Context Flow Architecture

Timothy E.A. Lees

Ph.D.

University of Edinburgh

1990

Abstract

Good computer architecture is, in many ways, very similar to good building architecture. Its effectiveness can only be judged by the way in which the implementation of the design — whether a computer or a building — fulfils its given role. In computer architecture, this judgement is based on one criterion — *speed.* In the best computers, speed is obtained by coupling the best current technology and materials with the best design. This Thesis presents a novel way in which to design pipelined computers. Speed is achieved by maximising the use of hardware resources to provide an environment in which many independent processes can execute concurrently in a single system. The design method is called *context flow.*

Two different facets of context flow are discussed. An underlying theory of context flow is established which is used to prove certain properties of context flow systems. These theoretical results show context flow machines to be implementable. Using these results, a practical approach to the creation of context flow systems is presented, leading to the design and analysis of an example context flow processor. The result is an architectural design technique with a formal foundation which can be used to build efficient pipelined computers.

Acknowledgements

Like any voyage of discovery, production of this Thesis, although credited to one person, requires the help of many others. I would like to record my gratitude to all the people helped me on my four-year mission, which although not "seeking out new life forms and civilizations", did attempt to "boldly go where no man has gone before".

I would like to thank my supervisor, Nigel Topham, for his patience and guidance over the past four years; Doug Gurr, for helping to organize the ramblings of a non-theorist; and Tom Stiemerling, Steve Proctor and Richard Eyre-Todd, with whom I have shared the tears of frustration and laughter in Room 2412.

For my eight years at Edinburgh University, the Boathouse by the Union Canal has been a second home, and the members of Edinburgh University Boat Club — Hamish, Sam, Tony, Tom, Alison, Mary, Steve, Geoff, Simon, Brian, Simon, Vince, Bill, Will, Rob, Ed, Rupert, Mike, Kate, Rob, Richard, Ben, Cameron, Ian and Gary to name but a few — a second family. I cannot thank each of you enough.

This work was funded by a Research Studentship from the Science and Engineering Research Council.

Publications

The following papers were published during the production of this Thesis.

- T.E.A. Lees. "On Context Streams and the Boundedness of Context Flow Graphs", *Internal Report CSR–2–90*, Dept. of Computer Science, Edinburgh University.

- T.E.A. Lees. "On Context Stream Tuples and Higher-Order Context Flow Graphs", *Internal Report CSR–4–90*, Dept. of Computer Science, Edinburgh University.

- T.E.A. Lees. "Context Streams — A Theoretical Basis for a Generic Form of MIMD Pipelining". In *Proc. 2nd IEEE Symposium on Parallel and Distributed Processing*, (Dallas, Texas; Dec. 9–11 1990).

Table of Contents

List of Figures

List of Tables

Preface

This Thesis is concerned with the efficient implementation of pipelined computers. It presents a design technique called *context flow*, which overcomes a number of the inefficiencies of current pipelined systems by removing the source of these inefficiencies from the pipeline, rather than merely attempting to lessen the effects when they arise. This Thesis attempts to combine a theoretical foundation for context flow, with practical designs for pipelined context flow systems, to show context flow as an effective means of supporting multiple concurrent processes in a highly-pipelined environment.

Chapter 1 argues the case for pipelining, as opposed to replication, as a means to exploit parallelism while retaining generality of purpose. The principles of pipelining are presented, together with existing solutions for the problems found in conventional pipelined systems. Chapters 2 presents a theoretical basis for context flow, which is developed in Chapter 3. Chapters 4 and 5 concern the implementation and performance of context flow systems, providing a collection of context flow versions of common architectural elements, and a design for a context flow processor. Chapter 6 discusses the implementation and applications of context flow systems and outlines directions for future research.

To my parents.

Please accept this as part repayment

of the huge debt of gratitude I owe you.

Chapter 1

Introduction

Many significant improvements in computer performance have been brought about through the application of technological advances. For example, improved integrated circuit processing techniques have allowed denser and more complex very large scale integration (VLSI) devices to be fabricated, and materials research has yielded substrates such as gallium arsenide which can sustain faster switching speeds than silicon. Desires to increase performance still further, or to avoid the high costs associated with innovative technologies and materials, have prompted the search for methods to improve performance by non-technological means. One area which has been particularly fruitful is the introduction of techniques to exploit parallel or concurrent activities in a computer system and thus allow existing resources to be utilized to the full. This chapter describes current methods which provide temporal and spatial solutions to the exploitation of parallelism, and presents some of the associated problems.

1.1 Parallel Architectures

Parallel processing can be defined as

> "an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Parallel events may occur in multiple resources during the same time interval; simultaneous events, at the same time instant; and pipelined events during overlapped time spans."
>
> [Hwang and Briggs, 1984]

Parallelism can be introduced into a computer system by either hardware or software means at a number of different levels. Software solutions predominate at higher levels of abstraction where parallelism is introduced either between jobs or programs by means of multiprogramming or time sharing of resources; or between procedures within the same program. Hardware mechanisms tend to be used to extract parallelism at a lower level — either between instructions or within the execution cycle of a single instruction. Design of a parallel architecture is therefore a trade-off between the costs of software and hardware approaches.

1.1.1 Classification of Parallel Architectures

The above definition of parallel processing covers a diverse range of architectures, which need to be classified if meaningful comparisons of performance are to be drawn. Four characteristics commonly used to distinguish architectures are:

Generality of purpose A general purpose machine attempts to provide a given level of performance across a wide spectrum of applications. Special purpose machines perform their designated task well, but other tasks

poorly or not at all, and often require the data either to be in a rigidly defined format or to be programmed in a particular type of language.

Granularity The range of sizes of basic processing units at which a system performs most efficiently determines many other characteristics. A parallel system can be *coarse-grained*, consisting of a few complex processors; or *fine-grained*, containing hundreds or thousands of simple processors.

Topology and reconfigurability The way in which processors are connected is important, as is the ease with which the interconnection pattern can be changed to suit a given algorithm.

Coupling This feature describes the distribution of clock signals and the location of memory relative to the processing units. *Tightly-coupled* systems have a global shared memory, whereas in *loosely-coupled* systems, memory is distributed physically amongst the processors.

Classification also serves to relate past and present architectural developments, and to aid clarification of design concepts.

The diverse nature of parallel architectures has lead to the evolution of several design classification schemes. Flynn [1972] identifies four basic machine types, based on the characteristics of the instruction and data streams:

- Single Instruction Single Data (SISD) — a conventional uniprocessor in which each instruction operates on a single datum.

- Single Instruction Multiple Data (SIMD) — a single instruction specifies the operation to be applied to several data.

- Multiple Instruction Single Data (MISD) — included largely for completeness sake.[1]

- Multiple Instruction Multiple Data (MIMD) — several instruction streams are processed simultaneously, each operating on its own data.

This taxonomy is somewhat crude, as the SIMD and MIMD classifications each encompass machines with very different architectures. Under Flynn's scheme, the SIMD group includes *vector* processing machines, in which one instruction initiates operations on every element of a vector in turn; and *array* processors, in which a collection of processors operate synchronously on regular arrays of data under centralized control. Hockney [1987] differentiates these two very different groups as pipelined and replicated-SIMD, respectively.

Skillicorn [1988] presents a more complete classification scheme, which subsumes those of Flynn and Hockney. Twenty-eight machine classes are identified according to the number of instruction processors and data processors and their interconnection topology. This taxonomy covers most current architectural forms including graph reduction and data-flow machines, von Neumann uniprocessors, array processors, and a wide variety of multiprocessors.

Johnson [1988] provides an alternative taxonomy for machines in Flynn's MIMD category, which allows distinctions to be drawn according to factors which are more pertinent from a programmer's point of view. Under this scheme, multiprocessor machines are divided into four classes:

---

[1] Krishnamurthy [1989] cites pipelined systems as examples of MISD processing. However, it is somewhat tenuous to describe the stages in a pipeline as instruction streams. Flynn could offer no examples of a machine in this class.

- Global Memory Shared Variable (GMSV). Machines in this class have a common memory, accessible at equal cost by each of the processors, and use shared variables to provide process synchronization. Examples include the New York Ultracomputer [Gottlieb *et al.*, 1983], and the University of Illinois Cedar [Kuck *et al.*, 1987].

- Distributed Memory Shared Variable (DMSV). Machines in this class have memory modules connected to each processor, but which are globally accessible via a common address space. An example of a DMSV architecture is the BBN Butterfly [Rettberg and Thomas, 1986].

- Distributed Memory Message Passing (DMMP). This class includes machines such as NCUBE [Hayes *et al.*, 1986], in which each processor maintains its own address space and communicates with other processors across an interconnection network.

- Global Memory Message Passing (GMMP). This category completes the taxonomy, but contains few actual machines. Each process would have an isolated address space within a common memory.

It is difficult to see the development of a classification scheme which can capture both architectural and operational aspects of all machines. Indeed, with Flynn's scheme in such widespread use, it seems likely that its ambiguities will be tolerated for some time to come.

1.1.2 Exploitation of Parallelism in Uniprocessors

Several improvements, both architectural and in software, can be made to single processor machines to maximize use of processor resources, and thus to achieve faster program execution.

Pipelining

The instruction execution sequence of a processor is typically broken down into several phases — *instruction fetch*, *instruction decode*, *operand fetch*, and *instruction execution*. In a non-pipelined processor, each instruction proceeds through each of the execution phases before processing of the next instruction begins. This is wasteful of processor resources, as at any given time the hardware which supports all but one of the phases is idle. A *pipelined* processor begins execution of a new instruction after each has passed through the first stage, allowing all stages to remain in continuous use and, in the case of a k stage pipeline, producing a k-fold increase in performance. Pipelining is covered in greater depth in Section 1.2.

Multiple Function Units

A conventional arithmetic and logic unit (ALU) contains circuits to perform several operations, typically addition, subtraction, multiplication, and the boolean logic functions, but can only perform one operation at a time. By transferring execution of each function to a separate unit, and replicating commonly used operators, increased completion of these operations can be achieved. The Control Data Corporation (CDC) 6600 [Thornton, 1964] was the first computer to include functional parallelism as a major design feature. It provided an execution unit containing ten functional units (FUs) under the control of a device called the *scoreboard* which regulated access to the FUs, and resolved data dependency conflicts between instructions. The IBM 360/91 [Tomasulo, 1967] provided separate execution units for floating-point (FP) and scalar instructions, with the floating-point unit (FPU) capable of performing concurrent addition and multiplication operations, data dependencies permitting.

Input-output and Memory Systems

The mismatch in speed between input-output (I/O) operations and processor operations can be overcome by the introduction of dedicated processors whose operations are initiated by the central processor but run independently thereafter. This allows the central processor to continue to perform useful computation while waiting for completion of an I/O operation. The I/O processor can communicate with the memory via a direct memory access (DMA) channel, allowing transfer of data at the maximum possible rate.

The mismatch in speed which exists between the main memory and the processor can be overcome in a number of ways. By judicious placement of instructions or data in an intermediate high speed memory or *cache*, the traffic between the processor and main memory can be reduced substantially. This technique is particularly effective when a program exhibits considerable locality either in the form of iterative loops or in repeated access to a small number of memory locations. A technique called *interleaving*, can be used to exploit the sequential nature of memory references. A k-way interleaved store has k independently accessible memory modules with consecutive locations stored in adjacent *modules*, giving a k-fold increase in access rate, although the time taken to access individual locations remains the same.

Multiprogramming

Multiprogramming is a software technique which allows processor-intensive and I/O-intensive programs to share the resources of a computer system. The mix of programs being executed can be controlled by a process called *scheduling*, to balance access to available resources according to the needs and priority of each program. By dividing processor availability into discrete intervals or *time-slices* a single high-priority process can be prevented from monopolising a given resource.

A combination of multiprogramming and time-slicing in an operating system provides an illusion of concurrency to the user of a uniprocessor system.

Vector Processing

Many scientific and numerical problems require evaluation of expressions of the form

$$\overline{a} = \overline{b}\ op\ \overline{c}$$

where $\overline{a}$, $\overline{b}$ and $\overline{c}$ are vectors of the form

$$\overline{x} = (x_1, x_2, \cdots, x_n)$$

and the operation performed is

$$a_i = b_i\ op\ c_i\ \forall i \in \{1, \ldots, n\}$$

By using a single instruction to accomplish this operation, rather than repeated application of the operator within a loop, a considerable amount of the overhead associated with processing the instructions can be saved. The performance of vector processors is difficult to estimate as it is very dependent on the size and nature of the application. The CRAY 1, the first commercial processor to include vector processing facilities, has a typical performance of 20 MFLOPS (floating-point operations per second), and a theoretical peak performance of 160 MFLOPS. Currently, the fastest available computer is the CRAY Y-MP/8, with a peak performance of 2.667 GFLOPS [Bell, 1989]. Vector processing is, however, an efficient method of solving particular classes of numerical problems.

1.1.3 Array Processors

An *array processor* is a regular collection of ALUs or *processing elements* (PEs) which operate synchronously under lockstep control. Each PE is a device capable of applying a series of simple operations to small data objects stored in a

local memory. Each PE executes the same operation simultaneously, with a single stream of instructions being broadcast to all elements by a central controller. PEs are interconnected by a routing network to allow exchange of data. Array processors tend to be less general purpose and more difficult to program than other parallel machines, being best suited to solving problems with data whose structure closely matches that of the interconnection network. Array processors are classified according to the size of data each PE operates upon, either single/few-bit or floating-point quantities, and the physical interconnection of the processors. Research interest in array processors was initiated by the SOLOMON project [Gregory and McReynolds, 1963], a 32×32 PE array developed by the Westinghouse Electric Corporation.

Bit-serial Arrays

The ICL Distributed Array Processor (DAP) was one of the earliest array processors to be developed commercially. The DAP consisted of a 64×64 array of single bit processors connected to a master control unit. The design of the DAP was similar to that of SOLOMON but with the introduction of two novel features — separation of column-wise and row-wise access to PEs, and integration of the array as part of the memory of an existing system (one of the 2900 series). Each PE consisted of a 4 kbit memory and a 1-bit full adder, together with input and output multiplexers and three 1-bit registers. The DAP had two modes of operation, *matrix* mode in which 4096 words were processed concurrently in a bit-serial manner, and *vector* mode in which sixty-four 64-bit words are processed in parallel.

It was, perhaps, the DAP's dependence on its 2900 series host that ultimately limited its commercial success, the host machine being expensive and superseded largely by other machines and technologies soon after the introduction of the DAP. In 1986, Active Memory Technologies (AMT) launched the mini-DAP, a 32×32 PE

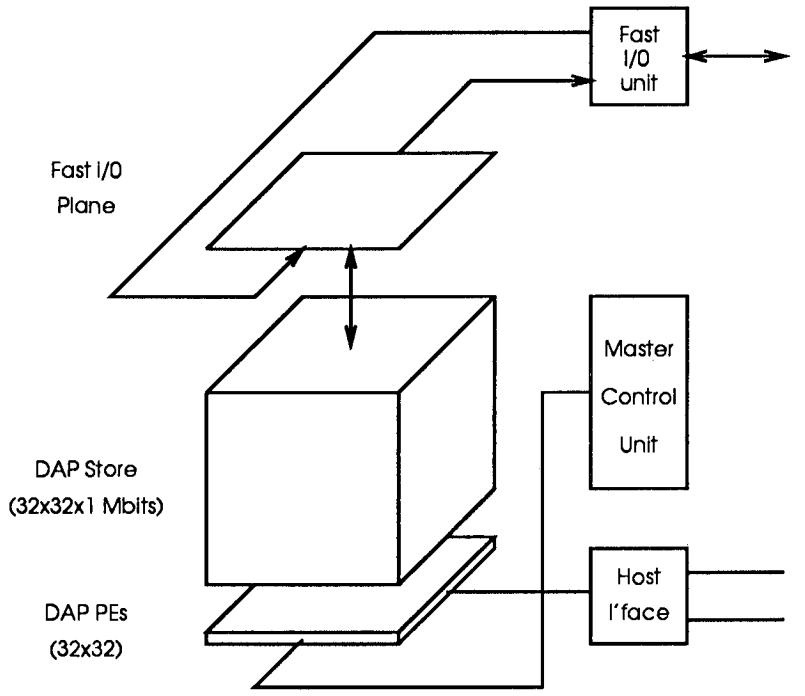


Figure 1–1: AMT mini-DAP Architecture

array built from LSI components. The design, as shown in Figure 1–1, is similar to the mainframe DAP, with the addition of a fast I/O unit, a more generalized host interface, and 1 Mbit memories for each PE.

The Connection Machine [Hillis, 1985] is a very large, fine-grain parallel computer developed for artificial intelligence applications at Massachusetts Institute of Technology, and implemented commercially as the CM-1 by Thinking Machines Corporation. A connection machine consists of a large number of processors (65,536 in the CM-1), connected by a communication network. Each PE has a small amount of local memory and a 1-bit ALU, and is assigned a unique address in the network. The PEs are controlled by a central processor via a global instruction bus. Programs written for the connection machine [Christman, 1984] consist of two parts — a description of the *connection*, an arbitrary graph having

one PE at each vertex, and a description of the operations the controlling computer is to send to each PE. The power of the connection machine lies not in the power of the individual processors, but in their quantity. This approach has been taken a stage further with the development of neural network systems [Hopfield, 1979].

Bit-parallel Arrays

The SOLOMON project was also the stimulus for a series of architectures whose processors operated on whole words of data rather than single bits. The first of these was ILLIAC IV [Barnes *et al.*, 1968], developed at the University of Illinois and which became operational in 1972. It consisted of 64 PEs arranged on an 8×8 grid. Each PE performed addition, multiplication, logical and shifting operations on several formats of data, from 64-bit floating-point numbers to 8-bit characters. A 2048-word memory provided local storage for each PE. Despite problems with reliability, delivery and budget, the ILLIAC IV made a valuable contribution to research in parallel architectures, and influenced many subsequent designs.

The Burroughs Scientific Processor (BSP) [Jensen, 1978] developed many features found lacking during production of ILLIAC IV. The size of memory associated with each PE was increased to 128 kwords, and the processing capability of the controlling processor was increased to allow more complex scalar operations to be performed. Due to problems within Burroughs Corporation, the design did not meet its performance goals, and the BSP never went into production.

1.1.4 Multiprocessors

Multiprocessor systems are single computers which contain several processing units which communicate and cooperate at various levels on the solution of a single problem. No matter where in a multiprocessor system the memory is located, the

structure of the network linking the processing elements and memory modules has an important bearing on system performance, with the choice of topology being a trade-off between connectivity and cost.

Interconnection Networks

In any multiple processor system, making the right data available to the right processor at the right time is essential for the full exploitation of available parallelism. The physical interconnection of the processors has been the subject of great interest for designers of multiprocessor systems [Feng, 1981],[Haynes *et al.*, 1982]. There are two types of interconnection network, *static*, with dedicated buses, or *dynamic*, with reconfigurable interprocessor links.

Static interconnection networks cover a large variety of topologies, all characterized by the existence of fixed dedicated links between processors. Shared bus, star and ring networks provide simple, low-cost interconnection between a few processors. However, their usefulness is severely constrained by bandwidth limitations, and lack of fault-tolerance. Although these three topologies have been successfully applied to local area networks, their suitability for use in tightly coupled multiprocessor environments is limited.

A binary tree would seem to be a very natural way of organizing an interconnection network, as many problems can be described in terms of a tree structure, and a natural correspondence exists between the hardware processors and the software processes. However, the performance of the tree is potentially limited by the root node if a large amount of communication has to take place between the two halves of the tree, and the depth of the tree is limited by the number of processors. An analysis of the number of messages transferred through a node, in the case when each node sends a message to all other nodes in a network of n nodes [Horowitz and Zorat, 1981], shows that a binary tree $[O(n \log_2 n)]$ compares favourably with a linear array $[O(n^2)]$, and with a 2-dimensional array $[O(n^{3/2})]$.

A n-Cube can connect $k = 2^n$ processors, providing a rich interconnection network. The ensemble can be thought of as a cube in n-dimensional space, with the processors at the vertices and the connections forming the edges, with the distance between any two processors $\propto \log_2 n$. A possible source of difficulty is that n connections are required per processing element, and that for $k > 1000$ wiring problems may be encountered. The cube-connected cycles [Preparata and Vuillemin, 1981] is similar to the n-Cube, but with the vertices replaced by a cycle of processors. This has the advantage of a constant number of connections per processor. In an n-dimensional hypertorus, each processor is a member of n orthogonal rings, the distance between any two processors being proportional to the n^{th} root of the number of processors.

A large number of dynamic routing networks have been designed offering a variety of trade-offs between hardware complexity and operational efficiency. A *single stage* network is composed of a set of switching elements connecting n inputs to n outputs. The single stage network is also known as a *recirculating network* as data items may have to circulate through the single bank of switching elements several times before reaching their correct destination. A *multistage* network consists of more than one stage of switching elements, and is capable of establishing arbitrary connections between inputs and outputs. The multistage networks can be further split into *blocking*, *non-blocking* and *rearrangeable* types. In a blocking network, the establishing of a connection between two ports may cause a conflict when subsequent connections are attempted. This class includes delta and omega networks. A rearrangeable network can alter existing connections to accommodate a new path between two ports, for example the Benĕs network. The third type, the non-blocking network, can handle all possible connections without the occurrence of blocking, for example the Clos network. Multistage networks such as the banyan network tend to have a hardware complexity of

$O(n \log_2 n)$ compared to connections schemes like the crossbar switch which have a hardware cost that grows $O(n^2)$

Shared Memory MIMD Computers

Shared memory systems consist of separate processing and memory units, connected by means of a network allowing uniform accessibility to all memory locations by any processor. Memory contention and its management are important factors in determining machine performance.

A bus provides a simple and easily configurable means of connecting processor and memory units. The Sequent Balance has a 32-bit bus-connected shared memory architecture. Up to 24 processing units, each constructed from NS32032 processors with an 8 kbyte cache and floating-point unit, and 28 Mbytes of memory can be attached to a 52-bit pipelined packet bus. Use of caches reduces the number of accesses to the common memory and hence the effect of contention on performance, at the expense of maintaining coherency between the contents of each cache and the memory. The Tandem–16 Nonstop system illustrates another feature of multiprocessors — fault tolerance. Replication of resources within this system allows continuous operation and repair of faults without bringing down the entire system.

A crossbar switch offers complete connectivity between processors and memories, with the rate of transfer limited only by the number of memory modules, rather than the availability of connective paths. With current technology, a crossbar switch is only practical if the number of processors is relatively small; and is the preferred interconnection topology for sixteen or fewer processors. One of the earliest MIMD computers, the C.mmp [Wulf and Bell, 1972] developed at Carnegie Mellon University, used a 16×16 crossbar switch (S.mp) to connect Digital Equipment Corporation (DEC) PDP–11/40 minicomputers to shared memory modules. A derivative of C.mmp, the Livermore S-1 [Farmwald, 1984], consists of 16 Cray–1

processors fully connected to 16 memory modules by a crossbar switch, providing a 16 Gbyte address space and an estimated performance of 1 GFLOP.

Multistage interconnection networks provide a cost-effective means of connecting large numbers of processors and memory units. The New York Ultracomputer project has produced a series of designs for machines based around an Omega network connecting a large number of processing elements to a shared memory constructed from distinct memory modules. A novel network switching element is used to combine access requests to the same memory location, thereby reducing contention. A 512 processor implementation is under construction as the IBM Research Parallel Processor Project (RP3) [Pfister *et al.*, 1985].

Distributed Memory MIMD Computers

Each processing element in a distributed memory MIMD computer has an associated memory module which can be accessed by it more easily than by other processors. Access to non-local memory is provided by an interconnection network. The non-uniformity of memory access requires that placement of code and data among the processors be optimized to achieve maximum performance.

Star and ring networks have only been used in a few multiprocessor systems, for example the IBM ℓCAP and the CDC CyberPlus. Hierarchical network structures have been employed successfully in some projects, notably Cm*, the successor to C.mmp, based on bus-interconnected clusters of microprocessors. The hypercube has received much interest as a network topology for MIMD computation. The Cosmic Cube [Sietz, 1985] built at California Institute of Technology is a 6th-order hypercube, the nodes being constructed from Intel microprocessors. Several commercial machines have been derived from this design, including the NCUBE and the Floating Point Systems T-Series.

The BBN Butterfly consists of up to 256 computing nodes connected by a Banyan network. Each node consists of a Motorola 68020 with a 68881 floating-point co-processor and a 4 Mbyte local memory. Although the memory is *physically* distributed among the nodes, it forms a continuous *logical* shared address space with non-local memory addresses being accessible via the network.

The INMOS Transputer is a microprocessor designed as a building-block for MIMD computers. A processing unit and memory are combined with hardware support for process queues on a single chip. Interprocess communication is supported by instructions for passing messages either internally through a common memory location, or externally through one of the four hardware links provided on each transputer. The transputer has been used in a variety of parallel processing applications, from large high performance systems such as the Edinburgh Concurrent Supercomputer, to specialist processing systems such as the ALICE graph reduction machine [Darlington and Reeve, 1981].

