THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF MECHANICAL AND MANUFACTURING ENGINEERING
and
NATIONAL ICT AUSTRALIA

Operating System for a Flow-Graph Machine

Aleksander Budzynowski
3218701

Bachelor of Engineering

October 2011

Supervisor: Prof. G. Heiser
Co-Supervisor: Dr M. Chowdhury

Certificate of Originality

I declare that this submission is entirely my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any degree or diploma in any institute of higher learning, except where due acknowledgement is made.

Aleksander Budzynowski October 2011

Abstract

The flow-graph machine currently being developed by Wave Semiconductor is a computer processor based on the dataflow model of computation, and has the potential to provide more energy-efficient, resource-efficient and scalable computing than sequential computer processors. In order to make the most of this potential, an operating system must be created for the machine. Such an operating system would need to be designed differently to traditional operating systems, and this area has not received much study in the past. This project is a first step towards the creation of such an operating system. It includes an exploration of the various ways such a machine might be used, an explanation of how an operating system would support this, identification of challenges in the implementation, and suggestions for further development of the machine, the operating system, and the software that will use it.

Acknowledgements

I would like to thank my supervisor Professor Gernot Heiser for giving me the opportunity to take part in this exciting and enjoyable project. Also thanks to my co-supervisor Dr Mac Chowdhury for agreeing to co-supervise at very late notice. Finally, thanks to Karl Fant for teaching me to view many things in a different way.

Contents

1. Introduction **1**

2. Background and Related Work **5**

2.1. The Consequences of Sequences 5

2.2. Scalability, Energy Efficiency and Fault Tolerance 7

2.3. The von Neumann bottleneck 9

2.4. Dataflow Graphs 11

2.5. Dataflow Machines 13

2.6. The Manchester Machine 15

2.7. The Connection Machine 16

2.8. NULL Convention Logic 18

2.9. The Software-Defined ASIC 20

2.10. Operating System Architectures 24

3. Exploration **27**

3.1. Potential Applications 28

3.2. Scope 29

3.3. Understanding Flow Graphs 30

3.4. Rethinking the Role of an Operating System 34

3.4.1. Hardware management 35

3.4.2. Multiplexing and abstraction . . . 37
3.4.3. Services . . . 43
3.5. The Outside World . . . 45
3.6. Management . . . 47
3.7. Graph Manipulation . . . 53
3.7.1. The graph interpreter . . . 54
3.7.2. Hardware-supported graph manipulation . . . 55
3.7.3. Opaque references . . . 57
3.7.4. Templates . . . 58

4. Investigation **60**
4.1. Simulator . . . 61
4.2. Assembly Language . . . 63
4.3. Graph Manipulation Instructions . . . 65
4.4. Compiler . . . 69
4.5. Code Loading . . . 71

5. Synthesis and Future Work **78**
5.1. Hardware . . . 79
5.2. Operating System . . . 81
5.2.1. Architecture . . . 82
5.2.2. Adapting to the FGM . . . 83
5.3. Languages . . . 86

6. Conclusion **88**

A. Graph copy algorithm **90**

List of Figures

2.1. Dataflow graph for evaluating $ax^2 + bx + c^2$. 12
2.2. Dataflow graph for calculating gcd(a,b) using Euclid's algorithm. 14
2.3. Design of the Manchester dataflow processor. 16
2.4. Layout of instructions in the SD-ASIC. 21

3.1. Parents of an instruction. 50
3.2. Simulating nodes with more references. 56

4.1. Addition of a node to a linked list. 66
4.2. The **write** instruction 67
4.3. The **read** instruction 68
4.4. The flow graph compiler in operation. 70
4.5. An illustration of towers and bridges. 73

Nomenclature

Application-specific integrated circuit (ASIC)

An integrated circuit designed for a specific purpose, with a large initial investment but low unit cost.

Concurrency

A way of writing software in which multiple computations can be in progress at the same time. The actual work behind all of these computations need not proceed in parallel; it can be interleaved (i.e. limited processing resources can be time-shared between several computations).

Dataflow computing

A computational model without a notion of a "currently active instruction" — instead instructions activate once their inputs are available, and direct their outputs to subsequent instructions.

Data structure

A system of organising information in a computer. Many data structures support internal cross-references.

Distributed system

A computing system consisting of many independent computers arranged in a network.

Field-programmable gate array (FPGA)

An integrated circuit which can be reprogrammed to behave like an essentially arbitrary digital circuit, useful for prototyping, with a relatively high unit cost.

Global

Involving the entirety of an arbitrarily large system.

Graph

A set of entities (nodes) and particular relationships that exist between pairs of these entities (arcs) (see Section 2.4).

I/O

Input and output.

Local

Involving a strictly limited portion of an arbitrarily large system.

PE

See *processing element*.

Parallelism

A means of increasing computational efficiency by performing several tasks at the same time (*in parallel*) using multiple processing elements.

Partial ordering

Partially specified ordering constraints on a set of items, e.g. “A comes before B, and A comes before C, and C comes before D”, in which case the relative orders of B and C, and of B and D, are not specified and any order is acceptable.

Processing element

A unit of hardware which performs computational work, generally one of several similar units.

Pointer

A memory address, usually an integer. A pointer is a type of reference.

Pointer arithmetic

Arithmetic operations on memory addresses, taking into account the relative placements of items. One example is selecting an element from an array by taking advantage of the fact that array elements are in consecutive memory locations.

Reference

A name for an object, or some other way of locating it.

Tree

A data structure frequently used in computing where information is stored in a hierarchical fashion, for example a directory structure in a file system.

1. Introduction

Hardware wants to be parallel.

However, most of computer science is built on the assumption of sequential execution. Users don't care how the computer works, as long as they get good performance from their applications. Processor manufacturers appease users by making their processors ever more parallel, while appeasing computer scientists by preserving the illusion that processing is done sequentially.

Processor clock frequencies plateaued years ago, and since then processors have delivered faster sequential execution by including more functional units and ensuring more of them are exercised through many complicated techniques. It is easier to squeeze more processor cores onto a chip than to improve sequential execution, however, and thus processors now expose this kind of coarse-grained parallelism to software.

Unfortunately, the shared-memory multi-threaded model of parallel programming which seems a natural fit for multicore hardware is in general difficult to use [Lee, 2006]. Synchronisation is expensive, races and deadlocks are dangerously easy to introduce, and scalability degrades as more cores contend to access a shared global memory. The fact that parallelism is so easy in hardware but so difficult in software suggests that we are doing something wrong.

[Hillis, 1988] has noted that the human brain has about 10 billion neurons, each with a switching time of at least a millisecond. Compare this to an 8-core Nehalem chip, which

has over 2 billion transistors, each with a switching time of less than a nanosecond. This makes the theoretical limit for switchings per second about a million times higher in the processor. Yet it is clear that despite this seemingly higher computational capacity, a single computer is far from reaching human intelligence. While this argument is in no way conclusive, it suggests that there are much better ways to wire transistors together than the way it is done in a typical processor.

A major problem with existing computer architectures is the von Neumann bottleneck. Many of the transistors sit idle because the bandwidth of internal communication channels is easily saturated, and this gets worse as we try to scale our systems.

Performance is not the only important consideration. Energy efficiency, heat dissipation, and fault tolerance are particularly important both in embedded computing and high-performance computing.

Wave Semiconductor (Wave Semi) are working on a processor design that should address all of these issues, and has the potential to revolutionise computer science. It is a clockless dataflow processor, designed to be extremely scalable from the ground up, with thousands of simple processing elements constantly interacting with each other rather than a small number of complex cores. It allows very flexible power/performance trade-offs, and has the potential to be more energy-efficient overall.

The first hardware implementation of this design is destined for use in embedded systems as a replacement for field programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). However, Wave Semi expect the design to be widely applicable, and plan to develop it for high performance computing, and ultimately as a general-purpose processing fabric.

As the technology matures, the demands placed on it by software will become more ambitious. Special software will be needed to help manage available resources, keep order between programs and provide services that increase the utility of the processor —

or in other words, an *operating system* (OS).

One aim of this project was to look at ways such a processor might be used, and investigate what an operating system would have to do to support these uses. Another aim was to look into the differences between this processor and conventional processors, and the impact these differences will have on the operating system. In particular I tried to identify challenges that might be faced in the design of the OS and suggest how these challenges might be overcome.

Since Wave Semi are keen to modify the design of the processor in order to better support the work of an operating system, exploring how the hardware could be improved was an important part of this project. This, coupled with the fact that the area has not been explored much in the past, meant that the project was very exploratory. My intention was to be inclusive rather than focussed, approaching the problem in its entirety so that future work would rest on strong foundations.

As there is much to be discovered about the design of dataflow processors, the kinds of problems they are suited for and how to write software for them, many such things were found in the course of this project. Since an operating system has the role of integrating diverse software and hardware components of a computer system into a coherent whole, such considerations are certainly relevant.

I will refer to the hypothetical future computer on which a mature OS will run as the FGM (flow graph machine). The current product being developed by Wave Semi is branded the software-defined ASIC (SD-ASIC). It is impossible to specify exactly how the FGM will work, however, the important aspects should be generally faithful to the SD-ASIC.

In Chapter 2 I provide the motivations behind the FGM, provide a summary of related work, give an overview of the design of the SD-ASIC, and give an overview of some choices in OS architecture. In Chapter 3 I document my conceptual exploration of all the

key issues of the FGM and of an OS in the context of the FGM which I have identified. This includes an exploration of various approaches to how management is to occur in the FGM, as well as motivations for adding graph manipulation capabilities to the FGM and a discussion of how this might occur. In Chapter 4 I explain the more concrete investigation I performed. This includes my implementation of an FGM simulator, assembly language for the simulator, graph manipulation extensions to the design, and a simple compiler that runs within the simulator, as well as an exploration of how programs might be loaded onto the FGM by an OS. In Chapter 5 I bring everything together and discuss directions for future work.

2. Background and Related Work

2.1. The Consequences of Sequences

The concept of the sequential algorithm originated in the field of mathematics and was picked up by computer engineers as a convenient model of computation. However, processor designers quickly realised that sequential processing was not the way to get computational work done efficiently. Performing several calculations in parallel allowed more work to be done in less time, and the diminishing cost and size of transistors led to this being cheap and practical. While computer processors develop longer pipelines and gain more functional units and more cores, little progress has been made in moving away from software interfaces based on sequential instructions.

To paraphrase [Silc et al., 1998], currently the path from inception to execution for a program looks something like this:

1. A programmer conceives a partially ordered algorithm, but then expresses the algorithm in total ordering because of the use of a sequential von Neumann language.

2. The compiler extracts a partial ordering from the program by analysing data dependencies, and generates reordered "optimised" sequential machine code.

3. The microprocessor dynamically extracts a partial ordering from the machine code, and executes it using *micro dataflow*, then reestablishes the arbitrary serial program

order at the instruction completion stage.

One way of understanding this is that sequential programs are arbitrary and limited, and though we do what we can to get around their limitations, we keep returning to a sequential formulation because that is the interface that was adopted long ago. Some computer scientists respond to the increasing need for parallelism with more sequence! Parallel execution is often modelled as interleaved sequences of instructions, or with multi-tape Turing machines.

Modern processors work very hard to get parallelism out of sequential programs. The techniques they use include pipelining, out-of-order execution, and speculative execution. All this parallelism is hidden behind an illusion of sequential execution. The parallelism is not exposed to software, except in the very coarse-grained form of multiple cores, and this means there are some workloads which are very difficult to implement efficiently on traditional processors.

For applications that have the economic incentive to go into hardware, this is not a problem. Implementing a workload in an FPGA or ASIC exposes a high degree of parallelism to software. However, developing for and deploying these targets is expensive. It would be good to have an option that is less expensive, that is about as easy to program as a traditional processor, and that makes parallelism almost as accessible as an FPGA.

I do not claim that designing something with these characteristics is easy. General-purpose graphical processing units (GPGPUs) are an affordable alternative to FPGAs and ASICs, with a more familiar sequential programming model, but the programming model is quite restrictive and a good understanding of the hardware is required to use it effectively. This makes GPGPUs still quite difficult to write software for, and excludes certain workloads from being implemented efficiently. The Intel Itanium processor attempted to expose some parallelism at the assembly language level, but still with a very sequential form, and pushing a lot of complex decisions into compilers. Maybe we need a new

approach.

Most digital circuits, including processors, use a form of global synchronisation called a *clock*. This is a signal that oscillates at the processor's *clock frequency*, co-ordinating electrical activity throughout the chip and introducing sequence points in the work done.

Even though clock frequencies have essentially stopped increasing, modern processors continue to improve sequential execution speeds [Intel, 2010] using techniques like those mentioned above, but this adds hardware complexity and increases the cost of certain operations like context switches. I do not see how we can continue seeing performance improvements indefinitely if we continue in this fashion.

The clock frequency of a digital circuit must be slow enough to encompass the worst-case propagation and switching delay in the circuit. As transistors continue to become smaller, the difference between average case and worst case increases. Consequently the use of a clock is becoming more of a burden.

2.2. Scalability, Energy Efficiency and Fault Tolerance

Techniques which are ideal for small-scale problems can become unwieldy for problems of a larger scale. As systems grow in size, bottlenecks between components tend to become more prominent, and the ability of components to deal with more interactions is tested. This is true of all manner of productive activities and systems, not just computing.

Designing scalable systems is challenging. Sometimes rearranging existing systems is sufficient, but other times a complete redesign is required. One way to address bottlenecks is to make the system decentralised. This is done by making multiple (or all) components capable of fulfilling certain functions rather than relying on a single component for particular functions. A method of restricting the number of interactions between components is to organise the system into a hierarchy. Often components within the hierarchy are re-

sponsible for filtering information before it reaches the top of the hierarchy. Making the components themselves more efficient and improving the capacity of the links between them can help.

Parallelism in many forms is exploited to improve performance. Commodity computer processors use instruction-level parallelism to improve the performance of sequential software, and expose thread-level parallelism to software in the form of multiple cores. GPUs are good at solving problems with massive data parallelism, and many commodity processors include *vector processing* units which also take advantage of data parallelism, allowing the same operation to be performed on many data items. The term "single instruction, multiple data" (SIMD) describes vector processing units, while "multiple instruction, multiple data" (MIMD) describes the use of multiple cores.

Some programs are very easy to implement for vector processors. These are typically programs with frequently-used data-intensive operations like matrix operations. However, vector processing units lack flexibility and it is difficult to use them for problems without a regular structure. The obvious programming model for multi-core processors is a thread-based shared memory programming model. However, when the threads perform non-trivial communication, it can be extremely difficult to write bug-free programs in this way. This is because threads interact non-deterministically and the programmer must introduce determinism manually where necessary, which is a difficult task [Lee, 2006, Fant, 2007].

[Amdahl, 1967] put forward the argument that most tasks have some components which are strictly sequential and cannot be parallelised, and therefore increasing the amount of parallel resources available will bring diminishing returns (this has become known as *Amdahl's law*). Others have criticised Amdahl's assumptions as only applying to certain sets of problems, such as [Gustafson, 1988] who argued that often we are not trying to solve problems of a fixed size in less time, but rather trying to solve as large a problem

as possible in a reasonable amount of time. Nonetheless, an important consequence of Amdahl's law is that slower sequential execution will slow many programs down, no matter how large the increase in the availability of parallel resources.

For enterprise and high-performance computing (HPC), the demand for fault-tolerance, energy efficiency and scalability of computing hardware is already strong and will continue to grow. As the cost of computing hardware decreases and businesses expand their computing infrastructure, the cost of down time, failures, and cooling become quite significant, whilst scalability allows existing large-scale resources to be used more efficiently.

In the embedded and mobile computing space, fault-tolerance and energy efficiency are important. Some embedded devices are used in critical systems (such as automobiles, spacecraft and medical implants) where fault-tolerance is paramount, and energy efficiency is almost always important for mobile devices, either to conserve battery life or to fit within a constrained energy budget.

Many processors provide facilities to adjust the performance of the processor in order to reduce energy usage or heat dissipation. However, this is complicated because the safe clock frequency, operating voltage and temperature of the chip are all linked, and the range of performance scaling is limited. The trend is towards putting components to sleep rather than adjusting performance [Le Sueur and Heiser, 2010].

2.3. The von Neumann bottleneck

The von Neumann architecture is a model of computer design characterised by a central processor, and a separate memory that holds both instructions and data. It is generally understood that the processor is powerful enough and the memory is large enough to perform substantial computing tasks by itself. Adding more processors that share the memory (as in multicore systems) is seen as a performance optimisation that introduces

difficulties into the model.

The von Neumann bottleneck [Backus, 1978] refers to the limited speed with which data in memory can be accessed compared with the much higher speed with which processors can process data. This bottleneck becomes yet more strained as the number of cores increases because the cores contend for the memory, which makes efficiently scaling to hundreds of cores effectively impossible for a von Neumann architecture machine.

Shared-memory models are generally not used in large-scale computing with many computers networked together. Instead, message-passing programming models are used, where compute nodes must explicitly exchange messages with other compute nodes in order to share data. Message passing exposes the burden of deciding which data needs to be physically ferried around the system to software. While this may make programming more difficult, it reduces or eliminates the von Neumann bottleneck. [Lauer and Needham, 1978] showed that OSes based on shared memory are a dual of OSes based on message passing.

