨r,lp⟩   given fire
         when
         lp = ⟨rator,rand,annot,Notnf,Inactive,q⟩ ∧
         (unary(self) ∨ (¬ unary(self) ∧ rator = ⟨r,_⟩))
         → ⟨r,⟨rator,rand,annot,Notnf,Active,q⟩⟩
             then
             newag !! reduce(self)
             where
             newag = new(agent,⟨r,self,0⟩)

[remote-fire]
⟨r,lp⟩   given fire
         when
         lp = ⟨rator,rand,annot,Notnf,Inactive,q⟩ ∧
         rator = ⟨r',_⟩ ∧ r' ≠ r ∧
         ¬ unary(self)
         → ⟨r,⟨rator,rand,annot,Notnf,Active,q⟩⟩
             then
             r' !! p-balance(rator,rand,annot,newagent)
             where
             newagent = new(agent,⟨r,self,1⟩)
```

The Paragon for need is similar. The only difference is that the needing agent must be added to the packets pending queue.

```
⟨r,lp⟩   given need(agent)
         when
         lp = ⟨rator,rand,annot,isnf,Active,q⟩
         → ⟨r,np⟩
             where
             np = ⟨rator,rand,annot,isnf,Active,agent::q⟩

⟨r,lp⟩   given need(agent)
         when
         lp = ⟨_,_,_Nf,Inactive,_⟩
         → self
             then
             r !! p-wakeup(agent)

[local-need]
```

```
⟨r,lp⟩   given need(agent)
         when
         lp = ⟨rator,rand,annot,Notnf,Inactive,q⟩ ∧
         (unary(self) ∨ (¬ unary(self) ∧ rator = ⟨r,_⟩))
         → ⟨r,⟨rator,rand,annot,Notnf,Active,agent::q⟩⟩
             then
             newag !! reduce(self)
             where
             newag = new(agent,⟨r,self,0⟩)

[remote-need]
⟨r,lp⟩   given need(agent)
         when
         lp = ⟨rator,rand,annot,Notnf,Inactive,q⟩ and
         rator = ⟨r',_⟩ ∧ r ≠ r' and
         unary(self)
         → ⟨rator,rand,annot,Notnf,Active,agent::q⟩
             then
             r' !! p-balance(rator,rand,annot,newagent)
             where
             newagent = new(agent,⟨r,self,1⟩)
```

There might be other reasons for exporting agents to remote processors. For example each processor could maintain a map of which of its neighbours are idle, and export the agent to them. However these are not considered here.

Local packets

We only provide one method for local packets. This is rewrite. We always know that all rewrites take place on the processor on which the packet resides. Therefore we can keep the method almost exactly the same as before. The only complication is that we need to send p-wakeup messages to all the agents waiting on the packet via their processors.

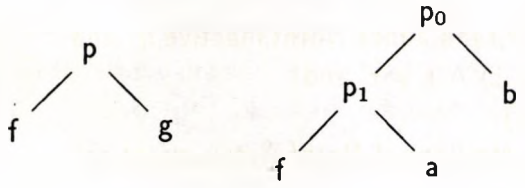
```
⟨_,_,_,_,act,q⟩
         given rewrite(op,arg,ann,Notnf)
         → ⟨op,arg,ann,Notnf,act,q⟩

⟨_,_,_,_,act,as⟩
         given rewrite(op,arg,ann,Nf)
```

→ ⟨op,arg,ann,Notnf,act,nil⟩
then
q !! p-wakeup

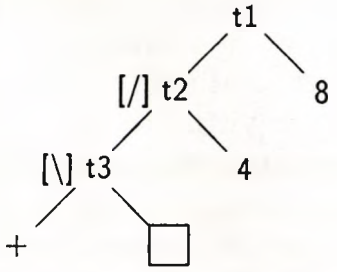
4.2.3 Agents

The method for the reduce message is most affected by the changes we have made. However due to our transformations of section 3.5, we know that every rewrite of the graph in response to the reduce message is one of two forms, a unary rewrite or a binary rewrite as shown below.



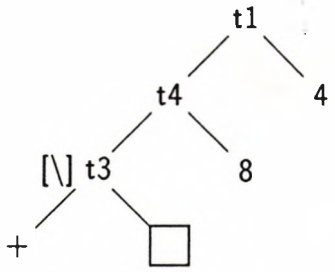
A unary rewrite can always be executed immediately because all the information is available to the agent. Also, the agent can always tell if a unary rewrite applies. However if a binary rewrite applies, and if the p_1 is associated with a different processor than the agent, then the agent must get the body of p_1 before it can decide which rule applies, and before it can continue.

Before giving the full specification, a short example is in order. Consider a simple program as follows:



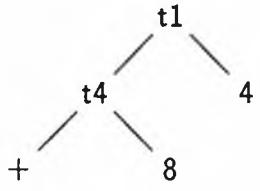
Say that we have two processors numbered zero and one, and that packets t1 and t2 are associated with processor 0, and that t3 is associated with processor 1.

To execute the program we create an agent on processor 0 to reduce packet t1. We need to know which rule applies, and to do this we need to look at the left subgraph of t1. That is, we need to look at the annotations and the left and right subgraphs of t2. We can do this easily because t2 is associated with the same processor as t1 and the agent. A rule for executing the / director applies, and the graph rewrites to the following form:



That is, we create a new packet called t4 which is associated with processor 0, and rewrite t1.

The next rule to apply is to t4. However in order to know which rule applies, we need to know about packet t3. As this is associated with another processor, we cannot look at its body directly, but must ask that remote processor to look at it. So we send a message to processor 1 to ask it to send back the body of packet t3. The agent suspends while the body is returned, and when the body does arrive back, the correct rule can be chosen. The graph is transformed into the following:



and finally the plus operator is executed and t1 rewritten to 12.

By asking the processors to handle delivering messages to agents and packets, and to handle the copying of bodies of packets, we enforce a locality of access. That is, an agent associated with one processor cannot directly look at the body of a packet that is associated on another processor, therefore the embodiment of the agent need not be directly connected to the packet heap of the remote processor.

This mechanism is specified in Paragon as follows. These first two rules specify when the reduction can be done locally. A reduction can be performed locally when a unary rule applies, or when a binary rule applies and the appropriate packet is local.

```
[reduce-unary]
⟨r,_,_⟩  given reduce(p)
         when
         unary(p)
         → ⟨r,p,0⟩
```

```
        then
        self! reduce_unary(p)

[binary-local]
⟨r,_,_⟩  given reduce(p)
        when
        ¬ unary(p) ∧
        p = ⟨r,lp⟩ ∧
        lp = ⟨rator,rand,annot,Nf,_,_⟩
        → ⟨r,p,0⟩
            then
            self ! reduce-binary(p,rator,rand,annot,Nf)
```

When the packet is not local, we send a message to its processor.

```
[binary-remote]
⟨r,_,_⟩  given reduce(p)
        when
        ¬ unary(p) ∧
        p = ⟨r,lp⟩ ∧
        lp = ⟨r',_⟩ ∧r ≠ r'
        → ⟨r,p,1⟩
            then
            r' !! p-getbody(p,self)
```

The remote processor will return the body of the packet to the agent via the local processor. The Paragon for this is as follows:

```
[return-body]
⟨r,p,1⟩  given return-body(rator,rand,annot,Nf)
        → ⟨r,p,0⟩
            then
            self ! reduce-binary(p,rator,rand,annot,Nf)
```

As the agent requesting the remote copy has been put to sleep, when the body of the requested returns, it must be woken, and the reduction process begun.

The reduce-binary message and reduce-unary are introduced above. These effectively do the job of the reduce message in the specification of chapter 3.5. For example the paragon for two of these messages are given below.

```
[Y]
⟨r,_,0⟩  given reduce_unary(p)
         when
         p = ⟨_,lp⟩ ∧
         lp = ⟨Y,f,nil,_,_,_⟩
         → ⟨r,p,0⟩
             then
             lp ! rewrite(f,p,nil,Notnf) ;
             self !! reduce(p)

[\1]
⟨r,_,0⟩  given reduce-binary(p,rator,rand,annot,isnf)
         when
         p = ⟨r,lp⟩ ∧
         annot = nil ∧
         rator = ⟨op,arg1,\::ann,_⟩
         → ⟨r,p,0⟩
             then
             p ! rewrite(op,newp,ann,Notnf) ;
             self !! reduce(p)
             where
             newp = new(packet,⟨rand,arg1,nil,Notnf,Inactive,nil⟩)
```

The final method we consider handles the case when a balance message has been replied to by a processor as explained in the previous section.

```
[return-balance]
⟨r,p,1⟩  given return-balance(p)
         when
         p = ⟨r,lp⟩ ∧
         lp = ⟨_,_,_,_,Active,q⟩
         → ⟨r,p,0⟩
             then
             lp ! rewrite(p,nil,nil,Notnf,Active,q)
```

Another rule not shown here simply ignores the message if the above rule does not match.

4.2.4 The Processor Class

Processors act as a collators of requests that come in from remote processors to operate on packets or agents. They also deal with requests for remote copies, requests for spontaneous rewrites, and requests for the export of work.

Most of the methods are fairly straightforward, and simply embody the purpose of the processor as a global collator of requests. Note that the state of the processor never changes, but that we need to know its name. The following three rules simply forward the appropriate message to the object which resides in the processor.