1.2 Pipelined Architectures

Pipelining is a technique developed to exploit the temporal parallelism that exists in the execution of instructions within a computer. A *pipelined* processor allows operations to be initiated before previous instructions have been completed, resulting in an increased instruction execution *rate*, while the instruction execution *time* remains almost unchanged. Instruction completion rate then becomes a function of instruction issue rate, rather than total processing time. Concurrent execution of different instructions is made possible by dividing the computational process into hardware-independent *stages*. In an ideal pipeline, each stage is of equal complexity, requiring the same time to complete its operation. Any discrepancy between execution time results in a bottleneck, as rate of progression through the pipeline is dependent on the longest time taken to complete a single stage, thus the rate

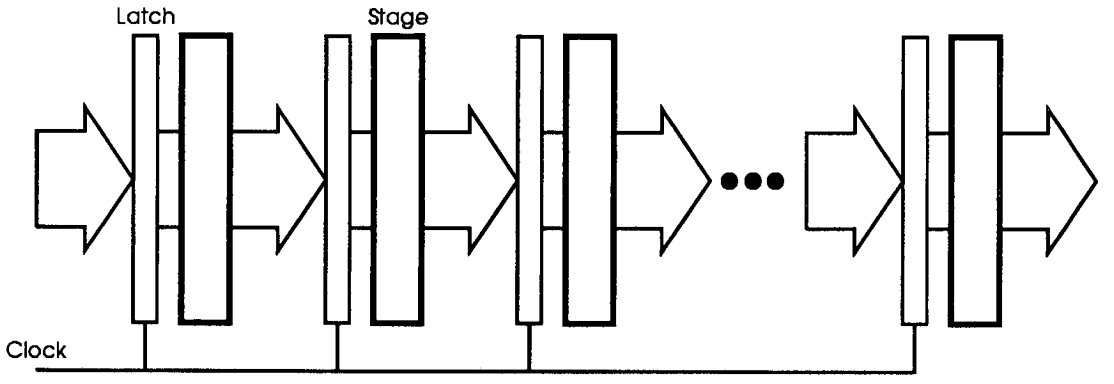


Figure 1–2: Linear Pipeline Structure

of instruction completion is reduced. Stages are connected by *latches* or *buffers* which store intermediate results. Figure 1–2 shows the basic structure of a linear pipeline.

Given a time τ_s to perform the operation of each stage, and a delay τ_l associated with each latch, the *beat time* τ_p of the pipeline is given by

$$\tau_p = \max(\tau_s)_1^k + \tau_l \tag{1.1}$$

As the execution time for individual instructions is not changed, the time taken to complete a given number of tasks is the time taken to complete one task — each stage taking τ_p, plus the time to complete the remainder — a result being produced every τ_p. Thus for a k-stage pipeline, the time to complete N tasks is

$$k\tau_p + (N-1)\tau_p \tag{1.2}$$

The time to complete the first task is also known as the *fill time* of the pipeline, and has an important bearing on pipeline performance in the presence of discontinuities in an instruction stream.

The *speedup* of a k-stage pipeline over an equivalent non-pipelined machine is defined as

$$S_k = \frac{Nk}{k + (N-1)} \tag{1.3}$$

where, for $N \gg k$, $S_k \rightarrow k$, thus the maximum speedup achievable in a k-stage linear pipelined is a factor of k.

The *efficiency* of a pipeline is the percentage of the total time taken to execute a given number of tasks for which useful results are produced

$$\eta = \frac{Nk\tau_p}{k\left(k\tau_p + (N-1)\tau_p\right)} = \frac{N}{k + (N-1)} \tag{1.4}$$

From Equation 1.4, $\lim_{N\rightarrow\infty} \eta = 1$, implying that a greater efficiency can be obtained with a greater number of tasks. From Equation 1.3, an alternative definition of efficiency can be derived,

$$\eta = S_k/k \tag{1.5}$$

as the ratio of *actual* speedup to *theoretical* speedup.

The *throughput* of a pipeline is the rate of task completion

$$w = \frac{N}{k\tau_p + (N-1)\tau_p} = \frac{\eta}{\tau_p} \tag{1.6}$$

the maximum throughput being one result per clock cycle, attained when $\eta = 1$.

Pipeline techniques can be applied at several levels within a computer system, a common instance being the instruction processing section of a processor. Instruction execution is easily divisible into separate operations which can be applied sequentially:

- Instruction fetch: obtain the instruction code from memory.
- Instruction decode: determine operation to be performed and how to locate operands.

- Address generation: calculate the effective address of the operand(s) with indirection and offset as necessary.
- Operand fetch: obtain the operands from memory.
- Instruction evaluation: execute the instruction on the specified operands.
- Operand store: update memory with result of evaluation.
- Program counter update: generate address of next instruction.

Pipelining is also found in the arithmetic units (AUs) of many processors. The process of floating point addition, for example, can be divided into four steps — exponent subtraction, mantissa alignment, mantissa addition and normalization. Early examples of machines with pipelined AUs were the Texas Instruments Advanced Scientific Computer (ASC) [Watson, 1972], which contained four pipelined AUs, and the IBM System/360 Model 91 [Tomasulo, 1967]. Incorporation of pipelines in microprocessors is more recent, but nonetheless gaining in importance. The Intel i860 microprocessor [Intel Corporation, 1989] contains pipelined FP addition and multiplication units which can run concurrently with the core (integer and control) unit, yielding a maximum performance of 80 MFLOPS. This level of performance is made possible by limiting the length of the FP pipeline to three stages.

Pipelines can be classified according to three main features, the function performed, the configuration of the stages, and the data being operated on. A pipeline is said to be *uni-functional* if it performs a single dedicated function and, by definition, has a *static* configuration. A *multi-functional* pipeline can be configured in a variety of ways to perform different functions, as in the Texas ASC. A static multifunctional pipeline can only be reconfigured between batches of data, while a *dynamic* multifunctional pipeline permits several simultaneous configurations.

Processing streams of data in a vector pipeline is a natural application of pipelining, as is repeated processing of a sequence of scalar operands.

In the presence of a regular flow of instructions, a pipeline is an effective means of increasing processing ability. However, most instruction streams are irregular, requiring the following problems to be addressed:

- Conditional branch instructions require the outcome of a test to be known before the target address can be determined. Any instructions entering the pipeline after the branch but before the test has completed, may need to be discarded.

- An instruction must be delayed in the pipeline if it requires the result produced by an earlier instruction.

- An instruction must not overwrite a memory location or register whose previous contents are required by an earlier instruction.

- The pipeline must not change the order of updates of a memory location by successive instructions.

1.2.1 Instruction Stream Discontinuities

A smooth flow of instructions through a pipeline is essential if efficient operation is to be achieved. This, however, can be disrupted by a change in the expected sequence of instructions, due to control transfer instructions. Branch instructions play an important part in structured languages in one of two forms, *unconditional* branches, for example to subroutines, and *conditional* branches in loops and conditional statements.

The address of the instruction to be executed after a conditional branch may not be known until the branch instruction has almost completed its passage through the pipeline. If the

branch is to be taken, the pipeline must be refilled from the target address. Not only is processing time wasted in partial evaluation of the instructions immediately after the branch, but these instructions may also have changed register or memory values.

The Branch Penalty

The penalty associated with branching becomes more severe as pipeline length is increased. For a given pipeline, let k denote the number of stages which may be required to be flushed when a branch instruction is encountered, p_b denote the proportion of instructions which are branches in a program, p_t denote the probability that a branch will be taken, and CPI denote the average number of cycles required to execute a single instruction, after the initial start-up latency. Then

$$\text{CPI} = 1 + (k-1)p_b p_t \tag{1.7}$$

and the effective performance of the pipeline [Lilja, 1988], F_0 is

$$F_0 = \frac{1}{1 + (k-1)p_b p_t} \tag{1.8}$$

Thus, maximum performance may only be obtained if

- the pipeline has only a single stage ($k = 1$),
- there are no branch instructions ($p_b = 0$), or
- the branch instructions are never taken ($p_t = 0$).

The first two conditions cannot be met, therefore the objective of pipeline design must be to reduce the value of p_t. While it is not possible to ensure that branches are never taken, always fetching the next instruction from the correct

target address has the same effect. Equation 1.8 also shows that the branch penalty increases with pipeline length.

If no steps are taken to reduce p_t, performance drops on average by 14% for a two stage pipeline, and by 55% if the pipeline length is increased to 6 stages, using the values $p_b = 0.242$ and $p_t = 0.676$ measured by Lee and Smith [1984].

Hardware Solutions

A simplistic solution to the branching problem would be to replicate the initial stages of the pipeline, and process the instructions following both the branch and its possible target. Despite additional problems of resource contention between the instruction streams, occurrence of multiple branch instructions in the pipelines, and the cost of replicating the hardware, this method was implemented in the IBM System/370 Model 168 and 3033 machines. The hardware cost can be reduced if duplication is confined to sufficient logic to prefetch the instruction at the branch target address. If the branch is taken, the target instruction can be loaded without delay. A similar approach employed in several machines is *loop catching.* A small high speed memory is provided which acts as a "loop cache", allowing instructions from repeatedly executed segments of code to be presented to the pipeline without the additional delay of fetching them from memory. The latter two strategies can, however, only limit the effects of taking a conditional branch, and the pipeline must always be refilled.

Branch prediction strategies can either be static, for example, always assuming branches are taken when generated by certain instructions, and pre-fetching accordingly; or dynamic, for example, maintaining a taken/not-taken bit for each branch instruction, updated according to some heuristic algorithm. Assuming that a correct prediction incurs no penalty, and that the penalty is the same whether

the branch is taken or not taken, then the effective performance becomes

$$F_{pred} = \frac{1}{1 + (k-1)p_b p_w} \tag{1.9}$$

where p_w is the probability of a wrong prediction. Branch prediction provides a performance improvement as long as the probability of a wrong prediction is less than the probability that a branch will be taken anyway ($p_w < p_t$). McFarling and Hennessy [1986] show that prediction accuracies of between 81% and 85% are possible, yielding a value of p_w between 0.150 and 0.190, which is considerably less that the value of p_t measured by Lee and Smith. Ditzel and McLellan [1987] report accuracies as high as 95%.

An alternative approach caches branch instructions with destination addresses in a *branch target buffer* (BTB). This approach was used in the instruction buffer unit of the MU5 [Ibbett, 1982], which contains an associative store in which are held eight pairs of branch addresses and their targets. When a new instruction address is generated, the associative store is searched, and if a match is found, the target address replaces the original address, and instructions are fetched from the target. These instructions are flagged as "out of sequence" so that if the prediction was incorrect, they can be discarded.

Given a BTB hit ratio p_h and an additional m cycles to fetch an instruction in the event of a miss, the effective performance becomes

$$F_{btb} = \frac{1}{1 + [((k-1)+m)(1-p_h)]\, p_b p_t} \tag{1.10}$$

For use of a branch target buffer to be effective,

$$p_h > \frac{m}{((k-1)+m)} \tag{1.11}$$

Branch folding [Ditzel and McLellan, 1987] is a technique which eliminates unconditional branches from pipelines. In the CRISP processor [Ditzel *et al.*,

1987] a three stage execution pipeline is used in an architecture comprising a prefetch and decode unit, a decoded instruction cache and an execution unit. Instructions are fetched in encoded form from main memory, decoded and stored in the instruction cache in 192-bit form. Associated with each decoded instruction is a next-address field, having the effect of making each instruction a branch. This eliminates the need for the execution of separate branch instructions, which, instead are *folded* with the preceding instruction and deleted from the pipeline. Conditional branches contain an additional alternative-address field, not unlike words of microcode, which is used in conjunction with a single static prediction bit.

The effective performance of branch folding is

$$F_{bf} = \frac{1}{1 + (k-1)p_{cb}p_t} \tag{1.12}$$

where p_{cb} is the probability that an instruction will be a conditional branch. In the case of CRISP, which employs branch prediction and a three stage pipeline

$$F_{CRISP} = \frac{1}{1 + 2p_{cb}p_w} \tag{1.13}$$

generating a performance improvement if

$$p_{cb}p_w < p_b p_t$$

As, by definition, p_{cb} is always less than p_b, even the simplest static prediction scheme, which always assumes that a branch will be taken, provides a performance enhancement.

Software Solutions

The previous attempts to solve or ameliorate the branching problem all assume that if instruction i were a taken branch, then instruction $i + 1$ would be out of sequence. However, it is possible to promote the branch instruction so that the

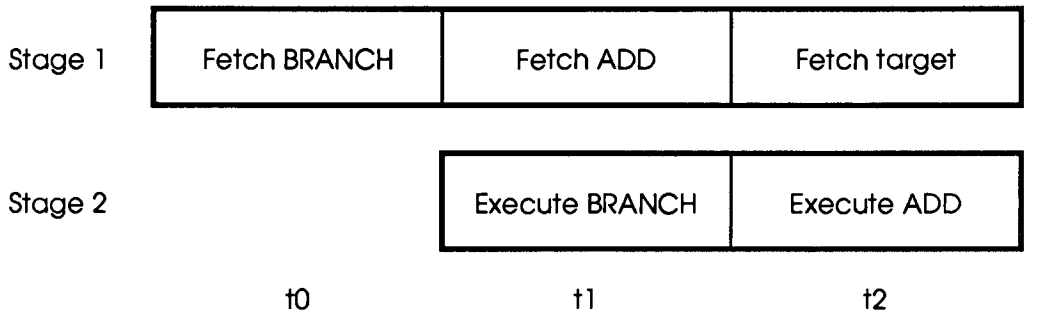


Figure 1–3: Delayed branch timing in a 2-stage pipeline

$i + b^{th}$ instruction is affected instead. This technique is known as *delayed branching*. If, for a k-stage pipeline $b > k$, then the target address of the branch will always be known in time to fetch the current instruction. Figure 1–3 shows the timing for such an instruction in a fetch/execute pipeline with $k = b = 2$. At time t_0, the branch instruction is fetched from memory. At t_1 it is executed, updating the value of the next instruction pointer. At the same time, the **add** instruction is fetched from memory. At t_2, the target instruction is fetched, but is not available for execution until t_3. If the **add** instruction is discarded, the time spent fetching it will have been wasted, and no completed instruction will appear from the pipeline during t_2, thus reducing the pipeline efficiency. Instead, the **add** instruction can be executed if the programmer or compiler can arrange that its execution would have no effect on the outcome of the branch. Thus, the branch instruction has been redefined to mean "execute the next instruction and then branch conditionally". A pipeline of the type shown in Figure 1–3 is employed successfully in reduced instruction set computers (RISCs) and microcoded architectures [Hennessy *et al.*, 1982] [Radin, 1983]. Delayed branching is most easily implemented for short pipelines where it is possible to delay execution for one instruction. Great reliance is placed on the use of compilers, as there are certain dangers inherent in the use of this type of instruction in human generated and maintained assem-

bly code. Conditional branches are implemented in the Intel i860 both with and without delayed branching, which allows for easier compiler optimization [Kohn and Margulis, 1989].

If $p_{no\text{-}op_i}$ is the probability that the i-th instruction following a branch will perform no useful work, then the effective performance of delayed branching is

$$F_{db} = \frac{1}{1 + (k-1)p_b p_n} \tag{1.14}$$

where

$$p_n = \sum_{i=1}^{k-1} p_{no\text{-}op_i} \Big/ (k-1)$$

Thus, performance is improved providing $p_n < p_t$. Radin [1983] reports a 60% utilization of a single delayed branch slot in the IBM 801. Filling subsequent slots becomes pregressively more difficult, with McFarling and Hennessy's [1986] figure of 70% for first slot utilization in MIPS dropping to less than 25% for second slot utilization.

Branch preparation is employed in PIPE [Goodman *et al.*, 1985] where a prepare-to-branch instruction which specifies a condition and the number of instructions to be executed irrespective of the outcome of the condition. The longer pipeline in PIPE, compared to that of the IBM 801, allows up to seven instructions to be executed after the branch. Analysis of several benchmark programs confirms this as a useful property, although use of all available slots cannot always be made by the compiler.

Certain comparisons, including equality, inequality, and comparison with zero, can be performed outwith the processor ALU and therefore incorporated into a compare-and-branch instruction [Katevenis, 1985]. These *fast* comparisons can be performed as soon as the register operands are available, thus reducing the delay to a single cycle, thereby allowing their use in delayed branches.

Delayed branches provide an improvement of between three and eight percent, dependent on the number of instructions executed between branches, compared

with architectures with non-delayed branches [DeRosa and Levy, 1987]. However, in cases where the proportion of taken branches increases to 75%, for example, as a result of more frequent subroutine calls, assumption that all branches will be taken accompanied by appropriate pre-fetching, provides the best performance.

1.2.2 Interinstruction Dependencies

The second major source of disruption in a pipeline is the existence of dependencies between partially executed instructions. These dependencies manifest themselves as one of four types of access conflicts for shared resources — **read-after-read**, **read-after-write**, **write-after-read** and **write-after-write** [Ramamoorthy and Li, 1977].

A **read-after-read** interaction in fact poses no problem as the correct value is accessed by both instructions. Two consecutive update operations could leave the memory location or register containing the wrong value if they are performed out of the intended order. The **read-after-write** conflict is characterized by an attempt to fetch data from a location which an earlier instruction is in the process of updating, and can arise during both register and memory accesses. This situation, and its somewhat rarer converse, the **write-after-read** conflict, require detection and prevention in hardware.

In the CDC 6600, the time required to complete an arithmetic operation differs between functional units, creating the possibility that instructions may not be executed in the correct order. This problem, and others caused by dependencies are resolved by the *scoreboard*, which buffers information about register availability with each of the FUs and unit usage with each register, and controls data transfer within the processor. The scoreboard handles three types of conflict. First order conflicts, as shown in Figure 1–4(a), are requests for units that are currently in use, or instances of **write-after-write** conflicts, and are resolved by delaying issue of the second operation. Second order conflicts, Figure 1–4(b), are cases of the **read-**

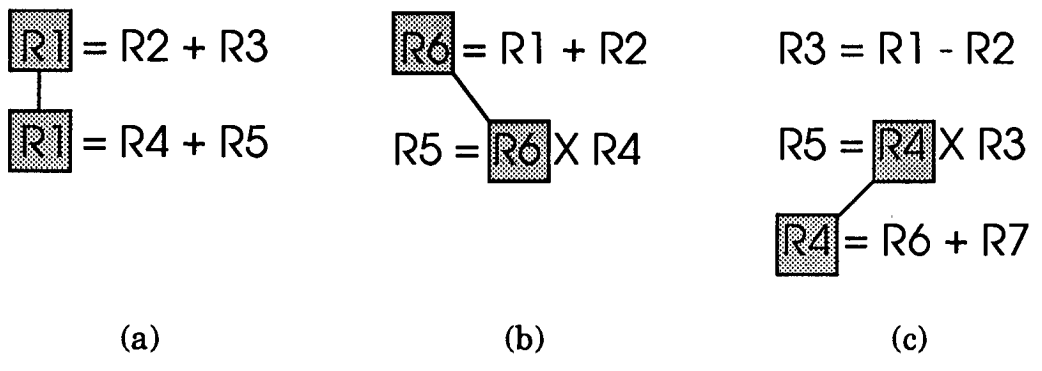


Figure 1–4: Instruction dependencies detected in the CDC 6600 Scoreboard

after-write problem. In these instances, the second instruction is issued and passes through early stages of the pipeline, so that its other operands can be fetched, but reading of the operand in contention is delayed until the earlier instruction is completed. Third order conflicts, Figure 1–4(c), are instances of the write-after-read conflict which arise as the result of an earlier second order conflict. Although the final instruction may complete before the second order conflict is resolved, it is prevented from storing its result until the previous read operation has completed.

A similar scheme is implemented in the Motorola 88000 series of RISC processors [Melear, 1989]. The processor architecture is based around two execution units, an integer unit executing single-cycle operations, and a FP unit containing separate addition and multiplication pipelines. These units are connected to a register file containing thirty-two 32-bit registers. Associated with the register file is a scoreboard register which prevents read-after-write conflicts from occurring. This register contains one bit corresponding to each member of the register file which is set on issue of a multiple-cycle instruction. If another instruction attempts to read from a register whose scoreboard bit is set, it is held in the instruction pipeline until the appropriate bit is cleared.

1.3 Micromultiprogramming

The problems which can reduce the efficiency of pipelines stem from a single common cause — delays in making data available for processing — whether as a result of the existence of a dependency between two instructions, or the memory latency when fetching the target instructions of a branch. Delays of a similar nature are to be found at other levels within a computer system, notably at the operating system level, when pages of virtual memory not held in the working set have to be loaded from backing store. Due to the mismatch in speed between semiconductor memory and the magnetic disks on which the swapped-out pages are kept, a large amount of processing time would be wasted if the processor remained idle while the page was fetched. Instead, the processor saves the state of the current task and begins execution of another, until the page request has been satisfied. This process of context switching is equally applicable at lower levels, where similar speed mismatches exist when high latency tasks are initiated.

Context switching below the instruction level is termed *micromultiprogramming*, and was first proposed by Chen [1971] as a means of improving performance of interleaved memory. Given an i-way interleaved store, randomly generated address requests result only in a $\sqrt{i}$ improvement in service time, rather than the approximate factor of i improvement shown with incremental requests. If the requests were handled by memory module availability, rather than arrival sequence, the improvement in performance for both random and incremental address requests would be close to i, albeit at the expense of a loss of ordering of data responses.

1.3.1 Sub-Instruction Level Context Switching

The use of *skeleton processors* as a means of exploiting the available resources of a multiprocessor architecture was suggested by Flynn and Podvin [1972]. Each skeleton processor appears to the programmer as a logically independent computer, but exists in hardware as a small register set supported by a minimum of control logic. The machine is organized as four rings, each consisting of eight skeleton processors, with a different machine resource being accessible at each point on the ring. Each processor is supplied from its own instruction stream, and is responsible for preparing its own instructions. Decoded instructions are then passed to the execution unit. As the rings "rotate", a given resource is accessed by a new processor. Contentions, if they arise, are dealt with on a priority basis. A performance of 500 MIPS was predicted, using the then current technology, but the machine was never built. In this machine, context switching at the sub-instruction level is implicit between the phases of the instruction execution process.

Kaminsky and Davidson [1979] propose the use of a multiple instruction stream pipeline as a means of increasing utilization of integrated circuit chip area in LSI uniprocessors. Execution is divided into fixed-length cycles, each consisting of a number of phases equal to the number of stages in the pipeline. During each phase, instructions from distinct streams are each at a different stage of processing, therefore no additional hardware is required to resolve dependencies.

Context switching is performed at the microinstruction level in the Xerox Alto [Thacker *et al.*, 1982]. The micromachine is shared by sixteen *tasks*, which perform instruction decoding, device control and general system maintenance operations. The address of the next microinstruction for each task is held in a register, allowing rapid switching between tasks in response to requests from device controllers. Instruction prefetch and execution are overlapped, and delayed branching is employed for all conditional control transfers. The task-switching system provides

a means to share the system resources between the consumers of these resources, and provides a greater integration of I/O devices with the central processor.

1.3.2 MIMD Pipelining

The Denelcor Heterogeneous Element Processor (HEP) [Jordan, 1985] is a shared memory pipelined multiprocessor which takes the notion of micromultiprogramming a stage further. At the outermost level, the HEP appears similar to many other MIMD machines — consisting of up to 16 process execution modules (PEMs) and 128 data memory modules (DMMs) connected by a message passing interconnection network. However, each PEM can also be considered a MIMD computer in its own right. Each PEM contains an eight stage instruction execution pipeline through which flow instructions *and* their operands. Independence of instructions is ensured by interleaving the instruction streams so that no two instructions from the same process exist in the pipeline concurrently. An implicit context switch thus takes place between each pipeline stage. Figure 1–5 contrasts instruction processing in a conventional pipelined machine (a), with MIMD pipelining in the HEP (b). Work in the HEP is disseminated among a maximum of sixteen *tasks*. Each task is described by a task status word (TSW) that identifies its associated protection domain in memory. Tasks are composed of up to sixty-four *processes* of which there are a maximum of 128 in each PEM. A process is characterized by a process status word (PSW), containing a program counter and other state information. Active processes are represented by a process tag (PT). During execution, PTs migrate from the scheduler, a hardware queue containing the PTs for each task, to the execution pipeline. Instructions which refer to external memory locations are queued before routing to the appropriate DMM via a pipelined network.