In a Turing Award lecture, [Backus, 1978] describes the von Neumann bottleneck as an "intellectual bottleneck" that keeps us reliant on "von Neumann languages": programming languages which operate on one machine word at a time, and which embody the von Neumann bottleneck in the assignment statement. And conversely: "The dominance of von Neumann languages has left designers with few intellectual models for practical computer designs beyond variations of the von Neumann computer."

Remarkable advancements have been made in the field of programming languages since that time, many that seem in line with the claims made by Backus. Yet the von Neumann architecture is still dominant. Perhaps the availability of mature non-von Neumann languages and the mounting pressure to improve scalability will help us overcome our reliance on the von Neumann architecture. The availability of FPGA technology is allowing research into alternative hardware designs without huge investment, as in [Naylor

and Runciman, 2010]. Yet the colossal investment that has been made into von Neumann machines gives them a huge head start, and so to compete with them we need a model that is inherently much superior. Backus did mention dataflow as an alternative model, and the model seems attractive, but that doesn't make it easy to implement well.

2.4. Dataflow Graphs

A *graph* is a collection of *nodes* (also called vertices) and *arcs* (also called edges), where each arc connects two nodes. The salient features of a graph are the identities of the nodes, and which nodes are connected to which. The positioning of the nodes and arcs is not important. Two pictures of graphs represent the same graph precisely when the nodes are the same and arcs join the same pairs of nodes.

Graphs can be used to model many kinds of systems. The world wide web has a graph structure: web pages are nodes, and links between web pages are arcs. The Internet has a graph structure: computers and routers are nodes, and the communication channels connecting them are arcs. Social networks can be modelled using graphs in a similar way. There is a graph structure to many forms of data. Hierarchically organised data, such as files in a directory structure, is stored in a *tree*, which is a special kind of graph. Information stored in a relational database has a graph structure, for instance, students and classes could be the nodes, and enrolment information the arcs, in a database. Finally, many data structures commonly used by software have either restricted or arbitrary graph structures.

Computations can be represented as graphs where each node represents some kind of computation (indicated by the node's *label*), and each arc indicates a flow of data. Arcs have a direction to show which way the data flows. Most computations combine multiple input items together to produce one or more outputs, and often the identities of the inputs

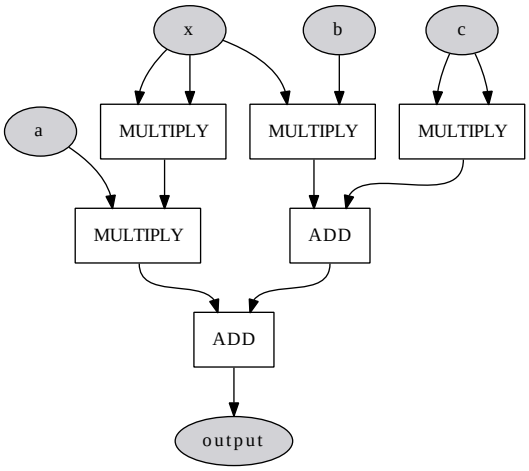

Figure 2.1.: Dataflow graph for evaluating $ax^2 + bx + c^2$.

and outputs are significant. For example, reordering the inputs to a subtract operation would change the meaning. So strictly speaking, each arc joins an *output port* of a node to an *input port* of another node (possibly the same node). Figure 2.1 shows a simple computation expressed as a dataflow graph.

A dataflow graph is similar to a data dependency graph, and similar analysis can take place. However, sometimes multiple sources can fulfil a data dependency at different times, which makes cyclic dataflow graphs sensible, as seen in Figure 2.2. This also allows nondeterminism in the execution (which arises from variations in computation times), in addition to any nondeterminism within the computations themselves. Cycles in dataflow graphs are similar to loops on sequential processors.

Dataflow graphs can be indefinitely composed to form larger dataflow graphs. Subgraphs can be collapsed into individual nodes, for example a graph which computes the greatest common divisor of two numbers can be represented as a single node in a more complex program. Groups of edges between two nodes can be collapsed into a single edge, along with the node ports involved, for example, groups of 32 edges which can each carry a single bit can be replaced with a single edge that represents a 32-bit machine word. This allows an arbitrary level of abstraction to be applied, without any change in

the representation we are using.

Dataflow is certainly an important part of how we think about software, but it tends to appear at a coarse-grained level. Streams, sockets, pipes and message passing are examples of this. Some programming environments are entirely dataflow based, such as National Instruments LabVIEW, and Yahoo! Pipes is a way to use dataflow to assemble the data sources and data processors on the Internet into arbitrary computations. Spreadsheets are a familiar programming model which is dataflow-oriented (and should be amenable to efficient implementation on a dataflow processor). Electronic circuits can also be viewed from a dataflow perspective. Circuit diagrams are essentially a specific type of dataflow graph.

By exposing a dataflow-based instruction set, dataflow processors allow us to make software dataflow much more fine-grained than it normally can be. The FGM allows dataflow in the software to map onto dataflow in the machine's circuits.

2.5. Dataflow Machines

There was a lot of interest in dataflow machines in the 1980s, with many implementations produced [Silc et al., 1998]. [Kavi et al., 1986] give a definition of a dataflow model of computation. The defining feature of a dataflow processor is that there is no "current instruction" (and therefore no program counter); instead instructions can execute whenever they are ready, that is, when their inputs are available. Machine code takes the form of dataflow graphs.

The obvious advantage of dataflow machines is that dataflow programs are easy to parallelise: the processor necessarily keeps track of which instructions are ready to run and so multiple computations can occur in parallel across multiple functional units. In Figure 2.1, the horizontal groups can be executed in parallel. This is obviously faster than

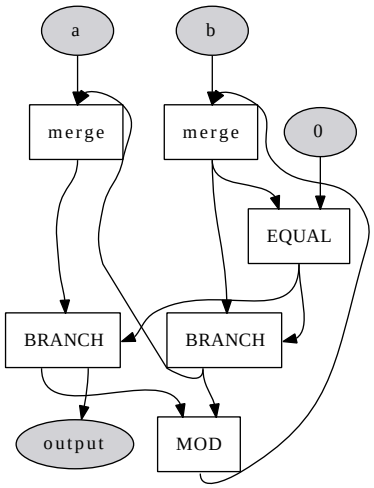

Figure 2.2.: Dataflow graph for calculating $\gcd(a,b)$ using Euclid's algorithm. The explicit merge nodes are not necessary, but added for clarity.

computing the multiplication and addition operations one after another. Whilst on a sequential processor this computation would be described by a sequence of instructions, modern sequential processors would reverse-engineer a similar dependency graph to Figure 2.1 from the instruction stream and use this to distribute the work between multiple execution units, thus making them capable of achieving similar performance to dataflow machines, at least on a simple example like this one.

Some dataflow processor designs only allow one data item (or *token*) per arc. The Wave Semi design (see Section 2.9) follows this principle. [Silc et al., 1998] points out that this limits the re-usability of code, since multiple iterations of a loop, or multiple invocations of procedure cannot proceed independent of one another in parallel, and it is suggested that this is a "serious drawback". I disagree: if you hope to beat the von Neumann bottleneck, then it is inevitable that you will need multiple copies of the program code in physically different places.

The alternative design, whereby tokens travel with "tags", as in the Manchester machine (see Section 2.6) seems to seal the von Neumann bottleneck into the implementation. Machines of this design are generally implemented with separate matching and

processing units. The matching unit receives all incoming tokens, and, assuming it is destined for a two-operand instruction, the unit attempts to locate a matching partner for the token. If there is none, the item is stored in expectation of a partner arriving later. If a match is found, the operands are combined with the instruction code and sent to the processing unit, which will allocate a free execution unit to the instruction. After execution, the outputs of the instruction would be sent back to the matching unit. In practice there were usually more stages in this pipeline, as show in Figure 2.3.

There are several problems with such a design. There is a scalability problem: the matching unit needs a global memory of tokens to check against, and the instruction memory is also global. This is essentially the von Neumann bottleneck in another form. The interconnects between the units may also have limited bandwidth — another bottleneck. The second problem is the length of the pipeline: the number of stages tokens have to pass through, and the amount of time they take, limit the execution speed of algorithms that are impossible to parallelise.

2.6. The Manchester Machine

One prominent experiment was conducted by the University of Manchester [Gurd, 1985]. Its pipeline contained a token queue, a matching unit with an attached overflow unit, an instruction store, a processing unit, and an I/O switch that linked the dataflow processor to the (sequential) host processor, shown in Figure 2.3.

As discussed in Section 2.5 we see bottleneck and performance issues with this machine, however, there is still much useful research behind this machine. Work was done on the assembly language and programming models for the machine, in particular single-assignment languages. The tagged dataflow model in the machine supported various types of interesting behaviour. It may be worthwhile investigating these behaviours and attempt-

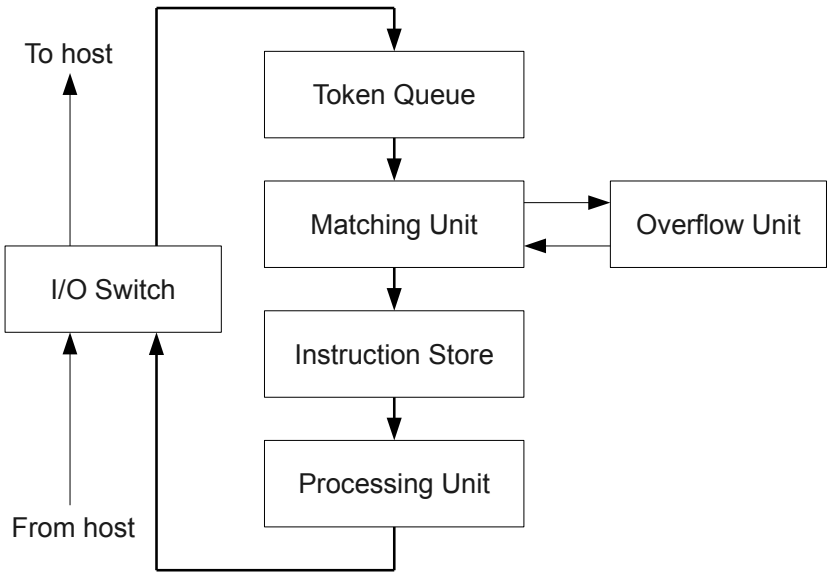

Figure 2.3.: Design of the Manchester dataflow processor.

ing to translate them to the FGM if they seem useful.

2.7. The Connection Machine

The Connection Machine CM-2 [Hillis, 1988] was not a dataflow processor in the usual sense. It had a sequential host processor that broadcast instructions to an array of processing elements (PEs) so would probably be classified as a single instruction, multiple data processor. However, these elements each had local memory and were part of a routing network enabling them to exchange data with each other. The prototype had 65,536 PEs each with 512 bytes of memory.

Hillis introduces the concept of *active data structures* — essentially graphs of processing nodes each with some memory attached, but perhaps better thought of as data structures with a degree of autonomy, or data structures where each component of the data structure is capable of performing simple computations and communicating with its

neighbours.

Hillis describes how such structures can be used to efficiently sort and perform matrix operations and other data-intensive operations, and mentions that they can be used for things like semantic networks. Trees seem a good encoding for many sorts of objects. If one wants to find the maximum value or sum of an unsorted collection of numbers, it is much faster if the collection is represented as a tree than if it is represented as a linked list, for example. Trees provide a good trade-off between performance and space consumption. With trees it is always possible to limit the memory required by each node by limiting the fan-out and increasing the depth of the tree, which is important when each PE has quite a limited amount of memory.

The primary programming language for the Connection Machine is a variant of Lisp, with a new type of object called a "Xector" which can contain code or data, and interesting operators for doing parallel work with them, somewhat reminiscent of pointer operations in the C language.

The name "Connection Machine" derives from the importance to the machine's designers of being able to dynamically form essentially arbitrary connections between any pair of PEs. The key to this was the machine's hypercube structure and sophisticated routing system, which guaranteed that a message could be sent from any PE to any other in a maximum of 12 hops (assuming no faulty PEs).

I think this machine is better thought through than the Manchester machine and have assimilated some of the motivations behind it. Hillis concludes with a commentary on computer science as a science, and, among other things, predicts that computer science will begin to look more like physics as we push the limits of computation. I think this passage captures the sentiment: "Machines will have three-dimensional connectivity because space is three-dimensional. They will have limited propagation rates because space has a finite speed of light. As less is wasted between function and implementation, the physics

begins to show through." [Hillis, 1988] Currently, processors are designed to hide physical considerations from software. Changing this seems like the way to more computing power, but that doesn't mean it is obvious how that change should be made.

2.8. NULL Convention Logic

The SD-ASIC is built out of an asynchronous circuit logic called NULL Convention Logic (NCL) [Fant, 2005].

Instead of a global clock, circuit elements co-ordinate locally by interleaving data values with a special value called NULL. This introduces a distinction between the presence of data and the lack of it, which enables circuit elements to distinguish successive presentations of data from one another without relying on a clock line.

When NCL gates are connected together, every data path needs a corresponding *acknowledge* (ack) path in the opposite direction to ensure that data items never collide. Essentially a "wave" of data propagates through the circuit, followed by a "wave" of NULL. The combination of data paths and ack paths results in a structure of partially overlapping cycles, or *coupled cycles*, where the overlaps create synchronisation between neighbouring cycles. One can imagine the cycles as interlocking cogs in a clockwork mechanism.

This structure of coupled cycles suggests a more general framework for concurrent computation [Fant, 2007], where elements co-ordinate with elements upstream and downstream and form part of cycles that are well-behaved, composable, and do not contain timing assumptions. A loop built out of coupled cycles is itself a cycle which can itself be coupled with other cycles, and this can be continued on even larger scales. Also, a new set of "gates" can be built and linked together like NCL gates are, essentially forming a new layer of a similar nature. The FGM is such a layer built out of NCL, and as such many of

the lessons learned at the circuit level also apply to programs that run on the FGM.

Timing assumptions are fundamental to a traditional processor, where all electrical activity must stabilise within a single clock tick to be sampled. The clock frequency is therefore limited by the worst-case propagation time on the chip. Improving process technologies are leading to ever smaller transistors. As a result the variation in switching times between individual transistors is increasing, and hence the worst-case switching time is becoming larger as a proportion of average-case switching time.

NCL circuits do away with almost all timing assumptions[1]. The top-down style of operation co-ordinated by a clock is replaced with a bottom-up style: essentially a web of feedback loops interacting with each other.

Clocked circuits exhibit a metastability problem: when sampling an input from the outside world or from a different clock domain, it can take an unbounded amount of time for the sampled value to reach 0 or 1. Processing an input that has not settled can cause metastability to propagate through a circuit, potentially leading to total failure. The probability can be rendered arbitrarily small by allowing settling time, but cannot be eliminated entirely. By contrast, asynchronous logic (including NCL) waits however long it takes for the input to settle [Bainbridge, 2002]. This not only removes the possibility of total failure (at the expense of sampling taking an unbounded period of time), but also decreases average sampling time to the average case rather than adhering to a pessimistic but "safe" time period.

NCL has some appealing properties from the perspective of energy management. NCL circuits tolerate very large ranges of operating voltage, responding to voltage changes by switching at different speeds. Reducing the operating voltage will reduce energy consumption at the expense of performance. NCL circuits are also very energy-efficient, and generate very little electromagnetic interference [Bailey et al., 2008].

[1] As far as I am aware, all timing assumptions are locally constrained within the NCL gate library and thus their effect does not influence larger circuits.

2.9. The Software-Defined ASIC

Wave Semi are currently developing the SD-ASIC for use in embedded computing, and an important part of this is the ability to compile legacy hardware description code to efficient flow graph programs. This will allow the SD-ASIC to take over the role of FPGAs and ASICs, hopefully delivering a solution that combines good performance with low cost and energy efficiency. They believe the SD-ASIC will in time be capable of fulfilling the role of embedded processors as well. This would allow a single general-purpose chip to perform the functionality currently spread over multiple chips. Wave Semi have indicated that the FGM should not be treated as a computational resource subordinate to a host processor (like a GPU or the machines previously discussed) but a host processor in its own right. As I state in Chapter 3, I suspect that the FGM may actually be *better* for controlling itself than using a separate processor to do this.

Further down the track, Wave Semi plan to move into high-performance computing, where workloads are frequently very parallelisable and much work is invested in implementing things efficiently. At this point it seems a usable operating system will become a necessity. Beyond that, the further the FGM is able to spread, the better.

While it is very important to have a feel for how the SD-ASIC works, a detailed description of its design is not important here. After all, its design is in a state of flux and this project is more concerned with what sort of software should be written for it, and how it should be written, than with actually creating software for the SD-ASIC.