```
[p-reduce]
-       given p-reduce(p)
        → self
          then
          ag ! reduce(p)
          where
          ag = new(agent,⟨self,p,0⟩)

[p-need]
-       given p-need(ag,rp)
        → self
          then
          rp ! need(ag)

[p-fire]
-       given p-fire(rp)
        → self
          then
          rp ! fire
```

The next two rules implement the remote copy. The first deals with a request for a copy of a local packet. The agent that requested the copy is sent copies of the instance variables of the packet in question via the processor on which that agent resides.

The second rule deals with the copies being received by a processor. The agent that requested the copy is woken and sent the binary-reduce message.

```
[p-getbody]
-       given p-getbody(lp,ag)
```

```
        where
        lp = ⟨rator,rand,annot,Nf,_,_⟩
        → self
            then
            r !! p-return-body(rator,rand,annot,Nf,ag)
            where
            ag = ⟨r,_,_⟩

[p-return-body]
-       given p-return-body(rator,rand,annot,Nf,ag)
        → self
            then
            ag !! return-body(rator,rand,annot,Nf)
```

The management of agent exporting is handled by the following messages. When a balance message is received, the processor creates a packet, and an agent to reduce the packet.

```
[p-balance]
-       given p-balance(f,g,annot,agent)
        when
        agent = ⟨r,_⟩
        → self
            then
            r !! p-return-balance(newp,ag) ||
            newag !! reduce(newp)
            where
            newp = new(Packet,⟨self,lp⟩)
            lp = ⟨f,g,annot,Notnf,Inactive,agent⟩
            newag = new(Agent,⟨self,newp,0⟩)

[p-return-balance]
-       given p-return-balance(agent,newp)
        → self
            then
            agent !! return-balance(newp)
```

Finally, to deal with spontaneous rewrites on the graph, we have the following methods. The first clause deals with when the graph is in a state that can be rewritten. The first parameter to the message is the packet (that is associated with processor y) that is to be rewritten. The second

parameter is the name of the local packet whose state is being tested. A rule not shown simply accepts the message without changing anything if the guards are not true. This is provided so that the p-spontaneous message is not kept pending.

```
[p-spontaneous]
-       given p-spontaneous(rp,lp)
        when
        rp = ⟨self,⟨rator,nil,annot,Nf,_,_⟩⟩ ∧
        lp = ⟨r,_⟩
        → self
            then
            r !! p-reply-spontaneous(lp,rator)

[p-reply-spontaneous]
-       given p-reply-spontaneous(rp,new-rator)
        when
        rp = ⟨_,lp⟩ ∧
        lp = ⟨_,rand,_,_,_,_⟩
        → self
            then
            lp ! rewrite(new-rator,rand,annot,Nf);
```

4.3 Design and Simulation

In this section we outline a design that has been produced from the specification in the previous section. The design has been derived by applying the methodology from chapter 3.4 to the specification. We then discuss the metrics needed for measuring the performance of the machine. This design has been prototyped as a simulator written in C, and we then present some of the results of running programs on the simulator.

4.3.1 The design

The design that we produce is a result of applying the method of chapter 3.4 to the specification in the previous section. As the specification has been written with a loosely coupled multiprocessor in mind, we first confirm that the specification matches up to our expectations.

We begin by identifying the classes. These are:

- Processors abbreviated $\mathcal{R}$

- Packets $\mathcal{P}$
- Local packets $\mathcal{L}$
- Agents $\mathcal{A}$

The system equations are as follows:

$$\begin{array}{lll} S(\mathcal{P}) = \{L, P\} & A(\mathcal{P}) = \{A, R\} & D(\mathcal{P}) = \{A\} \\ S(\mathcal{L}) = \{L\} & A(\mathcal{L}) = \{P\} & D(\mathcal{L}) = \emptyset \\ S(\mathcal{A}) = \{A, P, L\} & A(\mathcal{A}) = \{A, P, R\} & D(\mathcal{A}) = \{A, P, L\} \\ S(\mathcal{R}) = \{A, P, L\} & A(\mathcal{R}) = \{P, A\} & D(\mathcal{R}) = \{P, A\} \end{array}$$

The first thing to note is that nothing is dynamic in $\mathcal{R}$, so we can implement this system using a finite number of processors. Although we need heap processors for packets and agents, we can distribute these by observing that the objects only create or change packets and agents if they are on the same processor.

The following is a description of a design for a single processor that satisfies the above equations. In the machine, it will be replicated several times. Each instance will be connected to the communications network.

Our system consists of a number of class processors $P(\mathcal{R})$. As these are stateless, there is no need for $H(\mathcal{R})$. Processor are connected by a network. The methodology states that the communications network connecting the processors must be synchronous, and that it is controlled by a task pool $T(\mathcal{R})$. However we note that $\mathcal{R}$ is not synchronous in $\mathcal{R}$, so it can be asynchronous.

Packets, Local Packets and Agents are stored in $H(\mathcal{P}, \mathcal{L}, \mathcal{A})$ and connected to $P(\mathcal{P}, \mathcal{L}, \mathcal{A})$.

This can be physically implemented as in figure 4.3.

The design is clearly divided into a communications element and a processing element. The Processing element consists of a heap and a processor. The heap embodies $H(\mathcal{A}, \mathcal{P}, \mathcal{L}, T(\mathcal{A}, \mathcal{P}, \mathcal{L}))$, and the processor embodies $P(\mathcal{A}, \mathcal{P}, \mathcal{L}, T(\mathcal{A}, \mathcal{P}, \mathcal{L}))$. The communications element embodies $P(\mathcal{R}, T(\mathcal{R}))$ and $H(T(\mathcal{R}))$.

The simulation

A prototype of this design has been developed in the form of a simulator written in C.

It implements all the defined methods for the classes, with space for agents and packets being allocated from a local heap.

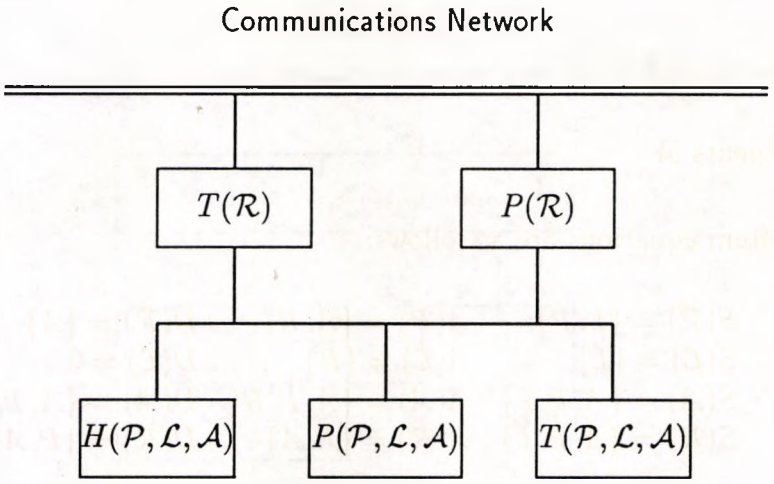


Figure 4.2: A single processor

4.3.2 Assumptions

The primary assumption with the metrics is that one hop equals one reduction step. This includes the time taken to read from the CE buffer, calculate a new direction, and to forward the packet in that new direction.

The second assumption we make is that the garbage collection has little effect on the overall performance. This is confirmed in practice by the GRIP architecture [HP90], where garbage collection is found to take no more than 2% of the total time spent executing programs.

4.3.3 Code distribution

There are many different ways that code may be distributed throughout the machine. [Kel89] introduces the functional language Caliban which allows the programmer to place functions on physical processors taking account of their topology. However, no automatic methods have been implemented for Cobweb. For these experiments, the code has been distributed by hand.

Several experiments were attempted based on the distribution of the code for the function nfib. The simplest is to put all the code on one processor. Because the machine does not attempt to balance the load, this meant that the entire expression is evaluated on one processor.

The next experiment simply distributed the code at random throughout all the processors available.

The third experiment was based on distributing identical copies of the code onto all the processors. For example, nfib can be rewritten as follows.

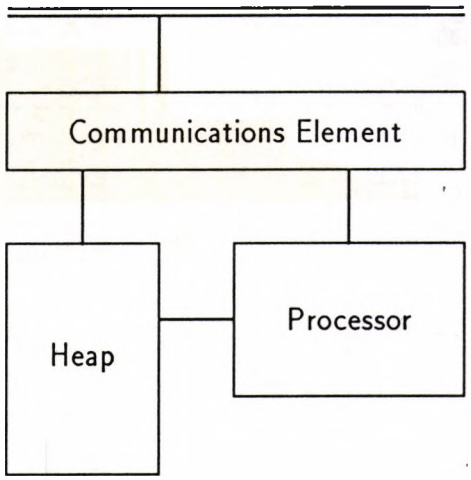


Figure 4.3: A physical implementation of a Cobweb node