The HEP architecture, as shown in Figure 1–6, provides solutions for four major problems concerning multiprocessor designs. Hardware support for process

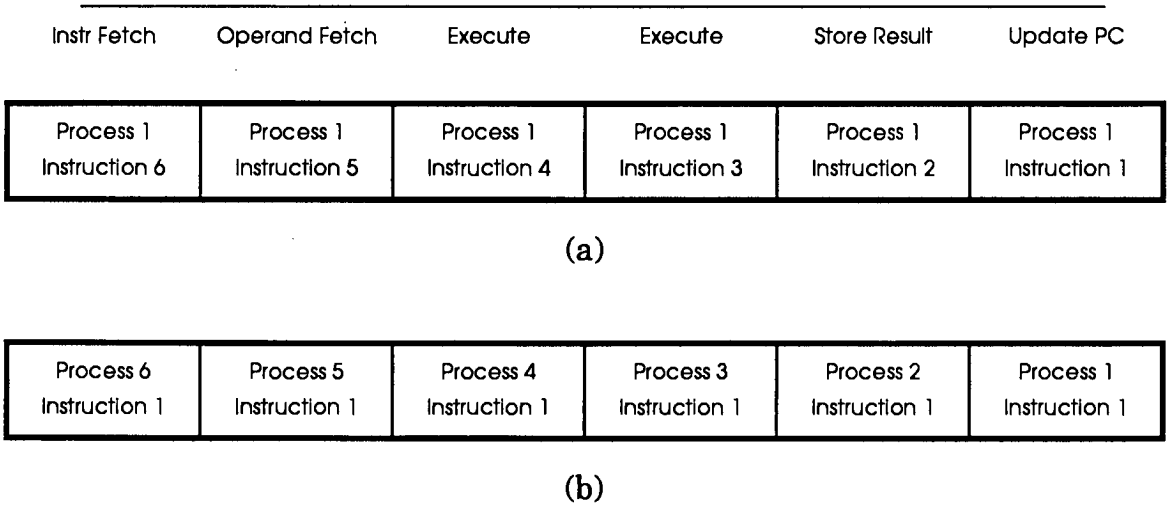


Figure 1–5: Conventional and MIMD Pipelining

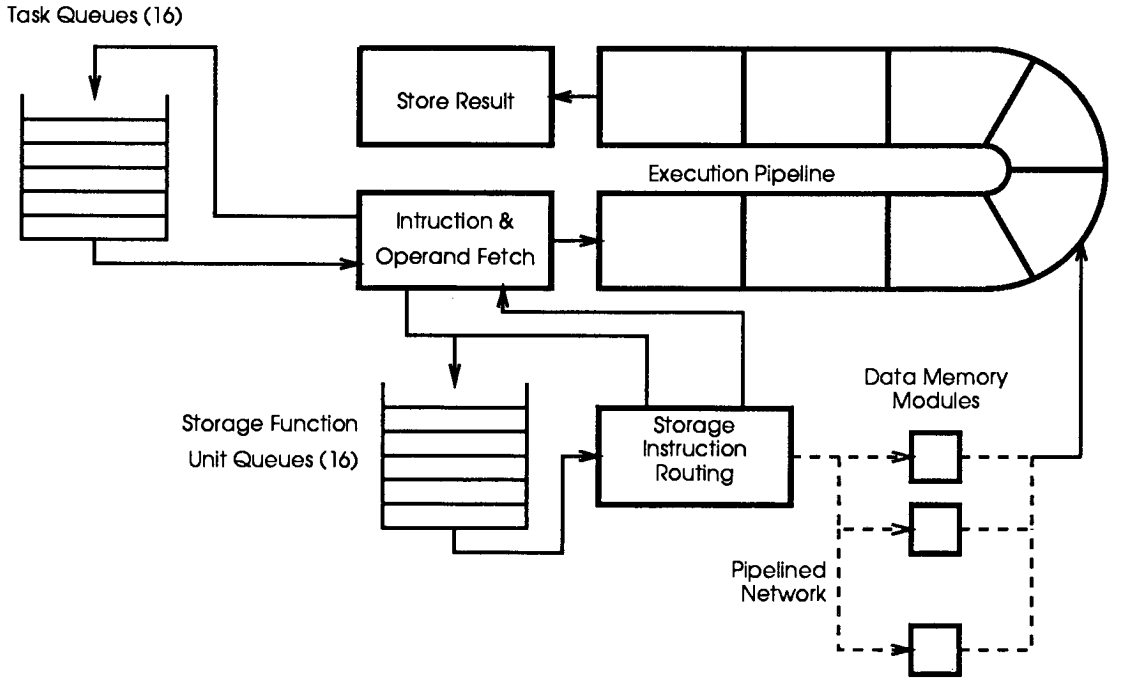


Figure 1–6: Architecture of a HEP PEM

creation is provided alongside management of process context queues, thus relieving the operating system of one of its major burdens. Efficient process synchronization is important if processes are to be partitioned with finer granularity, to extract the maximum parallelism, but still be executed effectively. The HEP provides a synchronization tag bit on each memory word that can be read by setting the appropriate control bits in instructions. This scheme allows implementation of algorithms requiring intensive synchronization and communication. The HEP's solution to the problem of memory latency, the isolation of non-immediately satisfiable memory requests in a separate queue, together with the interleaving of several instruction streams, allows the processor to execute instructions at the maximum possible rate, limited only by the parallelism of the problem. This last feature makes the architecture easily scalable, with a commensurate increase in processing capability.

1.3.3 Context Processing

The concept of interleaving instruction streams in a single processor is developed in the Circulating Context Multiprocessor (CCMP) [Butner and Staley, 1986]. The generic CCMP machine, illustrated in Figure 1–7, is a circular pipeline in which each stage implements part of the instruction processing cycle. Processes, represented by their state information contained in packets or contexts, are passed around the ring with the operations performed at each stage modifying the context as required. Queues can be inserted between stages to smooth packet flow and allow more concurrent active processes. To further increase the available parallelism, the queues can feed multiple instances of each functional unit.

The CCMP model has several attractive features. The mobility of process information, and the implication that processes are not tied to any one processor, permits easy implementation of process migration for load-balancing or reliability purposes. The replication of functional units within the processor allows packets

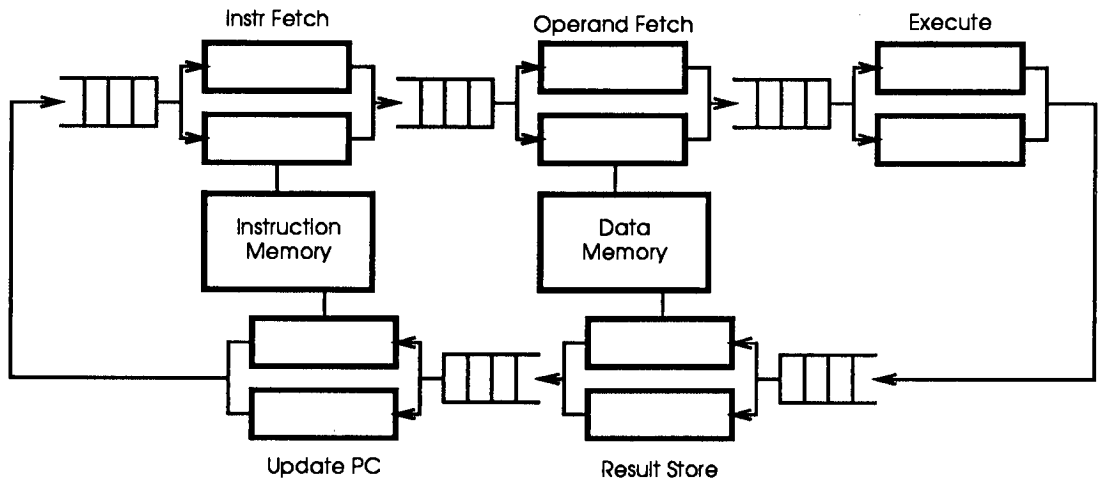


Figure 1–7: The Circulating Context Multiprocessor

to be absorbed from the queues at greater rate, and therefore directly improve processing ability.

Ianucci [1988] uses hardware contexts which can be switched in a single cycle. These *continuations* are therefore very small, and contain only the program counter and a pointer to data space. Nikhil and Arvind [1989] combine the use of continuations with a RISC-style execution pipeline. The continuations flow through the pipeline identifying instructions and *frames* — segments of memory usable as a register set by a given process — to be fetched from local memory. Access requests to main memory are diverted from the main pipeline to a heap controller, preventing the pipeline from stalling while they are being serviced. To allow more than one read request to be outstanding for a given process, a limitation which also exists in the HEP, *fork* and *join* primitive instructions are provided. As long as sufficient parallelism exists within a given application to generate enough continuations to fill the pipeline, processing is able to take place at the maximum possible rate.

Weber and Gupta [1989] propose a system of hardware contexts, each with a dedicated register set, purely as a means of circumventing memory latency. Context switches are initiated on cache misses or writes to shared locations. Using four contexts, improvements of up to 80% have been measured when the architecture exhibited large memory latency, the context switch overhead was low, and the cache interference was minimal.

None of the architectures described above have taken micromultiprogramming to its logical conclusion — the direct manipulation of process context, rather than data, by pipeline stages. The benefits of interleaving multiple instruction streams are clear. If two instructions from the same process can be guaranteed not to exist in a state of partial execution concurrently, then there can be no dependency or branching problems. Dependencies between instructions still exist within processes, but do not affect pipeline operation as there is only a single instruction active at any given time. Conditional branching still occurs, but presents no problem as the condition is evaluated, and the target address known, before the next instruction is fetched. Removal of these two obstacles allows a pipeline to produce results at the maximum possible frequency. It must, however, be noted that although one instruction is completed every cycle in an MIMD pipeline, each process requires as much time to execute as if there were no pipelining.

The requirement to switch contexts between pipeline stages necessitates a minimal context associated with each process. This requirement is not met by the CCMP model, in which the reliance on a von Neumann processing paradigm yields large process contexts that have proven unimplementable [Staley and Butner, 1986]. Instead, the bulk of the process status information has a fixed location, and only the program counter, the current data object, and a process identification tag are circulated. While this may seem only a slight deviation from the idealized model, the process' relocatability is entirely lost, nullifying a significant feature of the original model. Implementations of machines which employ sub-instruction

level context switching have confirmed the validity of this approach as a means of achieving high performance from a pipelined architecture, although each appears to have used the technique to circumvent a different problem. Kaminsky and Davidson [1979] proposed the multiplexing of processor resources between a fixed number of instruction streams as a means to improve the utilization of chip area and minimize the number of off-chip connections. CCMP was developed as a means to provide a fault tolerant machine based around "trusted" first-in-first-out (FIFO) queues [Butner, 1984].

Summary

Parallelism exists in two exploitable forms within any computation which may be realized providing sufficient machine resources exist. The spatial parallelism which exists in program data may be extracted by replication of execution units, either within a processor as multiple functional units or in the form of a multiple processor ensemble. The temporal parallelism which exists during the execution of an instruction may be exploited by pipelining the stages which comprise the execution sequence.

Replication and pipelining both have their own associated problems. The problems with replication tend to be of a more algorithmic nature, for example, loss of generality of application, synchronization and inter-process communication, and placement of code and data. Although pipelining maintains generality of purpose, it is at the expense of an increase in hardware complexity required to handle dependencies which arise between pipeline stages.

While many solutions which attempt to ameliorate the problems caused by instruction and data dependencies have been proposed and adopted, none fully addresses the source of the problem by removing dependencies from the pipeline. Al-

though dependencies will always exist in program data, their effects could be eliminated if processing were performed on mutually independent instruction streams.

The incoherent approach to previous uses of sub-instruction level context switching suggests the need for an architectural theory which handles all sources of discontinuities in a unified manner. This need is addressed by a design technique called *context flow.*

"la dernière chose qu'on trouve en faisant un ouvrage,
est de savior celle qu'il faut mettre la première."
— BLAISE PASCAL (1623–1662), *Penseés i. 19*

Chapter 2

A Theory of Context Flow

Several problems with instruction pipelines which degrade their performance below the anticipated maximum have been identified. The problems arise as a result of interinstruction dependencies and are exacerbated by memory latency. A set of key features has been outlined which smooth the disruptions found in conventional pipelines, resulting in improved performance in pipelined multiprocessor systems, namely interleaving of multiple instruction streams with only one active instruction per stream, removing interinstruction dependencies; efficient process creation and management, including provision in hardware for process queuing; pipelining of memory access requests, resulting in the hiding of memory latency; and relocatability of processes between sites of execution, providing inherent support for load balancing and fault tolerance. *Context Flow* (CF) [Topham *et al.*, 1988] is an architectural paradigm in which process contexts rather than process data are manipulated. This chapter presents the central concepts of context flow.

2.1 The Principles of Context Flow

The behaviour of a context flow system is encapsulated in a structure called a *context flow graph* (CFG). A CFG is a directed graph which describes the operation of the CF system in terms of the transformations applied to contexts which pass through the graph. A CFG is not a program, rather an abstraction of the hardware on which programs may be executed.

2.1.1 Contexts

The notion of a context is well established in the field of operating systems. The *volatile context* of a process is the subset of shared system resources which are accessible to the process. A description of the volatile context is contained in a structure called a *process context block* (PCB). A typical PCB may contain a unique process identification number, the contents of the processor register set etc. It is natural, to use the term *context* to describe the process state information in a system which is an implementation of multiprogramming at the instruction level. The requirement for relocatability imposes two main constraints on the nature of a context. A context should be entirely self-contained, encapsulating all information pertinent to its related process and should be small enough to be physically transmittable between the sites of different processing phases. A context is defined as follows:

Definition 2.1 A context c at time t, $t \in \mathbb{N}$, is a tuple $\langle f, D, D' \rangle_t$ where D, D' are sets and $D \xrightarrow{f} D'$. The function f can be an identity id_D, where $\forall d \in D : d \xmapsto{f} d$, and can be abbreviated to the context $\langle D \rangle$. The set of all contexts is written C.

Remark 2.2 Time is represented as integer multiples of a machine cycle. Therefore for a given instant t, $t \in \mathbb{N}$.

Sometimes D may be very large, in which case it is useful to differentiate between the frequently used and less frequently used data. The frequently used data constitutes the data set of a *dynamic context.* The infrequently used data is stored at a fixed location as the data set of a *static context.*

Definition 2.3 Let $c = \langle f, D, D' \rangle$ be a context. Its static context $\sigma(c)$ is a pair $\langle i, D_s \rangle$ where i is an integer label denoting the location where D_s is stored. Its dynamic context $\delta(c)$ is the pair $\langle f, D_d \rangle$ where $D_d \cup D_s = D$, such that $D_d \cap D_s = \varnothing$.

In many cases, $D_s = \varnothing$ and all process data is held in D_d. The term "context" is used to refer to the dynamic context of a process, unless otherwise stated. The empty or *null* context, which contains no function or data, is defined as follows:

Definition 2.4 A null context ν is a tuple $\langle f, D, D' \rangle$ where $f = D = D' = \varnothing$.

The above definitions place no restrictions on the format of a context, nor do they require that a context be of constant size throughout an application. This differs significantly from the CCMP model in which the same packet of process information circulates round the processing ring. The dynamic nature of the CF contexts significantly eases implementation.

2.1.2 Nodes

The sites at which function application occurs, and where static contexts can be stored, are called *nodes.* CFGs are constructed from such nodes.

A *transformation* (τ) node is used to perform a manipulation of a single context. A τ-node has a single incoming arc on which contexts are accepted, and a single outgoing arc along which processed contexts are delivered. All manipulations performed at a given τ-node take place within one unit of time, also called a *graph cycle period.* The operation performed within a τ-node may be dedicated, or

Figure 2–1: Transformation Node Operation

selectable from a group of functions by a field within the context. For example, a τ-node could implement a simple ALU, with the context containing the operands and the function selector. Connected in sequence, a series of τ-nodes implement a simple pipeline. A τ-node may incorporate locally addressable storage, in the form of a queue, stack, random access memory, or a register. Storage of this type is private to the node to which it is attached. Control signals and data for memory are provided from and returned in contexts.

Definition 2.5 A transformation node can perform the following actions

$$\tau : \nu \rightsquigarrow \nu$$
$$\tau : \langle \varnothing, D, D' \rangle_t \rightsquigarrow \langle D' \rangle_{t+1} \ , \ D \xrightarrow{f_\tau} D'$$
$$\tau : \langle f, D, D' \rangle_t \rightsquigarrow \langle D' \rangle_{t+1}$$
$$\tau : \langle f, D_d \cup D_s, D' \rangle_t \rightsquigarrow \langle D' \rangle_{t+1}$$

where $\rightsquigarrow$ denotes the transformation during one graph cycle, and f_τ represents a dedicated function performed by the node.

Figure 2–1 shows the operation of a transformation node in diagrammatic form. In these diagrams, a null context is denoted by o and a non-null context by •. Diagrams of this form only indicate the flow of contexts through the graph, providing no information about the values contained in the contexts. A *branch* (β) node provides a means of introducing spatial parallelism in the form of multiple

context streams. A β-node has a single input arc and two output arcs. A β-node examines one or more fields in the incoming context and routes the context to one of the two output arcs, dependent on the outcome of its internal decision process. A null context is output simultaneously on the other arc. A β-node contains no local memory, therefore routing can only be performed based on the contents of the current context, and not on any previous state information.

Definition 2.6 A branch node performs the following actions

$$\begin{aligned}
&\beta : \nu \rightsquigarrow (\nu, \nu) \\
&\beta : \langle id_D, D, D \rangle_t \rightsquigarrow (\langle D \rangle_{t+1}, \nu) \quad \textit{if} \quad f_\beta D \\
&\beta : \langle id_D, D, D \rangle_t \rightsquigarrow (\nu, \langle D \rangle_{t+1}) \quad \textit{if} \quad \neg f_\beta D
\end{aligned}$$

where f_β is the decision function implemented within the node.

The operation of a branch node is shown in Figure 2–2.

The introduction of parallelism by means of the branch node, requires the definition of a complementary node to combine context streams. This function is provided by the *merge* (μ) node, as shown in Figure 2–3, which has two input arcs and a single output arc. The merge node accepts a pair of contexts on its inputs and delivers one context on its output each graph cycle period.

Definition 2.7 A merge node performs the action

$$\mu : (\langle f_1, D_1, D_1' \rangle_t, \langle f_2, D_2, D_2' \rangle_t) \rightsquigarrow \langle f_3, D_3, D_3' \rangle_{t+1}$$

As in the branch node, no transformation is applied to a context within a merge node.

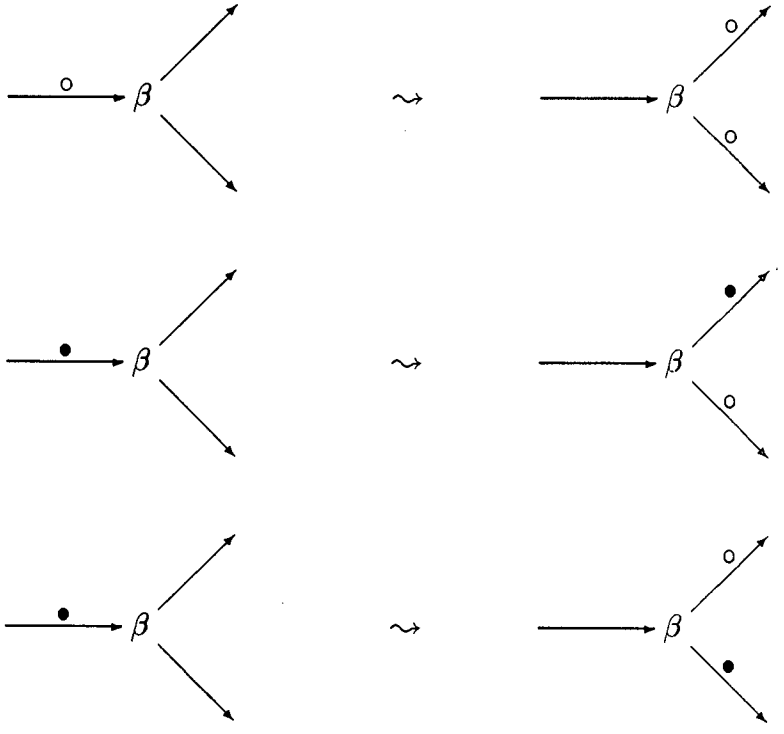


Figure 2–2: Branch Node Operation

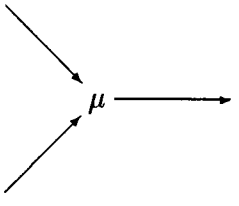


Figure 2–3: Merge Node Representation

2.1.3 Graphs

A context flow graph is a representation of the evaluation algorithm of the abstract machine being implemented. The vertices of the graph are the nodes which perform the transformations, and the edges encode their temporal ordering.

Four classes of graph can be defined:

- Class I The class of acyclic graphs which contain no branch or merge nodes, equivalent to static, uni-functional arithmetic pipelines.

- Class II The class of acyclic graphs which contain equal numbers of branch and merge nodes, connected in such a way that the paths between a branch and its corresponding merge node contain only matched pairs of branch and merge nodes. Graphs in this class corresponds to dynamic multi-functional pipelines.

- Class III The class of acyclic graphs which contain any numbers of branch and merge nodes in any combination. An example of a Class III graph is an instruction pipeline with a connection to memory.

- Class IV The class of cyclic graphs, equivalent to complete pipelined systems with pipelined access to memory.

A context flow graph is *empty* if every node contains a null context. A context flow graph is *open* if it contains an edge for which the source or destination is undefined, otherwise it is *closed.* Acyclic graphs are open, but cyclic graphs need not be closed. The parallelism identified in a CFG is similar to the intra-algorithmic parallelism revealed when data dependencies are drawn as a data flow graph [Davis and Keller, 1982].

Notation	Meaning
$\langle\!\langle\, \rangle\!\rangle$	the empty stream
$\langle\!\langle c \rangle\!\rangle$	a stream with a single context
$\langle\!\langle c_1, c_2, \ldots, c_n \rangle\!\rangle$	a stream with n contexts
(s_1, s_2)	a pair of parallel streams
$c \subseteq s_1$	context c is contained in stream s_1
$s_1 \approx s_2$	stream s_1 is equivalent to stream s_2
s^n	stream s repeated n times
$s_1 \wedge s_2$	concatenation of streams s_1 and s_2
$\#s$	the number of contexts in stream s
$s_\circ$	head of stream s
s'	tail of stream s

Table 2–1: Streams and stream operations

2.2 Context Streams

The state of a computation in a context flow graph is represented by the *stream* of contexts which pass a given point. Borrowing notation from CSP [Hoare, 1985], Table 2–1 defines the set S of streams, and several basic stream operations. Before any of these operations can be defined, a notion of stream equivalence must be established. As the contexts in a stream represent independent processes, their order is unimportant. Their relative position in the stream determines only the relative speed with which they are operated on, not the outcome of the computation.

Definition 2.8 Two streams s_1 and s_2 are equivalent if and only if each context in s_1 is contained in s_2, and each context in s_2 is contained in s_1.

$$s_1 \approx s_2 \quad \textit{if} \quad \forall c \subseteq s_1, c \subseteq s_2 \wedge \forall c \subseteq s_2, c \subseteq s_1$$

Lemma 2.9 $\approx$ *is an equivalence relation.*

Proof

- $\approx$ *is reflexive. Clearly,* $\forall c \subseteq s_1, c \subseteq s_1$, *therefore* $s_1 \approx s_1$.
- $\approx$ *is symmetric. Suppose* $s_1 \approx s_2$ *then by definition* $\forall c \subseteq s_1, c \subseteq s_2 \wedge \forall c \subseteq s_2, c \subseteq s_1$ *therefore* $\forall c \subseteq s_2, c \subseteq s_1 \wedge \forall c \subseteq s_1, c \subseteq s_2$ *therefore* $s_2 \approx s_1$.
- $\approx$ *is transitive. Suppose* $s \subseteq s_1$, *then* $s_1 \approx s_2 \Rightarrow s \subseteq s_2$ *and* $s_2 \approx s_3 \Rightarrow s \subseteq s_3$. *Conversely, suppose* $s \subseteq s_3$, *then* $s_2 \approx s_3 \Rightarrow s \subseteq s_2$ *and* $s_1 \approx s_2 \Rightarrow s \subseteq s_1$. *Thus* $s_1 \approx s_3$.

This completes the proof. □

The empty stream $\langle\!\langle\,\rangle\!\rangle$ is the identity element for all stream operations, and is simply a stream which contains no contexts. The singleton stream contains only one non-null context, and is therefore equivalent to the context itself. The stream $\langle\!\langle c \rangle\!\rangle$ may be written as c. A pair of streams (s_1, s_2) denotes a grouping of two independent streams of contexts.

2.2.1 Stream Operators

The concatenation operator $\wedge$ is used to join two or more streams to form a single stream, and defines a function:

$$S \times S \xrightarrow{\wedge} S$$

The following axioms show that concatenation is associative and commutative, and distributive when applied to pairs of streams.

Axiom 2.10 $s_1 \wedge (s_2 \wedge s_3) \approx (s_1 \wedge s_2) \wedge s_3$

Axiom 2.11 $s_1 \wedge s_2 \approx s_2 \wedge s_1$

Axiom 2.12 $s \wedge \langle\!\langle\,\rangle\!\rangle \approx \langle\!\langle\,\rangle\!\rangle \wedge s \approx s$

Axiom 2.13 $s_1 \wedge s_2 \approx s_1 \wedge s_3 \Rightarrow s_2 \approx s_3$

Axiom 2.14 $s_1 \wedge s_2 \approx \langle\!\langle\,\rangle\!\rangle \Rightarrow s_1 \approx s_2 \approx \langle\!\langle\,\rangle\!\rangle$

Axiom 2.15 $(s_1, s_2) \wedge s_3 \approx (s_1 \wedge s_3, s_2 \wedge s_3)$

Axiom 2.16 $(s_1, s_2) \wedge (t_1, t_2) \approx (s_1 \wedge t_1, s_2 \wedge t_2)$

The replication operator performs an n-fold concatenation of a stream with itself. $\underbrace{s \wedge s \cdots s \wedge s}_{n}$ is written as s^n. The length function $\#$ is defined as the number of contexts which a stream contains, according to the following rules:

Axiom 2.17 $\#\langle\!\langle\,\rangle\!\rangle = 0$

Axiom 2.18 $\#\langle\!\langle c \rangle\!\rangle = 1$

Axiom 2.19 $\#(s_1 \wedge s_2) = \#s_1 + \#s_2$

$\#\langle\!\langle c \rangle\!\rangle$ may also be written as $\#c$. For pairs of streams, the maximum $\#^+$ and minimum $\#^-$ lengths of streams are defined.

Definition 2.20

$$\#^+(s,t) = \max(\#^+s, \#^+t)$$

where

$$\#^+s = \#s \quad \text{and} \quad \#^+(s \wedge t) = \#^+s + \#^+t$$

Definition 2.21

$$\#^{-}(s,t) = \min(\#^{-}s, \#^{-}t)$$

where

$$\#^{-}s = \#s \quad \text{and} \quad \#^{-}(s \wedge t) = \#^{-}s + \#^{-}t$$

The *stream difference* is defined as the difference in lengths of the streams in a pair.

Definition 2.22 $\#_d(s,t) = \#^{+}(s,t) - \#^{-}(s,t)$, where $\#_d s = 0$.