The SD-ASIC is a 2D grid of many thousands of processor elements (PEs) each connected to its (eight or so) nearest neighbours. Each PE has a table where it can store a few hundred instructions. Each instruction specifies where its outputs are sent after it executes; the destination can be any instruction in one of the nearby PEs. Thus there is essentially a 3D lattice in which multiple graphs of instructions can be stored, but data flows between instructions can only cover spatially short distances. It is likely that there

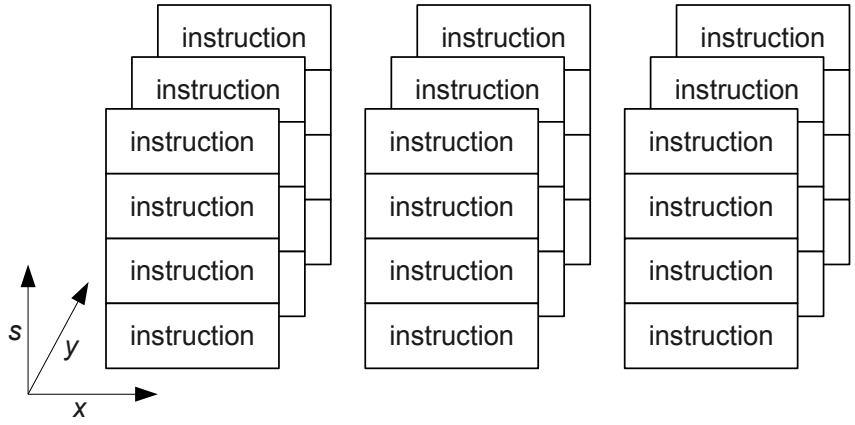

Figure 2.4.: Layout of instructions in the SD-ASIC.

will also be some longer links strategically placed, but clearly there will be limitations on how long they are or how often they appear.

To illustrate, a group of 9 simplified PEs is shown in Figure 2.4. The PEs are distributed along the *x* and *y* axes, and the instructions within the PEs are indexed by the *s* axis. In an SD-ASIC program, data flows between instructions can extend a maximum of 1 step in the *x* dimension and 1 step in the *y* dimension. Unlimited travel in the *s* dimension may occur.

The limited length of flow graph links enforces a kind of metric on flow graph programs that reflects the underlying physics of the machine. This, combined with the use of feedback loops rather than global timing seems like a partial response to Hillis' prediction (Section 2.7).

A few kilobytes of memory (SRAM) are attached to each PE. This is where the instructions and their inputs are stored. As inputs arrive, they "park" within the instruction they are destined for. When all of an instruction's inputs are present, it is ready and the PE will execute it when it is not otherwise busy (subject to some kind of hardware scheduling policy). After execution, the outputs of the instruction are immediately sent off to their

destinations. At the assembly language level, the SD-ASIC is akin to a distributed system with asynchronous unbuffered message passing.

The SD-ASIC currently has two data types: 16-bit words (signed or unsigned integers), and Boolean predicates. This is due to the applications the SD-ASIC will be used for in the immediate future and larger word sizes or floating-point support might be added to the architecture as needed.

There is no global synchronisation (no clock). There is no assumption of a global memory or bus, and no global addresses. This helps make the processor inherently scalable. In practice we may often want an off-chip memory and a bus in order to communicate with other components, however I expect this will be more for economic reasons that technical ones. The important thing is that the von Neumann bottleneck has been practically eliminated: adding PEs means simultaneously adding processing power, memory capacity and memory bandwidth. Access to external memory and device I/O would not be possible from all PEs, but would probably be limited to the edges of an array and possibly some special nodes within the array, since connecting all nodes to a bus or other communication channel would add complexity to the circuit, which seems unnecessary as the parts of a program which might require access to external resources are usually known ahead of time and so can be located appropriately.

The lack of global synchronisation means there is no sampleable state space. For instance, at any one time some NCL gates may be in the process of switching. However since traditional multiprocessors routinely violate expected state space transitions due to out-of-order execution etc., this doesn't seem like much of a problem.

As with NCL, SD-ASIC assembly programs need to be "wired" in such a way that ensures data items will never collide. This is done through the use of coupled cycles and not using timing assumptions. Malformed programs should not be executed. It seems best for compilers etc. to work at the level of coupled cycles right until the last step of code

generation.

The SD-ASIC allows nondeterministic choices to be made between multiple inputs, essentially choosing the first that arrives. This is useful in decoupling the synchronisation between multiple components that do not need to be synchronised. One example might be for a subroutine that may be used in multiple places in a program.

[Silc et al., 1998] mention that token traffic is doubled in single-token-per-arc machines, due to every data flow needing an ack. This is not entirely true of the SD-ASIC, which can encompass multiple instructions in a single cycle, thereby decreasing the amount of effective ack traffic.

It is my hope that this design will expose the parallelism of hardware to software, making it much easier for performance-critical software to benefit from parallelism, without making software that is not performance-critical any more difficult to develop. Compile-time optimisations should allow code written in a sequential style to still take advantage of the parallelism of the hardware. However, programmers should be able to resort to a dataflow style when the occasion calls for it.

I believe adopting a dataflow programming style when performance is necessary would be easier than, say, writing assembly code for the Itanium. With the SD-ASIC, one may resort to data flows at any level of a program. Concurrency and local co-ordination are first-class citizens of the processor, and thus concurrent programming need not be the difficult, dangerous activity we expect it to be.

[Fant, 2007] argues that concurrent software is not inherently difficult to write and reason about, rather that our choice of initial assumptions makes it artificially difficult. If one starts with a network of nodes that use communication both to exchange information and for local synchronisation, rather than a global memory and clock, then concurrency can become orderly and natural. [Lee, 2006] makes some similar arguments from a different viewpoint. The design of the SD-ASIC appears to be well adapted to executing

concurrent programs written in the styles suggested by Fant and Lee.

There are instructions in the current architecture for reading and writing the local memory of a PE. This includes modifying the instructions that may be stored there. This means that code that creates or modifies other code should be possible. There are no instructions for accessing external memory, so pointers essentially don't make sense beyond a single PE.

The fact that the SD-ASIC is wired much more like a network of neurons than a von Neumann computer suggests that we may be able to compete with human intelligence with such a machine. It will certainly be useful to take inspiration from physics and biology when trying to come up with OS techniques within the machine.

2.10. Operating System Architectures

Operating systems generally consist of a *kernel*, which is a trusted portion of software with full control of the machine, and *user-level* components which are only trusted to perform certain functions. Hardware protection features, for example memory protection and certain instructions which can only be executed by the kernel, are usually used to prevent untrusted programs from interfering with other programs or taking control of the machine (but see [Wilkinson et al., 1981, Hunt et al., 2007] for examples of systems that do not).

The kernel is completely trusted which means that bugs in the kernel can easily crash the whole system, and security holes in the kernel allow malicious programs to take control of the system. A similar problem is also seen outside of the kernel: if a bug in a program is triggered or the program is compromised, then anything the program has influence over might be affected. These problems are addressed by designing OSes with less code in the kernel, and by separating programs into components which have limited in-

fluence over each other (both inside and outside the kernel). The THE multiprogramming system [Dijkstra, 1968] used a layered designed for the OS, and later MULTICS [Corbató and Vyssotsky, 1965] used hardware protection to enforce such a layered model. However, the hardware protection that is used to enforce the separation between components can result in performance penalties, so taking this approach to the extreme can be infeasible.

Monolithic kernel systems place many operating system components, including device drivers, in the kernel. *Microkernel* systems reduce the size of the kernel to a minimum, pushing various operating systems components to user level. The Linux kernel is monolithic. Modern versions of Microsoft Windows and Mac OS X are based on so-called "hybrid kernels", which are fairly large kernels that generally include device drivers, however the kernels are internally componentised similarly to microkernel systems. Mach [Accetta et al., 1986] and L4 [Härtig et al., 1997] are examples of microkernels. EROS [Shapiro, 1999], Mungi [Heiser et al., 1998], and MINIX 3 [Herder et al., 2006] are examples of operating systems built on top of microkernels.

Component architectures make building an operating system easier, particularly on microkernel platforms. They allow the use of models and tools to help automate the assembly of operating systems and other software components into complete systems and analyse such systems [Heiser et al., 2007].

Capability systems are based on a security mechanism involving the use of *capabilities* [Dennis and Van Horn, 1966]. A capability is both a reference to an object and the right to perform certain operations on that object. EROS and operating systems based on the seL4 microkernel [Klein et al., 2009] are capability systems, and there is prior and ongoing work on building reliable systems using capability models.

Single-address-space operating systems (SASOSes) perform a pervasive unification of the machine — in a SASOS, all tasks run in the same address space, and resources

such as I/O are accessible through this address space. Data objects have a single name (their memory address) across all tasks and all compute nodes in the system, and no distinction is made between volatile memory and persistent storage — objects persist until they are explicitly deleted. Examples of SASOSes include Mungi [Heiser et al., 1998] and Opal [Chase et al., 1992].

It has been argued that a SASOS provides a good platform for programming distributed systems, and some SASOSes designs considered the distribution of the OS across many compute nodes which all shared the same address space [Skousen and Miller, 1999, Heiser et al., 1993]. The view of the FGM as a special kind of distributed system suggests we may be able to apply some SASOS principles.

In the FGM, it will be expensive (and perhaps impossible) for any part of the machine to have direct control over every aspect of the machine. This raises questions about what the word "kernel" actually means on such a system and whether it is appropriate at all. Breaking a complex system into components seems very well suited to the FGM. More will be said about capability systems in Chapter 5. SASOSes suggest a route for unifying the FGM, by giving everything a unique global name. The use of fixed-width global addresses doesn't sit well with the FGM philosophy, though. I think that the parallelism of the FGM will make cataloguing and searching very efficient, which will provide an alternative route to managing the activity and information within the FGM.

3. Exploration

Rather than picking out a small, well-defined problem and solving it, I chose to conduct an extensive exploration of the problem domain at several levels, so as to help determine what sort of problems there are to be solved, which are the most important, and what kind of approaches might be taken to solve them. I hope this coverage gives a good starting point for deciding how to approach further work.

I tried to identify things we might want to achieve, understand the resources we have, and explore ways of using those resources to achieve those things. In searching for approaches to solving problems, I favoured those which could be done in the FGM and which were true to the local-only philosophy underpinning the design of the FGM. Such approaches are likely (but not guaranteed) to be scalable. I also looked for alternative approaches.

Initially I suspected that many of the problems which might appear difficult about the FGM, such as how to efficiently map multiple program graphs into the machine, might be less difficult if we tried to solve them within the machine, taking advantage of the massive parallelism it offers. For example, even though software for embedded systems is normally compiled, assembled and linked on a more powerful computer, perhaps some or all of that task would work far better on the embedded FGM being targeted, or on a more powerful FGM, rather than on a sequential computer. During the course of this project I have only become more convinced that this is the right approach.

3.1. Potential Applications

It is hoped that the FGM will be able to take over the role of sequential processors (including data-parallel processors like GPUs), as well as being able to compete with FPGA and ASIC chips. If this is the case, then the FGM's applications could extend to all types of devices which contain any kind of processor or signal-processing hardware. The list would include embedded, robotic, mobile, desktop, enterprise and scientific computing. Embedded computing is a very broad category, including such things as automotive, aerospace and military applications, household appliances, computer peripherals and medical implants.

Probably a more pertinent thing to consider is the areas that the FGM will excel at. The FGM is a multiple instruction, multiple data machine with significantly lower forking, merging and communication costs than a multicore processor. Unlike vector processing units, it does not require problems to have a regular (e.g. matrix-based) structure. This should allow it to solve problems with irregular parallelism more efficiently than other machines can. Additionally, the FGM appears to be easier to write software for than FPGAs and ASICs since less hardware understanding is required, and easier to write software for than GPUs and vector processors since it has a more flexible programming model, supporting instruction-level parallelism, thread-level parallelism and data parallelism while GPUs and vector processors only excel at highly data-parallel tasks.

Functional languages are based on a graph-reduction computational model, and attempts have been made to perform this more efficiently than can be done on sequential processors using an FPGA [Naylor and Runciman, 2010]. It seems intuitively very difficult to use vector machines to parallelise graph-reduction problems. The manipulation of symbol systems, for instance within interactive theorem provers, seems like it fits in a similar category. I suspect the FGM would be a good platform for solving these kinds of problems.

The FGM seems like an ideal platform for implementing things with mainly short-distance communication. Many signal processing problems, such as video processing, and physical simulations, such as computational fluid dynamics, involve a lot of computations that only work with small pieces of the problem and hence are a perfect fit for the FGM. Neural networks should also be easy to implement efficiently on the FGM. Graph problems, such as finding the shortest path between two places, should also be easy to solve on the FGM. By building a program that has the same shape as the graph being explored, we can take advantage of the parallelism of the FGM.

Designing parallel algorithms can be difficult. Neural networks and genetic algorithms are ways of letting the computer develop its own parallel algorithms through trial and error. These sorts of techniques might become more important as machines like the FGM allow us to scale up the amount of parallelism available in our computers.

Amdahl's law tells us we need to worry about sequential execution speed. In the FGM, sequential programs do not force many cores to sit idle, unlike vector hardware. Also, since the FGM can take advantage of instruction-level parallelism, it does not seem to be at a big disadvantage to sequential processors. Therefore sequential execution does not seem to be an inherent problem.

3.2. Scope

We have seen that the FGM might find uses in many quite different environments. It would be extremely beneficial if a single OS could be adapted to all these uses, rather than requiring multiple separate OSes. The Linux kernel is capable of running on a huge assortment of different hardware and is used in all of the categories mentioned in the previous section, so this does not seem to be an unrealistic goal, and it is something I have aimed for. However, all those potential applications make for a lot of things to consider,

so it makes sense to define some more specific applications of the OS to concentrate on whilst considering its design.

I think it is a good idea to focus on scientific computing, since scientific computing needs the performance and scalability, many kinds of simulation are ideal for the FGM, and Wave Semi are interested in expanding into this area. I think I should also keep embedded computing as a secondary consideration, as this is Wave Semi's focus at the moment, and innovations that can be applied there might be rapidly tested and potentially bring benefits quickly.

Considering both scientific and embedded computing will help us look at what we are doing from two different perspectives. Scientific computing will place greater demands on an operating system. Since problems are larger and often less well-defined than on embedded systems, things like dynamic memory allocation and graph manipulation will be more important. Since it is not known in advance what tasks will be run or how long they will take, more will be required by way of task and resource management.

Since I believe that using the FGM to solve FGM problems is an appropriate tactic, I want to also think about how determining program layout, loading programs, unloading programs, and even compiling programs might be performed from within the FGM.

3.3. Understanding Flow Graphs

Understanding flow graphs and how they are put to use in the FGM is important to being able to write low-level programs and ensure that suggested OS features actually meet all the needs of programs and hardware. However, this understanding is difficult to develop without hands-on experience with writing FGM programs, and does not need to be very deep for most questions of OS design. In this section I will cover what I have learned about flow graphs.

Modern processors break up instruction execution into multiple stages, and the chain of stages is called a pipeline. In one sense, the FGM can be thought of as a fully programmable pipeline. Acyclic (i.e. loop-free) dataflow programs can be thought of as a series of stages, allowing multiple instances of data to be passing through at a time. For example, in Figure 2.1, three separate instances of the variables (x, a, b and c) might be provided as inputs before the first sum is output. Eventually all the sums will be output, in the order corresponding to the order the input variables were presented in. Note that the graph in Figure 2.1 does not directly correspond to the SD-ASIC's assembly language, which requires extra nodes and arcs to facilitate the co-ordination necessary to meet the one-token-per-arc condition.

Graphs may contain cycles which can obviously be used for iteration, as in Figure 2.2, but I also believe they will be important for active data structures (explained in Section 2.7), for example a dictionary that can have items added, removed, and be queried, which internally might use a balanced binary tree.

There is an interesting connection between the study of dataflow graphs and process management. Process management involves trying to efficiently co-ordinate processes which transform inputs to outputs within resource constraints (finances, capital, labour, space, time). Dependency analysis is one of the problems of process management, cf. Gantt charts.

Many formal models for concurrency exist. The FGM is a partial order machine, in other words we cannot always decide the order in which two instructions are executed. Some interesting models seem to be Petri nets, trace theory, the actor model, and some process calculi [Baeten, 2005], in particular join calculus [Fournet and Gonthier, 2002]. Communicating Sequential Processes is a process calculus that seems rather inappropriate given the FGM design. Fant proposes a language for describing concurrent systems [Fant, 2007] which is heavily related to FGM principles, but little is provided in terms of formal

semantics or reasoning. [Kavi et al., 1986] give a definition of the dataflow model.

At the assembly level, FGM programs rely on a cycle structure where a cross-section of each cycle is initialised with data (I say "cross-section" since cycles can contain branches and merges), and this condition is maintained for the life of the program.

An abstract view of this (as used in Figure 2.1 and Figure 2.2) can hide the acknowledge paths and presents each "forward" arc as a buffer which can either be empty or hold a single data item. If more buffering is required, buffer nodes can be added. Other abstractions might be possible and potentially more efficient. More abstract graphical views of computation are possible, for instance where each node represents an entire program and arcs represent communication in between programs as well as with the outside world. There are many choices for means of co-ordination and buffering.

Mathematically proving that a "pipeline" graph does what it is supposed to do seems as easy as converting the graph to a mathematical function that it is equivalent to. Proving that more general graphs behave correctly, for instance proving that loops eventually produce output, requires a consideration of the evolution of program execution. In the most general case, I think, one would want to reason about a partial ordering of the inputs and outputs of the graph, because you may only be able to derive a partial ordering of the outputs of one graph, which may then be used as inputs to another graph, or even influence future inputs to the same graph! Because of the piece of data that must be present in every cycle, graphs can quite easily have state-holding behaviour, even if we assume that instructions cannot read and write local memory. However, it should be possible to assume that a graph cannot modify any external state (except as a result of producing an output).

If we think of a pipeline, it is clear that for maximum efficiency all stages of the pipeline should take a similar length of time. If one stage is significantly slower, it will form a bottleneck, with inputs banking up against it, leaving prior and later stages under-utilised.