```
nfib n  = 1, n < 2
        = 1 + (mfib (n−1)) + (ofib (n−2)), otherwise
mfib n = 1, n < 2
        = 1 + (ofib (n−1)) + (nfib (n−2)), otherwise
ofib n  = 1, n < 2
        = 1 + (nfib (n−1)) + (mfib (n−2)), otherwise
```

For this experiment, with three processors, the code for nfib was placed on processor zero, and the code for mfib and ofib on processors one and two respectively. The application code was placed on processor zero.

4.3.4 Results

The results for the first experiment, where all the code is on processor zero are unsurprisingly identical to the results for the single processor case. The execution profile for nfib 9 is shown in figure 4.4. The profile graph includes one extra feature — the network traffic. This is a measure of how many messages are in transit in the network. Being 'in transit' can include waiting in a CE buffer. The single message shown in figure 4.4 is the message sent to begin the computation.

The results for the random case are interesting. Figure 4.5 shows the execution profile.

The third experiment was when the code for the main function was repli-

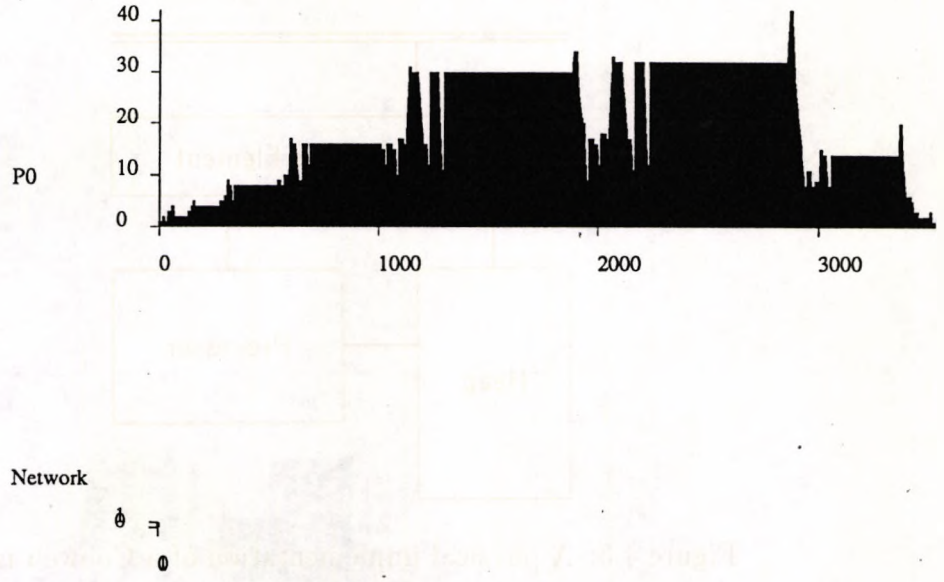


Figure 4.4: Execution profile for when all code is on one processor

cated across three processors, with the application code on processor zero. Figure 4.6 shows the execution profile for this case.

The results for when the code is distributed are disappointing. The above show little speedup in time. This is a result of the large expense associated with spontaneous rewrites. At low values of the argument to nfib, although there are three processors sharing the work, the overhead of managing the sharing causes it to take longer to execute than on one processor.

The following is a table showing speedups for a three processor COBWEB executing nfib n.

n	nfib n	one	rfib	speedup	msfib	speedup
4	9	358	531	0.67	486	0.73
5	15	564	811	0.69	669	0.84
6	25	892	1233	0.72	894	1
7	41	1403	1887	0.74	1214	1.15
8	67	2219	2926	0.76	1756	1.26
9	109	3523	4601	0.77	2352	1.49

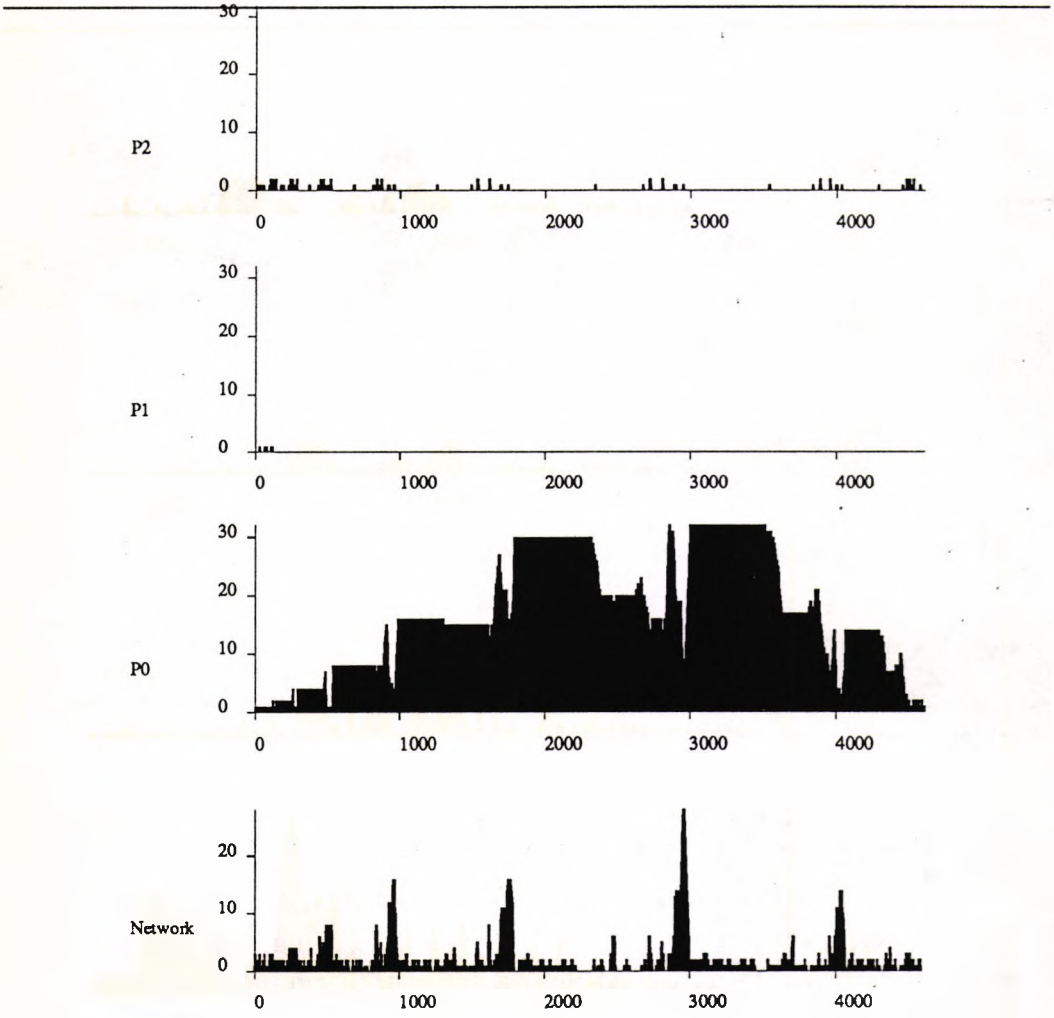


Figure 4.5: Execution profile for when code is randomly distributed

The one column shows the number of steps taken to evaluate the expression on one processor. The rfib column shows the result when the code is randomly distributed. The msfib column shows the result when the code is replicated. Each of the latter two columns have associated speedup columns which show the absolute speedup over the single processor version.

4.4 The performance of COBWEB

Having done the simulation, we can now estimate the performance of a multiprocessor COBWEB. We will look at the results for calculating nfib 9, first for the single processor COBWEB. The number of reduction steps needed to

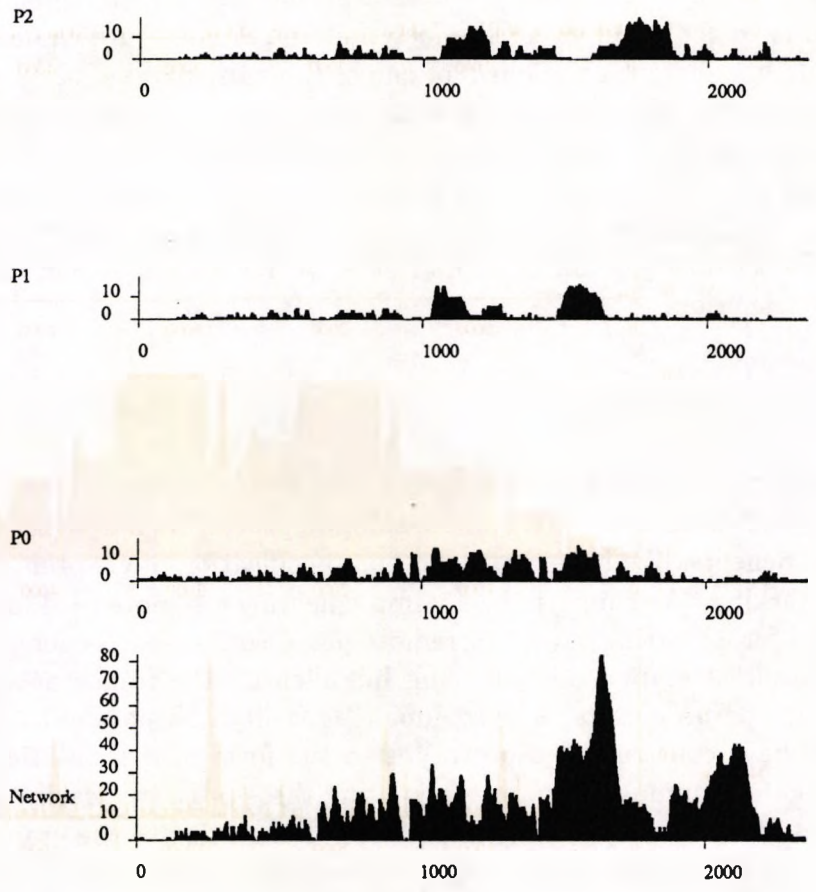


Figure 4.6: Execution profile for when code is replicated

evaluate the expression is 3523. As nfib 9 = 109, this give us approximately 35 steps per function call. This is approximately the number of rewrites for all values of n. A very rough estimate at the number of memory accesses taken to execute the average reduction is 40. At 70ns per memory access we have 10,000 nfibs per second on a one processor COBWEB. As a three processor COBWEB goes 1.5 times faster for this computation, we can say that a three processor COBWEB does 15,000 nfibs per second.

This result compares fairly unfavourably with current graph reduction machines. For example, GRIP delivers a scalable 36,000 nfibs per second per processor [HP90]. We see from the table of results that the parallelism exploited increases as the size of the problem increases. Unfortunately it is not at all clear if the parallelism scales well in COBWEB as the tools used to

generate the results break down at higher levels. However, we can see from this small set of data in conjunction with the network profile, that the level of communications are a problem.

The reason that COBWEB does not perform as well as hoped is undoubtedly due to the grain size. The fact that the architecture cannot efficiently exploit the inherent parallelism of one of the most parallel of all programs is due to the communications overhead associated with the grain. Figure 4.6 shows that huge demands are made of the network despite the problem being fairly small. We chose WSI as a technology mainly because of the very large communications bandwidth. However we see here that despite this, directors are still too fine a grain of computation to be a competitive technique for graph reduction.

4.5 Summary

We have constructed a performance model for multiprocessors and identified that in the context of WSI, multi-tasking in processors has significant performance benefits. We have expanded our specification of chapter 3 to support multi-tasking. We support this by implementing a remote read facility, and a method for exporting agents to remote processors so as to enchance locality. We do not attempt load balancing but identify the importance of placing programs on processors so as to enhance locality and parallelism.

We have constructed a prototype in the form of a simulation and have investigated running programs on it using three program placing techniques. The results are not especially impressive. This is due to the grain of computation, which has resulted in a too large communications and synchronisation overhead, and little speedup is obtained with modest sized problems.

Chapter 5

Conclusion

The initial aim of this work was straightforward. It was to determine if the implementation of a parallel graph reducer on a wafer was feasible. The emphasis here was on the word feasible — the intention was never to produce a competitive graph reduction implementation. Although a definitive conclusion on this may still be in some doubt, there is no question that several positive results have emerged from this work.

5.1 Communications for WSI

The first positive result concerns communications for wafer scale integration. At an early stage we identified the need to separate the communications and processing functions of a WSI device. We then devised and investigated three different routing algorithms for random point to point communications between nodes in a highly regular arrays of processors on a wafer. Each algorithm has been shown to be useful for different applications.

We have identified that the signpost algorithm is the superior solution for the design presented. We have estimated its performance and found that its average latency under a wide range of conditions is low. It is simple to implement, and thus yields well — a crucial property for WSI. We have indicated that the algorithm has untapped potential because of its ability to be configured dynamically by the wafer controller.

Although the communications algorithms were designed with graph reduction in mind, they have more general properties and have potential applications in many areas of computer architecture. For example, we could design a wafer memory device by having a large block of memory in place of the processor at each node. This would be superior to the Anamartic wafer memory device because it would have lower latency, and concurrent access.

Another application, which is being investigated in this department, is neural networks. Each node consists of a number of artificial neurons which

can communicate directly, or remotely using the communications architecture.

It is clear that there are many potential applications that could exploit the high communications bandwidth of WSI using one of these communications architectures.

5.2 Formal Specification of Hardware

We have found that the specification language Paragon with some extensions is indeed suitable for specifying hardware. It allows expression of high level constructs such as dynamicism and asynchronous communications, yet at the same time can be used to specify much lower levels. It captures well the nature of hardware designs as collections of communicating objects. It is however limited in its scope, as in its present form it is not suitable for expressing very low level behaviour such as the control of timing.

The second positive result is that we have shown that there exists a route from our very high level specification language to hardware designs that can potentially be shown to be correct with respect to the specification. This differs from previous work in this area in that the specification is at a much higher level. We have shown how the methodology can be applied by hand to a specification of a parallel graph reducer. The methodology has demonstrated itself to be useful in several respects. First, it has allowed us to spot undesirable properties in our specification — for example objects being synchronous when we do not want them to be. Second it has provided us with a prototype in the form of a simulator that has allowed us to experiment with the architecture, and that has allowed us to estimate its performance.

5.3 Graph Reduction for WSI

As far as the feasibility of graph reduction on a wafer is concerned, the case is not yet proven beyond all doubt. Although the communications latency of WSI communications is extremely low, the overhead associated with such a fine grain of computation as directors still seems to be a limiting factor. However, having said that, the performance at least seems to be within an order of magnitude of contemporary parallel graph reduction architectures.

In addition, we have shown just how important is the mapping problem with such a fine grain of computation. The performance of our machine was substantially impaired by careless mapping of programs onto processors.

To complete the study of whether graph reduction is feasible on a wafer, we need to attempt computation at a higher grain, and we need to be able to map programs onto processors with more confidence. Supercombinators

would probably offer a better grain of computation for WSI graph reduction. Any grain coarser than this would probably be unable to exploit the potential of on-wafer communications. Static program analysis techniques, and programmer annotations will lead to better solutions on the program mapping problem.

5.4 Further Work

It is difficult to see how further study of the wafer scale communications architecture would further advance knowledge in the area. We believe that any of the communications architectures could easily be combined with a simple processor and taken all the way to design and manufacture. An appropriate application might be a wafer disk, or a communications switch.

Much of the work on the formal derivation of hardware from Paragon would benefit from further investigation and experimentation. Although the route has been sketched, some of the major problems have not been addressed. This includes proofs of the correctness of the method, which is probably the most difficult of the problems.

Automation of the method, including facilities for the static analysis of specifications is another area that would benefit from further research. In fact, automation would not be useful unless tools for the analysis of specifications were available, because it is only from such analysis that we can derive efficient designs.

Finally to decide if wafer scale is a suitable technology for graph reduction, an investigation into a higher grain of computation with appropriate program mapping techniques is needed.

Bibliography

[ABH+89] Paul Anderson, David Bolton, Chris Hankin, Paul Kelly, and Peter Osmon. COBWEB-2—a Declarative Language Multiprocessor Architecture for Wafer Scale Integration. In Fountain and Shute [FS89].

[AC78] Russell Aubusson and Ivor Catt. Wafer-Scale Integration - A Fault-Tolerant Procedure. *IEEE Journal of Solid State Circuits*, Vol.SC-13, June 1978.

[AH87] S. Abramsky and C.L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[AHK+87] Paul Anderson, Chris Hankin, Paul Kelly, Peter Osmon, and Malcolm Shute. COBWEB-2: Structured Specification of a Wafer-Scale Supercomputer. pages 51–67. Springer-Verlag, LNCS 259, 1987. City University TCU/CS/1987/10.

[AKW90] Paul Anderson, Paul Kelly, and Phil Winterbottom. The Feasibility of a General-Purpose Parallel Computer using WSI. *Fifth Generation Computer Systems*, July 1990.

[AN87] Arvind and R.S. Nikhil. Executing a program in the MIT tagged-token dataflow architecture. In de Bakker et al. [dBNT87].

[AO88] Paul Anderson and Peter Osmon. A Fault Tolerant Communications Architecture for Wafer Scale Integration. In *Proceedings of the Alvey Technical Conference*, pages 504–507, 1988. City University TCU/CS/1988/13.

[BBG87] Jacek Błażewicz, Jerzy Brezeziński, and Giorgio Gambosi. Time-Stamp Approach to Store-and-Forward Deadlock Prevention. *IEEE Trans. Communications*, Vol Com-35, No.5:490–495, May 1987.