The notion of membership can be extended to streams as follows:

Definition 2.23 $s_1 \subseteq s_2$ *if* $\exists t, u : t \wedge s_1 \wedge u \approx s_2$

Remark 2.24 $c \subseteq s$ *if* $\langle\!\langle c \rangle\!\rangle \subseteq s$

The head operator $_\circ$ is used to isolate the first context in a stream from the remainder or *tail.* The head operator, and tail operator $'$ define functions

$$S \xrightarrow{\circ} C \qquad S \xrightarrow{'} S$$

Definition 2.25 $\langle\!\langle c_1, \ldots, c_n \rangle\!\rangle_\circ \approx \langle\!\langle c_1 \rangle\!\rangle \qquad \langle\!\langle c_1, \ldots, c_n \rangle\!\rangle' \approx \langle\!\langle c_2, \ldots, c_n \rangle\!\rangle$

The head of the empty stream is defined to be a null context, and the tail of the empty stream to be the empty stream itself.

Definition 2.26 $\langle\!\langle \rangle\!\rangle_\circ \approx \nu \qquad \langle\!\langle \rangle\!\rangle' \approx \langle\!\langle \rangle\!\rangle$

Remark 2.27 Due to the possible variable order of contexts in equivalent streams, $_\circ$ does not extend to a function on the equivalence class of streams.

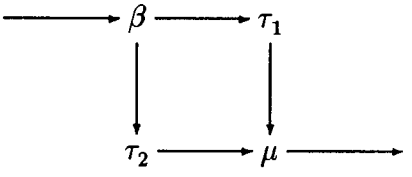
2.2.2 Graphs as Functions on Streams

Having defined a stream as the representation of the state of a computation, it is only natural to view a context flow graph as a function which operates on an input stream and returns an output stream[1].

Definition 2.28 A graph g defines a function on S

$$S \xrightarrow{g} S$$

As a graph simply represents the ordering of transformations applied to a context stream, it can be written as the composition of the functions defined by the nodes in the graph. For example, the graph



can be written as

$$g = \beta \circ (\tau_1, \tau_2) \circ \mu$$

Application of a graph to a stream is governed by the following rules, where $\phi \in \{\beta, \tau, \mu\}$, s represents a stream and c, a context.

Axiom 2.29 $\quad \{s\}\phi \equiv \langle\langle s_\circ \phi \rangle\rangle \wedge s'$

Axiom 2.30

$$\{s\}\tau \rightsquigarrow \begin{cases} \langle\langle\,\rangle\rangle & \text{if } s \approx \langle\langle\,\rangle\rangle \\ \langle\langle \nu, \{s'\}\tau \rangle\rangle & \text{if } s_\circ = \nu, s' \not\approx \langle\langle\,\rangle\rangle \\ \langle\langle s_\circ \tau \rangle\rangle & \text{if } s_\circ \neq \nu, s' \approx \langle\langle\,\rangle\rangle \\ \langle\langle s_\circ \tau, \{s'\}\tau \rangle\rangle & \text{if } s_\circ \neq \nu, s' \not\approx \langle\langle\,\rangle\rangle \end{cases}$$

---

[1] $\{a\}f$ is used to denote the application of function f to argument a, and af to denote the result of applying f to a

Axiom 2.31

$$\{s\}\beta \rightsquigarrow \begin{cases} (\langle\!\langle\,\rangle\!\rangle, \langle\!\langle\,\rangle\!\rangle) & \text{if } s \approx \langle\!\langle\,\rangle\!\rangle \\ (\langle\!\langle\,\rangle\!\rangle, \langle\!\langle\,\rangle\!\rangle) \wedge \{s'\}\beta & \text{if } s_\circ = \nu, s' \not\approx \langle\!\langle\,\rangle\!\rangle \\ (s_\circ, \nu) \wedge \{s'\}\beta & \text{if } s_\circ f_\beta, s' \not\approx \langle\!\langle\,\rangle\!\rangle \\ (\nu, s_\circ) \wedge \{s'\}\beta & \text{if } \neg s_\circ f_\beta, s' \not\approx \langle\!\langle\,\rangle\!\rangle \end{cases}$$

Axiom 2.32 $\{(s_1, s_2)\}\mu \rightsquigarrow s_3$

Axiom 2.33 $\{\langle\!\langle c_1, \ldots, c_n \rangle\!\rangle\}(\phi_1 \circ \phi_2) \rightsquigarrow^{n+1} \langle\!\langle c_1\phi_1\phi_2, \ldots, c_n\phi_1\phi_2 \rangle\!\rangle$

2.2.3 Combining Context Streams

A merge node, as introduced in Definition 2.7 above, accepts two contexts on its input and produces one one its output during each graph cycle period. While the branch node always generates one null context per cycle, the merge node does not necessarily perform the converse operation.

Proposition 2.34 *A merge node must be able to accept two valid contexts during a single graph cycle period.*

Proof *Given a graph*

$$g = \beta \circ ((\tau_2 \circ \tau_3), \tau_1) \circ \mu$$

and stream

$$s = \langle\!\langle \langle D_1 \rangle, \langle D_2 \rangle \rangle\!\rangle : D_1 f_\beta, \neg D_2 f_\beta$$

$$\begin{aligned} \{s\}g =& \{\langle D_2 \rangle\}\,(\{\langle D_1 \rangle\}\beta \circ ((\tau_2 \circ \tau_3), \tau_1) \circ \mu) \\ \rightsquigarrow& \{\langle D_2 \rangle\}\beta \circ ((\{\langle D_1 \rangle\}\tau_2 \circ \tau_3), \tau_1) \circ \mu \\ \rightsquigarrow& ((\{\nu\}\tau_2 \circ \{\langle D_1\tau_2 \rangle\}\tau_3), \{\langle D_2 \rangle\}\tau_1) \circ \mu \\ \rightsquigarrow& \{(\langle D_1\tau_2\tau_3 \rangle, \langle D_2\tau_1 \rangle)\}\,\mu \end{aligned}$$

This completes the proof. □

Corollary 2.35 *A merge node must be able to store incoming contexts if they cannot be output during the current graph cycle period.*

Proof *From Proposition 2.34 and Axiom 2.32, the expected outcome is*

$$\langle\!\langle \langle D_1\tau_2\tau_3\rangle, \langle D_2\tau_1\rangle \rangle\!\rangle$$

If μ has no ability to store incoming contexts, then the result can only be $\langle D_1\tau_2\tau_3\rangle$ or $\langle D_2\tau_1\rangle$. □

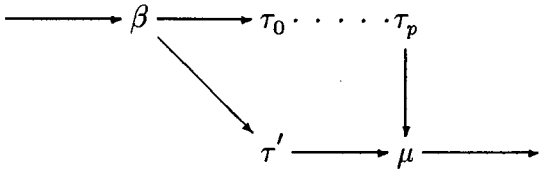
2.2.4 Characteristics of a Merge Node Buffer

Having proved that a merge node must buffer incoming contexts, the precise nature of the local storage mechanism must be determined.

Definition 2.36 μ^n is used to denote a merge node containing n buffered contexts, and $\mu^{(c_1,\ldots,c_n)}$, a merge node containing the buffered contexts $c_1, \ldots, c_n$.

Proposition 2.37 *A merge node may have to store more than one context.*

Proof *By induction on the difference in path length p between branch node β and merge node μ. Consider the class of graphs which contain one matching pair of branch and merge nodes:*



$$g_p = \beta \circ ((\tau_0 \circ \cdots \circ \tau_p), (\tau')) \circ \mu$$

and stream

$$s_p = \langle\!\langle c_0, \ldots, c_{p-1}, c_p, \ldots, c_{2p-1} \rangle\!\rangle$$

where

$$c_i : \begin{cases} 0 \leqslant i \leqslant p-1 & c_i f_\beta \\ p \leqslant i \leqslant 2p-1 & \neg c_i f_\beta \end{cases}$$

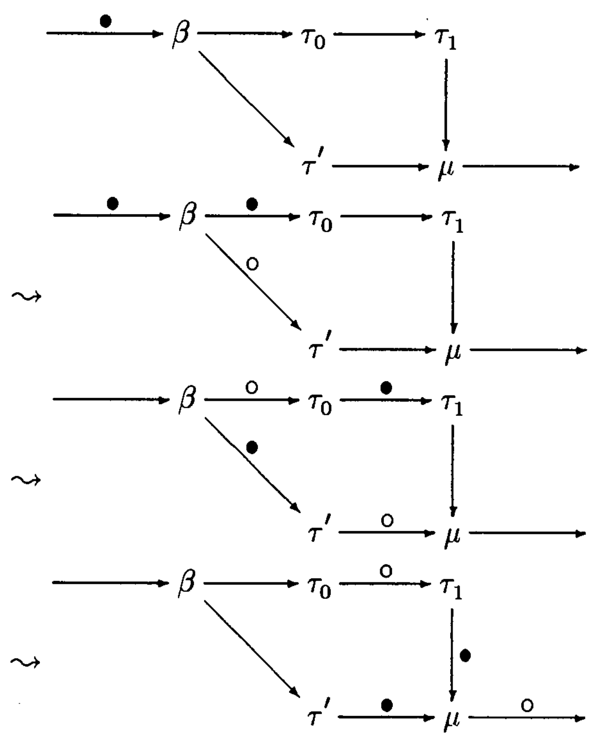
Here, and in all subsequent proofs, the graph is initially empty.

- *Let* $p = 0$.

$$
\begin{aligned}
\{c_0\}g_0 =& \{c_0\}\beta \circ (\tau_0, \tau') \circ \mu \\
\rightsquigarrow & \begin{cases} (\{c_0\}\tau_0, \{\nu\}\tau') \circ \mu & \textit{if } c_0 f_\beta \\ (\{\nu\}\tau_0, \{c_0\}\tau') \circ \mu & \textit{if } \neg c_0 f_\beta \end{cases} \\
\rightsquigarrow & \begin{cases} (c_0\tau_0, \nu)\mu \\ (\nu, c_0\tau_0)\mu \end{cases}
\end{aligned}
$$

as one of the inputs is always ν*, no context needs to be stored in* μ*.*

- *Let* $p = 1$.



From Corollary 2.35, μ *must store one context during the next graph cycle.*

- *Assume that the proposition is true for* $p = n$*, that is given graph* g_n *and stream* s_n *where*

$$g_n = \beta \circ ((\tau_0 \circ \cdots \circ \tau_n), (\tau')) \circ \mu$$

$$s_n = \langle\langle c_0, \ldots, c_{n-1}, c_n, \ldots, c_{2n-1} \rangle\rangle$$

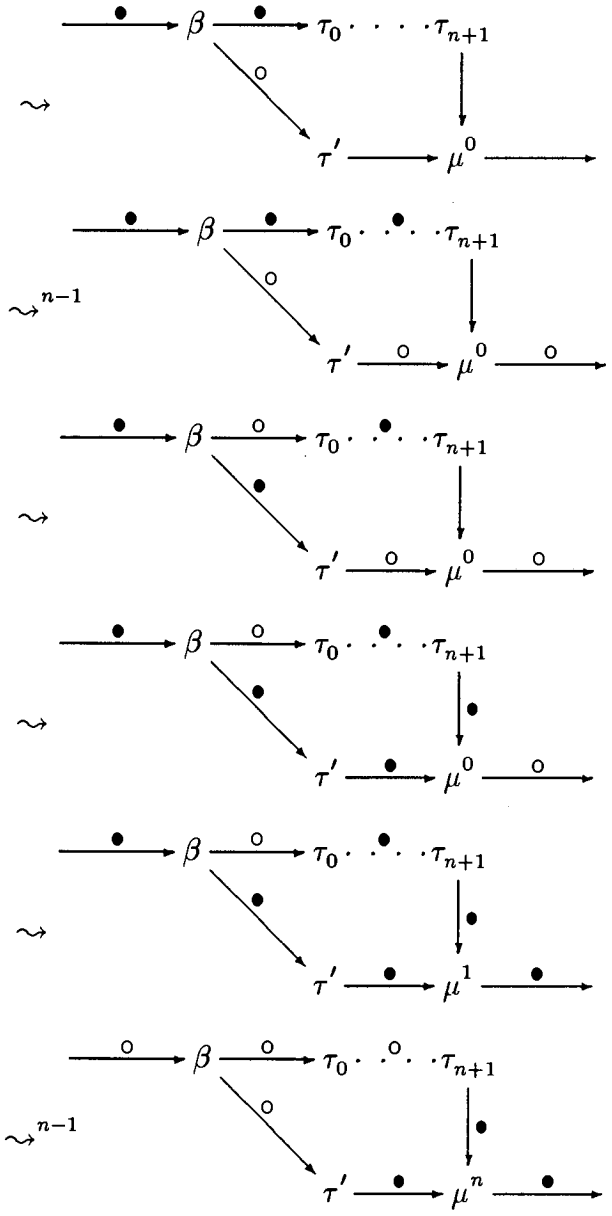
then

$$\{s_n\}g_n \rightsquigarrow^{2n} \left\{(\{c_{n-1}\tau_0 \cdots \tau_{n-1}\}\tau_n, \{c_{2n-1}\}\tau')\right\} \mu^{n-1}$$

- *Let* $p = n + 1$.

$$g_{n+1} = \beta \circ ((\tau_0 \circ \cdots \circ \tau_{n+1}), (\tau')) \circ \mu$$

$$s_{n+1} = \langle\langle c_0, \ldots, c_n, c_{n+1}, \ldots, c_{2n+1} \rangle\rangle$$



Therefore, in cases where a difference in path length exists, the merge node is required to store more than one context. □

Proposition 2.37 implies that it is not possible to use a register as the internal storage mechanism of a merge node, as a register would only be able to hold one input context. A register set, or some type of random access memory might be a possible mechanism, except that the location at which the context is to be stored would require specification in the context. This is undesirable, as the process of buffering contexts should be automatic and not under the control of the application being evaluated by the graph. This restricts the choice to a sequential access memory — either a stack or a FIFO queue. Although a stack would satisfy the ordering property of Definition 2.8, the first context to be buffered in the node would be the last to be forwarded, which although not incorrect, would impose an unnecessary and execution-specific delay on the particular process. Use of a FIFO queue maintains ordering and imposes the minimum delay on contexts. With the above restriction in mind, the operation of a merge node is defined as follows:

Axiom 2.38

$$\{(s,t)\}\mu^0 \rightsquigarrow \begin{cases} \langle\langle \nu, \{(s',t')\}\mu^0 \rangle\rangle & \text{if } s_0 = t_0 = \nu \\ \langle\langle s_0, \{(s',t')\}\mu^0 \rangle\rangle & \text{if } s_0 \neq \nu, t_0 = \nu \\ \langle\langle t_0, \{(s',t')\}\mu^0 \rangle\rangle & \text{if } s_0 = \nu, t_0 \neq \nu \\ \langle\langle s_0, \{(s',t')\}\mu^{(t_0)} \rangle\rangle & \text{if } s_0 = t_0 \neq \nu \end{cases}$$

$$\{(s,t)\}\mu^{(c_0,\ldots,c_n)} \rightsquigarrow \begin{cases} \langle\langle c_0, \{(s',t')\}\mu^{(c_1,\ldots,c_n)} \rangle\rangle & \text{if } s_0 = t_0 = \nu \\ \langle\langle c_0, \{(s',t')\}\mu^{(c_1,\ldots,c_n,s_0)} \rangle\rangle & \text{if } s_0 \neq \nu, t_0 = \nu \\ \langle\langle c_0, \{(s',t')\}\mu^{(c_1,\ldots,c_n,t_0)} \rangle\rangle & \text{if } s_0 = \nu, t_0 \neq \nu \\ \langle\langle c_0, \{(s',t')\}\mu^{(c_1,\ldots,c_n,s_0,t_0)} \rangle\rangle & \text{if } s_0 = t_0 \neq \nu \end{cases}$$

Figure 2–4 illustrates the operation of a merge node.

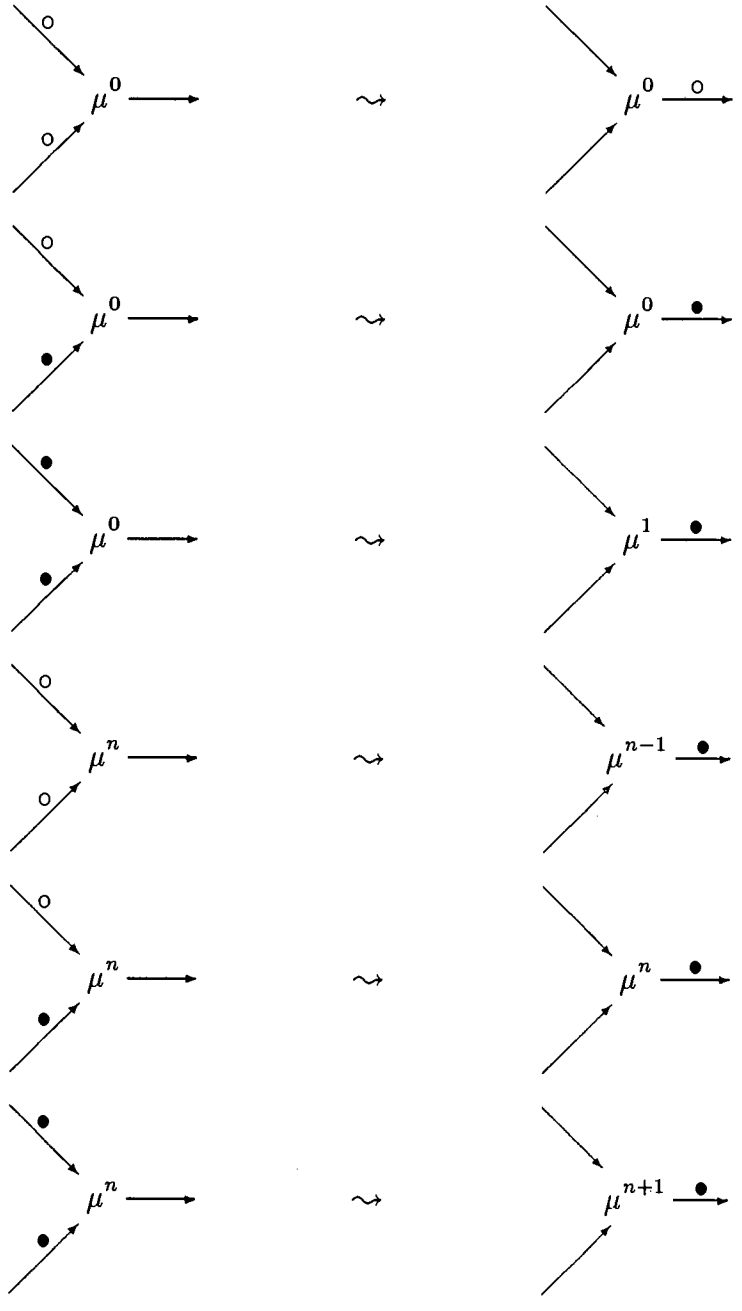
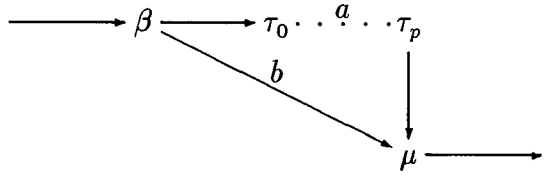


Figure 2–4: Merge Node Operation

2.3 Boundedness of Queues in Acyclic Graphs

A FIFO queue is potentially capable of storing an infinite number of elements should a mismatch in input and output flow rates be sustained for a considerable period of time. In any implementation of a merge node, however, the queue must be implemented within a strictly finite memory. Before any CF machine can be built, the maximum length of all queues must be determined. In order to do this, a means to induce a maximal length queue in a merge node must be established. Consider the following graph



which is initially empty. Suppose a stream of contexts passes through β and are all routed via a so that each node $\tau_0, \ldots, \tau_p$ contains a non-null context. During this time, null contexts pass along b to μ. If the next context is routed via a then μ will merge a null context from b and non-null context from τ_p and no contexts are queued. If, however, the next context is routed via b, μ will merge two non-null contexts, queueing one. If non-null contexts continue to to be routed along b, the queue at μ will continue to grow until all the contexts in $\tau_0, \ldots, \tau_p$ have been merged. At this point, the queue cannot grow as at most one non-null context can arrive at μ. This is stated formally in the following Lemma.

Lemma 2.39 *The input stream* $s = \langle\langle c_0, \ldots, c_{p-1}, c_p, \ldots, c_{2p-1}\rangle\rangle$ *operated on by graph* g *of Proposition 2.37 induces a maximal length queue in* μ.

Proof *If for* $c_j \subseteq s$

$$\exists j : \begin{cases} 0 \leqslant j \leqslant p-1 & \neg c_j f_\beta \\ p \leqslant j \leqslant 2p-1 & c_j f_\beta \end{cases}$$

then

$$\{s\}\beta = \begin{cases} \left(\langle\langle c_0, \ldots, \nu_j, \ldots, c_{p-1}, \nu, \ldots, \nu\rangle\rangle, \langle\langle \nu_0, \ldots, c_j, \ldots, \nu, c_p, \ldots, c_{2p-1}\rangle\rangle\right) \\ \left(\langle\langle c_0, \ldots, c_{p-1}, \nu, \ldots, c_j, \ldots, \nu\rangle\rangle, \langle\langle \nu_0, \ldots, \nu, c_p, \ldots, \nu_j, \ldots, c_{2p-1}\rangle\rangle\right) \end{cases}$$

After $p+1$, $p+j+1$, and $p+j+2$ cycles, the following inputs to μ would occur.

$$\begin{aligned} & \{s\}\beta \circ \mu \\ & \vdots \\ \rightsquigarrow^{p+1} & \{(\langle\langle c'_0, \ldots\rangle\rangle, c'_p)\}\mu^0 \\ & \vdots \\ \rightsquigarrow^{j} & \begin{cases} \{(\langle\langle \nu_j, \ldots\rangle\rangle, c'_{p+j})\}\mu^j \\ \{(\langle\langle c'_{j-p}, \ldots\rangle\rangle, \nu_j)\}\mu^j \end{cases} \\ \rightsquigarrow & \begin{cases} \{(\langle\langle c'_{j+1}, \ldots\rangle\rangle, c'_{p+j+1})\}\mu^j \\ \{(\langle\langle c_{j-p+1}\prime, \ldots\rangle\rangle, c'_{j+1})\}\mu^j \end{cases} \end{aligned}$$

In both cases, the result is a maximum queue length in μ of one less than in Proposition 2.37. The input stream in Proposition 2.37 generates the maximum queue length in μ since between the $p+1^{th}$ to $2p^{th}$ graph cycles there is always a non-null context at the head of the input streams to μ. Any other input stream introduces a null context during this time, allowing one context to be forwarded from the merge node while only a single context is added, thus maintaining queue length. □

Having determined the nature of the input streams to a merge node which cause a maximal length queue to form, the relationship between the lengths of those streams and the queue size is established in the following Theorem and Corollaries.

Theorem 2.40 *In Class II graphs, the maximum number of contexts which a merge node may be required to store is p, the difference in path length between a merge node and its matching branch node.*

Proof *Let* $t_p = \langle\!\langle c_0, \ldots, c_{p-1}, c_p, \ldots, c_{2p-1} \rangle\!\rangle \wedge c_{2p}$. *Applying* t_p *to the initially empty graph* g_p *of Proposition 2.37 yields*

$$
\begin{aligned}
& \{t_p\} g_p \\
& \vdots \\
\rightsquigarrow^{2p-1} & \left\{ \{c_{2p}\} \left(\{c_{2p-1}\} \beta((\{c_{p-1}\tau_0 \cdots \tau_{p-2}\}\tau_{p-1} \circ \{c_{p-2}\tau_0 \cdots \tau_{p-1}\}\tau_p), \{c_{2p-2}\}\tau') \right) \right\} \mu^{p-2} \\
\rightsquigarrow & \left\{ \left(\{c_{2p}\} \beta((\circ \{c_{p-1}\tau_0 \cdots \tau_{p-1}\}\tau_p), \{c_{2p-1}\}\tau') \right) \right\} \mu^{p-1} \\
\rightsquigarrow & \begin{cases} \{(\langle\!\langle\,\rangle\!\rangle, \langle\!\langle\,\rangle\!\rangle)\}\, \mu^p & \text{if } c_{2p} = \nu \\ \left\{ ((\{c_{2p}\}\tau_0 \circ \cdots \tau_p), \{\nu\}\tau') \right\} \mu^p & \text{if } c_{2p} \neq \nu, c_{2p} f_\beta \\ \left\{ (\langle\!\langle\,\rangle\!\rangle, \{c_{2p}\}\tau') \right\} \mu^p & \text{if } c_{2p} \neq \nu, \neg c_{2p} f_\beta \end{cases}
\end{aligned}
$$

As all possible input streams result in one or both inputs to μ *being the empty stream, the queue size cannot increase during the next graph cycle.*

$$
\rightsquigarrow \begin{cases} \langle\!\langle c, \{(\langle\!\langle\,\rangle\!\rangle, \langle\!\langle\,\rangle\!\rangle)\}\, \mu^{p-1} \rangle\!\rangle \\ \langle\!\langle c, \left\{ ((\{c_{2p}\tau_0\}\tau_1 \circ \cdots \tau_p), \{\nu\}\tau') \right\} \mu^{p-1} \rangle\!\rangle \\ \langle\!\langle c, \{(\langle\!\langle\,\rangle\!\rangle, \langle\!\langle\,\rangle\!\rangle)\}\, \mu^p \rangle\!\rangle \end{cases}
$$

where c *is the head of the merge queue in* μ. *As* t *contains the stream which induces the largest possible queue in* μ, *from Lemma 2.39, the result follows.* □

This gives rise to the following corollaries.