Introducing multiple parallel copies of the offending stage might alleviate the problem. Additionally, if consecutive stages are misaligned, horizontal sets of data may be delayed. Consider that in Figure 2.1 the input a cannot progress until x^2 has been computed, which means a's source might stall with a on the output arc. The solution to such a problem is to add a buffer node. Problems like these have been studied in queueing theory.

Given the implicit buffering each edge provides as well as any explicit buffer nodes, a pipeline computation has some "slack" — if input is presented on its inputs faster than it is consumed on its outputs, then the computation starts to accumulate more in-process data; if data is consumed on the outputs faster than input is presented, the data in the pipeline will drain out. This can help smooth out fluctuating levels of inflow or outflow on the computation.

The *kanban* system, invented by Ohno for Toyota [Ohno, 1988], appears to be a real-world implementation of the initialised-cycles system. It is a form of distributed process control, where parts are produced if a kanban card is present, and the card travels with the parts to the consumer of the parts, and when the parts are consumed the kanban is sent back to the producer.

One of the driving design principles in NCL and the FGM is *no timing assumptions*. However, this does not equate to no timing analysis. Indeed for the purposes of pipeline optimisation, timing analysis is important. However this is generally concerned with the average case rather than the worst case.

If we unroll an execution of a dataflow program into a data dependency graph (the equivalent of an instruction trace), we can easily find the longest path through the graph and this becomes our worst-case execution time. This assumes we know how long each instruction takes (which we can perhaps come up with a worst case for, but the worst case will probably be quite bad), and that instructions are executed as soon as they are ready, or at least some kind of fair scheduling where we can bound how long it will take for an

instruction to be executed (which will probably be extremely pessimistic). Additionally, due to nondeterminism, the space of possible trace graphs may be huge. This becomes more complicated under voltage scaling.

In process management, one does not have complete guarantees on how long elements of a process take, and usually one needs to be able to cope with failures. Still, one optimises for the common case and uses estimates of duration to do a best-effort arrangement of tasks and an estimate of how long the entire process takes.

3.4. Rethinking the Role of an Operating System

In this section I will discuss the things an operating system is typically responsible for, and examine their relevance to the FGM and explore what form they might take, in light of the FGM hardware and the applications it may be used for. The radically different hardware has a significant impact on many things, and will open up new possibilities. In a sense we are returned to the infancy of operating systems: the rules are not yet written and we have the opportunity to shape the next generation of computing.

The output of a dataflow program must flow somewhere — to another program, into some OS code such as a device driver, or directly out of the processor. If it flows into another program, then there is at least one arc between the two programs. If it flows into the OS, then there is an arc between it and the OS code. This means that a dataflow graph can conceivably encompass all the user programs and system software on the processor. Thus that the boundaries between programs and the operating system will be blurred. This suggests that the kernel/user mode privilege distinction may be too simplistic, and that the notions of system calls, services and libraries may need to be extended or replaced.

Also, the fact that software has an explicit graph structure rather then being stored as a formless blob means that there may be more opportunities for optimisations and

security checks at the point of program compilation or loading. Loading programs into the processor and laying out their instructions so as to make the best use of resources, as well as halting and unloading programs, and instrumentation, debugging, and code generation, all seem to involve graph manipulation. I believe that graph manipulation will be an important activity in the machine (I expand on this in Section 3.7) and it may be a very appropriate OS "service", not to mention a tool used by the OS to manage other programs.

Specifically the ability of instructions to modify PE memory in the SD-ASIC makes me wonder if it would be possible to relocate instructions belonging to a program while the program is running in the FGM, or even dynamically add new ones. When it would be beneficial to move a computation, either to take advantage of underutilised PEs, or to bring it closer to a resource it needs to access, it would be convenient if the OS or hardware could do that transparently. This would effectively make all code position-independent.

We have a rare chance to start from scratch, to leave behind some of the legacy of bad decisions (such as bad instruction sets), and to rethink things and challenge assumptions. At the same time we must not make it too difficult for people to transition to this technology, which means we must consider compatibility with existing systems, and must not be so arrogant as to reject sequential programming languages and the like.

3.4.1. Hardware management

Device drivers

To provide a hardware-independent interface for other software, an operating system provides device drivers to interface with the other devices in the machine.

Perhaps in time such devices will adopt NCL and dataflow! Perhaps they will integrate directly into the array of PEs. However, for the moment the OS will definitely need to be able to communicate with legacy devices over some kind of shared bus.

This does not seem like a big challenge, though some issues like timing might need careful consideration.

Interrupt handling

Hardware interrupts fit naturally into an asynchronous data flow model, particularly as most hardware platforms include an interrupt-acknowledge procedure. However the word "interrupt" becomes a misnomer, as they no longer interrupt anything. They would simply be hardware notifications that the OS responds to. The FGM may even be able to fulfil the role of an interrupt controller (see Section 3.5).

Fault tolerance

Dealing with hardware failures in a way that causes minimum software disruption is an important role of an OS. Since dataflow programs do much in parallel, they may accumulate long pipelines of work that needs to be discarded if, for instance, a network connection closes abruptly.

Given the thousands of homogeneous PEs on a chip, and the flexibility of flow graphs, it is conceivable that faulty PEs will be tolerable (provided they can be detected and routed around). The ability to cope with a small number of faulty PEs would bring down the cost of manufacturing the FGM. The OS might need to help support such fault tolerance.

Resource management

Two common OS activities become quite different in the dataflow processor, namely processor scheduling and memory management.

Processor scheduling becomes more of a spatial than a temporal problem. The OS does not need to time-multiplex PEs as they do this themselves. However, the machine requires communicating computations to be spatially proximate.

Memory management becomes tied to processor scheduling and becomes a largely

local problem. Virtual addresses are not really necessary because addresses are local, and in Section 3.7 I suggest making references opaque, which would make memory addresses entirely inaccessible.

To allow arbitrary data structures to be created, some form of memory allocation will be necessary. In Section 3.6 I discuss approaches for controlling allocations.

Management of external resources such as external RAM, flash memory, disk space, human-computer interfaces, and networking will probably involve a combination of traditional methods like file systems and access rights and new abstractions which are appropriate for the FGM like presenting external data as graphs or data flows. Questions of how graphs and data flows are represented on external media will need to be addressed.

Hardware protection

Hardware protection mechanisms such as kernel mode and memory protection may be unnecessary. The OS loader could ensure that a program contains no "sensitive instructions" when it is loaded, and if the machine does not have addressable memory then no memory protection is required either. This question will be discussed further in later sections.

Perhaps some form of hardware protection would simplify OS implementation. However I am not sure what such protection might look like.

3.4.2. Multiplexing and abstraction

Procedures, system calls and libraries

Procedure calls, and, by extension, system calls, are not first class operations in a dataflow processor. Rather, the invoked code must form part of the data flow — akin to being inlined. Obviously a request can be serviced concurrently with a

program's continued execution. Perhaps a system service should be viewed as part of the program.

Of course, programming languages can provide the appearance of synchronous system calls, either by inserting explicit sequencing or by attempting to preserve the appearance of synchronicity whilst taking advantage of parallel hardware.

Inlining procedures, system calls and libraries everywhere they are used wastes space if they are not heavily utilised. Instead, instances of procedures or libraries could be shared between an appropriate number of clients that require them, and the OS could be responsible for creating new copies, and cleaning up and merging existing ones according to demand. Choosing the right granularity for this may be challenging. I see this as something of an analogue to hardware and software caching on traditional multicore and distributed systems.

Tasks, processes and threads

In order to control and manipulate computational work and the resources allocated to it, users and the OS need a "unit" of computation by which to delineate a logical grouping of instructions and data that is coarse-grained enough to be manageable, something like a "task" or "job".

This is of particular import in a machine where the boundaries between interacting programs, libraries, and the OS can be very blurry. For example, a program is not only a graph in itself — it is a subgraph of a larger graph. The program, including any OS glue code right up to the point where data is sent to and received from outside the processor, as well as all programs it directly and indirectly communicates with — in other words, possibly all instruction is in the FGM — together form a graph. There are many ways to divide this graph up into subgraphs — most of them useless.

It is doubtful that we need threads, which imply a unit of sequential execution, as an abstraction, although it would be best if threaded programming models could be adapted to run on whatever abstractions will be provided.

One idea is to summarise the tasks in the FGM in terms of their important components and the data flows and dependencies between them. This might be represented as a hierarchy of components. The hierarchy might be organised into logical groupings of code, building up from subroutines to modules to processes to the tasks within the machine. The hierarchy could be stored as a data structure in the FGM, possibly going all the way down to the program's actual instructions. This could be particularly useful for instrumentation and debugging.

Separation and virtualisation

Most operating systems present the illusion that a program is running in a machine of its own. What does this actually mean? It means that the program has a private memory address space and set of registers. In the FGM it seems that as long as programs cannot conspire to direct their data flows into other programs, and any use of local memory is somehow kept restricted to a single program, separation is enforced.

The Burroughs B6700 computer used trusted compilers to ensure that programs were incapable of interfering with each other by construction, however such a system required only a small flaw somewhere in the system in order to gain control over the system. A successful attack is described by [Wilkinson et al., 1981]. The problem with this system is that there is no mechanism to verify that a program is trustworthy — it is either trusted or it is not.

The Singularity operating system [Hunt et al., 2007] uses programs which are guaranteed by construction not to interfere with each other in order to remove the

need for expensive hardware protection and complicated management of shared resources. Singularity uses cryptographic techniques to verify that a program was generated by a trusted compiler, which is a big improvement in trustworthiness but still relies in the compiler being correct.

In the FGM, it could well be possible to inspect the entire program graph of a program when it is loaded and ensure it has no instructions that could be used in a dangerous way. This requires the instruction set to be designed in a way that makes this plain. This will be developed further.

The FGM should be fairly good at executing programs with different instruction sets — dynamic binary translation seems like an efficient way to do things. The machine should be able to efficiently simulate NCL and probably electrical circuits by building programs that take the shape of these circuits.

Virtual memory, swapping and persistence

If it is possible for the OS to move individual instructions around in the processor without disrupting a program's execution, it should be possible for the OS to serialise a graph and save it to external memory. Thus an entire program state can be saved and loaded without any fuss. If data structures are represented as graphs then swapping data to external memory (in order to free up more processing nodes) can be done more intelligently as it is possible to examine the distance between a computation and its data, perhaps in a similar way to that described by [Mazzola Paluska et al., 2011]. This means that memory scheduling may not be the problem it is on von Neumann machines [Denning, 1968].

Shared memory

Shared PE-local memory without synchronisation is downright dangerous. Any sharing of memory should be synchronised in the same way that use of shared code

is synchronised. When it comes to external memory we might resort to classical techniques like file locking.

Events

Event-based programming fits naturally within the dataflow processor, but events do not need to be handled sequentially. Asynchronous notifications, which have traditionally been clumsy, are a fundamental feature of the hardware.

Periodic or time-delay events would be important for certain applications. This might involve providing an abstraction over an external timing source, or regulating access to timing sources in the FGM.

Files, pipes and sockets

These seem like timeless abstractions. Certainly pipes and sockets can fit very well in the dataflow model. Files are traditionally seekable, which is a bit more complicated but certainly achievable.

A stream of data can be sent around serially (as a single dataflow) or in parallel (as many parallel dataflows of one data item), or something in between. The appropriate choice depends on whether it *can* be processed in parallel, and how many resources we wish to dedicate to it.

I think graphs should be a different type of primitive object that can be stored and retrieved.

The FGM fabric might not provide a convenient way for spatially remote programs to communicate with each other. The OS might be responsible for providing a routing network as an abstraction over whatever long-range links the FGM might offer. This might tie in with a socket abstraction of some sort.

I/O caching and prefetching

Caching becomes more challenging as programs that wish to access the cache may

need to be located closer to it. Perhaps caches will be subjected to the same treatment as other components which can service multiple components as mentioned above.

When I/O is asynchronous, prefetching seems like something that could potentially be exported to a language runtime.

Heterogeneous hardware

Operating systems provide abstractions over hardware that provide a common interface to hardware despite the details of the hardware differing.

Different areas of the human brain appear to be dedicated, or at least better suited, to certain tasks. Perhaps in the FGM we will see different domains of the processor better suited to different tasks — some domains might be faster, some might have more memory, some might consume less power, some might not have support for floating-point arithmetic. The OS might migrate programs around and even translate them to make the best use of resources.

Another option would be to scatter heterogeneous PEs throughout the array. This approach seems to make it easier to arrange programs in a way that suits their needs dynamically, since decisions about moving a program to where available resources are would be based on local information.

On the other hand, many programs exhibit a tendency to use particular resources. Consider that a program with large memory usage and low needs for processing power would do best if it were running *only* on PEs with more memory, or that a numerical simulation would make extensive use of floating-point arithmetic operations, while a web server would use virtually none. Thus creating domains of similar PEs might be sensible.

3.4.3. Services

Inter-process communication

The hardware is built for allowing communication by passing messages between software components. What sort of abstractions will be presented to help take advantage of this needs further investigation, however there is sure to be a lot we can learn from distributed systems and component systems.

Debugging, instrumentation and auditing

Without sequential execution, debugging becomes more difficult. This is a necessary consequence of embracing parallelism at all levels of a program. The ability to single-step a program written in a sequential language, even when it has been optimised to exploit concurrency, seems desirable.

However, presumably copying the entire state of a program is fairly cheap in terms of time, because the act of copying can take advantage of parallel hardware. This suggests that checkpoint-based debugging could make a lot of sense.

Instrumentation will also be very important, particularly as it is more difficult to reason about timing in the FGM. Collecting and summarising information from all over a flow graph is something that needs further exploration.

The ability to measure PE utilisation, activation counts and so on will also be useful in terms of laying out dataflow graphs efficiently, and for auditing and accounting purposes.

Methods of annotating code could be quite important for the purposes of debugging and instrumentation. Instructions may form part of a larger data structure.

I would like to mention that the operation of the human brain is investigated by measuring activation with MRI and similar machines. This kind of visual approach to debugging or instrumentation might sound too impractical to be of much use, but

it might be useful for large-scale problems, such as large self-modifying programs, and resource allocation for the entire collection of programs in the machine.

Energy management

The OS should assist in taking advantage of the energy/performance flexibility of NCL (see Section 2.8) to balance energy consumption and heat dissipation with performance.

The usefulness of single-ISA heterogeneous multicore architectures for energy management have been demonstrated [Kumar et al., 2003]. An array of PEs (which do not need to have uniform performance characteristics) whose energy usage and performance can be scaled in whole or in part, quite possibly with the option of putting large domains of the array to sleep, sounds like an even more flexible solution. Presumably the OS would be able to take advantage of that.

Namespaces

Although the hardware tries to avoid global addresses and the like, the OS will need to have some kind of control over the entire machine. Users and administrators will want to list tasks, and refer to them by some kind of name to start, stop, debug, and manipulate them.

Some kind of namespace of software components might be useful so that systems can compose themselves dynamically. Namespaces of users and devices seem useful as well.

The obvious namespace provided by most operating systems is a file system. We must take into account that fast searching of vast amounts of data could be practical, and that semantic connections and classifications might be more easily stored in the machine. Possible choices include a flat address space as in SASOSes, a hierarchical file system structure, a more complex graph-based namespace, perhaps based

on semantic relations, or a formless easily searchable space of objects.

Keeping these namespaces and operations on their referents coherent across the processor without introducing unnecessary delay or inconsistency might be a challenge. Replicated copies of objects will be a must. Established distributed systems techniques like two-phase commit [Gray, 1978] are likely to be useful and the experience of SASOSes may help us.

Security

Dataflow programs that do not access PE-local memory are unable to interfere with other programs. I think it would be possible to extend this even to programs that modify themselves, so that they are secure *by construction* rather than due to hardware security enforcement. Changes to the instruction set may be necessary to make this work. However, perhaps it would be easier if some kind of hardware privilege enforcement were provided. This is further discussed in later sections.

There is no reason why things like filesystem security policies and sandboxing cannot be implemented in the OS for a dataflow processor. Further, things like fine-grained taint tracking should be possible to implement where the dataflow structure of programs is so plainly exposed to the OS.

3.5. The Outside World

While most programs accept some form of input when they begin and many continually accept input during operation, every single useful computer program produces some kind of output. It can take many forms, including text, graphics, sound, network packets, or control signals to a robotic system.

The FGM is a system of interlinked feedback loops. Some of these loops include elements that are outside the FGM, for example a loop that passes through an electric

motor to a shaft encoder before returning to the FGM.

This view seems conceptually more challenging than a view where the outside world is subordinate to the computer, where the computer pulls in data when it is ready (by reading device registers) and pushes data out as it calculates it (by writing to device registers). Instead, inputs can arrive at any time. Nothing stays fixed for a predictable clock cycle. There is nothing to "stand" on while contemplating the behaviour of the system.

Of course, it is possible to emulate the behaviour of a traditional computer: an FGM program *can* be constructed so that the entire program requires a special signal to progress — a clock done in software rather than in the hardware. Hardware inputs can be synchronised as soon as they enter the FGM. This is not an efficient way to operate but it shows that the FGM is at least as general as a traditional computer.

In a traditional computer, sometimes external devices do send notifications to the processor without being asked: *interrupts*. Traditionally these are handled by a device called an *interrupt controller* which prioritises and masks incoming interrupt signals. Since the FGM can deal with asynchronous inputs, using nondeterministic choices, to my understanding it is possible for the FGM to perform the functions normally performed by an interrupt controller. In other words, the FGM is general enough to implement an interrupt controller in software!