[BHK88] David Bolton, Chris Hankin, and Paul Kelly. Parallel Object-Orientated Descriptions of Architectures Specified by Graph Rewriting Systems. City University TCU/CS/1988/11, 1988.

[BHK90] David Bolton, Chris Hankin, and Paul Kelly. Parallel object-oriented descriptions of graph reduction machines. *New Generation Computing*, 1990.

[Bur87] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, Department of Computing, Imperial College of Science and Technology, 1987.

[Bur89a] G.L. Burn. Deriving a parallel evaluation model for lazy functional languages using abstract interpretation. In de Bakker [dB89].

[Bur89b] G.L. Burn. Overview of a parallel reduction machine project ii. In Odijk et al. [ORS89].

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.W Glauert, J.R. Kennaway, M.J Plasmeijer, and M.R. Sleep. Term graph rewriting. In de Bakker et al. [dBNT87].

[CCC+89] A. Contessa, E. Cousin, C. Coustet, M. Cubero-Castan, G. Durrieu, B. Lecussan, M. Lemaitre, and P. Ng. MaRS, a combinator graph reduction multiprocessor. In Odijk et al. [ORS89].

[Cur89] L. Curran. Wafer Scale Integration arrives in 'disk' form. *Electronic Design*, 37(22):51–54, October 1989.

[dB89] J.W. de Bakker, editor. *Languages for Parallel Architectures. Design, Semantics, Implementation Models.* Wiley series in parallel computing, 1989.

[dBNT87] J.W de Bakker, A.J. Nijman, and P.C. Treleaven, editors. *Parallel Architectures and Languages Europe, Vol.I: Parallel Languages.* Springer-Verlag, LNCS 258, 1987.

[DD89] W. Damm and G. Döhmen. AADL: A net-based specification method for computer architecture design. In de Bakker [dB89].

[DS87] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Computers*, Vol C-36, No.5:547–553, May 1987.

[Ell86] *The ELLA User Manual*, 2.0 edition, 1986.

[FS89] T.J. Fountain and M.J. Shute, editors. *Multiprocessor Computer Architectures.* North Holland, 1989.

[Gaj88] Daniel D. Gajski, editor. *Silicon Compilation.* Addison Wesley, 1988.

[GH86] B. Goldberg and P. Hudak. Alfalfa: Distributed graph reduction on a hypercube multiprocessor. Preprint, Yale University Department of Computer Science, November 1986.

[GKS87] J.R.W. Glauert, J.R. Kennaway, and M.R. Sleep. DACTL: a computational model and compiler target language based on graph reduction. Technical Report Report SYS-C87-03, School of Information Systems, University of East Anglia, 1987.

[GKW85] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), January 1985.

[Gla90] Hugh Glaser. Personal communication on the fast project, 1990.

[Gor86] M.J.C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design.* North Holland, 1986.

[HOS85] Chris Hankin, Peter Osmon, and Malcolm Shute. COBWEB—A Combinator Reduction Architecture. In *Functional Programming and Computer Architecture, LNCS 201*, pages 99–112, Nancy, France, September 1985. Springer Verlag.

[HP90] Kevin Hammond and Simon Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer, 1990.

[HS88] Jim C. Harden and Noel R. Strader II. Architectural Yield Optimization for WSI. *IEEE Transactions on Computers*, 37(1), January 1988.

[Hug84] R.J.M. Hughes. *The design and implementation of programming languages.* PhD thesis, Programming Research Group, Oxford, 1984. PRG 40.

[Hun90] Sebastian Hunt. *HFC — a* HOPE$^+$ *to FLIC Translator*, May 1990.

[IEE90] *The 17th International Symposium on Computer Architecture.* The IEEE Computer Society Press, May 28–31 1990.

[JHL90] I.P. Jalowiecki, S.J. Hedge, and R.M. Lea. 'WASP': a demonstrated wafer scale technology. In *Proceedings of the UK IT90 conference, Southampton University*, 1990. Alvey Conference Publication no. 316.

[Kar87] R.J. Karia. *An Investigation of Combinator Reduction on Multiprocessor Architectures.* PhD thesis, Westfield College, University of London, 1987.

[Kel89] Paul H.J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Pitman/MIT Press, 1989.

[Kir89] C. Kirkham. The Manchester Dataflow project. In Fountain and Shute [FS89].

[KS81] J.R. Kennaway and M.R. Sleep. Director Strings as Combinators. Technical report, University of East Anglia, 1981.

[KS86] Paul Kelly and Malcolm Shute. Cartesian Routing and Fault Tolerance in a Wafer-Scale Multi-computer. In *Proceedings of the IFIP Workshop on Wafer Scale Integration*, Grenoble, France, 17-19 March 1986.

[LB90] David R. Lester and Geoffrey L. Burn. An Executable Specification of the HDG-Machine. GEC Hirst Research Centre, East Lane, Wembley, 1990.

[Lea87] R.M. Lea, editor. *WSI II. Proceedings of the Second IFIP WG 10.5 Workshop on Wafer Scale Integration.* North Holland, Sept 23–25 1987.

[LKID89] R. Loogen, H. Kuchen, K. Indermark, and W. Damm. Distributed implementation of programmed graph reduction. In Odijk et al. [ORS89].

[MK87] D. May and C. Keane. Compiling occam to silicon. Technical Report 23, Inmos Ltd., 1000 Aztec West, Almondsbury, Bristol, 1987.

[ML86] R.D. McKirdy and R.M. Lea. Physical design issues for WSI. In Saucier and Trihle [ST86].

[NSS89] R. Negrini, M.G. Sami, and R. Stefanelli. *Fault Tolerance through Reconfiguration in VLSI and WSI arrays.* MIT press, 1989.

[ORS89] E Odijk, M. Rem, and J.C. Syre, editors. *Parallel Architectures and Languages Europe, Vol.I: Parallel Architectures.* Springer-Verlag, LNCS 365, 1989.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In IEEE [IEE90].

[PCS89] S.L. Peyton Jones, C. Clack, and J. Salkild. High-performance parallel graph reduction. In Odijk et al. [ORS89].

[Pel83] D.L. Peltzer. Wafer Scale Integration: The limits of VLSI? *VLSI Design*, September 1983.

[Per88] Nigel Perry. HOPE$^+$. Reference IC/FPR/LANG/2.5.1/7, Functional Programming Research Group, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK, February 1988.

[Pey87] Simon L. Peyton Jones. *The Implementaton of Functional Programming Languages.* Prentice Hall, 1987.

[Pit87] K.D. Pitt. Wafer Scale Integration Packaging Problems. In Lea [Lea87].

[PJ89] S.L. Peyton Jones and M.S. Joy. FLIC — a functional language intermediate code. formerly UCL Department of Computer Science internal report 2048, July 1989.

[RD86] A.W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. Technical Report Technical monograph PRG-57, Oxford University PRG, 1986.

[Shu83] Malcolm J. Shute. *The role of Simulation in the Study of Multiprocessor, Control Flow and Data Flow Systems.* PhD thesis, Westfield College, University of London, 1983.

[SJ89] Ph. Schnoebelen and Ph. Jorrand. Principles of FP2: Term algebras for specification of parallel machines. In de Bakker [dB89].

[ST86] G. Saucier and J. Trihle, editors. *Wafer Scale Integration.* North Holland, 1986.

[Sum86] G.W. Sumerling. Cost Models for ULSI and WSI, 1986.

[Tew89] Stuart K. Tewksbury. *Wafer-Level Integrated Systems: Implementation Issues.* Kluwer Academic, 1989.

[Tur79] David Turner. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, pages 31–49, 1979.

[Veg84] Steven R. Vegdahl. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computing*, December 1984.

[Wad87] P. Wadler. Strictness on non-flat domains. In Abramsky and Hankin [AH87].

[WL87] K.D. Warren and R.M. Lea. Electrical Design Issues for Wafer Scale Integration. In Lea [Lea87].

Appendix A

Cobweb as a TRS

This appendix contains a complete description of COBWEB as a term rewriting system. As defined in 3.3.1 the syntax of a term is:

$$
\begin{array}{lcl}
\langle e \rangle & ::= & \langle e \rangle \langle e \rangle \\
 & | & ([\langle d \rangle] \langle e \rangle) \\
 & | & (\langle e \rangle) \\
 & | & \langle p \rangle \\
\langle d \rangle & ::= & \langle d' \rangle \langle d \rangle \\
 & | & \mathsf{nil} \\
\langle d' \rangle & ::= & \wedge \mid - \mid / \mid \backslash \mid \#
\end{array}
$$

where $\langle p \rangle$ represents a primitive operator, or constant.

A.1 Directors

The full set of directors is given here.

Send the operand to the right subgraph.

$$
\begin{array}{lcl}
([\backslash ::d]\ f\ a)\ x & \rightarrow & [d]\ f\ (a\ x) \\
([\backslash ::d]\ f\ \mathsf{I})\ x & \rightarrow & [d]\ f\ x
\end{array}
$$

Send the operand to the left subgraph.

$$
\begin{array}{lcl}
([/ ::d]\ f\ a)\ x & \rightarrow & [d]\ (f\ x)\ a \\
([/ ::d]\ \mathsf{I}\ a)\ x & \rightarrow & [d]\ x\ a
\end{array}
$$

Discard the operand.

$$([-::d]\ f\ a)\ x \rightarrow [d]\ f\ a$$

Send the operand to both subgraphs.

$$\begin{aligned}
([\wedge::d]\ f\ |)\ x &\rightarrow [d]\ (f\ x)\ x \\
([\wedge::d]\ |\ a)\ x &\rightarrow [d]\ x\ (x\ a) \\
([\wedge::d]\ |\ |)\ x &\rightarrow [d]\ x\ x \\
([\wedge::d]\ f\ a)\ x &\rightarrow [d]\ (f\ x)\ (a\ x)
\end{aligned}$$

Context sensitive strictness director.

$$([\#::d]\ f\ a) \rightarrow [d]\ f\ a$$

A.2 Strict built in operators

The full set of strict dyadic operators is:

INT_	INT+	INT−	INT×
INT/	INT<	INT% (remainder)	
INT≤	INT=	INT≥	INT>
INT≠			

There is a similar set for floating point:

FLOAT_	FLOAT+	FLOAT−	FLOAT×
FLOAT/	FLOAT<	FLOAT≤	FLOAT=
FLOAT≥	FLOAT>	FLOAT≠	

The only monadic strict operator for the integers is the negation operator INT_ . The set for floating point numbers is:

SQRT	SIN	COS	ARCTAN
EXP (natural exponential)		LN (natural log)	

And finally there are two conversion operators: INT→FLOAT and FLOAT→INT.

The generic rules for all these operators in terms of a monadic operator g and a dyadic operator f:

$$\begin{aligned}
f\ \perp\ b &\rightarrow \perp \\
f\ a\ \perp &\rightarrow \perp \\
f\ a\ b &\rightarrow \underline{f\ a\ b} \\
g\ \perp &\rightarrow \perp \\
g\ a &\rightarrow \underline{g\ a}
\end{aligned}$$

A.3 Primitives

The set of primitives includes the boolean operators, the selection operator, the context free parallelism operator and the fixedpoint operator.

$$
\begin{array}{lcl}
\mathsf{OR\ TRUE}\ x & \rightarrow & \mathsf{TRUE} \\
\mathsf{OR\ FALSE}\ x & \rightarrow & x \\
\mathsf{OR}\ \bot\ x & \rightarrow & \bot \\
 & & \\
\mathsf{AND\ FALSE}\ x & \rightarrow & \mathsf{FALSE} \\
\mathsf{AND\ TRUE}\ x & \rightarrow & x \\
\mathsf{AND}\ \bot\ x & \rightarrow & \bot \\
 & & \\
\mathsf{XOR}\ x\ \bot & \rightarrow & \bot \\
\mathsf{XOR}\ \bot\ x & \rightarrow & \bot \\
\mathsf{XOR}\ x\ x & \rightarrow & \mathsf{FALSE} \\
 & & \\
\mathsf{NOT}\ \bot & \rightarrow & \bot \\
\mathsf{NOT\ FALSE} & \rightarrow & \mathsf{TRUE} \\
\mathsf{NOT\ TRUE} & \rightarrow & \mathsf{FALSE} \\
 & & \\
\mathsf{IF}\ \bot\ a\ b & \rightarrow & \bot \\
\mathsf{IF\ TRUE}\ a\ b & \rightarrow & a \\
\mathsf{IF\ FALSE}\ a\ b & \rightarrow & b
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{K}\text{-}n\ i & \rightarrow & \mathsf{K}\text{-}n\text{-}i \\
\mathsf{K}\text{-}n\text{-}i\ x_0 \ldots x_n & \rightarrow & x_i \\
 & & \\
\mathsf{P}\ a\ b & \rightarrow & a\ b \\
 & & \\
\mathsf{Y}\ f & \rightarrow & f\ (\mathsf{Y}\ f)
\end{array}
$$

A.4 Data Constructors/Selectors

Data constructors and selectors are given below.

$$
\begin{array}{lcl}
\text{PACK-}n\ d & \rightarrow & \text{PACK-}n\text{-}d \\
\text{PACK-}n\text{-}d\ x_0\ \ldots\ x_{n-1} & \rightarrow & \langle d|x_0, \ldots, x_{n-1}\rangle \\
 & & \\
\text{SEL-}n\ i & \rightarrow & \text{SEL-}n\text{-}i \\
\text{SEL-}n\text{-}i\ \bot & \rightarrow & \bot \\
\text{SEL-}n\text{-}i\ \langle d|x_0, \ldots, x_{n-1}\rangle & \rightarrow & x_i \\
 & & \\
\text{UNPACK!-}n\ f\ \bot & \rightarrow & \bot \\
\text{UNPACK!-}n\ f\ \langle d|x_0, \ldots, x_{n-1}\rangle & \rightarrow & f\ x_0\ \ldots\ x_{n-1} \\
\text{UNPACK-}n\ f\ e & \rightarrow & f\ (\text{SEL-}n\text{-}0\ e)\ \ldots\ (\text{SEL-}n\text{-}(n-1)\ e) \\
 & & \\
\text{CASE-}r\ e_0\ \ldots\ e_{r-1}\ \langle d|x\rangle & \rightarrow & e_d \\
\text{CASE-}r\ e_0\ \ldots\ e_{r-1}\ \bot & \rightarrow & \bot \\
 & & \\
\text{TAG}\ \bot & \rightarrow & \bot \\
\text{TAG}\ \langle d|x\rangle & \rightarrow & d
\end{array}
$$

A.5 Sequencing, Strictness and Termination

$$
\begin{array}{lcl}
\text{SEQ}\ \bot\ b & \rightarrow & \bot \\
\text{SEQ}\ a\ b & \rightarrow & b \\
 & & \\
\text{STRICT}\ f\ \bot & \rightarrow & \bot \\
\text{STRICT}\ f\ x & \rightarrow & f\ x \\
 & & \\
\text{ABORT} & \rightarrow & \bot
\end{array}
$$

Appendix B

Cobweb in Paragon

This appendix contains the specification for COBWEB at two levels. The first section describes the highest level, and contains a complete paragon specification for all the classes defined for the machine as introduced in section 3.3.2.

Section 3.5 contains a discussion of some of the design decisions made for the machine. These were expressed as a set of simplified paragon rules. The second section in this appendix contains these new rules.

B.1 High level specification

B.1.1 Packets and Agents

We first need to describe the packet and agent class