Corollary 2.41 *In Class II graphs, the length of a queue in a merge node is bounded.*

Proof *Follows from Theorem 2.40.* □

Corollary 2.42 *In Class II graphs, the maximum number of contexts which a merge node may be required to store is* $\#_d$, *the difference in length of the two parallel streams applied to its inputs.*

Proof *From Proposition 2.37, and Theorem 2.40, after $p+1$ cycles, the state of the computation at the input to the merge node s_μ is:*

$$s_\mu = \{\langle\!\langle c_{p+2}, \ldots, c_{2p-1} \rangle\!\rangle\} \left(\{c_{p+1}\}\beta \circ ((\{\nu\}\tau_0 \circ \cdots \circ \{c_0\tau_0 \cdots \tau_{p-1}\}\tau_p), \{c_p\}\tau') \right)$$

This can be written as

$$s_\mu = (c_{p+2} \wedge \cdots \wedge c_{2p-1}) \wedge c_{p+1} \wedge \left(\nu \wedge \langle c_{p-1}\tau_0 \rangle \wedge \cdots \wedge \langle c_0\tau_0 \cdots \tau_p \rangle, \langle c_p\tau' \rangle \right)$$

By Axiom 2.15

$$\begin{aligned} s_\mu =& ((c_{p+2} \wedge \cdots \wedge c_{2p-1}) \wedge c_{p+1} \wedge (\nu \wedge \langle c_{p-1}\tau_0 \rangle \wedge \cdots \wedge \langle c_0\tau_0 \cdots \tau_p \rangle), \\ & (c_{p+2} \wedge \cdots \wedge c_{2p-1}) \wedge c_{p+1} \wedge \langle c_p\tau' \rangle) \\ \therefore \#_d s_\mu =& |\#(c_{p+2} \wedge \cdots \wedge c_{2p-1}) \wedge c_{p+1} \wedge (\nu \wedge \langle c_{p-1}\tau_0 \rangle \wedge \cdots \wedge \langle c_0\tau_0 \cdots \tau_p \rangle) - \\ & \#(c_{p+2} \wedge \cdots \wedge c_{2p-1}) \wedge c_{p+1} \wedge \langle c_p\tau' \rangle| \\ =& p \end{aligned}$$

By Definition 2.22, this can be extended to all paths $\tau_0 \cdots \tau_p$ which include pairs of branch and merge nodes. □

2.3.1 The Effects of Queues on Pipeline Operation

The use of queues in a pipeline — implicit in any CF system as a result of their inclusion in every merge node — has a profound effect on the performance of the processor. *Decoupled* architectures [Smith, 1984] are a class of pipelined system in which hardware queues play an integral role. A decoupled architecture separates memory access and instruction execution into two distinct modules, connected by queues, each module executing instructions from its own instruction stream. Although a dual instruction stream increases the effective instruction issue rate, a more complex compiler is needed, and synchronization between streams is required during the evaluation of conditional branches. The use of queues to isolate

main memory allows access delays and mismatches in instruction processing rate to be absorbed, allowing improved instruction issue and smoothing of long or unpredictable memory references. Queues are also used to counteract the effects of memory latency in the Fortran Optimized Machine [Brantley and Weiss, 1983]. When used in conjunction with a suitable compiler, memory requests can be issued in advance of their being required by a particular instruction.

Since the minimum queue size for any merge node is determined exactly by the position of the node in the CFG, as shown in Corollary 2.42, the need for inclusion of interlocks between nodes to prevent queue overflow is removed. This maximises the performance of any CF system, as hardware interlocks would impose a delay on all contexts, irrespective of their effect on queue length. Implementation of the CFG is also eased in that interlock hardware, which tends to be complex and non-regular [Hennessy *et al.*, 1982], is not required — particularly relevant in any VLSI implementation.

The finite size of the CFG imposes a limit on the granularity of tasks into which any problem that is to be solved using CF may be split.

2.3.2 The Effects of Memory Isolation on Pipeline Operation

The isolation of memory elements in a context flow graph within distinct nodes makes pipelining the natural mechanism for memory access in context flow. The incorporation of data memory into the execution pipeline causes memory references to delay execution no more than any other type of instruction [Smith, 1985]. Neither parallel nor interleaved memory can offer this performance guarantee, being limited by the size of block accessed, and the degree of interleaving respectively. Pipelined memory is the key to sustaining high instruction completion rates in the HEP, and is an integral part of the University of Tokyo Cyclic Pipelined Computer [Shimizu *et al.*, 1986].

Although memories are associated with a single node, shared memory structures can be implemented by merging streams of contexts containing access requests and using the resulting stream as input to determine the addresses to be accessed. This ensures that strict ordering of memory accesses can be enforced, and freedom from side-effects due to multiple accesses can be endured.

Many memory structures perform very localized functions, for example, queues in merge nodes. Clearly, such devices should not be accessible from any node in a graph. By restricting access to a memory to the node in which it is located, all memory structures, irrespective of their function or operational characteristics, can be treated in a uniform manner.

Summary

Context flow provides a simple yet powerful mechanism for constructing parallel pipelined systems from a small set of basic building blocks. The context flow model consists of a set of core nodes which can be used to construct representations of systems in the form of context flow graphs. These nodes provide simple functionality and make possible the creation of multiple independent paths through the graph. Accompanying the set of core nodes is a set of basic evaluation rules which govern the movement of contexts in the graph.

- Each context may represent only one process, but tasks may be divided into sub-processes to allow parallel evaluation using multiple contexts.

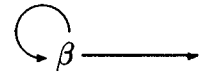
- Flow of contexts through the CFG is synchronous, with each node performing its operation in a single graph cycle period.

- Exactly one context may be transmitted along an arc during a graph cycle. In the case where no information is to be passed, a null context is generated.

- Only contexts of one type may be passed along a given arc during the entire period of graph activity. There is no other restriction on the types of arc attached to the ports of a node.

- Branch nodes route contexts solely on the basis of their current contents, not on any external state information.

- Cycles are permitted in a graph. However, self-loops of the form,



  cannot occur, as branch nodes cannot alter contexts.

- Each transformation node is distinct, and can only interact with other nodes through the arcs in the graph.

"πάντα χωρεῖ, ὀνδὲν μένει"

"Everything flows and nothing stays"

— HERACLITUS (c.535–c.475 B.C.), *Cratylus, 402a*

Chapter 3

Higher-Order Graphs

A formal basis for context flow has been established, and some properties concerning queue lengths in certain types of acyclic context flow graph proved. In this chapter, the notions of *stream tuples* and *named edges* are introduced. Using these concepts, it is shown that for both acyclic graphs with arbitrary numbers of branch and merge nodes and cyclic graphs, the length of queue in all merge nodes is bounded. The boundedness of queues is essential for the implementation of CF systems. As most systems are closed, and therefore contain cycles, it is necessary to show that boundedness extends to these types of graph. In order to do this, additional notation is required, in particular to describe loops. Methods for calculating a numerical value for the maximum queue length and for graph initialization are presented.

3.1 Stream Tuples

The transformation, branch and merge nodes, and the context stream operators provide a simple yet powerful means to describe certain classes of pipelined system. Their definitions avoid the inclusion of implementation specific features, and are therefore suitable for use in a wide variety of applications. These nodes form the basic or *core* nodes of a CF system. At the lowest level, a *first-order* CFG contains only core nodes. To permit a more structured approach to CF system design, *higher-order* graphs can be defined. In these graphs, restrictions as to the degree of the nodes are relaxed, allowing *structured nodes* to be created with multiple input and output arcs. Structured nodes are formed from interconnections of lower-order nodes or core nodes. As higher-order graphs are built from core nodes at the lowest level, they retain the property of passing only one context on each arc during a graph cycle.

As more than two context streams may exist in parallel in higher-order graphs, the concept of a pair of context streams is extended to groupings of an arbitrary number of independent streams, called *stream tuples.* Table 3–1 defines the set $\mathbb{S}$ of stream tuples, and the basic tuple operators.

Definition 3.1 A stream tuple $\mathcal{S} = (s_1, \ldots, s_n)$ is a collection of n independent context streams.

Definition 3.2 The cardinality of a stream tuple $\mathcal{S}$ is the number of context streams in $\mathcal{S}$:

$$|(s_1, \ldots, s_n)| = n$$

A context stream is defined as a member of a stream tuple if it is contained within the tuple.

Notation	Meaning
$\langle\!\langle\, \rangle\!\rangle$	the tuple containing empty streams
(s)	a tuple containing a single stream
$(s_1, \ldots, s_n)$	a tuple containing n streams
$\lvert\mathcal{S}\rvert$	cardinality of tuple $\mathcal{S}$
$s \subseteq \mathcal{S}$	stream s is contained in tuple $\mathcal{S}$
$\mathcal{S}_1 \cong \mathcal{S}_2$	tuple $\mathcal{S}_1$ is equivalent to $\mathcal{S}_2$
$\mathrm{S}\Pi_i$	projection of the i^{th} stream from tuple $\mathcal{S}$
$\mathcal{S}_1 \bowtie \mathcal{S}_2$	join of tuples $\mathcal{S}_1$ and $\mathcal{S}_2$
$\mathcal{S}_1 \wedge \mathcal{S}_2$	concatenation of tuples $\mathcal{S}_1$ and $\mathcal{S}_2$
$\#\mathcal{S}$	length of tuple $\mathcal{S}$
$\mathcal{S}_\circ$	heads of streams in tuple $\mathcal{S}$
$\mathcal{S}'$	tails of streams in tuple $\mathcal{S}$

Table 3–1: Tuple operations

Definition 3.3 $s \subseteq (s_1, \ldots, s_n)$ *if* $\exists i : 1 \leqslant i \leqslant n, s \approx s_i$

The notion of equivalence for stream tuples is slightly different to that for context streams, in that the ordering of streams within a tuple is significant. The position of a stream within a tuple determines the port of the node with which it is to be associated. Equivalence is defined only for tuples of the same cardinality.

Definition 3.4 Given two stream tuples $\mathcal{S}$,$\mathcal{T}$, where $|\mathcal{S}| = |\mathcal{T}|$, $\mathcal{S}$ and $\mathcal{T}$ are equivalent if each context stream in $\mathcal{S}$ is equivalent to the corresponding context stream in $\mathcal{T}$.

$$(s_1, \ldots, s_n) \cong (t_1, \ldots, t_n) \quad \textit{if} \quad \forall i : 1 \leqslant i \leqslant n, s_i \approx t_i$$

Lemma 3.5 $\cong$ *is an equivalence relation over stream tuples.*

Proof *Let* $\mathcal{S} = (s_1, \ldots, s_n), \mathcal{T} = (t_1, \ldots, t_n), \mathcal{U} = (u_1, \ldots, u_n)$, *such that* $|\mathcal{S}| = |\mathcal{T}| = |\mathcal{U}| = n$.

- $\cong$ *is reflexive. Clearly* $\forall i : 1 \leqslant i \leqslant n, s_i \approx s_i$, *therefore* $\mathcal{S} \cong \mathcal{S}$.

- $\cong$ *is symmetric. Suppose* $\mathcal{S} \cong \mathcal{T}$, *then by definition* $\forall i : 1 \leqslant i \leqslant n, s_i \approx t_i$, *therefore, as* $\approx$ *is symmetric,* $\forall i : 1 \leqslant i \leqslant n, t_i \approx s_i$, *therefore* $\mathcal{T} \cong \mathcal{S}$.

- $\cong$ *is transitive, Suppose* $s \subseteq \mathcal{S}$, *then* $\mathcal{S} \cong \mathcal{T} \Rightarrow s \subseteq \mathcal{T}$ *and* $\mathcal{T} \cong \mathcal{U} \Rightarrow s \subseteq \mathcal{U}$. *Conversely, suppose* $s \subseteq \mathcal{U}$, *then* $\mathcal{T} \cong \mathcal{U} \Rightarrow s \subseteq \mathcal{T}$ *and* $\mathcal{S} \cong \mathcal{T} \Rightarrow s \subseteq \mathcal{S}$. *Thus* $\mathcal{S} \cong \mathcal{U}$.

This completes the proof. □

3.1.1 Tuple Operations

The identity element for all operations on stream tuples is a tuple containing empty streams, and is denoted by $\langle\!\langle\!\langle\, \rangle\!\rangle\!\rangle$. The tuple containing only a single stream is written as (s).

It is useful to define a projection operator Π to extract one or more streams from a stream tuple.

Definition 3.6

$$(s)\Pi \cong s$$

$$(s_1, \ldots, s_n)\Pi_i \cong s_i \ : \ 1 \leqslant i \leqslant n, n \in \mathbb{N}$$

$$(s_1, \ldots, s_n)\Pi_{i,\ldots,j} \cong (s_i, \ldots, s_j) \ : \ 1 \leqslant i < j \leqslant n, n \in \mathbb{N}$$

Axiom 3.7 $\langle\!\langle\!\langle\, \rangle\!\rangle\!\rangle\Pi \cong \langle\!\langle\, \rangle\!\rangle$

It may also be necessary to perform the converse operation, joining a context stream with a tuple, or joining two tuples together. This is performed by the join operator $\bowtie$.

Definition 3.8

$$(s_1, \ldots, s_n) \bowtie t \cong (s_1, \ldots, s_n, t)$$

$$(s_1, \ldots, s_n) \bowtie (t_1, \ldots, t_m) \cong (s_1, \ldots, s_n, t_1, \ldots, t_m)$$

The join operator has the following properties:

Axiom 3.9 $\mathcal{S}_1 \bowtie (\mathcal{S}_2 \bowtie \mathcal{S}_3) \cong (\mathcal{S}_1 \bowtie \mathcal{S}_2) \bowtie \mathcal{S}_3$

Axiom 3.10 $\mathcal{S} \bowtie \langle\!\langle\!\langle \rangle\!\rangle\!\rangle \cong \mathcal{S}$

Axiom 3.11 $\mathcal{S}_1 \bowtie \mathcal{S}_2 \cong \mathcal{S}_1 \bowtie \mathcal{S}_3 \Rightarrow \mathcal{S}_2 \cong \mathcal{S}_3$

Axiom 3.12 $\mathcal{S}_1 \bowtie \mathcal{S}_2 \cong \langle\!\langle\!\langle \rangle\!\rangle\!\rangle \Rightarrow \mathcal{S}_1 \cong \mathcal{S}_2 \cong \langle\!\langle\!\langle \rangle\!\rangle\!\rangle$

The concatenation operator $\wedge$ is extended to operate on stream tuples.

Definition 3.13 $(s_1, \ldots, s_n) \wedge (t_1, \ldots, t_n) \cong (s_1 \wedge t_1, \ldots, s_n \wedge t_n)$

The following axioms define concatenation to be associative and commutative when applied to stream tuples.

Axiom 3.14 $\mathcal{S}_1 \wedge (\mathcal{S}_2 \wedge \mathcal{S}_3) \cong (\mathcal{S}_1 \wedge \mathcal{S}_2) \wedge \mathcal{S}_3$

Axiom 3.15 $\mathcal{S}_1 \wedge \mathcal{S}_2 \cong \mathcal{S}_2 \wedge \mathcal{S}_1$

Axiom 3.16 $\mathcal{S} \wedge \langle\!\langle\!\langle \rangle\!\rangle\!\rangle \cong \langle\!\langle\!\langle \rangle\!\rangle\!\rangle \wedge \mathcal{S} \cong \mathcal{S}$

Axiom 3.17 $\mathcal{S}_1 \wedge \mathcal{S}_2 \cong \mathcal{S}_1 \wedge \mathcal{S}_3 \Rightarrow \mathcal{S}_2 \cong \mathcal{S}_3$

Axiom 3.18 $\mathcal{S}_1 \wedge \mathcal{S}_2 \cong \langle\!\langle\!\langle \rangle\!\rangle\!\rangle \Rightarrow \mathcal{S}_1 \cong \mathcal{S}_2 \cong \langle\!\langle\!\langle \rangle\!\rangle\!\rangle$

Axiom 3.19 $(s_1, \ldots, s_n) \wedge t \cong (s_1 \wedge t, \ldots, s_n \wedge t)$

It is important to differentiate between joining and concatenation. Joining operates on *tuples*, and does not affect the streams within the tuples. Concatenation leaves tuple size unchanged, operating of the *streams* which make up the tuple.

The length operators $\#^+$ and $\#^-$ are extended over stream tuples, yielding the lengths of the longest and shortest streams in a stream tuple.

Definition 3.20

$$\#^+(s \bowtie t) = \max(\#^+s, \#^+t)$$

where

$$\#^+s = \#s \quad \text{and} \quad \#^+(s \wedge t) = \#^+s + \#^+t$$

Definition 3.21

$$\#^-(s \bowtie t) = \min(\#^-s, \#^-t)$$

where

$$\#^-s = \#s \quad \text{and} \quad \#^-(s \wedge t) = \#^-s + \#^-t$$

Definition 3.22 $\#_d(s \bowtie t) = \#^+(s \bowtie t) - \#^-(s \bowtie t)$ where $\#_d s = 0$.

The head operator $_\circ$, when applied to a tuple, returns a tuple whose members are the heads of each of the constituent streams. Similarly, the tail operator $'$ produces a tuple containing the tails of the member streams.

Definition 3.23 $(s_1, \ldots, s_n)_\circ \cong (s_{1_\circ}, \ldots, s_{n_\circ}) \qquad (s_1, \ldots, s_n)' \cong (s_1', \ldots, s_n')$

The head of the empty tuple is defined as a tuple of null contexts, and the tail, as the empty tuple itself.

Definition 3.24 $\langle\!\langle\, \rangle\!\rangle_\circ \cong (\nu, \ldots, \nu) \qquad \langle\!\langle\, \rangle\!\rangle' \cong \langle\!\langle\, \rangle\!\rangle$

3.1.2 Named Edges

In many structured design methodologies it is useful to provide a series of intermediate definitions which are then combined to form the final result — the definition of procedures or functions in high-level programming languages, for example. Not only is this often notationally convenient, but it also can allow more efficient implementation. A similar situation can arise in the definition of structured nodes where several parallel output streams are defined in terms of a common function of the inputs. In order to avoid repeating the function at each of the streams for which it is required, the notion of a *named edge* is introduced, which defines a function which can be reused at any point within the definition of the structured node.

Definition 3.25 Let ϕ_i be the function defined by node i, $\phi \in \{\beta, \mu, \tau\}$. The expression $\epsilon \triangleq \phi_0 \circ \cdots \circ \phi_n$ defines an edge associating the name ϵ with the composition of functions ϕ_1 to ϕ_n.

Remark 3.26 In the definition of the function representing a graph, the composition operator $\circ$ represents an un-named edge connecting two nodes.

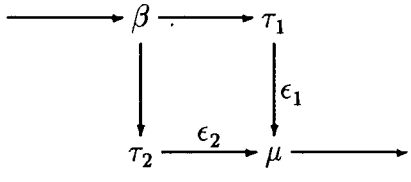
Once defined, named edges can be used to replace sequences of nodes connected by un-named edges in a graph. For example, the graph

$$g = \phi_0 \circ \cdots \circ \phi_n \circ \phi'$$

can be written as

$$g = \epsilon\phi'$$

using the value of ϵ from Definition 3.25; and the graph



can be written as

$$g = (\epsilon_1, \epsilon_2)\mu \ : \ \epsilon_1 \triangleq \beta \circ \Pi_1 \circ \tau_1, \epsilon_2 \triangleq \beta \circ \Pi_2 \circ \tau_2$$

3.1.3 Structured Nodes

A structured node has an internal hierarchy, and a functionality determined by the aggregate operation of the core nodes from which it is constructed. A structured node is itself a graph which is treated as a node for the purposes of constructing higher-order graphs. As projection and join operations are required to manipulate the multiple stream inputs and outputs permitted in structured nodes, the behaviour of branch and merge nodes can be redefined in terms of these operations, for tuples of cardinality two.

Axiom 3.27

$$\{s\}\beta \rightsquigarrow \begin{cases} \langle\langle\langle\rangle\rangle\rangle & \text{if} \quad s \approx \langle\langle\rangle\rangle \\ \langle\langle\langle\rangle\rangle\rangle \wedge \{s'\}\beta & \text{if} \quad s_\circ = \nu, s' \not\approx \langle\langle\rangle\rangle \\ (s_\circ \bowtie \nu) \wedge \{s'\}\beta & \text{if} \quad s_\circ f_\beta, s' \not\approx \langle\langle\rangle\rangle \\ (\nu \bowtie s_\circ) \wedge \{s'\}\beta & \text{if} \quad \neg s_\circ f_\beta, s' \not\approx \langle\langle\rangle\rangle \end{cases}$$

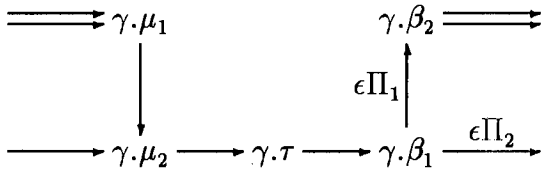
Axiom 3.28

$$\{\mathcal{S}\}\mu^0 \rightsquigarrow \begin{cases} \langle\langle \nu, \{\mathcal{S}'\}\mu^0 \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} = \mathcal{S}\Pi_{2\circ} = \nu \\ \langle\langle \mathcal{S}\Pi_{1\circ}, \{\mathcal{S}'\}\mu^0 \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} \neq \nu, \mathcal{S}\Pi_{2\circ} = \nu \\ \langle\langle \mathcal{S}\Pi_{2\circ}, \{\mathcal{S}'\}\mu^0 \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} = \nu, \mathcal{S}\Pi_{2\circ} \neq \nu \\ \langle\langle \mathcal{S}\Pi_{1\circ}, \{\mathcal{S}'\}\mu^{(\mathcal{S}\Pi_{2\circ})} \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} = \mathcal{S}\Pi_{2\circ} \neq \nu \end{cases}$$

$$\{\mathcal{S}\}\mu^{(c_0,\ldots,c_n)} \rightsquigarrow \begin{cases} \langle\langle c_0, \{\mathcal{S}'\}\mu^{(c_1,\ldots,c_n)} \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} = \mathcal{S}\Pi_{2\circ} = \nu \\ \langle\langle c_0, \{\mathcal{S}'\}\mu^{(c_1,\ldots,c_n,\mathcal{S}\Pi_{1\circ})} \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} \neq \nu, \mathcal{S}\Pi_{2\circ} = \nu \\ \langle\langle c_0, \{\mathcal{S}'\}\mu^{(c_1,\ldots,c_n,\mathcal{S}\Pi_{2\circ})} \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} = \nu, \mathcal{S}\Pi_{2\circ} \neq \nu \\ \langle\langle c_0, \{\mathcal{S}'\}\mu^{(c_1,\ldots,c_n,\mathcal{S}\Pi_{1\circ},\mathcal{S}\Pi_{2\circ})} \rangle\rangle & \text{if} \quad \mathcal{S}\Pi_{1\circ} = \mathcal{S}\Pi_{2\circ} \neq \nu \end{cases}$$

It should be noted that the only difference between the above Axioms and Axioms 2.31 and 2.38, on pages 50 and 54 respectively, is in the notation. There is no generalization of either merge nodes to accept more than two inputs, or branch nodes to produce more than two outputs.

Using the projection and joining operators, the operation of a node γ whose structure is



can be written as

$$\gamma = (\epsilon\Pi_1 \circ \beta_2) \bowtie \epsilon\Pi_2 \; : \; \epsilon \triangleq \left\{(\Pi_{1,2} \circ \mu_1) \bowtie \Pi_3\right\} \mu_2 \circ \tau \circ \beta_1$$

The definition of operations on stream tuples, coupled with the introduction of named edges, provides greater flexibility for the definition of graphs. Higher-order graphs are defined in exactly the same way as first-order graphs, with the restriction that cardinality of the result of a node must equal the sum of the input cardinalities of its successors. Class III and IV context flow graphs can also be defined using this notation.

3.2 Queue Boundedness in Class III Graphs

In Class II graphs, the paths between a branch node and a merge node contain equal numbers of other branch-merge pairs. In Class III graphs, this is not the case as the paths between branch and merge nodes may contain any combination of nodes. Using the concepts and notation of stream tuples, the property of queue boundedness that exists in Class II graphs can be shown to extend to Class III graphs.

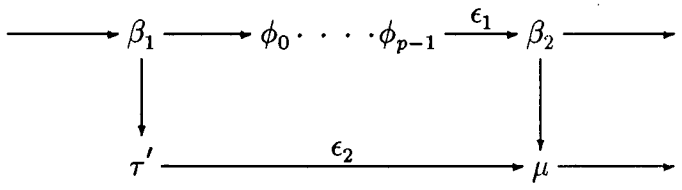
Proposition 3.29 *The number of non-null contexts output from a merge node is equal to the number of non-null contexts input to the node.*

Proof *From Corollary 2.35, a merge node cannot destroy or create a non-null context.* □

In order to show that the length of a queue in a merge node is bounded in a Class III graph, two cases must be considered — a graph which contains more branch than merge nodes, and a graph which contains more merge than branch nodes.