I believe that having an understanding of the role of input and output for the FGM is important for the design of an operating system, as the OS will have to mediate access to the outside world. In particular it may need to place device drivers between programs and the outside world, and the edge of the chip is also probably where programs are loaded and saved, and where commands from the user such as commands to terminate programs, enter the FGM. Developing such an understanding will probably require actually constructing software that has to deal with I/O.

A program may not send its output directly to the outside world, but instead send it to

another program. The interfaces between different programs, and between programs and the outside world, seem like places where the OS will need to be active, and might turn out to be good places for the OS to base its control over programs.

3.6. Management

An operating system needs to manage hardware resources and software which uses those resources. The challenge with the FGM is that each instruction has a small area of influence. How do you manage something that is beyond your reach?

In most real-world management structures like government and corporate management, no single person collects all the information, nor makes all the decisions, nor commands everyone else. Instead, information is collected and summarised by various people before being passed on, decisions are made at an appropriate level of management — sometimes by consensus, sometimes by individuals — and high-level commands often fan out to several individuals and gain detail as they travel to their points of implementation. Duties are delegated, appeals are made to higher levels when necessary. Some management structures are strictly hierarchical, while others are decentralised (involving a separation of powers) and operate on consensus.

These are exactly the kinds of approaches that can be applied in the FGM. Choosing the right sort of arrangement requires a good deal of analysis and experimentation, however there is no fundamental barrier to managing a system where each component of the management structure only has a small area of influence and a limited working capacity. The FGM is perhaps simpler than real-world management structures since we can be confident that all PEs will follow the rules.

Possible shapes for management structures that I have identified are broadcast, directed, hierarchical, and distributed. They are not all mutually exclusive and different approaches

can apply at different scales. For instance, a hierarchy can be used to broadcast a message to all nodes at the bottom of the hierarchy, or it can be used to direct the message to some parts of the hierarchy. A distributed approach could rely on the fact that starting from one instruction in a flow graph, one can follow data flows and eventually reach all the other instructions in the graph.

Distributed approaches seem most desirable as they can often function using nothing more than a program's internal structure, that is, no auxiliary control structure is needed. Information on where all the instructions of a program are located can be found within the program itself, so duplicating that information creates a potential inefficiency. Directed, hierarchical and broadcast approaches require either hardware support or extra structures to be constructed and maintained at the software level.

Two questions are quite illustrative of the resource management and task management problems that we would like to solve: "How can a running program be terminated?" and "In the presence of dynamic allocation, how can we control each program's allocations, or at the least, prevent a runaway program from bringing other programs to a halt?"

Consider the question of terminating a running program, and let us seek a distributed solution to the problem. A solution that sounds sensible is to start deleting the program's instructions from the boundary of the program. Each time an instruction is deleted, this triggers the deletion of each of the instructions that are specified as its outputs in a sort of domino effect. The instruction slots occupied by the deleted instructions may then be used for other purposes.

However, since part of the program is still running, a data word may be sent to a deleted instruction by an upstream instruction that has not yet been deleted. If the deleted instruction has by this time been allocated to another program, the stray data word can cause undesirable behaviour.

To avoid this problem, we can terminate a program in two passes: first a "halt" sig-

nal is sent out, starting from the program boundary and following the outputs of each instruction, putting each instruction in the program into a frozen state (e.g. by changing the instruction to a `NOP`, or setting a flag, or changing the priority to zero), and after this is completed, the deletion proceeds as described above.

How can we determine that the "halt" process has completed? We need to wait for some kind of signal to come back out of the program, indicating that every single node in the instruction graph has been visited. We want an algorithm that achieves a similar result to the *propagation of information with feedback* algorithm described by [Segall, 1983].

Looking at Segall's assumptions, we see some that are quite strong for our purposes:

- each link is bidirectional,
- all messages received at a node are stamped with the identification of the link from which they came, and
- each node has an identification; before the protocol starts, each node knows the identity of all nodes that are potentially in the network.

Such information might be very useful for housekeeping operations, but such operations are performed infrequently and it does not make sense to incur substantial costs during times when such information is unnecessary.

In the SD-ASIC, an instruction can have a large number of input flows since the instruction does not store the addresses of upstream instructions. This means that one cannot trace "backwards" through a flow graph based only on the information within the instructions. Adding space to *each* instruction for the addresses of its inputs would considerably increase the memory consumption of each program. This seems a high price to pay for something that needs to occur rarely or never.

Figure 3.1 illustrates an instruction (the *child*) and all instructions which contain a reference to it (the *parents*). Since the sources of an instruction's inputs must be located

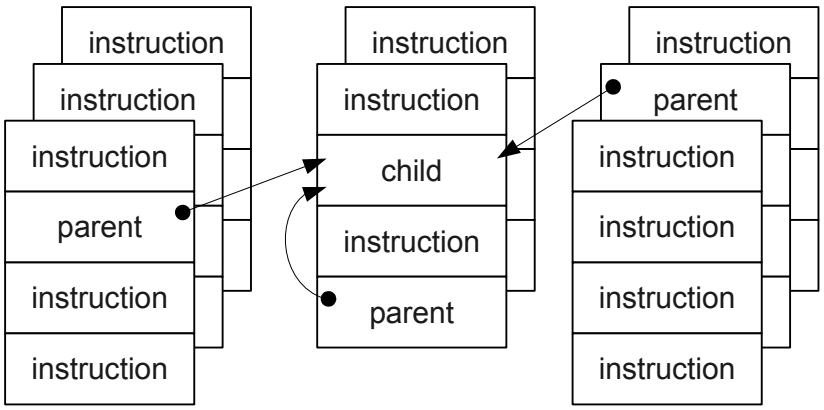

Figure 3.1.: Parents of an instruction.

on the PE itself, or one of its neighbours, locating all such instructions involves a scan of maybe a few thousand memory locations. I shall refer to this operation as a *parent search*. Since this would not need to be done very often, the cost seems reasonable, and the operation could be assisted by hardware optimisations.

It would also be very rare for an instruction to care where its input came from. Adding extra bandwidth between PEs so that this information can be sent around seems wasteful. Perhaps, when this is needed, it can be implemented in software by using an extra data word.

Can we design a distributed propagation of information with feedback algorithm without any significant changes to the SD-ASIC? If we assume there is a little spare bookkeeping space in each instruction, and that there is a path from every instruction to every other in an FGM program (which is reasonable because each instruction is part of a cycle and we can assume a program's graph is connected) then I suspect a system based on counting branches and merges would work, though I have not verified this. If each instruction included a count of how many upstream instructions reference it (a *reference count*), a simpler algorithm might be possible.

A hierarchical approach to the propagation of information with feedback is far simpler: if we have all the instructions in the program as leaves of a tree, we simply broadcast a message along the tree and let the acknowledgements filter back to the root.

Now let us consider the second question: in the presence of dynamic allocation, how can we limit a program's allocations? Let us frame this as a real-world problem to try to understand it better. If a man walks into a bank branch and withdraws some money, then quickly travels to the next town and withdraws more money from another branch, how does the bank prevent him from withdrawing twice his current balance? News of the first withdrawal must travel to the second branch faster than the man can travel. What if he has an accomplice — i.e. two men withdraw from the same account in different branches simultaneously?

Perhaps we do not need an entirely strict policy. Perhaps it is enough that a program's resource consumption is made known to the OS in a timely fashion, and that the program is incapable of allocating resources faster than the OS can chase it up. Certainly an OS would be able to enforce things about the program that could not be enforced in the bank analogy: in particular the program would not be capable of impersonating another program.

At one extreme, the OS could periodically check resource consumption in the FGM, and if resources are running low, it could conduct an audit of which programs are consuming too many resources, and take appropriate action. At the other extreme, we could require every single node allocation to be approved by the OS before it proceeds. This would involve a huge amount of communication and seems out of the question. Neither of these approaches seem efficient.

I thought of a reasonable alternative: programs that require allocation explicitly request "permits" from the OS in advance that permit them to perform a specified number of allocations. The program would be responsible for sending these (unforgeable) permits

along to the locations where allocation occurs. This means that allocation does not lead to overheads when no allocation is occurring, and it doesn't require every single allocation to be approved by the OS.

Obviously when nodes are deleted, the program should be credited for this somehow. It should either be able to trade them in for new permits, be able to re-use them in later allocations, or the OS should record a decrease in the program's resource usage. The exact mechanism will depend on how nodes are deleted.

In both of these methods, the amount of budget a program has is sent around within the program, and the program's "identity" is protected by the fact that the pressure/permits cannot be forged or leaked into other programs.

We could also track program identities by assigning each program a unique number, and modify the hardware so that program information can be broadcast relatively efficiently. This would not scale to an arbitrary sized FGM, so perhaps we could divide the FGM into *control domains* of (for instance) 50×50 PEs. Every instruction in the domain would be stamped with an identification number (for instance 0 to 255) indicating which program it belongs to, and within that domain any PE would be able to access a table of information about the programs within that domain. Where programs cross domain boundaries appropriate bookkeeping needs to take place, and domain boundaries would likely be where the bulk of OS activity occurred. While such a scheme might prove very convenient, I fear it simply delays the onset of difficult resource management problems until programs reach a larger scale — a scale at which the problem is managing control domains rather than instructions.

3.7. Graph Manipulation

Many real-world problems, from traffic flow to computer networking, can be posed as graph problems. There has been much study of graph algorithms, particularly for sequential machines. Graph algorithms are often difficult to run on multicore or vector hardware, because the graphs themselves are relatively amorphous and cannot be neatly arranged into n bins. It would seem that a reconfigurable graph fabric (like the FGM) would be the optimal choice for efficient execution of graph algorithms.

Consider the problem of finding the shortest path between two nodes in a graph. On a sequential machine, only one node can be explored at a time. It isn't obvious how to take advantage of vector hardware to work on this problem faster. Multicore machines don't seem a good match for the problem because thread creation and migration is expensive, and threads will frequently need to synchronise on shared data. On the FGM, a program could be created to have the same shape as the graph in question, thus allowing all nodes that are currently candidates for exploration to be explored in parallel.

Manually writing a program that does this for a specific graph should be fairly easy. However, what if we want to feed graphs into the program at runtime, and have it construct software to analyse the graphs in the most efficient way? Such a program needs to be able to generate flow graph programs dynamically.

Now consider a data structure like a FIFO queue or a binary tree (of unlimited size). On a von Neumann machine we might store a queue as a linked list, and a binary tree as a tree. Both of these structures are just special cases of graphs and in principle they should be quite natural to represent in the FGM.

As items are added or removed from these data structures, we need to allocate or release memory and adjust pointers between nodes, in the FGM as on von Neumann machines. I imagine that the memory would be allocated from a free list on the PE in question, and if no memory was available, a procedure to move nodes to neighbouring PEs and/or perform

some kind of garbage collection before retrying might be initiated. This will mean, for instance, that as the FIFO queue gets larger, its contents will begin to spread over multiple PEs.

It might be desirable for the binary tree to be an active data structure (see Section 2.7) to allow multiple updates or queries to be processed simultaneously, thus deriving benefit from the machine's parallelism. This means that not only would the *data* grow as items were added to the tree, but also new copies of tree *code* would be generated.

The FGM is certainly capable of representing graphs. The programs it runs have an explicit graph structure: each instruction is a node, and contains references to other nodes. These references specify the arcs of the graph. Perhaps we could extend this to represent and manipulate data with a graph structure. If that were possible, it should also be possible to manipulate graphs containing code, and hence create something like a compiler.

It appears that graph manipulation and generation of code (that is, dataflow graphs) at runtime will allow the FGM to be used on a much larger class of problems and make it easier to solve certain problems efficiently. I must admit that finding an efficient hardware or software implementation is likely to be challenging. Below I will explain why I believe it is possible, and discuss various implementation considerations. In Chapter 4, I propose a model for graph manipulation and demonstrate its utility.

3.7.1. The graph interpreter

One way to provide graph manipulation functionality with minimal changes to the architecture is to build a giant static program that covers many PEs and operates on some kind of data representation of a graph. It might treat the graph as a flow graph (but equipped with an instruction set affording graph manipulation). Alternatively, it might apply simple rewrite rules to the entire graph, a computational model that has been proposed and investigated in the past (see for example [Tomita et al., 2002], which describes a self-

replicating Turing machine embedded in a graph). Essentially it would be a simulation or virtualisation of a different kind of graph processor.

Implementing such a program might give insights into how to redesign the hardware to facilitate graph manipulation. Using a real SD-ASIC to run efficient simulations for further development of the FGM would be a good way to eat one's own dog food.

By virtue of being distributed across many PEs, this program would be able to take advantage of the parallelism of the machine, but would likely add a considerable degree of overhead and might need quite a large number of instructions per PE. On the other hand, it might turn out to be a good way to go about things, and become a library, or part of the OS.

3.7.2. Hardware-supported graph manipulation

The graphs that can be formed by taking fixed-size instruction slots as nodes and the references they contain as arcs (as described above) are limited in their structure. They are limited in the number of outgoing arcs permitted per node. [Hillis, 1988] examines this problem, discussing the different options for how many references a node contains, and explains that we can view a collection of physical nodes as a single logical node, which does allow us to encode arbitrary graphs.

Another issue is that, unlike the Connection Machine, the FGM fabric is not a *complete* graph, that is, nodes are not connected to all other nodes. In order to represent arbitrary graphs, we may need to consider a chain of nodes as a single arc, thus allowing arcs to span arbitrarily long distances.

The amount of data, code and references to other nodes that can be stored within a single node is a balancing act. Making nodes consume a fixed number of bits makes it much easier to manage allocation but leads to either wasted space within nodes or the overhead of spreading data across multiple nodes. Nodes with a single reference within

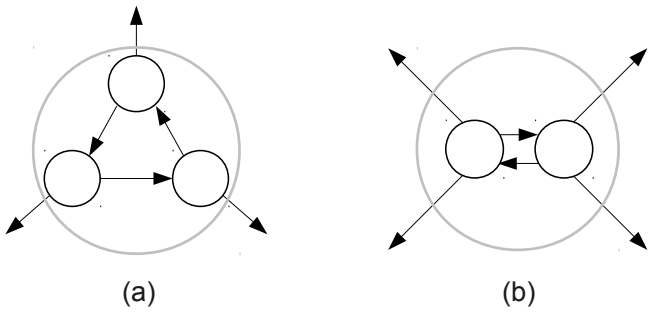

Figure 3.2.: Simulating nodes with more references: (a) using three nodes each with two references to represent one node with three references; (b) using two nodes each with three references to represent one node with four references.

them are not useful. Nodes with two references in them seem expressive enough for any needs, since a group of three such nodes can be used to simulate a single node with three references, and two nodes with three references can be used to simulate a node with four references, as shown in Figure 3.2. Adding more references reduces the number of nodes necessary to represent a graph, but creates wasted space when the references are unnecessary. A reasonable balance must be found, or variable-sized nodes used.

How can we manipulate such a data structure with only local information? I believe a principle which I call *transitive proximity* is important: loosely speaking, if A is near B and B is near C then A is near C. If we assume that B has uniform chances of being on the 9 PEs which are at most one hop from A, and the same holds for B and C, then there is a 60% chance that A and C are at most one hop from each other, and otherwise, they are two hops apart.

Thus, if B has a reference to C, and B passes this reference to A, then either the reference can be handed over trivially, or one of A or C needs to be moved by a limited distance. This distance is no larger than the maximum distance between two nodes. Moving either A or C to the same PE as B, for example, will be guaranteed to bring A and C near enough to each other.

Moving nodes around is not trivial. A node being moved will generally be referenced by other nodes, and these references must be updated. Doing so might be impossible without moving these nodes also, which would require further references to be updated, and so on. Waiting for all this work to complete just to be able to pass a reference around seems unreasonable. Instead, when a node needs to be moved, it can be moved to a new location immediately, and a placeholder for it left in the node's old location. The placeholder could contain a forwarding address, allowing references to it to be updated when convenient. Once no more references to the placeholder exist, it can be deleted to reclaim its space.

In the case where a node is an actual instruction with incoming data flows, the forwarding address can be used to ensure these flows don't get lost. In this way it seems that moving instructions is possible even while a program is running.

The movement of nodes could occur not just to support graph manipulation activity, but also to move instructions so as to balance out resource usage. For instance, if a PE is low on memory but one of its neighbours has plenty of memory, instructions could migrate so as to even this out.

For this to work on a larger scale, that is, so entire programs migrate to areas of the FGM that are little utilised, a method based on pressure and stiffness could be used — instructions would experience a "force" pulling them towards regions of lower resource utilisation and would in turn pull on their neighbours, thus tending towards a smoother pattern of resource consumption using local information only.

3.7.3. Opaque references

The ability to do pointer arithmetic is not particularly useful in the FGM. A reference points to one of some several hundred nodes, so the possible size of an array-like data structure is quite limited. Additionally, many problems that on a von Neumann computer

would be solved with the use of an array, would on the FGM be solved by operating on one or more incoming data flows, thus would not require anything being stored or retrieved from such a data structure.

Aside from pointer arithmetic, there seems to be no need for a program to inspect the numerical value of a pointer. So if pointer arithmetic is disallowed, references can be *opaque*, that is, their numerical value can be impossible for a program to obtain or manipulate.