```
class Agent ::= ⟨packet,integer⟩
class Packet ::= ⟨rator,rand,string,innf,act,list agent⟩
data rator = packet | basic-value
data rand = packet | basic-value
data innf = Nf | Notnf
data act = Active | Inactive
```

A *basic-value* can be a constant integer floating point value, or boolean; or it can be a basic operator or primitive. The full set of primitives is as follows:

Booleans IF,AND,OR,XOR,NOT,TRUE,FALSE

Selection K-n, K-n-i $n \geq 0, 0 \leq i < n$

Miscellaneous SEQ, STRICT, ABORT, P, Y

Data constructors/selectors for $n \geq 0, 0 \leq i < n$, $r > 0$
TAG, CASE-r, PACK-n, PACK-n-i SEL-n, SEL-n-i, UNPACK!-n, UNPACK-n

Packets respond to three messages:

rewrite indicates that the packet is to be rewritten. The arguments to this message are the new structure of the packet.

need indicates that the packet is needed by an agent.

fire indicates that the packet is to be evaluated to normal form.

Agents respond to two messages:

reduce indicates that the agent is to reduce the packet which is an argument to the message.

wakeup indicates that the agent can resume reducing a packet.

B.1.2 Rewrite

```
⟨_,_,_,_,_,p⟩
        given rewrite(op,arg,ann,Notnf)
        → ⟨op,arg,ann,Notnf,_,p⟩