Lemma 3.30 *If the number of branch nodes is greater than the number of merge nodes in the path between a branch-merge pair, the length of queue in the final merge node is bounded.*

Proof *Consider the graph*



initially empty.

$$g = \{\epsilon_1\beta_2 \circ \Pi_2 \bowtie \epsilon_2\}\mu :$$

$$\epsilon_1 \triangleq \beta_1 \circ \Pi_1 \circ \phi_0 \circ \cdots \circ \phi_{p-1},$$

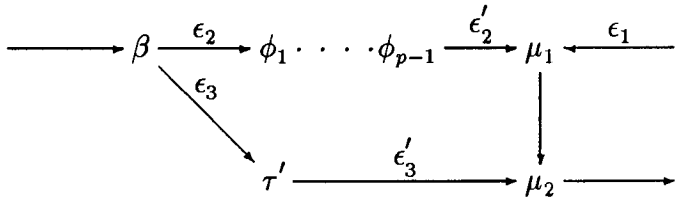
$$\epsilon_2 \triangleq \beta_1 \circ \Pi_2 \circ \tau'$$

The maximum length queue in μ occurs when $\forall c : c \subseteq s, s\epsilon_1\beta_2\Pi_1 \approx \langle\langle\rangle\rangle$ Therefore, if $\exists c : c \subseteq s, c\epsilon_1\beta_2\Pi_1 \neq \nu$ then $c\epsilon_1\beta_2\Pi_2 = \nu$ and, from Lemma 2.39, the queue will not reach its maximum length. □

Lemma 3.31 *If the number of merge nodes is greater than the number of branch nodes in the path between a branch-merge pair, the length of queue in the merge*

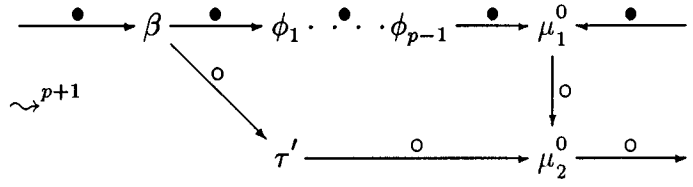
node is bounded.

Proof *Consider the graph*

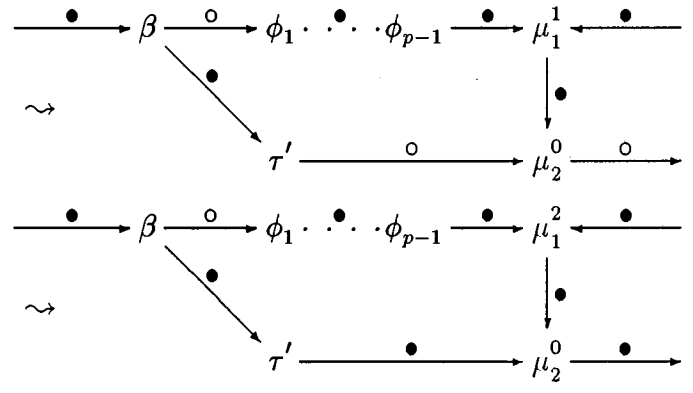


initially empty, where the queue length in μ_1 is bounded by q. There are three cases to consider:

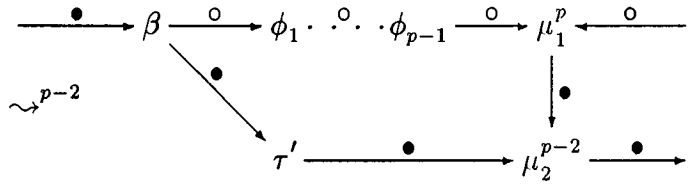
- *Suppose $p = q$. After $p + 1$ cycles,*



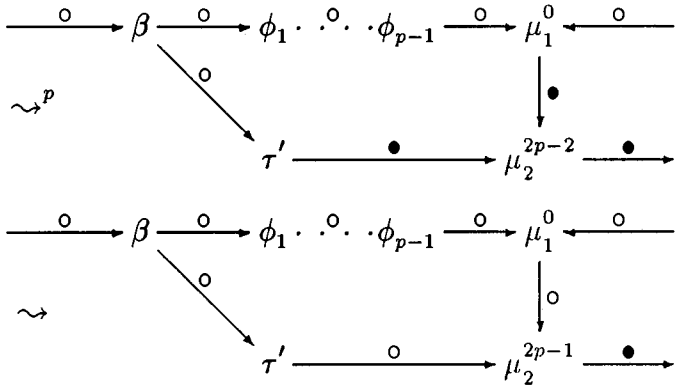
If contexts are now routed via ϵ_3,



μ_2 must now start to queue contexts. After μ_1 has merged the p length streams ϵ_1 and ϵ_2', p contexts are contained in its queue and p contexts have been forwarded. Of the latter, $p-2$ contexts have been merged with non-null contexts from ϵ_3'.

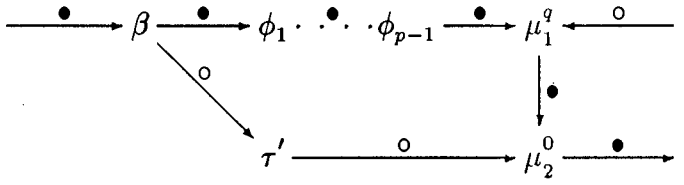


During the next p cycles, the queue in μ_1 empties into μ_2, merging with a stream of contexts from ϵ_3', causing the queue in μ_2 to grow to $2p-2$ elements.

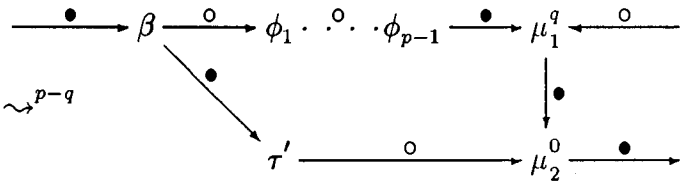


As the inputs to both merge nodes are null, the queue lengths remain constant.

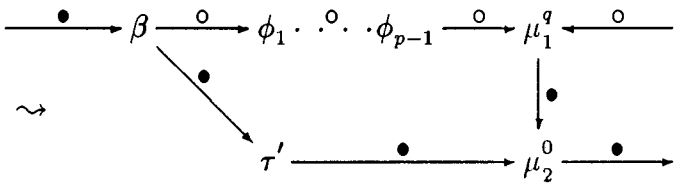
- *Suppose $p > q$. After $p+1+q$ cycles, edge ϵ_1 always passes a null context.*

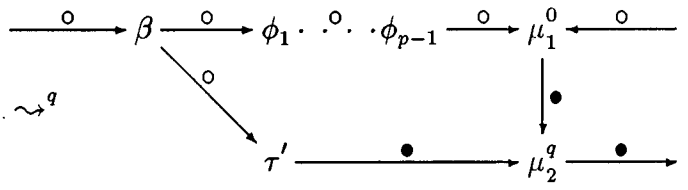


As $p > q$, non-null contexts are routed via ϵ_2 maintaining the length of queue in μ_1 at q elements. At the same time, null contexts are routed via ϵ_3, thus maintaining an empty queue in μ_2. After a further $p-q$ cycles, μ_1 holds q contexts in its queue, and has forwarded p contexts.



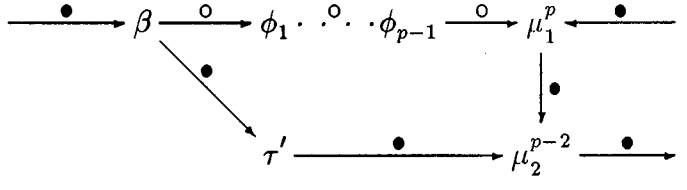
The q contexts remaining in μ_1 merge with those from ϵ_3'.



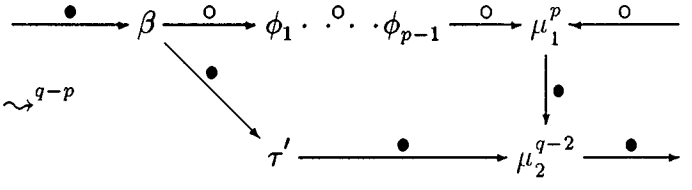


Therefore, the maximum queue length in μ_2 is $q+1$ which is less than that in the case for $p = q$.

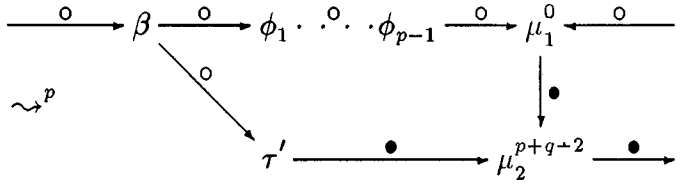
- *Suppose $p < q$. After $2p+1$ cycles, edge ϵ_2' always passes a null context. $q-p$ contexts remain to be input to μ_1 on ϵ_1, while the q contexts forwarded by μ_1 have merged with a stream of $p-2$ contexts from ϵ_3' at μ_2. As $p < q$, $p-2 < q$, and the queue in μ_2 grows to $p-2$ elements.*



After a further $q-p$ cycles, the queue length in μ_1 remains at p, and the queue length in μ_2 grows by $q-p$ to $q-2$ contexts.



After another p cycles, the queue in μ_1 is empty, and the queue in μ_2 contains $(p+q)-2$ contexts.



Therefore, maximum queue length in μ_2 is $(p+q)-1$, which is less than that in the case for $p = q$.

This completes the proof. □

Having proved that if the numbers of branch and merge nodes contained in a context flow graph are different, the queue lengths in the merge nodes remain bounded, the following Theorem follows by definition.

Theorem 3.32 *The length of queue in a merge node in a Class III context flow graph is bounded.*

Proof *Follows from the definition of a Class III graph, Lemma 3.30 and Lemma 3.31.* □

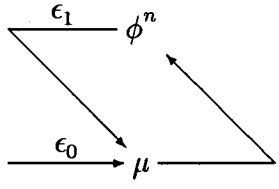
3.3 Queue Boundedness in Class IV Graphs

Class IV context flow graphs differ from Class III graphs in that cycles are permitted. It can be shown that queue boundedness extends to certain cyclic graphs, but that in others, queue length is directly proportional to input stream length. As any practical CF machine is closed and therefore cyclic, it is essential that all queue lengths are bounded.

Definition 3.33 A cycle or loop in a context flow graph is *open* if it contains either a branch node for which one of the outputs is not connected to any point in the loop, or a merge node for which one of the inputs is not connected to any point in the loop; otherwise it is *closed.*

Lemma 3.34 *Queue length is potentially unbounded in open context flow graphs containing closed loops.*

Proof *Consider the class of graphs*

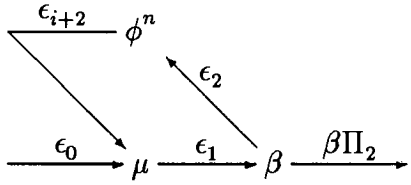


where ϕ^n denotes a sequence of branch, merge and transformation nodes containing up to n contexts.

Clearly, if any non-null context enters the graph at ϵ_0, it continues to circulate. If the number of contexts exceeds n then any additional contexts must be stored in the queue at μ. As the queue length is directly proportional to the number of non-null context in the stream at ϵ_0, its length is potentially unbounded. □

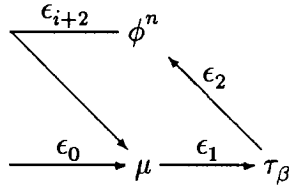
Lemma 3.35 *Queue length is bounded in open context flow graphs containing open loops if and only if the number of contexts entering the graph is equal to the number leaving.*

Proof *Consider the class of graphs*



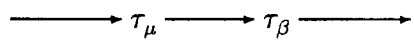
Let $s = \langle\langle c_1, c_2, \ldots, c_n \rangle\rangle$ such that $\forall i : 0 \leqslant i \leqslant n, c_i \not\approx \nu$. Applying s to the graph yields the following situations:

- *If $\beta\Pi_2 \approx \langle\langle\,\rangle\rangle$, then the graph is equivalent to*



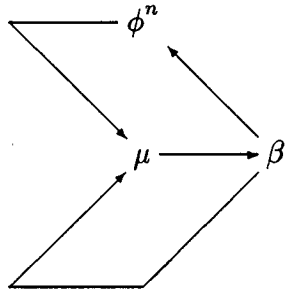
  and, from Lemma 3.34, the queue length of μ is unbounded.

- *If $\beta\Pi_2 \approx s$, then the graph is equivalent to*

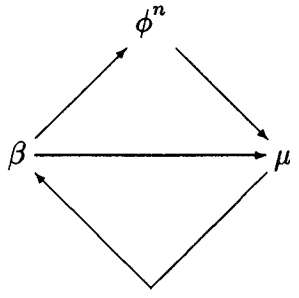


  and no contexts require to be queued.

As queue length is determined by the proportion of contexts which are routed out of the loop at the branch node, there must be a point at which the rate of flow of contexts out of the loop matches the rate of flow of contexts into the loop, resulting in a constant queue length. If the outflow is less than this, the queue will grow, and if greater, the queue will shrink. When this condition is met, the graph becomes equivalent to



which can also be drawn as



From Corollary 2.41, the length of queue in μ is bounded. This completes the proof. □

Theorem 3.36 *If a context flow graph contains cycles, it must be closed for the length of queues in the merge nodes to be bounded.*

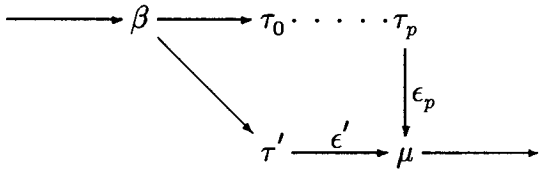
Proof *Follows from Lemmas 3.34 and 3.35.* □

3.4 Determining Maximum Merge Queue Length

The results of Corollary 2.41, Theorem 3.32 and Theorem 3.36 show that the queue lengths in merge nodes are bounded in all acyclic and closed cyclic context flow graphs. It is possible to determine a numerical value for the upper bound on queue length by the position of the merge node in the graph. The desirability of being able to determine the upper bound on queue length in an analytic manner, as opposed to requiring simulation, is clear, as simulation cannot assess the effects of all possible distributions of contexts which may arise. There are four configurations to consider.

3.4.1 Simple Transformation Sequences

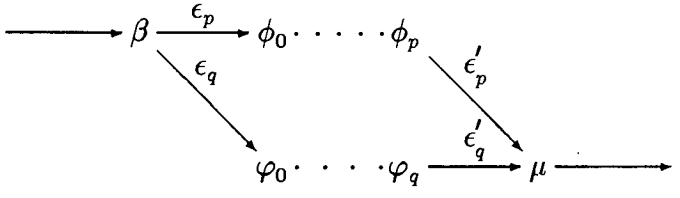
In the case where both inputs to the final merge node are streams which have only been operated on by transformation nodes, for example



the maximum length of queue at μ is given by the result of Theorem 2.40, as the difference in path length between β and μ, or by Definition 3.22 as $\#_d(\epsilon_p \bowtie \epsilon')$.

3.4.2 Matched Branch-Merge Pairs

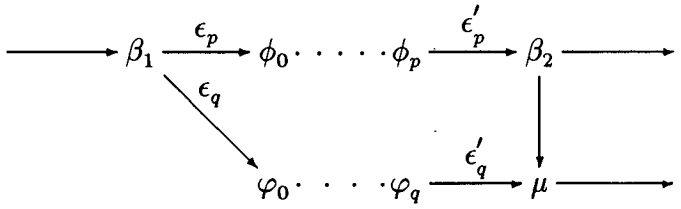
The above result can be generalized to the situation where the paths between the nodes themselves contain branch-merge pairs.



A maximum length queue is induced in μ if all contexts are first routed along the longest path between β and μ via ϵ_p and then by the shortest path via ϵ_q, which is given by $\#_d(\epsilon'_p \bowtie \epsilon'_q)$.

3.4.3 Unmatched Branch Nodes

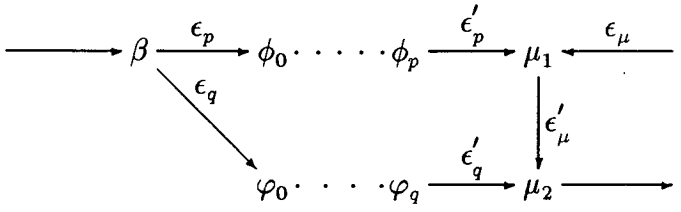
If one or both input paths to μ contains an unmatched branch, a maximal length queue in μ will occur if the unmatched branches route all contexts to the merge node. This corresponds to the graph



where $\epsilon'_p\beta_2\Pi_1 \approx \langle\langle\, \rangle\rangle$. The upper bound on the queue length in μ is given by $\#_d(\epsilon'_p\beta_2\Pi_1 \bowtie \epsilon'_q)$.

3.4.4 Unmatched Merge Nodes

The situation in which unmatched merge nodes exist in the input paths of a merge node is complicated by the fact that this creates two potential sources of contexts. In the graph



it is assumed that μ_1 has an upper bound on its queue length of m, given by the stream difference of its inputs. The maximum length of queue in μ_2 is determined by incorporating the queue into the input stream as a point source of m contexts,

and then evaluating the stream difference. In the above graph, the upper bound on queue length in μ_2 is given by

$$\#_d(\epsilon'_q \bowtie \epsilon'_\mu) = \max(\#^+\epsilon'_q, \#^+\epsilon'_\mu + m) - \min(\#^-\epsilon'_q, \#^-\epsilon'_\mu + m)$$

3.5 Graph Initialization

In each of the preceding situations it has been assumed that the graph is empty prior to application of a stream — each node containing a null context. Whilst this is convenient for purposes of clarity, it is somewhat unrealistic, especially in the case of closed cyclic graphs. It is quite conceivable that an acyclic graph or a particular section of cyclic graph be initially empty, however, a closed cyclic graph will require to be initialized by non-null contexts.

The only case which will be considered here is where contexts are initially stored in a single queue. If a graph represents an execution pipeline, it is clearly the case that no instruction should be started at a point which corresponds to a partial state of execution.

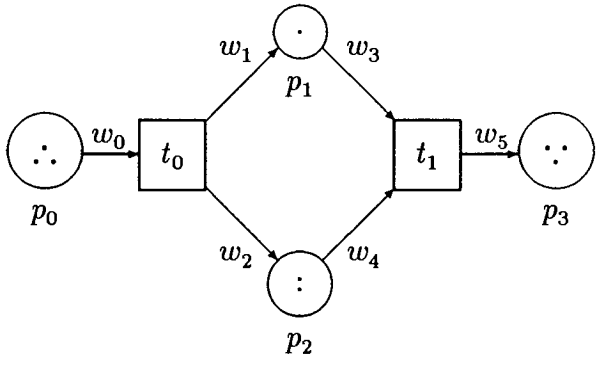
If a graph is initialized with a number of contexts which is less than the shortest path length in the graph, then the contexts will, after initial dispersal from the queue, distribute themselves throughout the graph, allowing maximum queue lengths to be calculated as detailed above. If, alternatively, the graph is initialized with a number of contexts which is greater than the shortest path length, the initialization queue will contain, on average, a quantity of contexts equal to the number used in the initialization less the mean context latency. In both cases the queue used for initialization must be capable of holding the initial number of contexts. Providing initialization is made at one node, calculation of other maximum queue lengths may be made according to the procedures of Section 3.4.

3.6 Context Flow as a Model of Parallel Computation

The definition of context flow avoids the inclusion of implementation-specific features, allowing the model to be applied in a wide variety of situations. Contexts are defined merely as tuples of information with some internal, but undefined, structure, and nodes are defined simply as the sites of function application. CF is an abstract mathematical model of parallel computation. In order to obtain an assessment of its relative strengths and weaknesses, it is instructive to compare it to two other existing representations of parallel processing — Petri Nets and data flow graphs.

3.6.1 Petri Nets and Data Flow Graphs

A *Petri net* [Reisig, 1985] is a directed graph containing two different types of node — *places* and *transitions*. Each place is weighted by a number of *tokens*, with places connected to transitions by weighted arcs. Tokens circulate in the net as a consequence of *firings*, events which enable a transition when, for all input arcs of a transition, the weight of a place and the arc which connects it to the transition are equal. For example, the net



has four places, p_0, p_1, p_2 and p_3, and two transitions t_1 and t_2, connected by arcs with weights w_0 to w_5. A net can also be represented by a matrix in which the rows represent transitions and the columns represent places. The above net can be written in matrix form as

$$\begin{pmatrix} -w_0 & w_1 & w_2 & 0 \\ 0 & -w_3 & -w_4 & w_5 \end{pmatrix}$$

Tokens are distributed in the graph according to an initial marking, which, in the above example, is

$$\begin{pmatrix} 3 \\ 1 \\ 2 \\ 3 \end{pmatrix}$$

A *data flow graph* (DFG) [Davis and Keller, 1982] is a directed graph which consists of three different types of node — *function* nodes, *distributors* and *selectors.* the arcs which connect the nodes denote the data dependencies present in a computation. Tokens are used to represent the flow of data between nodes, each token representing one data or boolean value. Tokens reside on arcs rather than in nodes, each arc storing only one token at a given time. Computation events are initiated when a token is present on all the input arcs to a node, and no tokens exist on the output arcs. The node then consumes the input tokens and generates tokens for the output arcs.

Treatment of Time

In both Petri nets and data flow graphs, operations are performed as soon as sufficient data, represented by tokens, are present — they are both *asynchronous* models. Neither requires an explicit denotation of time, initiation of events being governed by availability of data. Context flow, however, has a very strong notion

of time, encapsulated in the graph cycle period. Although time is not explicitly referenced, the duration of operations in all nodes is equal. CF is a synchronous model of parallel computation.

It is the synchronous nature of CF which necessitates the inclusion of null contexts in the model. In both Petri nets and DFGs, the concept of an empty token does not exist, and is indeed unnecessary. An empty token would simply represent the absence of information, which is equally denoted by the absence of a token which does contain information. Absence of a token causes a delay in the firing of a function, just as the inclusion of a null context in a context stream causes a delay of one graph cycle period in CF.

Parallelism

In CFGs, Petri nets and DFGs, the existence of parallelism is both inherent and obvious from the graphical description. In CF, the spatial parallelism indicated by the existence of multiple paths through the graph is augmented by the temporal parallelism which exists between each node. In DFGs, the parallelism is a result of, and hence limited only by, data dependencies.

Freedom From Side-Effects

All three models have no notion of a globally accessible memory. In Petri nets and DFGs, all computation is performed with values or fixed name-value bindings, all references to an identifier yielding the same value for the entire duration of the computation. CF is more general in its treatment of values in that changeable memory locations may exist; but as access is controlled by a single transformation node, non-local destructive updating of a memory location cannot be performed. While it may be possible for two contexts to access a common memory location,

this would represent some form of synchronization of, or communication between, the processes, which can be assumed to be intentional.

Locality of Effect

In Petri nets and DFGs, tokens generated at a transformation or function node only cause changes in the nodes which are its immediate successors. The same is also true for CF where applications of functions at one transformation node do not affect other contexts or the contents of memory locations at other nodes.

Non-determinism

If a Petri net has a place with two output arcs of weight one, and a single input arc also of weight one, then either of the successors may be fired non-deterministically. In data flow, a merge node combines tokens in a non-deterministic manner. The relative ordering of tokens from each of the two streams is preserved, but the stream from which a given token occurs in the output is arbitrary. CF performs a deterministic merge of context streams, the repeatability of a result for a given input being guaranteed by the synchronous arrival of contexts at the node.

Although there are several similarities between the three models, their differing treatments of time make comparisons difficult as this one property has an important bearing on the operation of the models.

Summary

Using the concepts of stream tuples and named edges, a mechanism for the description of arbitrary context flow graphs has been developed. From this, comes a formal method for introducing hierarchies, bringing the benefits of structured design techniques to the creation of CFGs. The property of queue boundedness, demonstrated for certain acyclic CFGs in Chapter 2, has been shown to extend to all acyclic and closed cyclic graphs. Methods to calculate the numeric value of the upper bound have been presented. These allow maximum queue lengths, an important facet of any CF implementation, to be determined analytically rather than by simulation. This has an important bearing on the feasibility of CF implementations. The relationship between CF and other parallel processing models has been explored.

"All things are a flowing."

— EZRA POUND (1885–1972)

Chapter 4

Context Flow Architecture

The principles of the context flow model, and the operation of the core nodes from which context flow systems are constructed, have been defined. The result is a formal framework for defining the structure and operation of context flow graphs, which has been used to prove that the queue length in any merge node has a fixed upper bound. This Chapter is concerned with more pragmatic aspects of context flow. Example designs for a number of example CF structures are presented which perform commonly needed functions in computer systems, such as decoding and access to shared memory, in a pipelined form. Their performance is measured with respect to a set of criteria which define utilization, throughput and latency for context flow structures. The design of a context flow routing element which can be used to construct pipelined multistage interconnection networks is discussed.

4.1 Performance Criteria

Two properties of context flow have an important influence on the implementability of CF systems — the ability to divide a context into static and dynamic components, and the dependence of the upper bound on queue length solely on the position of a merge node within the graph. In order that the performance of context flow structures may be compared meaningfully both with conventional structures and other CF devices, it is important to establish a set of performance criteria. The performance of CF systems is, however, very different from other more conventional pipelined architectures. If a CF machine contains a single context, the execution time will be greater than on an equivalent conventional pipelined machine — indeed no better than a non-pipelined system. As the number of contexts is increased, the performance also increases, the additional processes being executed during the times when stages would otherwise be idle. Three main features of performance — utilization, throughput and latency — are of interest, together with their relationships to the number of contexts contained in the system.

4.1.1 Graph Loading

A graph is *fully loaded* if the number of contexts contained in the graph is equal to the number of nodes in the graph. When containing fewer contexts, the graph is *under* or *lightly loaded*, and with more contexts, *over* or *heavily loaded.* Overloaded graphs may require contexts to be queued outwith merge nodes in transformation nodes with an associated queue. As maximum queue length in merge nodes is solely dependent on graph structure, and not on the functions performed by nodes, provision of additional queueing capacity in transformation nodes will not effect queue limits. Loading is expressed either as a percentage of the input stream

which is non-null, in the case of acyclic graphs; or as the number of contexts placed initially in a cyclic graph.