Opaque references help enforce separation between programs. References can not be created out of nothing. They can only be copied from existing references, and this allows references to be validated when they are created rather than when they are used. Opaque references also seem like a very useful idea if nodes can be relocated by the hardware at will, since the actual numerical value of a reference is subject to change at any time.

Opaque references can be seen as a form of capability system (see Section 2.10). If node A has a reference to node B, then A has certain *rights* over node B. If A is a normal instruction then presumably it has the right to send data to node B. However, A might be a graph-manipulation instruction, and be capable of reading one of B's references. To prevent violation of the separation between programs, a model would need to be adopted that could prevent graph manipulation instructions from following a chain of references into a separate program.

3.7.4. Templates

Just as constructing arbitrary pointers is dangerous, constructing arbitrary instructions could be dangerous too. Instead of manipulating the binary value of instructions and so forth, I propose the use of *code templates*. Rather than generating the machine code as a sequence of bytes, existing instructions would simply be copied and stitched into the desired configuration by graph manipulation.

An assembler program would need to have a copy of each possible machine instruction available. A compiler might have small graphs of instructions. These instructions or graphs of instructions would be duplicated each time they were required and stitched together to form a program.

An active data structure would serve as its own template. For example, whenever a new node is added to a binary tree, the code for that node could be copied from its parent node. This relies on the similarity between all the different nodes in the tree. It seems efficient, too: the code template would be fairly near to where it is needed, and multiple copies could be created in parallel. It seems like recursive functions and recursive data structures are ideally suited to this because of their self-similarity.

Code templates might make it easier to transform programs between slightly different instruction sets and could have other benefits for safety/isolation or convenience.

4. Investigation

With my investigation I intended to gain a better understanding of the FGM, as well as demonstrate and evaluate some of the things I had conceived.

Early in the project, I began work on a simulator of the FGM. I did not yet know what it might be used for, but I was certain that it would be valuable. When this was complete, I created an assembly language to make it easy to write programs which could run on the simulator.

Since I was not sure how the simulator would be used, and furthermore I thought it likely that I would want to experiment with changes to the simulator and assembly language, I designed them to be somewhat abstract and easily extended and modified. I felt that building my own simulator was a better option than obtaining one from Wave Semi, since by building a simulator, I would learn far more and find it easier to work with and modify the simulator.

The programming language I chose for these programs was Haskell. The language was ideal for my purposes. It is a hotbed of programming language research, and allows the creation of sophisticated, extensible programs with a minimum of code. The Haskell type system helps prevent many types of bugs well before a program is run, which is important in a research environment where small changes are often being made to a program.

I spent some time writing simple programs to experiment with the simulator and gain a better understanding of the FGM.

As my ideas developed, I became convinced that graph manipulation might be an important tool within the FGM and ought to be investigated. I decided to write a *flow-graph compiler*, that is, a program running within the FGM, capable of creating a new FGM program. Since an FGM program is a graph, this would presumably require the use of graph manipulation capabilities.

Although a compiler is considered by some to be part of an operating system, generally this is not the case. Either way, my aim was not to write a useful compiler for the FGM, but to investigate issues that might be relevant to an operating system by writing a very simple compiler.

To this end I designed a set of graph manipulation instructions for the FGM, and added them to the simulator and assembly language. Due to the extensible way I had designed the simulator and assembly language, these changes did not break existing programs I had written for the simulator. I then implemented the compiler (adding a few more graph manipulation instructions to the simulator along the way as necessary).

Following the success of the compiler, I turned to a more pressing need: how to load a new program onto the FGM. The problem sounds deceptively simple, and if very restrictive assumptions are made, the problem is simple to solve. Naturally I chose some more liberal assumptions, which made the problem rather challenging. I came to a method which I can justify, but better methods may exist.

Below I cover these endeavours in more depth.

4.1. Simulator

At its core, the simulator maintains a queue of instructions which are *ready*, that is, their inputs are available. One at a time, it removes an instruction from this queue, performs the operation associated with the instruction, and propagates the outputs to wherever they

are destined. If this results in other instructions becoming ready, then these instructions are added to the ready queue.

Initially the simulator included a notion of PE locations and took the passage of time into account, but I found these were not useful for my experiments so I removed them to leave an extremely simple core amounting to about 250 lines of code (as I have said, Haskell allows one to describe a lot with very little code — the amount of thought that goes into each line can be quite large). The following code excerpt is illustrative of the simulator's main data structures:

```
data GraphNode = GraphNode (IORef ([Maybe Word], Opcode, [Maybe Port]))

type Opcode = (CompletenessFn, ComputeFn, String)

type Port = (GraphNode, Int)
```

In essence this defines a GraphNode as three things grouped together: a list of the input words that have arrived for processing ("Maybe Word" because they may or may not be present at a given time), the operation code, and port locations where outputs are to be sent ("Maybe Port" because some outputs are discarded rather than being sent anywhere). An Opcode is a group of three things: a function that determines whether the input is complete (i.e. the instruction is ready), a function that does the actual computation, and a string which contains the instruction's name (for debugging purposes). A Port specifies both a GraphNode and an integer identifying which input of that instruction is the target of a data flow.

Defining a new type of instruction is as simple as writing a completeness function and a compute function for the opcode. While writing programs within the assembly language, I found I was using a particular construct so often that I created a new instruction for it. I called this instruction `GATE`, and given two inputs x and y, it will wait for both inputs to

arrive, and then output x. In this way y controls a "gate" allowing x to pass.

The `PASS` instruction simply passes its input x to its output (possibly to multiple outputs, in which case x is duplicated). A chain of these instructions can be used to relay information over long distances.

4.2. Assembly Language

I implemented my assembly language as an embedded domain-specific language (EDSL) in Haskell. This meant the development effort was minimal and the language could utilise Haskell features.

The language is built around *flows*, which as the name suggests, represent data flows. Whereas in the hardware, each instruction specifies where its outputs go, in the assembly language you specify where the inputs of an instruction *come from*. This leads to a much more natural programming style where instructions look like functions of their inputs.

In fact, program components can be defined as Haskell functions and strung together as desired. These fulfil a similar function to *macros* in macro assemblers, and I will refer to them by this name.

The following fragment shows the `square` macro which uses the `mul` instruction to square its input flow, and the `quadratic` macro which evaluates ax^2+bx+c^2 given inputs x, a, b and c, utilising the `square` macro as well as the `add` and `mul` instructions.

```
square x = mul [x, x]

quadratic x a b c = add [mul [a, square x], add [mul [b, x], square c]]
```

The following fragment shows the macro `counter` which, upon receiving a request, will output an increasing integer, initially 0, then 1, 2, 3, etc.

```
counter request = value where
    value = I 0 <~ gate request nextValue
    nextValue = add [value, generate value (I 1)]
```

This example shows the <~ notation which gives flows initial values. In the case, `value` is initially the integer 0, and subsequent values come from nextValue gated by the request. `generate` is a macro which repeatedly generates the same value (in this case 1) gated by the presence of an input. Note the usage of `value` within the definition of `nextValue` — we have thus constructed a loop.

Instructions with multiple outputs use Haskell's tuple or list syntax to define multiple flows at once. In the following example, the flows `quotient`, `remainder`, `loop` and `completeness` are generated from multi-output instructions, and `toDecimal` is itself defined as a two-input, two-output macro:

```
toDecimal input digitAck = (completeness, remainder) where
    (quotient, remainder) = divmod [unsafeMerge loop input, generate' quotient $ I 10]
    (loop:completeness:_) = steer [completeness', quotient]
    completeness' = eq [gate digitAck quotient, generate' completeness' $ I 0]
```

The instructions `divmod`, `steer` and `eq` respectively find the quotient and remainder of the inputs, use one input to steer the other between multiple outputs, and determine if two values are equal.

As stated above, the macros are Haskell functions. When the program is assembled, the functions are evaluated, resulting in an intermediate recursive data structure of instructions and flows (but no macros). A recursive data structure can be traversed *ad infinitum* due to the cycles within it. I used the data-reify library by [Gill, 2009] to solve this problem. It is capable of taking a recursive Haskell data structure and *reifying* it into an explicit graph. My assembler takes the resultant graph and transforms it into a form the simulator understands, which mainly involves reversing all the arcs in the graph, so that instructions know where to send their outputs rather than where their inputs come from.

I have found that loops play a critical role in FGM programs and every loop needs initialisation somewhere within. I believe I have designed an effective tool for writing FGM programs. My experience in using it to write various experimental programs has been good.

4.3. Graph Manipulation Instructions

Consider the following C function which adds a node to a linked list:

```
void addNode(int data, struct node* insertionPoint)
{
    struct node* newNode = malloc(sizeof(struct node));

    newNode->data = data;
    newNode->next = insertionPoint->next;
    insertionPoint->next = newNode;
}
```

On a sequential machine, a memory address can be referred to multiple times as part of graph manipulation code. In the above example we see `newNode` being dereferenced twice and `insertionPoint` dereferenced twice. The sequential ordering ensures the operations occur in the correct order.

If we make the assumption that graph manipulation instructions are quite simple, then to achieve something like the above example in the FGM we will need multiple instructions working together, and so there will need to be a way for these instructions to receive a memory address to operate on, and to co-ordinate them so they operate in the correct order. The example above also has a memory allocation operation; some kind of analogue will be required in the FGM.

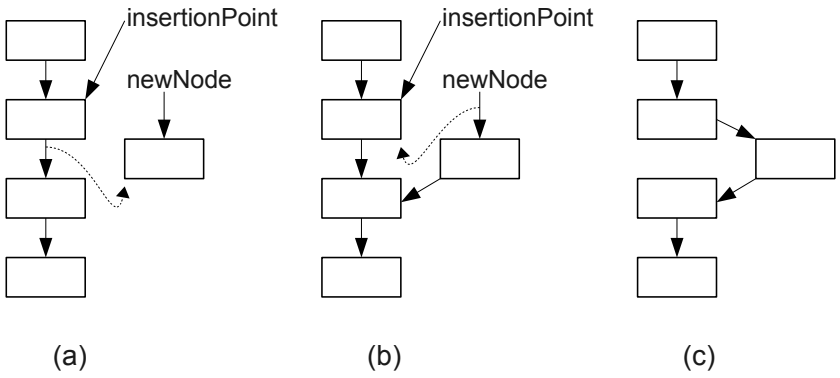

Figure 4.1.: Addition of a node to a linked list. Dotted lines show the references being transferred. (a) shows the state of the list just prior to addition, (b) shows the result of the first step, and (c) shows the final result after the second step.

First let us review the graph structure already present in the machine. Each instruction has up to (say) three references within it. These normally specify destinations for that instruction's outputs. Each instruction is a node, and the references specify the arcs between nodes. This infrastructure seems quite sufficient for *representing* graphs. We would like to augment it with the ability to modify them.

Linked list operations are graph operations — we are just accustomed to performing them on a computer with access to a global memory. If we illustrate the linked list algorithm graphically (see Figure 4.1, we see that arcs are transferred between nodes. This strongly suggests that we need to be able to pass instruction addresses between nodes in the FGM in order to perform graph manipulation.

The choice we have is either to allow references to be passed around like other pieces of data, or to somehow relegate reference exchanges to special instructions. In all cases we have the problem of passing a reference to a neighbouring node, potentially extending the reference to be further than nearest-neighbour. I believe this problem, whilst challenging, can be solved by relocating instructions as suggested in Section 3.7.

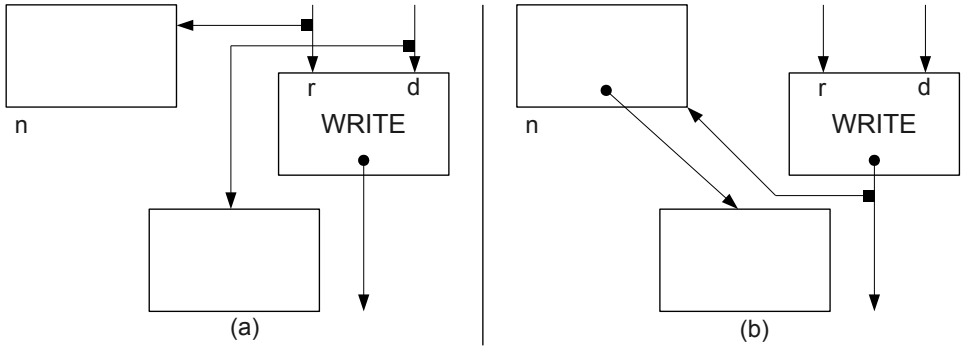

Figure 4.2.: The **write** instruction: (a) shows two incoming references, (b) shows the update to *n* and the reference output by the **write** instruction afterwards.

I chose a model which seemed consistent and elegant, and after implementation I found that it was usable. In this model, each graph manipulation instruction has a reference input which shall be referred to as *r*. Some instructions have a secondary input which shall be referred to as *d*. The node specified by *r* shall be referred to as *n* (in C notation, we would say that `n = *r`). Figure 4.2 illustrates this. Most instructions access *n*, and produce output when this access is complete. Short descriptions of the instructions follow.

read Reads a reference out of *n*, outputs the reference.

write Writes the reference *d* to *n*.

send Sends data word *d* to *n* like an ordinary data flow.

override Changes the port number in *r* to *d*, outputs the result.

duplicate Copies node *n*, outputs a reference to the new node.

null steer Sends *d* to one of two locations depending on whether *r* is a valid reference.

The **write** instruction must wait until the write is complete before producing an output. This is necessary to allow further instructions that must occur after the write to be properly

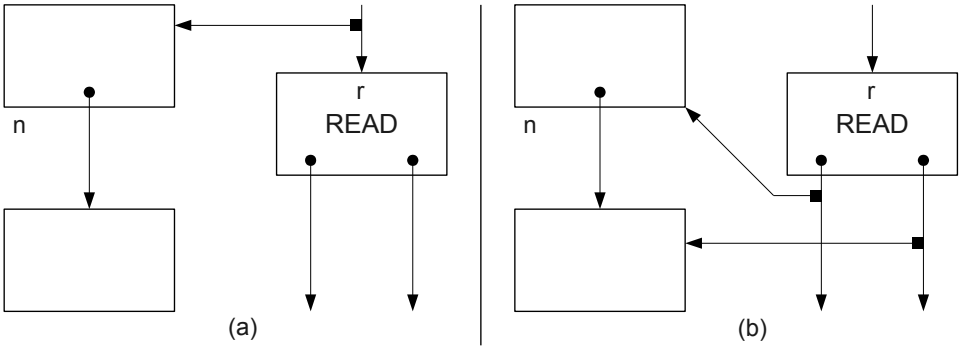

Figure 4.3.: The **read** instruction: (a) shows an incoming reference, (b) shows the references output by the **read** instruction afterwards.

synchronised. Similarly the **duplicate** instruction should not generate an output until the duplication is complete. (The same is true of the **read** instruction, however obviously it cannot produce output before the read is complete.)

As shown in Figure 4.2 and Figure 4.3, I made some of the instructions output a copy of *r* when they are done. My reasoning was that subsequent instructions may need *r* as an input. This is simply a convenience: references can be copied using the `PASS` instruction (not the **duplicate** instruction which copies an entire node).

Something which is notably absent from the instruction list is a means of deleting nodes. This is because it would be disastrous if an instruction were to be deleted despite references to it remaining in existence, particularly if we rely on program graph structure being sound to prevent programs from interfering with other programs.

Instead, I think the hardware or OS should automatically clean up nodes that are no longer referred to, rather than allowing programs to delete nodes arbitrarily, since it is extremely undesirable for a node to be deleted even though references to it still exist. Since my simulator runs on Haskell, nodes that are no longer referenced will eventually be *garbage collected* by the Haskell runtime.

Performing garbage collection in the FGM will probably be challenging due to the asynchronicity, lack of global memory, and presence of cycles in flow graphs. If an isolated cycle is created (by deleting all outside references to it), then the cycle will consume space needlessly. Worse, an active computation can continue in the cycle, wasting resources. Existing approaches to garbage collection, particularly those designed for distributed systems, should be reviewed. Work such as [Le Fessant et al., 1998] might be useful.

4.4. Compiler

The compiler I implemented is able to take a Polish notation numerical expression such as "`-*10 3 /2 1 `", which represents the mathematical expression $10 \times 3 - \frac{2}{1}$, and create a flow graph that would evaluate the expression, in this case giving the result 28. While the compiler is only capable of producing very simple programs, it demonstrates that, given a set of graph manipulation instructions, it is fairly straightforward for a program within the FGM to create a program for the FGM.

Such a compiler is a self-modifying graph. The program being generated must be somehow referenced by the compiler at all times, hence in a sense is part of the compiler graph during the compilation process.

Graph manipulation is used not only to construct the new program, but also to maintain an intermediate data structure used by the compiler. This data structure was a stack of "loose ends" of the program being strung together (as shown in Figure 4.4). Thus I was able to demonstrate the use of graph manipulation instructions in maintaining a simple dynamic data structure.

Initially the stack contains a reference to where the final output of the program should be sent. During operation, the compiler reads one character at a time from its input stream.