⟨_,_,_,_,_,p⟩
        given rewrite(op,arg,ann,Nf)
        → ⟨op,arg,ann,Nf,Inactive,nil⟩
          then
          p !! wakeup
```

Note that in the above rule p is a **list** of agents. There is an implied mapping of wakeup onto all the elements in the list.

B.1.3 Need

```
⟨_,_,_,_,Active,p⟩ given need(agent)
        → ⟨_,_,_,_,Active,agent::p⟩

⟨_,_,_,Nf,Inactive,p⟩ given need(agent)
        → ⟨_,_,_,Nf,Inactive,p⟩
          then
          agent !! wakeup

⟨_,_,_,Notnf,Inactive,p⟩ given need(agent)
        → ⟨_,_,_,Notnf,Active,agent::p⟩
          then
          new(Agent,⟨self,0⟩) !! reduce(self)
```

B.1.4 Fire

```
⟨_,_,_,_,Active,p⟩ given fire
        → ⟨_,_,_,_,Active,p⟩

⟨_,_,_,Nf,Inactive,p⟩ given fire
        → ⟨_,_,_,Nf,Inactive,p⟩

⟨_,_,_,Notnf,Inactive,p⟩ given fire
        → ⟨_,_,_,Notnf,Active,p⟩
          then
          new(Agent,⟨self,0⟩) !! reduce(self)
```

B.1.5 Wakeup

```
⟨_,c⟩   given wakeup when c ≥ 2
        → ⟨_,c−1⟩

⟨P,1⟩   given wakeup
        → ⟨P,0⟩
          then
```

```
        self!! reduce(P)
```

B.1.6 Reduce

Normal form and finding next redex

```
[onf]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨op,arg,a::x,_,_,_⟩ ∧a≠#
        → ⟨P1,0⟩
          then
          P1 !! rewrite(op,arg,a::x,Nf)

[ot]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,_,nil,_,_,_⟩ ∧
        P2 = ⟨_,_,_,Notnf,_,_⟩
        → ⟨P1,1⟩
          then
          P2 !! need(self)
```

Unlabelled rewrites

```
[ou1]
⟨P1,0⟩ when
        P1 = ⟨P2,a,ann,flag,_,_⟩ ∧
        P2 = ⟨f,nil,nil,Nf,_,_⟩
        → ⟨P1,0⟩
          then
          P1 !! rewrite(f,a,ann,flag)