4.1.2 Graph Utilization

Utilization is defined as the average number of nodes in the graph which contain a non-null context during a single graph cycle, usually expressed as a percentage of the graph size. The minimum, average and maximum utilizations can be measured. One of the aims of context flow is to provide maximal utilization, but this can only be achieved if the graph is either fully or heavily loaded.

4.1.3 Throughput

Throughput is the average number of contexts which either reach the end of an acyclic graph or pass a given point in a cyclic graph, during one graph period, and is expressed in contexts per cycle. Context flow attempts to provide a throughput of one context per cycle for each active context stream.

4.1.4 Latency

Latency is the average length of time a context takes either to traverse an acyclic graph, or to complete one circuit of a cyclic graph. In overloaded graphs, it is useful to differentiate between two forms of latency — *pipeline latency* which has the same definition as latency above, and *instruction latency* which excludes the additional time a context spends in queues, due simply to the loading of the graph rather than the graph structure. Latency is expressed in cycles per context.

4.2 A CF Arithmetic Unit

The arithmetic unit of a processor, characterized by a regular input of similar instructions, lends itself naturally to a pipelined, and hence context flow, implementation, and would play a central role in any context flow processor. The pipeline described here is intended only as an example to illustrate the construction of context flow systems. A more realistic implementation is given in the design of the context flow processor in Chapter 5.

The design is an example of a dynamically reconfigurable arithmetic unit which performs multiplication, reciprocation, addition and subtraction of fixed and floating point numbers. The unit, shown in Figure 4–1, is composed of three main sections — decoding and floating point pre-processing, execution, and floating point post-processing. During the first stage of processing, instructions are categorized into fixed and floating point operations, the floating point data undergoing exponent subtraction and mantissa alignment before decoding of the function is performed. Multiplication and subtraction both require the context to pass through two nodes to be completed, the first to form the partial products or negative respectively, followed by an addition step. The reciprocation function is provided to support division, which requires a context to circulate at least twice through the unit for evaluation. The results of floating point operations are then normalized and accumulated for use in scalar product or double length multiplication operations.

The simulated performance of the arithmetic unit is shown in Table 4–1. The input to the unit in each case is a stream of contexts with the different instructions occurring in different proportions and distributions over a period of 5,000 cycles. An exhaustive series of input streams determined those which produced

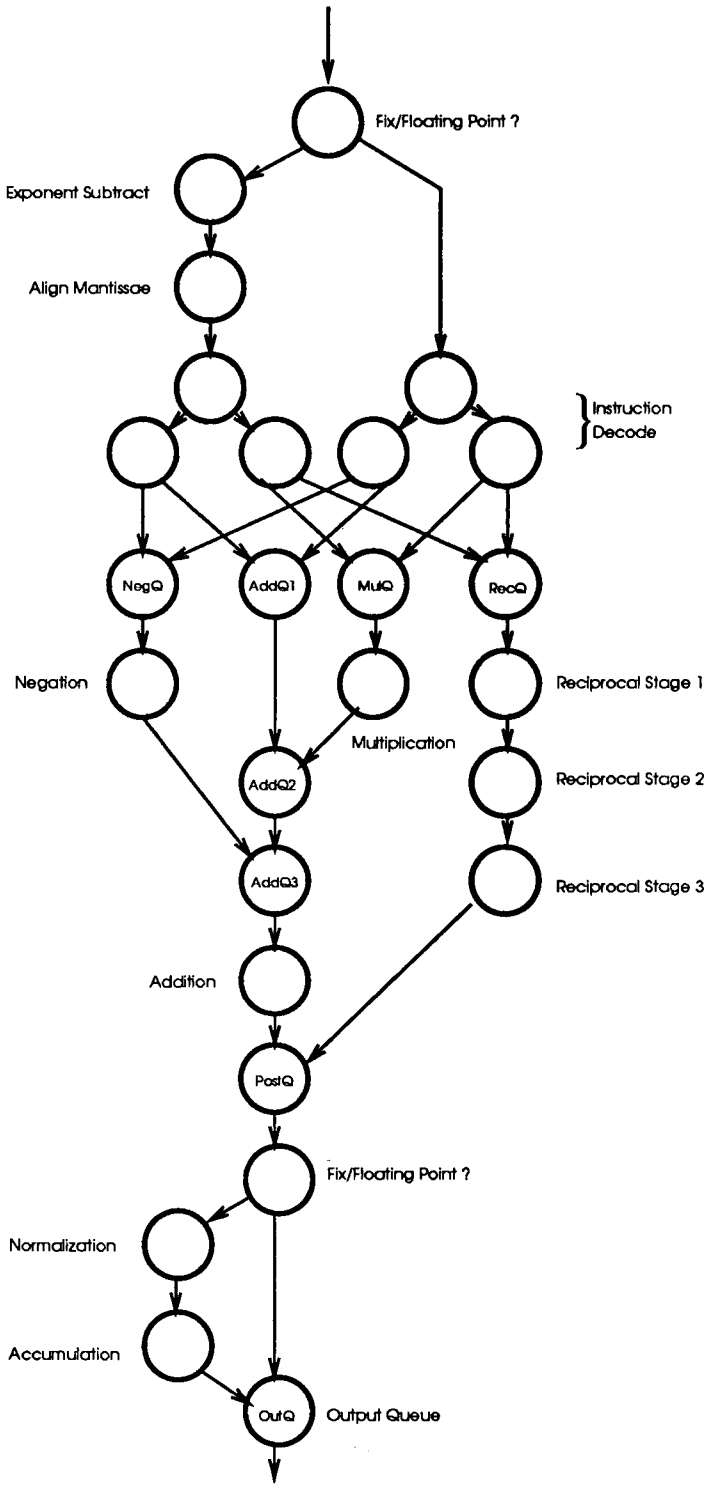


Figure 4–1: Structure of the CF arithmetic unit

Input Stream Composition	Minimum Utilization	Average Utilization	Maximum Utilization	Latency
Constant fixed pt. subtraction	3.85%	38.42%	38.46%	10.00
Constant floating pt. multiplication	3.85%	57.60%	57.69%	15.00
Alternate floating and fixed pt.	3.85%	45.03%	53.85%	14.00
Random uniform	3.85%	44.68%	57.69%	15.00

Table 4–1: Performance of CF arithmetic unit for various input streams

Input Stream Composition	MulQ	RecQ	AddQ1	NegQ	AddQ2	AddQ3	PostQ	OutQ
Constant fixed pt.	—	—	—	—	—	—	—	—
Constant floating pt.	—	—	—	—	—	—	—	—
Alternate	—	—	—	—	0.09	0.16	0.51	0.86
Uniform	0.02	0.02	0.02	0.02	0.09	0.23	0.90	0.76

Table 4–2: Average queue lengths in CF arithmetic unit

the best-case, worst-case, and intermediate performances, and those which induced maximal length queues within the merge nodes.

The best-case performance, an average execution time of 10.00 graph cycles per instruction, occurs with input streams consisting entirely of fixed point addition, subtraction and reciprocation operations. The worst-case performance, an average execution time of 15.00 graph cycles per instruction, occurs with a continuous stream of floating point multiplication instructions. The unit provides a consistent performance when given an input containing a mixture of fixed and floating point instructions. A stream of uniformly-distributed random instructions is executed in an average time of 15.00 graph cycles per instruction. The unit initiates one instruction per cycle and has unit throughput for all input streams.

The average and maximum lengths of queue in each of the merge nodes are given in Tables 4–2 and 4–3. Although the average queue lengths in many of

Input Stream Composition	MulQ	RecQ	AddQ1	NegQ	AddQ2	AddQ3	PostQ	OutQ
Theoretical Maxima	2	2	2	2	3	3	3	2
Constant fixed pt.	—	—	—	—	—	—	—	—
Constant floating pt.	—	—	—	—	—	—	—	—
Alternate	—	—	—	—	1	1	1	2
Uniform	2	2	2	2	2	3	3	2

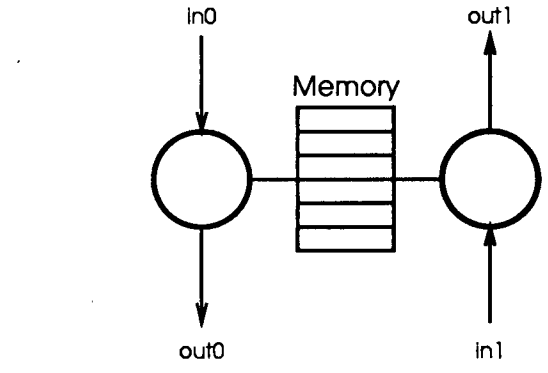
Table 4–3: Maximum queue lengths in CF arithmetic unit

the merge nodes is very short, these results show the maximum lengths to be in agreement with the theoretical limits.

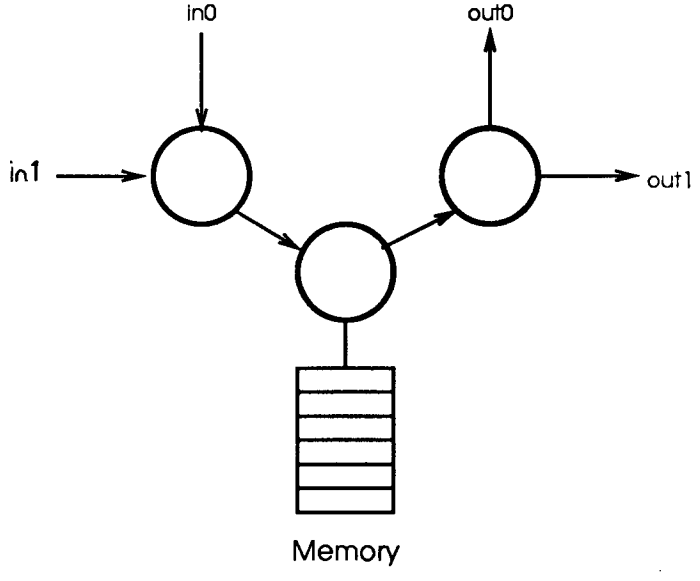
The design presented here is somewhat conservative, in that each distinct operation is assigned to a different node for the purpose of clarity. It would be possible to improve the performance of the design by combining certain functions into single nodes, thus reducing the latency of the unit.

4.3 A CF Shared Memory Unit

There are many instances when two streams of contexts require access to the same physical memory. For example, during instruction execution, values may be read from a register file at one point in the pipeline and results written to the same register file at another. The obvious method of implementing a memory of this type, by connecting it between two transformation nodes, as shown in Figure 4–2(a), is explicitly prohibited in context flow, due to the possibility of introducing unintentional side-effects during simultaneous access to the same location by two contexts. Shared memories can, nonetheless, be implemented in context flow with, in many cases, only a small additional performance penalty. The structure required to implement shared memory is shown in Figure 4–2(b). The unit consists of a merge node to combine the two streams of contexts which require access to the shared



(a)



(b)

Figure 4–2: Structure of the context flow shared memory interface

data; a single transformation node with the shared data stored in an associated memory; and a branch node to regenerate the two original streams after the memory access is complete. This method of shared memory organization has certain advantages — mutual exclusion on shared data is enforced in hardware providing an elegant implementation for semaphores and other synchronization primitives, and the structure is easily extensible to allow access by more than two streams with only the addition of merge nodes at the input and corresponding branch nodes at the output. There are, however, certain disadvantages to a memory structure of this type in that it can only process contexts at half the rate and is therefore a potential source of delay.

Figures 4–3 to 4–5 show the simulated performance of the shared memory unit under varying loads over a period of 2500 graph cycles. Changes in latency, measured as the average number of graph cycles each context requires to pass through the unit, are shown in Figure 4–3. Latency is almost constant at about 3 cycles for loads under 50%, after which it rises steeply to over 600 cycles for a 100% load. Similar behaviour is observed in maximum queue length, which remains almost constant at loads below 40% and rises rapidly with loads above 50%, as shown in Figure 4–4. Even at low levels of loading, a high degree of utilization, measured as the proportion of nodes that process non-null contexts, is achieved, as shown in Figure 4–5.

4.4 A CF Network Routing Element

In any multiprocessor system which includes a communications network linking either several processing elements or processing elements and memory modules, the performance of the switches which route information through the network has an important bearing on the overall performance of the system. The basic component of a multistage interconnection network is a 2×2 routing element, which can be

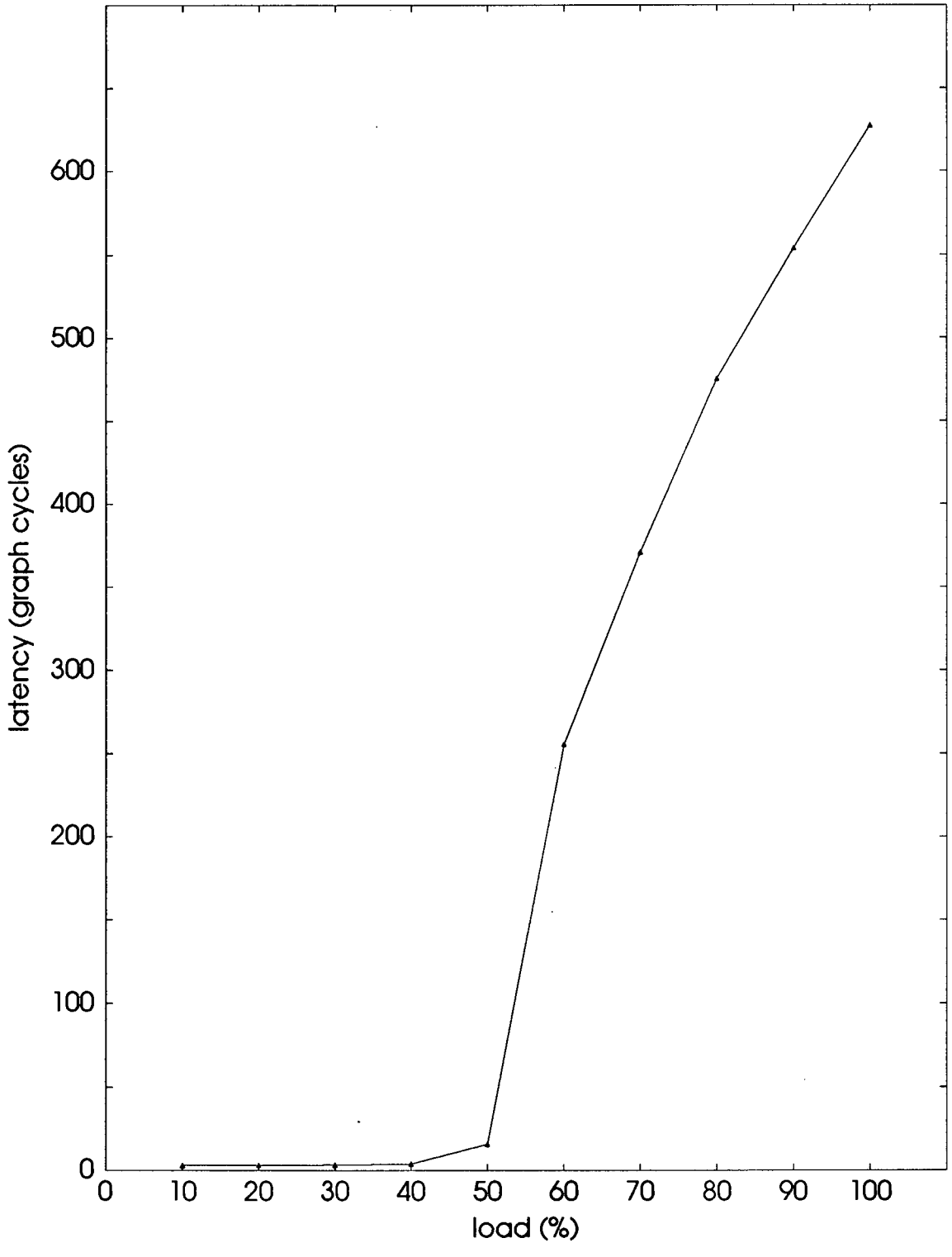


Figure 4–3: Latency v. load for CF shared memory interface

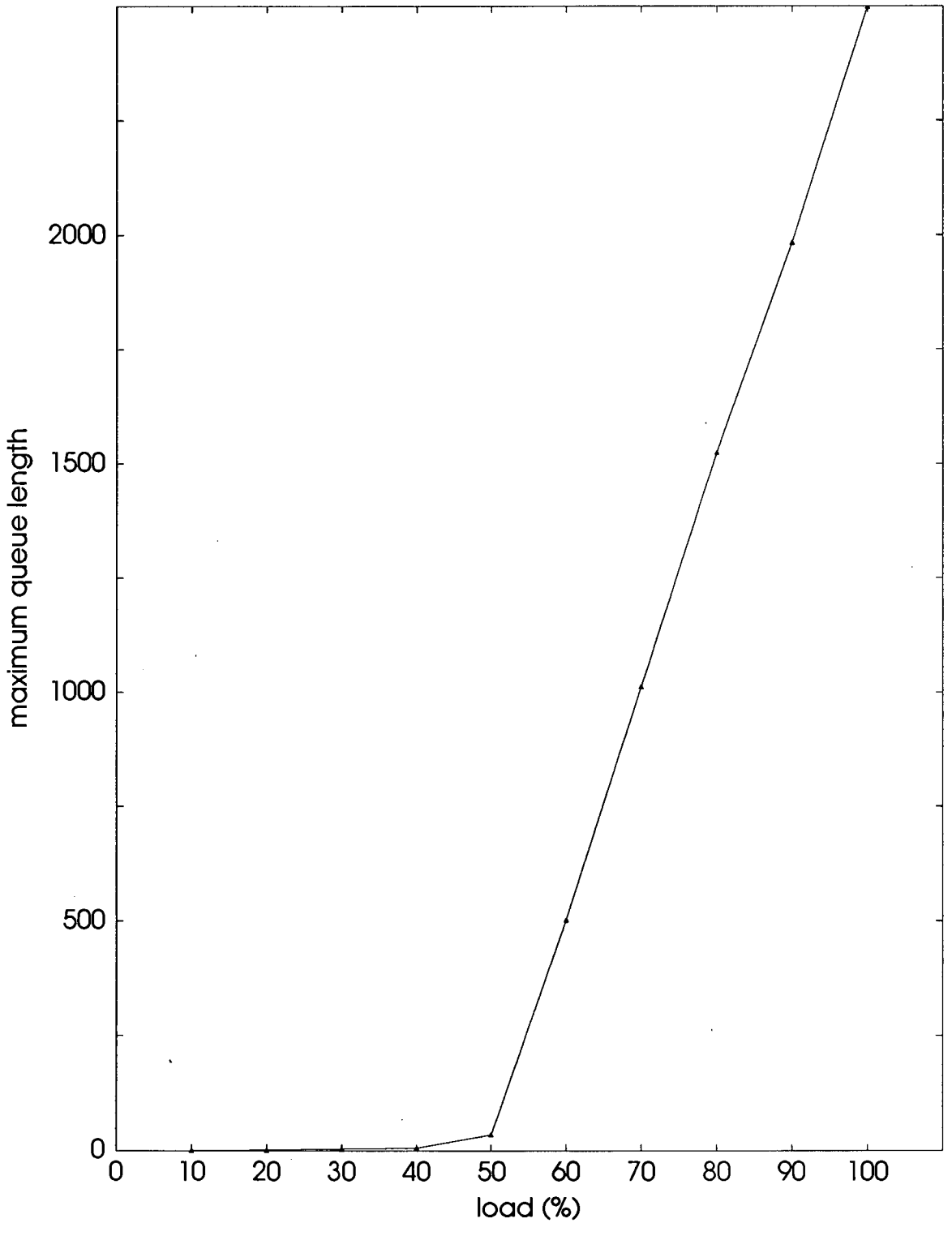


Figure 4–4: Maximum queue length v. load for CF shared memory interface

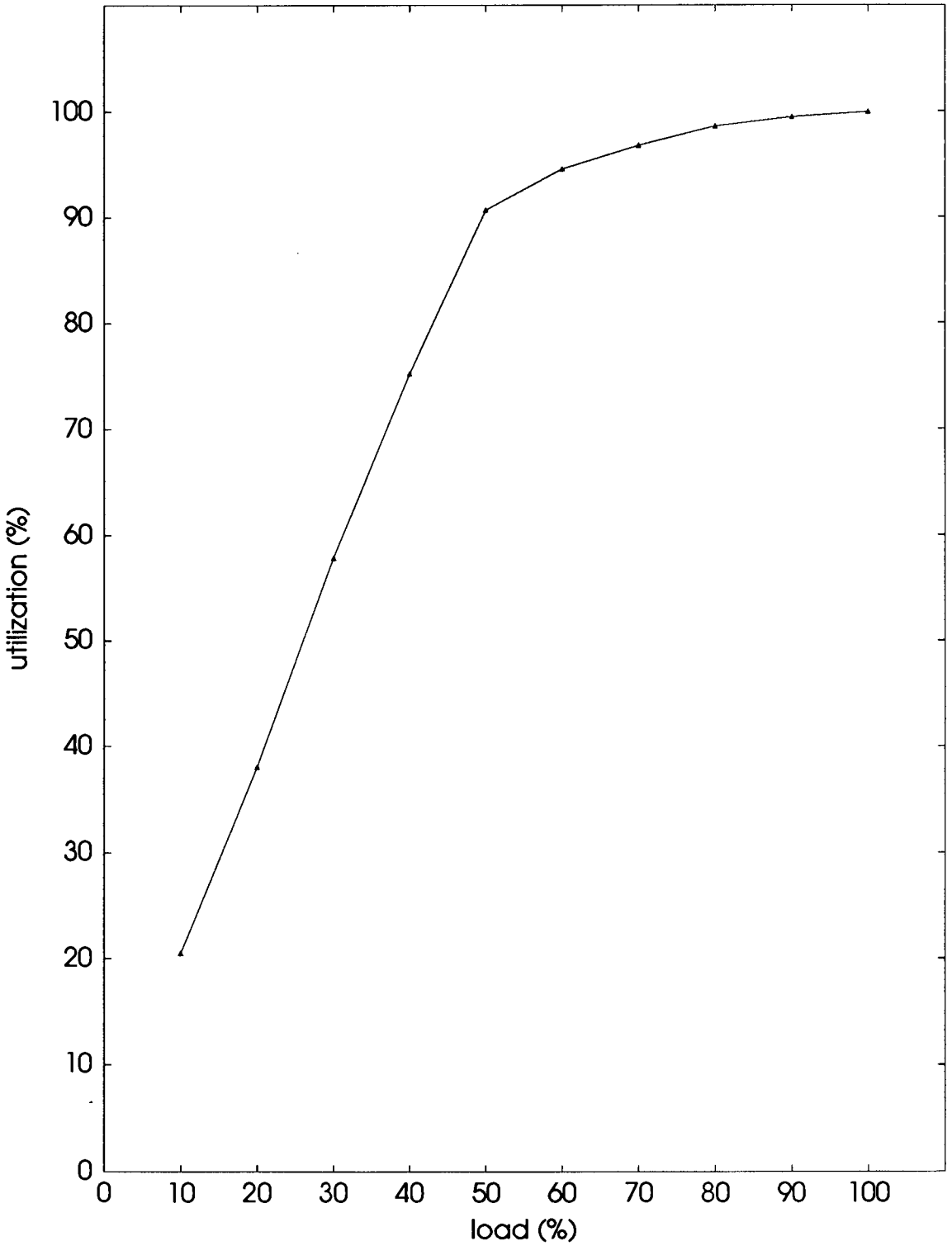


Figure 4–5: Utilization v. load for CF shared memory interface

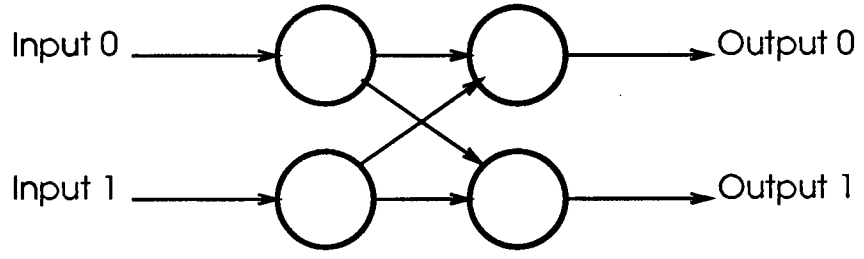


Figure 4–6: Structure of the CF routing element

used to construct a number of different network topologies implementing a variety of permutations.

The context flow implementation of a 2 × 2 routing element consists of two branch and two merge nodes, as shown in Figure 4–6. Four routing functions are implemented by this element

- *straight-through* — routing input 0 to output 0 and input 1 to output 1,
- *exchange* — routing input 0 to output 1 and input 1 to output 0,
- *combine 0* — routing both input 0 and input 1 to output 0, and
- *combine 1* — routing both input 0 and input 1 to output 1.

The router accepts two contexts per cycle on its inputs and outputs two contexts per cycle. In the case where one of the combining operations is required and two non-null contexts are input, the internal operations of the appropriate merge node ensure that both contexts are routed correctly.