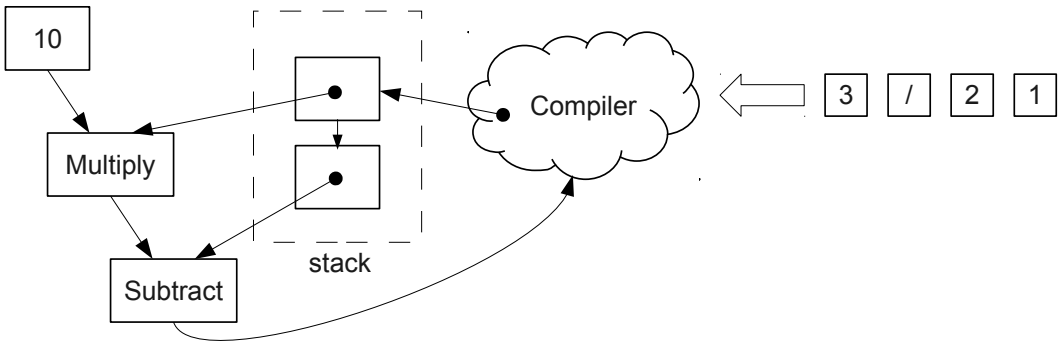

Figure 4.4.: The flow graph compiler in operation. The input `-*10 3 /2 1` has been partially processed.

If the character is an operator, the compiler creates the appropriate instruction, connects its output to the reference currently at the top of the stack (removing it from the stack), and then adds the two loose inputs of the instruction to the stack. For instance, upon seeing the `+` character the compiler will create an `ADD` instruction and connect it as described. If the character is a digit, it is treated as part of a decimal number. If the character is a space, it is understood that the end of a number has been reached. The number is sent to the reference at the top of the stack (removing it from the stack).

I implemented this design using the assembly language and verified that it behaved correctly using the simulator.

Each operator is processed by a different branch of the program, and numeric values are processed by yet another branch. It does not make sense to have separate stacks for each branch! The different branches must all share a single stack because they all need to work together on the same data. Therefore some mechanism is required to ensure that when information is retrieved from the stack, it is directed to the part of the program that requested it.

This can be achieved by assigning each possible user of the stack a different number, and when a part of the program sends a request to the stack, the number is appended so

the stack can direct its reply to the appropriate place. A drawback to this approach is that it requires all the stack users to be known in advance.

Instead of doing this, along with every request made to the stack, I included a reference to the node where the result of the stack operation was to be sent (either the value popped off the stack, or an acknowledgement that a push has been completed). This method has many similarities with the use of function pointers or callbacks in conventional programming languages. The performance cost of such an operation on real hardware would be considerably higher than the approach of using unique numbers described above, but it is more flexible and quite convenient.

4.5. Code Loading

Loading and running a program is effectively a pre-requisite for calling something an operating system. An operating system for the FGM will need to be capable of doing this.

The difficulty lies in the fact that loading data from outside the FGM must occur through the boundary nodes, but the instructions have to be distributed across processing elements that are far from the boundary without destroying the program's graph structure.

If we restrict ourselves to loading a single program at a time with a completely determined layout, the problem becomes simpler. The program's graph structure does not need to be maintained while it is being loaded so long as when loading is complete the structure is correct. A self-evident technique is to load the program, row by row, so it is pulled down through the FGM like a roller blind over a window. Instructions that have to travel the furthest must be loaded first so that they do not block the path of subsequent instructions. This operation could be globally co-ordinated, so that all the PEs work in lock-step to relocate the instructions they contain, or in a more distributed fashion where a push from one end ripples downwards and once it is complete an acknowledge is sent

in the opposite direction, allowing the next push to begin.

Detailed study of the problem thus restricted did not seem worthwhile. The problem appears to be fairly easy to solve, though deciding on which solution is best would require much knowledge of FGM hardware details. A solution might not be able to load multiple programs at once or vary the program layout.

Consider a situation where multiple programs are running within the FGM, and a PE has a large amount of work to do, or a lack of free memory slots. It would be good if a program being loaded could avoid this PE and favour PEs with more resources. Alternatively, consider a chip with a faulty PE that has been disabled. It would be useful if this PE could be evaded during loading.[1]

I chose to investigate the loading of programs with nothing more than the flow graph itself, that is, no extra information to assist loading, nor on how the program is to be laid out on the FGM. Only the instructions of the program and the arcs between them are available.

One problem I looked at was how to design the interface with external memory — how does the outside world appear to the FGM? I sought an interface which would make the outside world look like an extension of the FGM. In other words, external memory and I/O would be done through things that behaved like PEs at the boundary of the array, but with different properties to the other PEs in the FGM.

For example, the boundary nodes could look like PEs with very large memories attached, and which did not actually execute instructions. A program could thus be stored in these PEs and when required it would be moved out, instruction by instruction, and drawn across the FGM. I will refer to PEs equipped with a large number of instruction slots as *towers*.

References that refer to instructions in a tower must be large enough to index any of

[1] With the 8-way connected configuration of the SD-ASIC, it is actually not possible for a program graph to "jump" over a PE, so this motivation is somewhat weak.

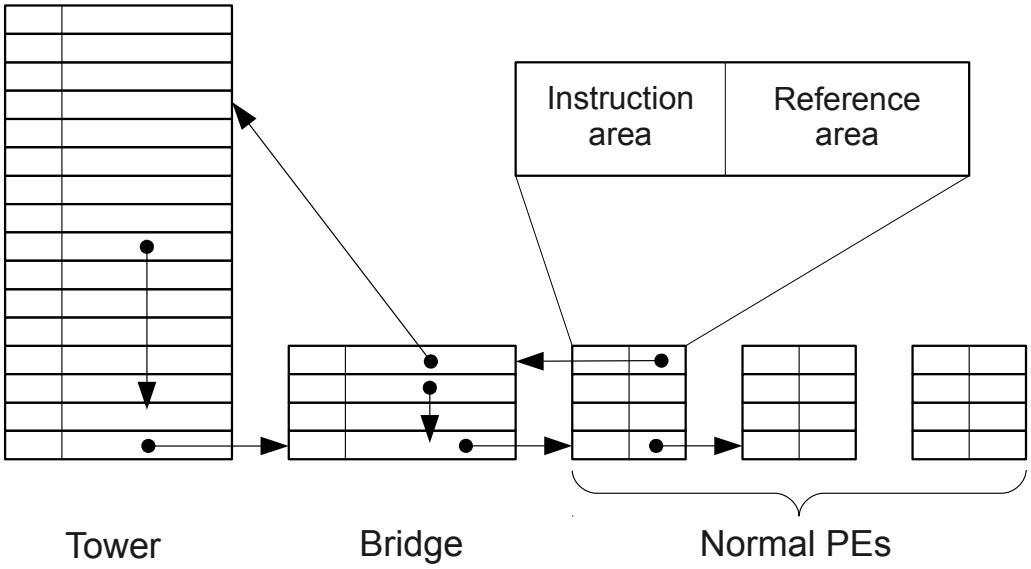

Figure 4.5.: An illustration of towers and bridges. Note that the amount of space required for references in a bridge is larger than that of a normal PE, because it must be capable of referencing all the instruction slots of a tower.

those instructions. Having such large references throughout the entire FGM, however, is undesirable. All we really need is that tower PEs themselves, *and their neighbours*, support these large references. I shall refer to the PEs which sit in between towers and regular PEs as *bridges*. These nodes have no more instruction slots than regular PEs, however each instruction slot is larger so as to accommodate the large references to nodes in the towers. See Figure 4.5 for an illustration.

Since towers have large amounts of memory attached, the extra space required to store references within them is not a big drawback. Bridges could either be equipped with slightly more memory than a regular PE, or simply offer fewer instruction slots to accommodate the larger references. The actual layout and implementation of this scheme could be done in many ways. A tower could simply be a bank of RAM with a memory controller attached. There could be many of these distributed across the FGM chip, which would

be good for parallelism, or fewer, larger ones, or the RAM (or ROM) could be entirely off-chip and accessed via a bus. Some of the instruction slots in a tower could represent I/O to external devices other than RAM, or serve special FGM functions.

This system of towers and bridges requires variable reference sizes. Making references opaque as in Section 3.7 would simplify this at the instruction level.

Given such a scheme, a program can be loaded by pulling it out of the towers and spreading it across the FGM, and it can be saved by folding it away into the towers. The next time the program is loaded it will immediately resume from the point when it was saved.

There seem many good reasons why we would want to begin a program from its initial state rather than resuming it. For example, using ROM instead of RAM to store the program might cost less or use less energy, or we may simply want to run several instances of the program. What if the program gets stuck and we would like to terminate it and start over? It would be good if we were able to *copy* a program from external memory rather than just moving it around.

Graph copies could have many applications beyond loading programs. They could be used to copy a program's entire state, essentially creating a checkpoint, which would be useful for development, debugging and error recovery. I suspect they will be useful in dealing with dynamic data structures, as at time large parts of a data structure require copying, and at least copying code templates (see Section 3.7) would be useful. Graph copies might be useful in program compilation, allowing program modules to be copied and wired together. It would clearly be good if the copy could take advantage of the FGM's parallelism.

The challenge in copying a program is in detecting cycles. If instruction A refers to B, B refers to C, and C refers to A, then after copying A, and then B, and then C, the algorithm should realise it has already copied A and use the copy, rather than making a

second copy of A. Various algorithms to do this have been published, with many different sets of assumptions on the amount of extra space available, and on memory layout. [Lindstrom, 1974] presents two algorithms; the first does not assume any extra storage and runs in $O(n^2)$ time where n is the number of nodes, while the second assumes a single tag bit per node in both the old and new graph and runs in $O(n \log n)$ time. [Fisher, 1975] presents an algorithm that does not assume any extra storage space and operates in $O(n)$ time, however it modifies the pointers in the original graph during construction of the copy, restoring them to their original values by the completion of the algorithm, and also assumes that the copied graph is allocated within a contiguous section of memory.

An implicit assumption of all these algorithms is that the copy is done on a conventional computer with a global memory. On the FGM, we do not have this luxury. Once a node is copied we need to send it away as soon as possible to create room for new nodes, because each PE has a limited amount of space. However we should aim to avoid over-extending arcs (including those that have not yet been created!). Assuming we are copying from a tower to a bridge, copied instructions must not leave the bridge until all the instructions that refer to them and which they refer to have also been copied.

Two methods of keeping track of this are bidirectional references and reference counts. In the case of bidirectional references, a node is ready to leave the bridge when all of its references refer to copied nodes. In the case of reference counts, a copy's reference count begins at 0 and is incremented each time a reference is made to it. When the reference count is equal to the reference count of the original node, and all references within the node refer to copied nodes, the node is ready to leave the bridge.

To look at this another way, the bridge has space for a few hundred nodes that are not fully linked with their new neighbours. These will run along some cross-section of the graph. In cases where the bridge PE runs out of space it may have to stash things in a RAM tower temporarily. Hopefully this will not occur frequently since dataflow graphs

are not strongly connected, and also bearing in mind that a program will be spread over multiple towers and bridges as it is being loaded, not just passing through a single bridge.

Since reference counts consume less space I decided to adopt that approach, although moving instructions with bidirectional links is far easier as discussed in Section 3.7.

To allow for loading programs from ROM, it would be good to minimise or avoid modifications to the original graph structure. Intuitively, there needs to be a way of detecting cycles in the original graph, and having each node in the original graph point to its copy (if one exists yet) seems like the simplest way to do this. Since a relatively small number of original nodes will be pointing to their copies at any one time in a one-to-one fashion, the amount of RAM space required could probably be made small with the help of some hardware optimisations.

In the algorithm I devised, a copied node initially contains references to original nodes. The aim is to replace these references with references to copied nodes. If it is found that copied nodes already exist, they are used. Otherwise a copy is created and a pointer to it stored within the original node that it is a copy of. When a node is copied, its reference count is set to zero. When its reference count reaches that of the original node, the link between original and copy is destroyed and the copy may move away from the bridge.

I created a simplified implementation of the algorithm in C. It was intended as a simple sanity check, and also to help communicate the algorithm. The implementation is given in Appendix A. I tested it with simple graphs and found that it behaved as expected, in particular that nodes are generally allowed to move away from the bridge fairly quickly, though in pathological cases a large number of nodes can be waiting for references to be made to them.

In principle there is no reason why this technique could not be used to duplicate graphs within the FGM and not just at the boundaries for loading programs. Some natural questions arise about how the graph copy process is initiated and whether it uses special hard-

ware features or instructions, or can be implemented atop more general graph manipulation instructions, and how the resource allocation problem is addressed, but I did not investigate this further.

I conceived a different approach to graph copy, inspired by cell division (mitosis), during which the organelles in a cell line up along the middle of the cell. Initially, instructions are shuffled around so that each node in the graph is located in an instruction slot which is just prior to a free instruction slot. Once this process is complete, each node is duplicated into the free slot. Since the location of each copy node is simply the location of the original plus one, the references between copy nodes are trivial to determine. The advantage of this technique is that nodes do not need extra storage for a pointer to their copies.

A final observation is that when loading a program, particularly from outside the FGM, it may be necessary to check the instructions of that program to ensure that there are no instructions that could subvert the FGM's separation measures. How this might occur depends on the instruction set, however if there is a clear set of privileged instructions, an instruction that determines if another instruction is privileged or not could be sufficient.

5. Synthesis and Future Work

An operating system is an exercise in integrating a vast collection of components into a coherent system. The components need to not only make sense individually, but also when brought together. Understandably, the previous chapters have covered quite a variety of topics. In this chapter I attempt to bring together the important elements and discuss some of the boundless avenues for further work.

A design that works well for a simple system may not work well for a more complex system, and it would be good to avoid investing a lot of effort into a design which later turns out to be limiting. This is why I have tried to give a broad coverage of this topic, at various levels and from various perspectives. I hope this coverage will be instructive in further design efforts on the FGM, enabling decisions that help build towards greater outcomes.

In my inquiries I have favoured solutions that can be distributed, solutions that only require local operations, because such solutions are likely to be scalable. The results have only increased my confidence that the challenges of this machine *can* be solved using distributed methods in almost all instances. Such methods may not be the most economical or practical solution in all cases, but if applied in key places could make a big difference.

5.1. Hardware

Programs will need to be deleted from the FGM's memory to make space for new ones. This might be a dedicated function of the hardware, but it might also be a special case of hardware-based garbage collection, or perhaps implemented in software using graph manipulation instructions. Deleted instructions could be added to a per-PE free list or some other efficient means of keeping track of free instruction slots.

Something I have appealed to many times is the ability for a program's instructions to be moved while the program is running to help even out computational and memory requirements (see Section 3.7). I believe the most important application of this is in making dynamic data structures feasible by allowing the structures to grow without hitting the limit of a PE's memory. I am reasonably confident that this problem can be solved in an efficient, distributed way, however I fear the solution will be rather complicated. Given its utility I would recommend further investigation.

The ability to set the priority of a program (of which pausing the program would be a special case) would be useful for debugging, controlling programs' usage of resources, and for operations which require a program to be stopped. This could be done with some kind of propagation of information with feedback algorithm (see Section 3.6) which would likely need hardware support.

The method of towers and bridges (see Section 4.5) seems like a useful hardware trick to support heterogeneous PEs. I explored it as a way to interface with external devices but found that it was also useful for large internal memories.

Towers and bridges rely on references being opaque, which means pointers do not have a meaningful numerical value and pointer arithmetic is impossible. Pointer arithmetic has limited utility on the FGM, so I do not think it will be missed. Opaque references seem like an ideal capability mechanism, and I think they could both take the place of global names for objects, and obviate the need for memory protection even in the presence of

dynamic allocation and graph manipulation.

Top-down approaches to resource management, such as control domains (see Section 3.6) could also be useful in practice, but to me they seem like short-term solutions. However, if the chip has heterogeneous cores, arranging them in domains might be appropriate, in which case some form of central control for these domains might be useful. It might be a good idea for such control to be implemented in software rather than hardware, for instance the OS can be active on the boundaries (bridges) between domains.

Although we may not realise it, sequential computers are used to do a lot of graph manipulation. The FGM is a graph machine, and it ought be more efficient at solving graph problems than a sequential computer. Designing a mechanism to do this is sure to be a challenge. I have proposed a set of graph manipulation instructions (see Section 4.3) and demonstrated that they can be used to solve interesting problems (see Section 4.4). The implementation of these instructions (or of any other mechanism for graph manipulation) is sure to be challenging. To explore different models of graph manipulation, it may be instructive to try to implement various abstract graph machines in software on top of the FGM. How graph manipulation interacts with things like node relocation, garbage collection, and allocation accounting will be important questions.

The ability for instructions to pass references to nodes between each other (which is one feature of my proposed graph manipulation instructions) allows behaviour similar to that of function pointers to be implemented in a distributed way. This convenience comes with a performance penalty as code (or at least a path to it) must be physically moved around, but the flexibility provided could be immense.

I have observed that features which are often taken for granted at the level of distributed systems, such as bidirectional links and identifying the source of a data item start to become rather costly if they are available to every single instruction in the machine. However, such features are useful particularly for management and bookkeeping. This

leads to questions such as: How far can we get without such features? Can we construct equivalent things in software? Can we implement them in such a way that we only pay the penalty when they are actually used? The parent search (see Section 3.6) illustrates the kind of idea that will be useful in answering these questions.

5.2. Operating System

In some ways, an operating system for the FGM will not be that different from other operating systems. It will have to manage the hardware, interface with software, and keep the software under control. The ultra-portable Linux kernel demonstrates that a lot of the core work an operating system does is independent of the hardware it runs on. For instance, once we have a means of controlling a program's resource allocation, choosing and enforcing an appropriate policy is the sort of problem that OS engineers already know how to deal with.