[ou2]
⟨P1,0⟩ when
        P1 = ⟨f,P2,ann,flag,_,_⟩ ∧
        P2 = ⟨a,nil,nil,Nf,_,_⟩
        → ⟨P1,0⟩
          then
          P1 !! rewrite(f,a,ann,flag)
```

Fixed point combinator

```
[oY]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨Y,arg,nil,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(arg,P1,nil,Notnf) ;
            self !! reduce(P1)
```

Directors

```
[o\1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg1,nil,_,_,_⟩ ∧
        P2 = ⟨op,arg2,\::ann,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(op,newpacket,ann,Notnf) ;
            self !! reduce(P1)
            where
            newpacket = new(Packet,⟨arg2,arg1,nil,Notnf,Inactive,nil⟩)

[o\2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg1,nil,_,_,_⟩ ∧
        P2 = ⟨op,I,\::ann,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(op,arg1,ann,Notnf) ;
            self !! reduce(P1)

[o/1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg1,nil,_,_,_⟩ ∧
        P2 = ⟨op,arg2,/::ann,_,_,_⟩
        → ⟨P1,0⟩
```

```
            then
            P1 ! rewrite(newpacket,arg2,ann,Notnf) ;
            self !! reduce(P1)
            where
            newpacket = new(Packet,⟨op,arg1,nil,Notnf,Inactive,nil⟩)

[o/2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,op,nil,_,_,_⟩ ∧
        P2 = ⟨I,arg,/::ann,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(op,arg,ann,Notnf) ;
            self !! reduce(P1)

[o∧1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg,nil,_,_,_⟩ ∧
        P2 = ⟨op,arg2,∧::d,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(newop,newarg,d,Notnf) ;
            self !! reduce(P1)
            where
            newop = new(Packet,⟨op,arg,nil,Notnf,Inactive,nil⟩)
            newarg = new(Packet,⟨arg2,arg,nil,Notnf,Inactive,nil⟩)

[o∧2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg1,nil,_,_,_⟩ ∧
        P2 = ⟨I,arg2,∧::d,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(arg1,newarg,d,Notnf) ;
            self !! reduce(P1)
            where
            newarg = new(Packet,⟨arg2,arg1,nil,Notnf,Inactive,nil⟩)

[o∧3]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P1,arg,nil,_,_,_⟩ ∧
        P2 = ⟨op,I,∧::d,_,_,_⟩
```

```
        → ⟨P1,0⟩
            then
            P1 ! rewrite(newop,arg,d,Notnf) ;
            self !! reduce(P1)
            where
            newop = new(Packet,⟨op,arg,nil,Notnf,Inactive,nil⟩)

[o∧4]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg,nil,_,_,_⟩ ∧
        P2 = ⟨I,I,∧::d,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(arg,arg,d,Notnf) ;
            self !! reduce(P1)

[o-]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg,nil,_,_,_⟩ ∧
        P2 = ⟨op,arg,-::d,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(op,arg,d,Notnf) ;
            self !! reduce(P1)

[o#]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨op,arg,#::d,n,_,_⟩ ∧
        ispacket(arg)
        → ⟨P1,0⟩
            then
            (arg !! fire ||
            P1 ! rewrite(op,arg,d,n)) ;
            self !! reduce(P1)
```

Context free parallelism

```
[oP]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,P3,nil,n,_,_⟩ ∧
```

$P_2 = \langle P,f,\_,\_,\_,\_\rangle \wedge$
ispacket(P_3)
$\rightarrow$ $\langle P_1,0\rangle$
then
(P_3 !! fire ||
P_1 ! rewrite(f,P_3,nil,n)) ;
self !! reduce(P_1)

Monadic strict primitive operators

[monop1]
$\langle \_,0\rangle$ given reduce(P_1) when
$P_1 = \langle$op,n,nil,$\_,\_,\_\rangle \wedge$
is_integer(n) $\wedge$arity(op) = 1
$\rightarrow$ $\langle \_,\_\rangle$
then
P_1 ! rewrite(op n,nil,nil,Nf)

[monop2]
$\langle \_,0\rangle$ given reduce(P_1) when
$P_1 = \langle$op,P_2,nil,$\_,\_,\_\rangle \wedge$
$P_2 = \langle _,_,_,$Notnf,$_,_\rangle$
$\rightarrow$ $\langle P_1,1\rangle$
then
P_2 !! need(self)

Dyadic boolean operators

[bool1]
$\langle \_,0\rangle$ given reduce(P_1) when
$P_1 = \langle P_2$,y,nil,$\_,\_,\_\rangle \wedge$
$P_2 = \langle$op,x,nil,$\_,\_,\_\rangle \wedge$
is_packet(x) $\wedge$
op = AND or op = OR
$\rightarrow$ $\langle P_1,1\rangle$
then
x !! need(self)

```
[and1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
        P2 = ⟨op,x,nil,_,_,_⟩ ∧
        op = AND ∧x = TRUE
        → ⟨_,_⟩
            then
            P1 ! rewrite(y,nil,nil,Notnf) ;
            self !! reduce(P1)

[or1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
        P2 = ⟨op,x,nil,_,_,_⟩ ∧
        op = OR ∧x = FALSE
        → ⟨_,_⟩
            then
            P1 ! rewrite(y,nil,nil,Notnf) ;
            self !! reduce(P1)

[and2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
        P2 = ⟨op,x,nil,_,_,_⟩ ∧
        op = AND ∧x = FALSE
        → ⟨_,_⟩
            then
            P1 ! rewrite(FALSE,nil,nil,Notnf) ;
            self !! reduce(P1)

[or2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,y,nil,_,_,_⟩ ∧
        P2 = ⟨op,x,nil,_,_,_⟩ ∧
        op = OR ∧x = TRUE
        → ⟨_,_⟩
            then
            P1 ! rewrite(TRUE,nil,nil,Notnf) ;
            self !! reduce(P1)

[IF1]
⟨_,0⟩   given reduce(P1) when
```

$P_1 = \langle P_2,b,nil,\_,\_,\_\rangle \wedge$
$P_2 = \langle P_3,a,nil,\_,\_,\_\rangle \wedge$
$P_3 = \langle IF,P_4,nil,\_,\_,\_\rangle \wedge$
is_packet(P_4)
then
P_4 !! need(self)

[IF2]
$\langle \_,0\rangle$ given reduce(P1) when
$P_1 = \langle P_2,b,nil,\_,\_,\_\rangle \wedge$
$P_2 = \langle P_3,a,nil,\_,\_,\_\rangle \wedge$
$P_3 = \langle IF,FALSE,nil,\_,\_,\_\rangle \wedge$
is_packet(P_4)
$\rightarrow \langle \_,\_\rangle$
then
P_1 ! rewrite(b,nil,nil,Notnf) ;
self !! reduce(P_1)

[IF2]
$\langle \_,0\rangle$ given reduce(P1) when
$P_1 = \langle P_2,b,nil,\_,\_,\_\rangle \wedge$
$P_2 = \langle P_3,a,nil,\_,\_,\_\rangle \wedge$
$P_3 = \langle IF,TRUE,nil,\_,\_,\_\rangle \wedge$
is_packet(P_4)
$\rightarrow \langle \_,\_\rangle$
then
P_1 ! rewrite(a,nil,nil,Notnf) ;
self !! reduce(P_1)

Dyadic strict primitive operators

[dyop1]
$\langle \_,0\rangle$ given reduce(P_1) when
$P_1 = \langle P_2,n,nil,\_,\_,\_\rangle \wedge$
$P_2 = \langle op,m,nil,\_,\_,\_\rangle \wedge$
is_integer(m) $\wedge$is_integer(n) $\wedge$
arity(op) = 2
$\rightarrow \langle \_,\_\rangle$
then
P_1 ! rewrite(op m n,nil,nil,Nf)

[dyop2]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,n,nil,_,_,_⟩ ∧
P_2 = ⟨op,P_3,nil,_,_,_⟩ ∧
P_3 = ⟨_,_,_,Notnf,_,_⟩ ∧
is_integer(n)
→ ⟨_,1⟩
then
P_3 !! need(self)

[dyop3]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,P_3,nil,_,_,_⟩ ∧
P_2 = ⟨op,n,nil,_,_,_⟩ ∧
P_3 = ⟨_,_,_,Notnf,_,_⟩ ∧
is_integer(n)
→ ⟨_,1⟩
then
P_3 !! need(self)

[dyop4]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,P_4,nil,_,_,_⟩ ∧
P_2 = ⟨op,P_3,nil,_,_,_⟩ ∧
P_3 = ⟨_,_,_,Notnf,_,_⟩ ∧
P_4 = ⟨_,_,_,Notnf,_,_⟩
→ ⟨_,2⟩
then
P_3 !! need(self) ||
P_4 !! need(self)

Primitives

[IF0]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,arg2,nil,_,_,_⟩ ∧
P_2 = ⟨IF,arg1,nil,_,_,_⟩ ∧
is_bool(arg1)
→ ⟨_,0⟩