Of course, many of the mechanisms and techniques used by traditional operating systems do not exist or are not appropriate for the FGM, and some abstractions don't fit well with the hardware. Developing replacements for these things is likely to be where the most effort needs to be directed.

In the FGM, the more widespread something is, the simpler it must be. Something that occurs on every PE must not take up more than a handful of instructions. This places some limitations on the complexity of the operating system, which in turn may need to rely on hardware assistance for certain things. The division between hardware and operating system will be in a different place to where it is in conventional systems, and may be less distinct. Many questions about OS design cannot be answered in isolation; they must also consider the practicality of modifying hardware designs.

5.2.1. Architecture

The tight integration between OS and hardware in the FGM will probably mean that many functions typically performed by an OS kernel will be performed by the FGM hardware. Additionally, instructions have a small area of influence which means that no code has control over the entire machine. This suggests that the notion of a kernel is not really appropriate for the FGM. In my view, the FGM hardware is roughly equivalent to a conventional computer running a microkernel, and the FGM OS would be equivalent to an operating system built around a microkernel (not including the microkernel).

While the hardware design of the FGM eschews global constructs in order to provide performance and scalability, an OS for the FGM will need some way of connecting everything together. In particular, there will need to be a way of cataloguing the main tasks being carried out in the system, so that a human operator can inspect and control what the system is doing. That said, not all systems require this, and in some systems it creates security problems.

In the SASOS approach (see Section 2.10), every piece of data and code has a permanent, unique global address. The notion of fixed-width global addresses immediately makes me wary. Treating the entirety of the machine's data equally creates the illusion that the cost of accessing the data is fairly uniform. As systems scale this will become further and further from the truth. SASOSes take the shared-memory von Neumann paradigm to the extreme, which seems totally at odds with the FGM's message passing approach.

Given that many copies of data and code will exist to take advantage of the FGM's parallelism, the ability to address any instruction on any PE seems wasteful and of limited value. If instructions can be moved around by the hardware at will, it becomes useless. Therefore fine-grained global addresses for code and data in the processor are inappropriate, but perhaps a coarse-grained solution would be useful, for instance giving a unique numerical tag to each large data structure or an entire program.

The single address space would be one way of providing a consistent interface to resources external to the processor, such as external RAM, persistent storage, and other I/O devices. I am not convinced this is any more useful than a file abstraction of hardware devices, particularly on a machine where only a small subset of processing nodes have direct access to external hardware.

Capability systems (see Section 2.10) seem like a much better model. As suggested in Section 3.7, the (opaque) references between instructions on the FGM could act as capabilities. These could be used as names or handles for various sorts of objects, for example file handles, network sockets, and communication channels between programs.

The component approach to OS architecture seems to be well suited to the FGM. The overhead of enforcing separation on the FGM is likely to be very low, and preventing components from sharing data structures will not hurt performance as it does on traditional systems (although as conventional processors gain more cores, the cost of sharing data structures increases). Component models specify the components in the system and the interfaces the components use to interact with each other. This seems a natural fit for dataflow.

5.2.2. Adapting to the FGM

Management is easier when the manager has complete control over the entire system, but is still possible without this, by distributing and delegating control appropriately. Broadcast approaches are the most direct but often incur a high penalty of some sort or do not scale well. Distributed methods are more challenging to implement and often require more information in each node. Striking the right balance will not be easy.

Every useful program produces an output. The point where a program interfaces with another program, or with the outside world, might be a good place from which the OS can monitor and control the program. Further exploration of how input and output and

communication between programs is set up, managed, and torn down will be important in designing an operating system. For instance, how will a server program accept connections from multiple clients, none of which it knows about in advance? When two programs communicate, how will the OS tell where one program ends and the next begins? Perhaps with the use of special instructions or arcs, which are not crossed by algorithms such as the one that carries out priority adjustments. This could potentially fit in with the capability model described above.

The permit system (see Section 3.6) looks like a promising approach to resource allocation control. Any such system would need to be integrated with whatever dynamic allocation instructions that would be provided (such as my **duplicate** instruction), and when nodes are deleted or garbage collected the program ought to be credited for them.

At first glance, completely evicting a large program from the FGM and bringing in another seems like a very expensive operation, as there are a lot of instructions to move. We must remember the large amount of parallelism in the machine, however. Context switches are expensive on conventional computers as well, since considerable amounts of code and data must be evicted from cache and a new set of code and data brought in. Conventional computers have the advantage that unmodified data (which generally includes all code) does not need to be written back to memory, whereas in the FGM the tight coupling between code and data means that they must be moved together. Also, conventional machines only pull the code and data which is actually used into caches. On the FGM, parts of the program not currently being used might be stored in memory while other parts are on PEs being executed, or the parts of the program experiencing low activity could be stored in PEs with larger amounts of RAM but less processing capacity.

Creating a checkpoint of a program would require a graph copy operation. Obtaining and analysing data traces would require adding some more data flows to collect data items. Single-stepping a program that has been optimised to run in parallel would be difficult,

although sophisticated tools might be able to give the illusion of single-step by collecting traces and "replaying" a single-step scenario. Allowing a developer to modify data values during single-step operation might be achieved with the help of checkpoints.

I suspect that separation between programs can be achieved by making instructions and references opaque, choosing the instruction set carefully, and treating references as capabilities with an appropriate rights model. This would require more investigation into existing rights models (for example, the Take-Grant model [Lipton and Snyder, 1977]) to see whether any are a good fit for the FGM and provide the necessary properties.

In a von Neumann machine, having all the code and data in the system as part of a task's address space allows the task to derive benefit from an arbitrary piece of code or data (subject to security checks). If you pretend the von Neumann bottleneck has a negligible effect, this seems simple. In reality, hardware caches have to ferry around a lot of data to make this work, and distributed SASOSes have to manage the problem explicitly (and that work may prove useful in the FGM). Replication of data and services in the FGM is in some ways analogous to caching systems on von Neumann machines, and the FGM OS will probably be responsible for performing it. Sharing code and data structures between different software components was demonstrated in Section 4.4. A further step would be to duplicate shared components when they are heavily used and merge them when the demands on them are low.

I have suggested that the hardware might be in charge of migrating instructions between PEs as appropriate. This leads to the question, does the operating system need to know anything about the physical layout of PEs at all? Maybe the OS need only see an abstract graph machine, or maybe it should be able to extract information about propagation costs or PE properties. Would any of that information need to be revealed to application software? These are difficult questions.

5.3. Languages

C is often described as a "portable assembly language". It hides many hardware details and does a lot of repetitive work for the programmer, but with a fairly small reduction in what is actually possible to implement. This combination of convenience and flexibility must be one of the reasons it is still popular as a language for OS programming and embedded systems programming, and even continues to be occasionally used for application software.

The FGM would surely benefit from a language of that calibre — or better! Such a language would be a very helpful tool for OS development as well as research into ways to use the FGM. Consider that if the hardware has opaque references, a clean concurrency model (no need to worry about things like write ordering), and garbage collection, a low-level language for it would be a few generations ahead of C in some respects.

While much more experience with the FGM will be necessary before such a language could be created, I believe the assembly language I have designed does quite well, both in terms of staying true to the FGM, and in convenience and usability.

Often, writing programs in the assembly language did not seem much more complicated than writing Haskell programs (debugging, however, was a major pain). When writing graph manipulation code, I noticed that I had to be more explicit about instruction dependencies than I would in a sequential language, but I did not notice this when writing other code. This is because graph manipulation instructions have *side effects*. Writing such code in a fully sequential style and having the compiler infer the dependencies seems a good way to deal with this.

Some existing programming languages will be more easily adapted to the FGM than others. However, it will be difficult for the FGM to gain acceptance unless it supports a variety of programming styles. Even functional languages, due to their dependence on things like closures and function pointers, are more challenging to implement on the FGM

than one might expect. It seems reasonable that strongly-typed programming languages will have an advantage in terms of efficient implementation on the FGM. Parallelising an operation such as multiplying each element of a list by 2 is easier if you know they are all integers, for instance.

I have imagined a programming language where one connects together sockets of different software components to create larger components. This could begin right down at the level of individual instructions, and go all the way up to the level of program modules, programs, or even systems of programs. To avoid deadlock, the sockets being connected need to have matching completeness criteria — essentially, they must co-ordinate with each other correctly. It would be good if the programming language could keep track of this and verify it is being done correctly, for example in its type system.

6. Conclusion

Software wants to be diverse. There are many different sorts of problems that people want to solve, and having a diverse set of tools is part of the reason that computers are such useful machines. Sometimes developers are willing to sacrifice performance for flexibility. Sometimes they are willing to sacrifice flexibility for correctness. Sometimes they are willing to sacrifice correctness for functionality. An operating system must try to do a good job whatever is thrown at it, and hardware that is more flexible is a benefit in this regard.

The FGM seems very flexible. It unifies sequentiality and parallelism. It encompasses instruction-level parallelism, thread-level parallelism, and everything in between, as well as data-parallelism, in a single stroke. It also supports nondeterministic choice.

This means it has a vast number of potential applications. In particular it is likely to be very effective at solving problems without a regular structure. I think that this kind of machine ought to be good at solving graph-related problems, and it so happens that many of the management problems associated with the FGM are graph problems. This suggests that the hardware should provide graph manipulation facilities, which will assist the OS in managing the machine, and which the OS can extend and provide to applications.

I have found that an operating system for the FGM will need to fulfil much the same role as traditional operating systems do. However, such an operating system will need to be designed for scalability, and much work is required to develop the management techniques

such an OS will need to employ to be consistent with the FGM. I have explored many of the challenges that I identified, outlining ways of approaching them, and in some cases potential solutions. I have also discussed ways in which the hardware might be changed to fulfil potential needs of the OS and applications.

Designing a general-purpose operating system for a dataflow processor is a very interesting problem, and I think such an operating system is both feasible to create and an important exercise for the development of computer science — whether or not this particular OS for this particular machine will take the world by storm. It is my hope that this project will help lead to computational potential that seems far beyond our reach today.

A. Graph copy algorithm

```
#define NUM_BOUNDARY_SLOTS 200
#define MAX_LINKS 3

struct towerNode;
struct bridgeNode {
    int refcount;
    int payload;
    struct towerNode *constructionRefs[MAX_LINKS];
    struct bridgeNode *refs[MAX_LINKS];
};
struct towerNode {
    int refcount;
    int payload;
    struct bridgeNode *copy;
    struct towerNode *refs[MAX_LINKS];
};

typedef struct bridgeNode bridgePE[NUM_BOUNDARY_SLOTS];


void graphCopy(bridgePE b) {

    int freeIndex = 1;
```

```
//search for nodes with references to the original graph
for (int i = 0; i < NUM_BOUNDARY_SLOTS; i++) {
    for(int j = 0; j < MAX_LINKS; j++) {
        if (b[i].constructionRefs[j]) {

            struct bridgeNode *target;
            struct towerNode *orig;
            orig = b[i].constructionRefs[j];
            if (orig->copy) {
                //a copy already exists; use it
                target = orig->copy;
            } else {
                //allocate a new node
                target = &b[freeIndex];
                freeIndex++;

                //record this copy
                orig->copy = target;

                //copy the original into the new node
                //(but set the reference count to 0)
                target->refcount = 0;
                target->payload = orig->payload;
                //target->original = orig;
                for (int k = 0; k < MAX_LINKS; k++) {
                    target->constructionRefs[k] = orig->refs[k];
                    target->refs[k] = NULL;
                }

            }
```

```
                    //replace the reference to original node with copy
                    b[i].constructionRefs[j] = NULL;
                    b[i].refs[j] = target;

                    //count the new reference we just made
                    target->refcount++;
                    if (target->refcount == orig->refcount) {
                        orig->copy = NULL;
                        //refcount is complete, the copied node may now
                        //move away from the bridge
                    }


                }
            }

        }
    }
```

Bibliography

Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings,*, pages 483–485, Atlantic City, NJ, USA, April 1967.

John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:613–41, 1978. 1977 ACM Turing Award Lecture.

J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335 (2-3):131 – 146, 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.07.036. URL http://www.sciencedirect.com/science/article/pii/S0304397505000307.

A.D. Bailey, Jia Di, S.C. Smith, and H.A. Mantooth. Ultra-low power delay-insensitive circuit design. In *Circuits and Systems, 2008. MWSCAS 2008. 51st Midwest Symposium on*, pages 503 –506, August 2008. doi: 10.1109/MWSCAS.2008.4616846.

John J. Bainbridge. *Asynchronous System on Chip Interconnect*. Springer-Verlag New York, Inc., 2002. ISBN 185233598X.

Jeff S. Chase, Henry M. Levy, Michael Baker-Harvey, and Edward D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 80–85, Key Biscayne, FL, USA, 1992. IEEE.

Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS Conference Proceedings, 1965 Fall Joint Computer Conference*, 1965. URL http://www.multicians.org/fjcc1.html.

Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11:323–333, 1968.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

Edsger W. Dijkstra. The structure of the "THE" multiprogramming system. *Communications of the ACM*, 11:341–346, 1968.

Karl Fant. *Logically Determined Design: Clockless System Design with NULL Convention Logic*. Wiley, 2005. ISBN 9780471684787.

Karl Fant. *Computer Science Reconsidered: The Invocation Model of Process Expression*. Wiley, 2007. ISBN 9780471798149.

David A. Fisher. Copying cyclic list structures in linear time using bounded workspace. *Communications of the ACM*, 18:251–252, May 1975. ISSN 0001-0782. URL http://doi.acm.org/10.1145/360762.360764.

Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332,

London, UK, 2002. Springer-Verlag. ISBN 3-540-44044-5. URL http://portal.acm.org/citation.cfm?id=647424.725795.

Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. URL http://doi.acm.org/10.1145/1596638.1596653.

J.N. Gray. Notes on database operating systems. In G. Goos and J. Hartmanis, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

John R. Gurd. The Manchester dataflow machine. *Computer Physics Communications*, 37(1-3):49 – 62, 1985. ISSN 0010-4655. doi: DOI:10.1016/0010-4655(85)90135-3. URL http://www.sciencedirect.com/science/article/pii/0010465585901353.

John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, May 1988. ISSN 0001-0782. URL http://doi.acm.org/10.1145/42411.42415.

Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Graham R. Hellestrand. A distributed single address space system supporting persistence. Technical Report UNSW-CSE-TR-9302, University of NSW, University of NSW, Sydney 2052, Australia, March 1993.

Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM Operating Systems Review*, 41(4):3–11, July 2007.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

William D. Hillis. *The Connection Machine*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, March 1988. URL http://dspace.mit.edu/bitstream/handle/1721.1/14719/18524280.pdf.

Galen Hunt, Mark Aiken, Manuel Fhndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the 2nd EuroSys Conference*, pages 341–354, Lisbon, Portugal, April 2007.

Intel. First the tick, now the tock: Next generation Intel microarchitecture (Nehalem), January 2010. URL http://www.intel.com/pressroom/archive/reference/whitepaper_nehalem.pdf.

Krishna M. Kavi, Bill P. Buckles, and U. N. Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, 35:940–948, November 1986. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1986.1676696. URL http://dx.doi.org/10.1109/TC.1986.1676696.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of

an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. *Microarchitecture, IEEE/ACM International Symposium on*, 0:81, 2003. doi: http://doi.ieeecomputersociety.org/10.1109/MICRO.2003.1253185.

H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Proceedings of the 2nd International Symposium on Operating Systems*, pages 3–19, Rocquerncourt, France, October 1978.

Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 152–161, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: http://doi.acm.org/10.1145/277650.277715. URL http://doi.acm.org/10.1145/277650.277715.

Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 Workshop on Power Aware Computing and Systems (HotPower'10)*, Vancouver, Canada, October 2010.

Edward A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.180. URL http://portal.acm.org/citation.cfm?id=1137232.1137289.

Gary Lindstrom. Copying list structures using bounded workspace. *Communications of*

the ACM, 17:198–202, April 1974. ISSN 0001-0782. URL http://doi.acm.org/10.1145/360924.360936.

R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322017.322025.

Justin Mazzola Paluska, Hubert Pham, and Steve Ward. Structuring the unstructured middle with chunk computing. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, Napa, CA, USA, May 2011.

Matthew Naylor and Colin Runciman. The reduceron reconfigured. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 75–86, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. URL http://doi.acm.org/10.1145/1863543.1863556.

Taiichi Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, March 1988. ISBN 9780915299140.

Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999. URL http://www.eros-os.org/papers/shap-thesis.ps.

J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Journal of Parallel and Distributed Computing Practices*, 1:1–33, 1998.

Alan Skousen and Donald Miller. Using a distributed single address space operating system to support modern cluster computing. In *Hawaii International Conference on System Sciences*, 1999.

Kohji Tomita, Haruhisa Kurokawa, and Satoshi Murata. Graph automata: natural expression of self-reproduction. *Physica D: Nonlinear Phenomena*, 171(4):197 – 210, 2002. ISSN 0167-2789. doi: DOI:10.1016/S0167-2789(02)00601-2. URL http://www.sciencedirect.com/science/article/pii/S0167278902006012.

A. L. Wilkinson, D. H. Anderson, D. P. Chang, Lee Hock Hin, A. J. Mayo, I. T. Viney, R. Williams, and W. Wright. A penetration analysis of a burroughs large system. *SIGOPS Operating System Review*, 15:14–25, January 1981. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1041454.1041455. URL http://doi.acm.org/10.1145/1041454.1041455.