```
            then
            P1 ! rewrite(K-2-n,arg2,nil,Notnf) ;
            self !! reduce(P1)
            where
            n = 0, if arg1 = True
            n = 1, if arg1 = False

[IF1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg2,nil,_,_,_⟩ ∧
        P2 = ⟨IF,arg1,nil,_,_,_⟩ ∧
        is_packet(arg1)
        → ⟨P1,1⟩
            then
            arg1 !! need(self)

[SEQ1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg2,nil,_,_,_⟩ ∧
        P2 = ⟨SEQ,arg1,nil,_,_,_⟩ ∧
        ⟨_,_,_,Notnf,_,_⟩ arg1
        → ⟨P1,1⟩
            then
            arg1 !! need(self)

[SEQ2]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg2,nil,_,_,_⟩ ∧
        P2 = ⟨SEQ,arg1,nil,_,_,_⟩ where
        not is_packet(arg1)
        → ⟨_,0⟩
            then
            P1 ! rewrite(arg2,nil,nil,Notnf) ;
            self !! reduce(P1)

[STRICT]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,arg2,nil,_,_,_⟩ ∧
        P2 = ⟨STRICT,arg1,nil,_,_,_⟩ ∧
        ⟨_,_,_,Notnf,_,_⟩ arg2
        → ⟨P1,1⟩
            then
```

arg1 !! need(self)

[STRICT]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,arg2,nil,_,_,_⟩ ∧
P_2 = ⟨STRICT,arg1,nil,_,_,_⟩ when
not is_packet(arg2)
→ ⟨_,0⟩
then
P_1 ! rewrite(arg1,arg2,nil,Notnf) ;
self !! reduce(P_1)

[ABORT]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨ABORT,x,_,_,_,_⟩ ∧
x ≠ nil
→ self
then
P_1 ! rewrite(ABORT,nil,nil,nf)

Selection

[K-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨K-n,i,nil,_,_,_⟩ ∧
is_integer(n)
→ ⟨P_1,0⟩
then
P_1 ! rewrite(K-n-i,nil,nil,Notnf) ;
self !! reduce(P_1)

[K-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨K-n,p,_,_,_,_⟩ ∧
is_packet(p)
→ ⟨_,1⟩
then
p !! need(self)

[K-n-i]
⟨_,0⟩ given reduce(p_n) when
p_n = ⟨p_{n-1},x_{n-1},nil,_,_⟩ ∧
p_{n-1} = ⟨p_{n-2},x_{n-2},nil,_,_⟩ ∧
⋮
p_0 = ⟨K-n-i,x_0,nil,_,_,_⟩
→ ⟨_,_⟩
then
p_n ! rewrite(x_i,nil,nil,Notnf) ;
self !! reduce(p_n)

Data constructors

[PACK-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨PACK-n,d,_,_,_,_⟩ ∧
is_integer(d)
→ ⟨_,_⟩
then
P_1 ! rewrite(PACK-n-d,nil,nil,Nf)

[PACK-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨PACK-n,p,_,_,_,_⟩ ∧
is_packet(p)
→ ⟨_,1⟩
then
P_1 !! need(self)

[SEL-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨SEL-n,i,nil,_,_,_⟩ ∧
is_integer(i)
→ ⟨_,_⟩
then
P_1 ! rewrite(SEL-n-i,nil,nil,Nf)

[SEL-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨SEL-n,p,nil,_,_,_⟩ ∧
is_packet(p)

```
        → ⟨_,1⟩
            then
            p !! need(self)

[SEL-n-i]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨SEL-n-i,s,nil,_,_,_⟩ ∧
        is_struct(s)
        → ⟨_,0⟩
            then
            P1 ! rewrite(x_i,nil,nil,Notnf) ;
            self !! reduce(P1)
            where
            s = ⟨d|x⟩

[SEL-n-i]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨SEL-n-i,p,nil,_,_,_⟩ ∧
        is_packet(p)
        → ⟨_,1⟩
            then
            p !! need(self)

[UNPACK-n]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,e,nil,_,_,_⟩ ∧
        P2 = ⟨UNPACK-n,f,nil,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(p_{n-2},s_{n-1},nil,Notnf) ;
            self !! reduce(P1)
            where
            s_j = new(Packet,⟨SEL-n-j,e,nil,Notnf,Inactive,nil⟩)
            p0 = new(Packet,⟨f,s0,nil,Notnf,Inactive,nil⟩)
            p_j = new(Packet,⟨p_{j-1},s_j,nil,Notnf,Inactive,nil⟩)

[UNPACK-1]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,e,nil,_,_,_⟩ ∧
        ⟨UNPACK-1,f,nil,_,_,_⟩
        → ⟨P1,0⟩
            then
```

P_1 ! rewrite(f,p,nil,Notnf) ;
self !! reduce(P_1)
where
p = new(Packet,⟨SEL-1-0,e,nil,Notnf,Inactive,nil⟩)

[UNPACK!-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,⟨d|x⟩,nil,_,_,_⟩ ∧
P_2 = ⟨UNPACK!-n,f,nil,_,_,_⟩
→ ⟨$node_n$,0⟩
then
P_1 ! rewrite(f,x_0,nil,Notnf) ;
self !! reduce($node_n$)
where
$node_1$ = new(Packet,⟨P_1,x_1,nil,Notnf,Inactive,nil⟩)
$node_j$ = new(Packet,⟨$node_{j-1}$,x_{j-1},nil,Notnf,Inactive,nil⟩)

[CASE-r]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨CASE-r,a,nil,_,_,_⟩
→ ⟨_,_⟩
then
P_1 ! rewrite(DCASE-r,⟨_|a⟩,nil,Nf)

[TAG]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨TAG,s,nil,_,_,_⟩ ∧
is_struct(s)
→ ⟨_,_⟩
then
P_1 ! rewrite(d,nil,nil,Nf)
where
s = ⟨d|x⟩

[TAG]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨TAG,p,nil,_,_,_⟩ ∧
is_packet(p)
→ ⟨_,1⟩
then
p !! need(self)

B.2 Transformed specification

The I combinator

[oI]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨I,P_2,_,_,_⟩
→ ⟨P_1,0⟩
then
P_1 !! rewrite(P_2,nil,nil,Notnf) ;
self ! reduce(P_1)

Selection

[K-1-0]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨K-1-0,arg1,_,_,_,_⟩
→ ⟨P_1,0⟩
then
P_1 ! rewrite(arg1,nil,nil,Nf) ;
self !! reduce(P_1)

[K-2-0]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,x,nil,_,_,_⟩ ∧
P_2 = ⟨K-2-0,a,nil,_,_,_⟩
→ ⟨_,0⟩
then
P_1 ! rewrite(a,nil,nil,Notnf) ;
self !! reduce(P_1)

[K-n-0]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,x,_,_,_⟩ ∧
P_2 = ⟨K-n-0,a,_,_,_⟩
→ ⟨P_1,0⟩
then
P_1 ! rewrite(K-(n-1)-0,a,nil,Notnf) ;
self !! reduce(P_1)

[K-n-i]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨K-n-i,a,_,_,_⟩
→ ⟨_,_⟩
then
P_1 ! rewrite(K-(n-1)-(i-1),nil,nil,Nf)

Data Construction/Selection

[PACK-n-d]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨PACK-n-d,a,_,_,_,_⟩
→ ⟨_,_⟩
then
P_1 ! rewrite(STRUCT-n-d,a,nil,nil,Nf)

[STRUCT-n-(n-1)]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_1,a,nil,_,_,_⟩ ∧
P_2 = ⟨STRUCT-n-(n-1),⟨d|x⟩,nil,_,_,_⟩
→ ⟨_,_⟩
then
P_1 ! rewrite(STRUCT-n⟨d|x++a⟩,nil,nil,Nf)

[STRUCT-n-n]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,a,nil,_,_,_⟩ ∧
P_2 = ⟨STRUCT-n-n,⟨d|x⟩,nil,_,_,_⟩
→ ⟨_,_⟩
then
P_1 ! rewrite(⟨d|x⟩,a,nil,Nf)

[STRUCT-n-i]
⟨_,0⟩ given reduce(P_1) when
P_1 = ⟨P_2,a,nil,_,_,_⟩ ∧
P_2 = ⟨STRUCT-n-i,⟨d|x⟩,nil,_,_,_⟩
→ ⟨_,_⟩
then

```
            P1 ! rewrite(STRUCT-n-(i+1),⟨d|x++a⟩,nil,Nf)

[DCASE-0]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,⟨d|y⟩,nil,_,_,_⟩ ∧
        P2 = ⟨DCASE-0,⟨_,x⟩,nil,_,_,_⟩
        → ⟨P1,0⟩
            then
            P1 ! rewrite(x_d,nil,nil,Notnf) ;
            self !! reduce(P1)

[DCASE-n]
⟨_,0⟩   given reduce(P1) when
        P1 = ⟨P2,a,nil,_,_,_⟩ ∧
        P2 = ⟨DCASE-n,⟨_|x⟩,nil,_,_,_⟩
        → ⟨_,0⟩
            then
            P1 ! rewrite(DCASE-(n-1),⟨x++a⟩,nil,Notnf) ;
            self !! reduce(P1)
```