The performance of the 2 × 2 router is given in Tables 4–4 to 4–6. The input streams both consist of 5,000 contexts with varying destination distributions. Modelling this element in isolation is complicated by the fact that an upper bound for the queue lengths cannot be determined, as it is dependent on the structure

Input Stream Composition (Constant routing functions)	Average Utilization	Latency	Throughput
straight-through	99.94%	2.00	2.00
exchange	99.94%	2.00	2.00
combine	37.49%	2502.00	1.00
combine, alternate null contexts	49.97%	2.00	1.00

Table 4–4: Performance of CF routing element for constant routing functions

Input Stream Composition (Random destinations)	load	Average Utilization	Latency	Throughput
50% → 0, 50% → 1	100%	90.98%	29.41	1.99
60% → 0, 40% → 1	100%	90.02%	116.78	1.90
70% → 0, 30% → 1	100%	88.29%	244.34	1.80
50% → 0, 50% → 1,	95%	87.60%	7.71	1.90
50% → 0, 50% → 1,	75%	70.52%	3.78	1.51
50% → 0, 50% → 1,	50%	47.78%	3.26	1.00

Table 4–5: Performance of CF routing element for random routing functions

Input Stream Composition (Random length bursts)	Load	Maximum Burst Length	Average Utilization	Latency	Throughput
50% → 0, 50% → 1,	100%	10	91.43%	88.23	1.95
	100%	100	88.63%	205.64	1.86
50% → 0, 50% → 1,	95%	10	86.79%	35.77	1.87
	95%	100	88.27%	70.53	1.88
50% → 0, 50% → 1,	50%	10	47.99%	3.85	1.00
	50%	100	47.92%	5.11	1.00

Table 4–6: Performance of CF routing element for random length sequences of requests for the same routing function

of the graph which generates the input streams. The routing element, and any network constructed using these elements, should be considered as part of a system which contains a fixed number of contexts. The input streams used to test the model of the routers can be taken as representing the heaviest load liable to be placed on an element.

When implementing a constant routing function the router can be very efficient, as shown in Table 4–4, forwarding each context in a constant time of two cycles and yielding two correctly routed contexts per cycle for both the *straight-through* and *exchange* functions. The *combine* functions produce the poorest performance, with an $O(l)$ average routing time and a unity completion rate for an input stream containing l contexts.

The routing element performs well when given streams containing randomized destinations, as shown in Table 4–5. If the destinations are uniformly distributed, contexts are routed in an average of 29.41 cycles with an almost maximal throughput. If the destinations are biased in favour of one of the outputs such that 60% of contexts are directed there and 40% to the other, the average routing time rises dramatically to 116.78 cycles and the throughput drops to 1.90 contexts per cycle. This attenuation in performance is even more marked if the distribution is changed to route 70% of contexts to one of the outputs.

All the above results are obtained with continuous streams of non-null contexts, a situation which is liable to occur relatively infrequently. Introducing null contexts into the input streams improves the performance of the routing element. If the load on the routing element is reduced by as little as 5%, latency is reduced by nearly a factor of four, from 29.41 cycles to 7.71 cycles per context; whilst the throughput is maintained. If the load is further reduced to 75%, latency drops to 3.26 cycles per context.

Many programs exhibit considerable locality, in that several successive memory references may access the same location. Input streams representing this situation,

containing random length sequences of contexts with the same destination, result in a performance consistent with other distributions — the inclusion of null contexts producing a large improvement compared to continuous non-null sequences, as shown in Table 4–6. All permutations of input stream yield a completion rate of at least one context per cycle. The maximum and average queue lengths for the routing element are given in Tables 4–7 to 4–9.

Using the CF routing element, a wide variety of pipelined interconnection networks can be constructed. The only change to the design of the element is the addition of a transformation node at each of the outputs to enable routing at the next stage to be performed using the same branch condition. This allows a network to be created by replication of a single type of routing element.

Two topologies which can both establish arbitrary connective paths between any of the inputs and outputs are the *omega* [Lawrie, 1975] and the *binary* n-*Cube* [Pease, 1977] networks, as shown in Figures 4–7(a) and (b) respectively. Figures 4–8 to 4–10 show the simulated latency and queue lengths of CF implementations of these networks connecting eight inputs to eight outputs for various network loads. The load imposed on a CF interconnection network is the proportion of non-null contexts presented as input. The graph in Figure 4–8 shows the change in latency for given network loads, for an input of 2500 contexts with uniformly distributed random destinations. As in the case of the routing element itself, inclusion of a small percentage of null contexts results in a considerable fall in latency. The same is true for maximum queue length, shown in the graph of Figure 4–9, with average queue length showing a similar, though less marked, decline, as shown in the graph of Figure 4–10 Throughput, the number of correctly routed contexts output per cycle, is directly proportional to the load imposed on the network.

Figures 4–11 to 4–14 show latency, throughput and queue length in the presence of *hotspots*, for a network load of 100%. A hotspot occurs when an additional percentage of input contexts are directed towards the same "hot" output. Fig-

Input Stream Composition (Constant routing functions)	Output 0		Output 1	
	average	maximum	average	maximum
straight-through	0	0	0	0
exchange	0	0	0	0
combine	1.00	5000	0	0
combine, alternate null contexts	0	0	0	0

Table 4–7: Queue lengths in CF routing element for constant routing functions

Input Stream Composition (Random Destinations)	Load	Output 0		Output 1	
		average	maximum	average	maximum
50% → 0, 50% → 1	100%	0.98	53	0.99	65
60% → 0, 40% → 1	100%	1.09	480	0.67	17
70% → 0, 30% → 1	100%	1.20	1007	0.42	7
50% → 0, 50% → 1	95%	0.81	15	0.86	23
50% → 0, 50% → 1	75%	0.36	6	0.37	6

Table 4–8: Queue lengths in CF routing element for random routing functions

Input Stream Composition (Random Length Bursts)	Load	Output 0		Output 1	
		average	maximum	average	maximum
50% → 0, 50% → 1	100%	0.99	200	0.89	230
	100%	0.98	600	0.76	700
50% → 0, 50% → 1	95%	0.92	137	0.80	80
	95%	0.73	200	0.77	146
50% → 0, 50% → 1	50%	0.21	10	0.18	9
	50%	0.23	13	0.23	15

Table 4–9: Queue lengths in CF routing element for random length sequences of requests for the same routing function

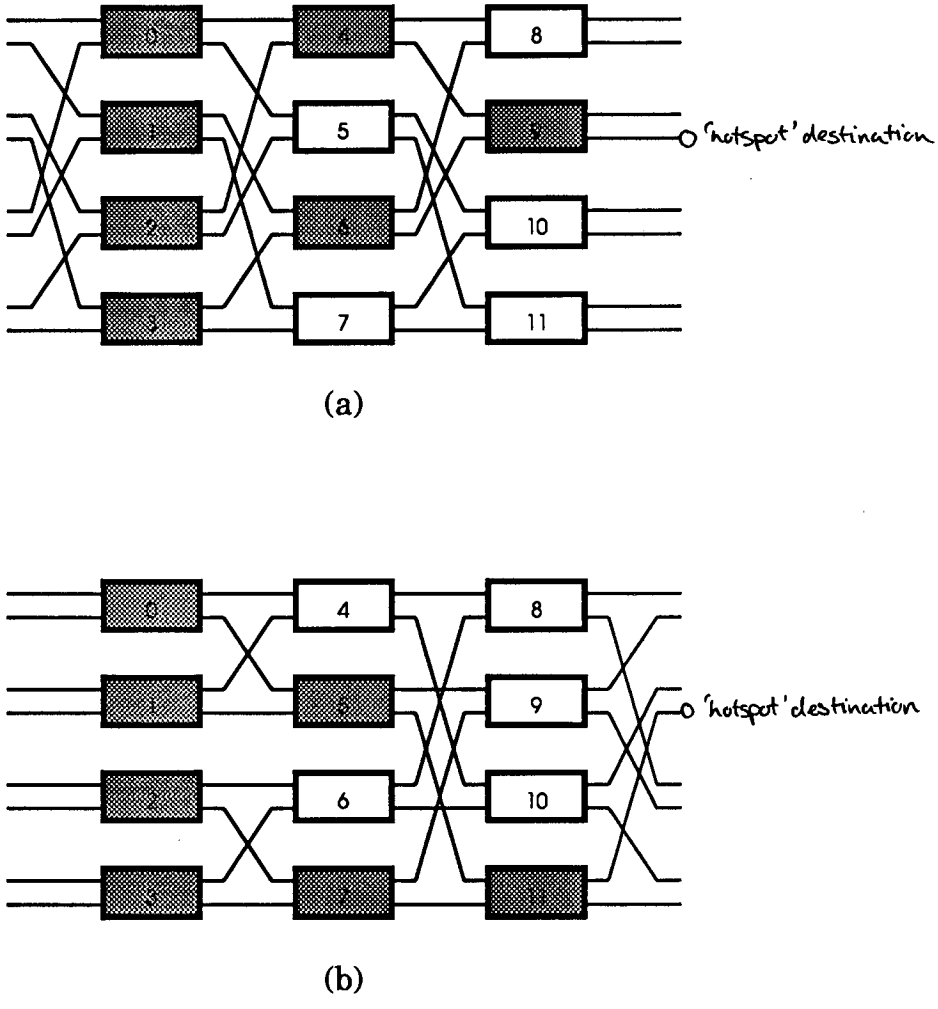


Figure 4–7: Topology of Omega and binary n-Cube networks showing locations of hotspots

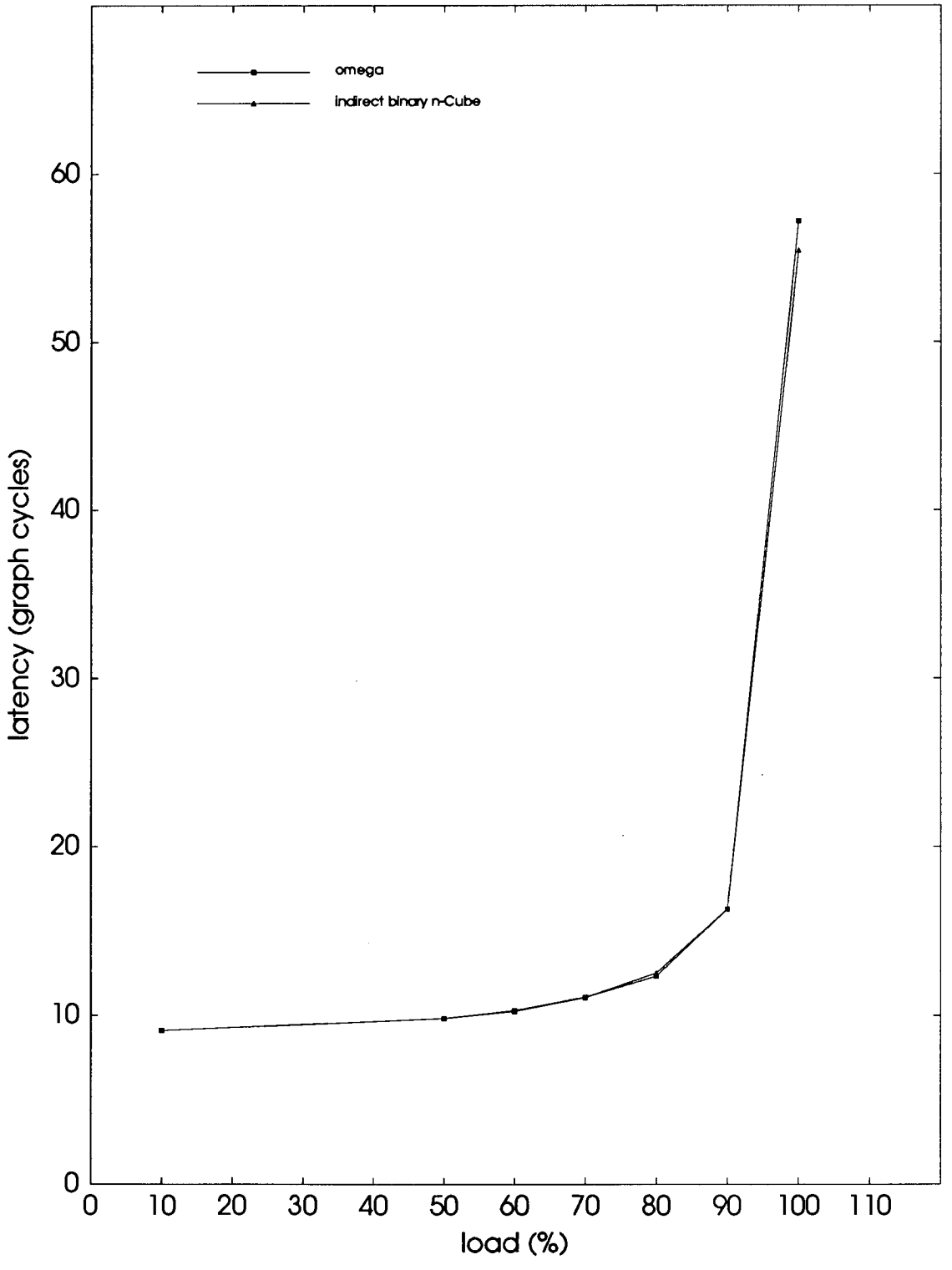


Figure 4–8: Latency v. load for CF Omega and binary n-Cube networks.

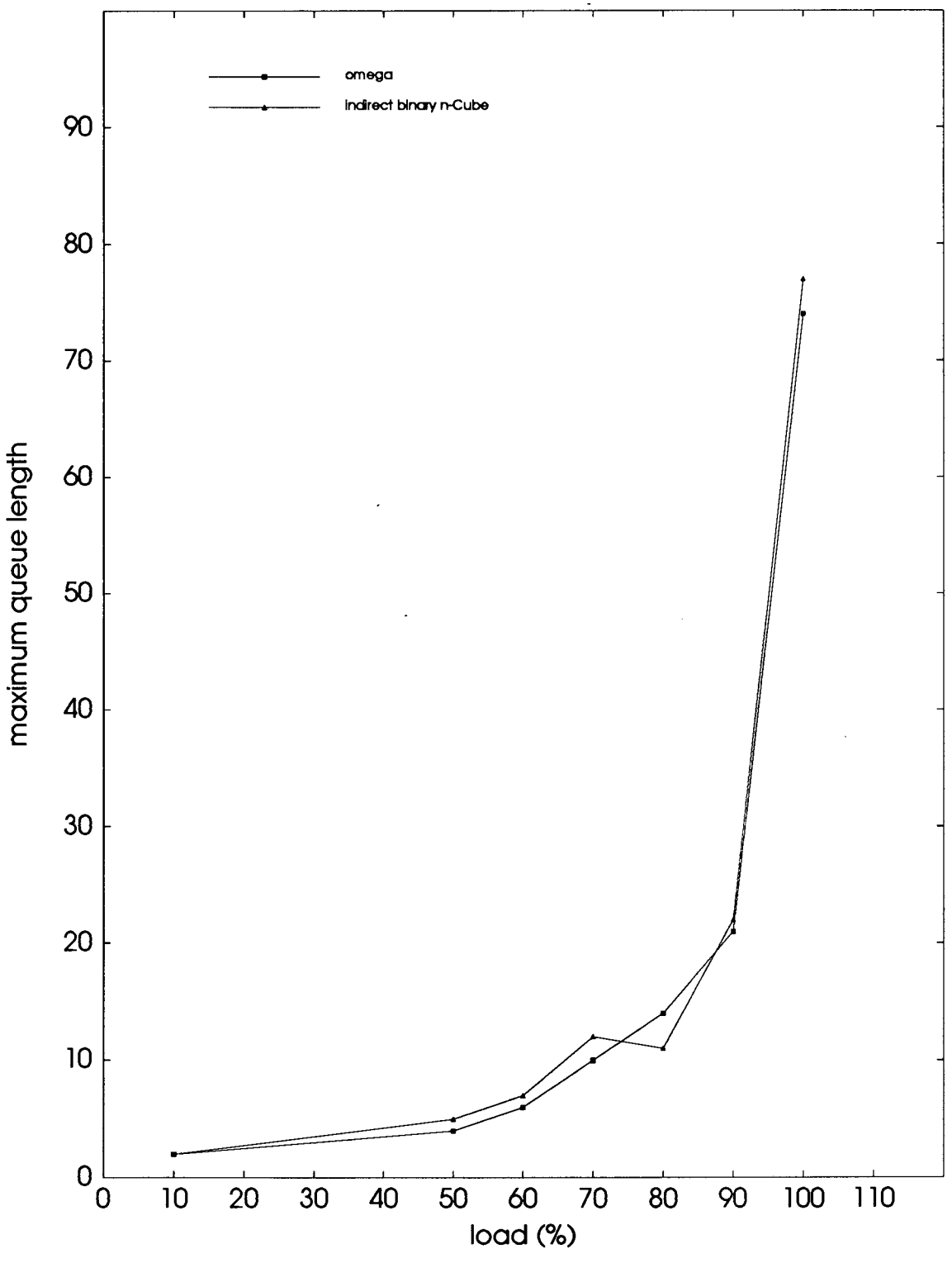


Figure 4–9: Maximum queue length v. load for CF Omega and binary n-Cube networks.

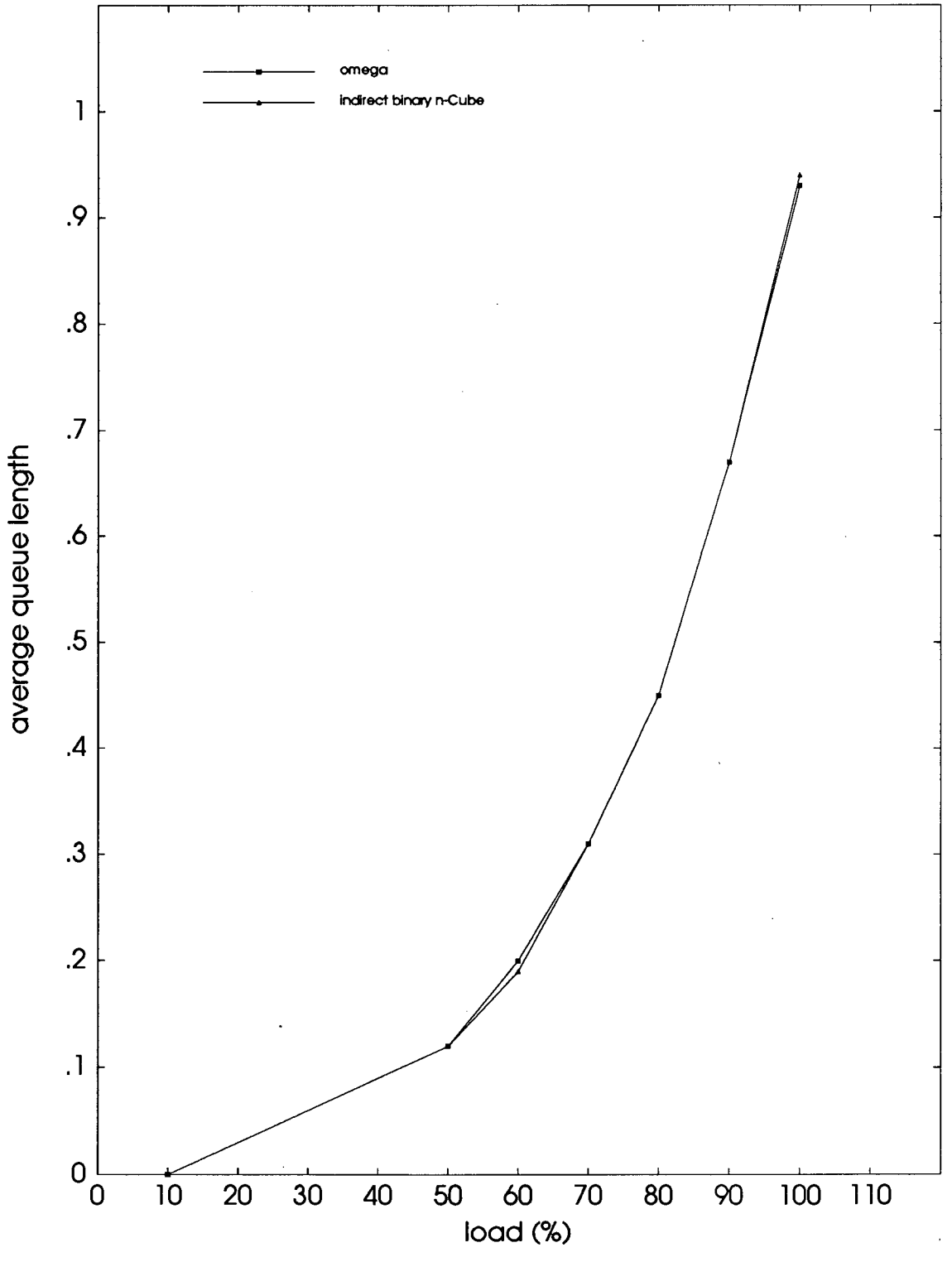


Figure 4–10: Average queue length v. load for CF Omega and binary n-Cube networks.

ure 4–11 shows the change in latency with increasing hotspot percentages. There is little or no change in latency below 1%, but above this, latency rises rapidly, reaching nearly 1100 cycles when all contexts are directed to the same destination.

Similarly, throughput is almost unchanged at a value close to the maximum of 8 contexts per cycle with hotspot occurrences of under 1%, declining towards a minimum of 1 context per cycle for a 100% hotspot, as shown in Figure 4–12.

Maximum queue lengths in both networks show a similar trend, remaining almost constant below 1% and rising rapidly thereafter. The shaded nodes in Figure 4–7 indicate the locations of rapid growth in queue length. Figure 4–13 shows the maximum queue lengths for four of the elements in the Omega network, each of which lies on the path to the hot output, together with the mean maximum length of the queues. Figure 4–14 shows the maximum queue lengths in equivalent positions in the binary n-Cube network. Both networks absorb hotspots of up to 1% with little or no degradation in performance.

Summary

Several features of the context flow model make the process of designing CF systems easier than that for conventional pipelined or concurrent systems. The interleaved nature of the process contexts eliminates all dependencies between instructions in the pipeline. The need for additional hardware or software to mitigate the effects of a conditional branch instruction is thus removed, as the condition will always have been evaluated before execution of the instruction commences. The need for data dependency checking in hardware, as provided by scoreboard devices, is also removed, since no dependencies can exist between instructions in a partial state of execution. Interaction between concurrent processes via shared variables is limited as all memory is associated with single transformation nodes, which may only contain one context during each graph cycle. Simultaneous access

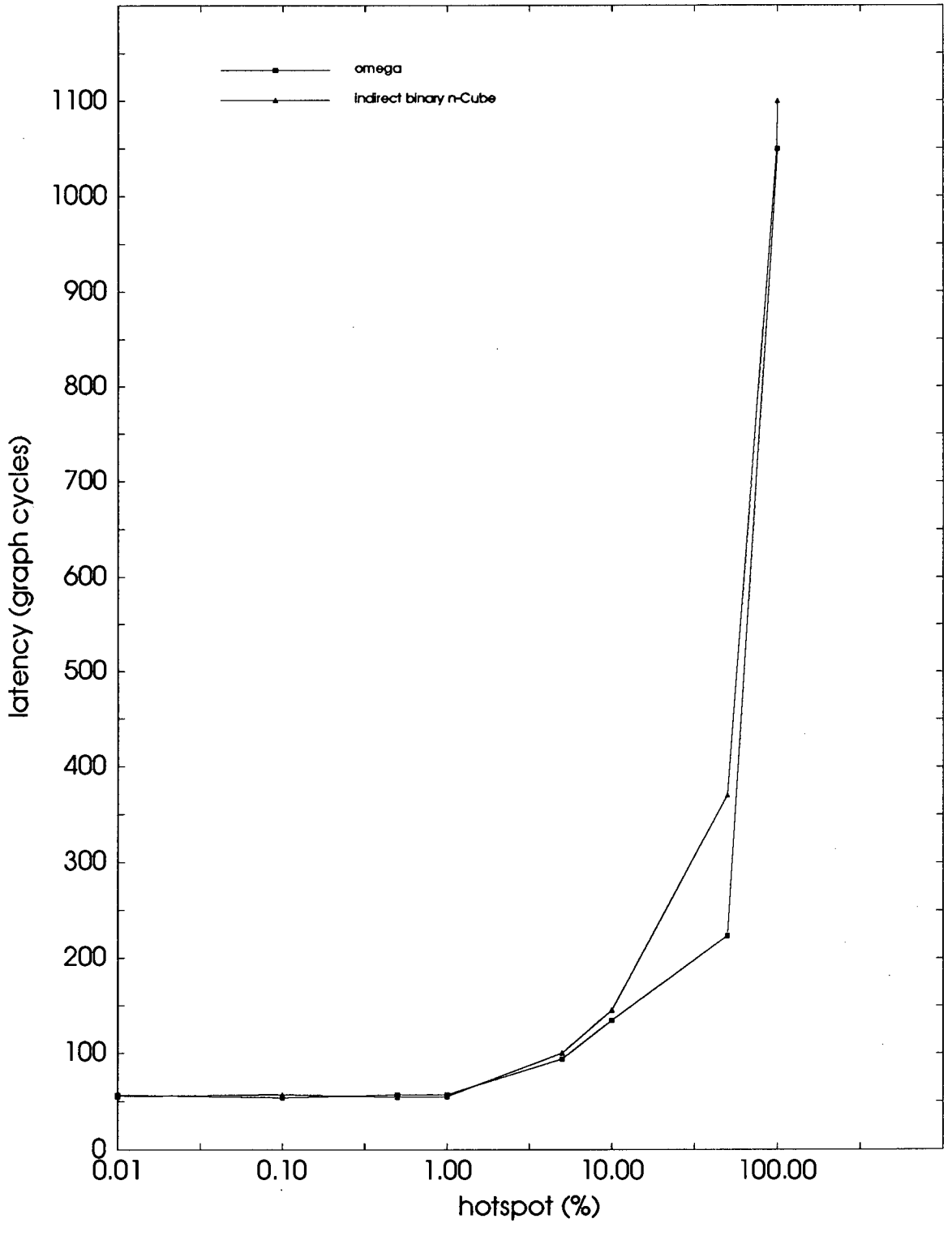


Figure 4–11: Latency v. hotspot occurrence for CF Omega and binary n-Cube networks with 100% load.

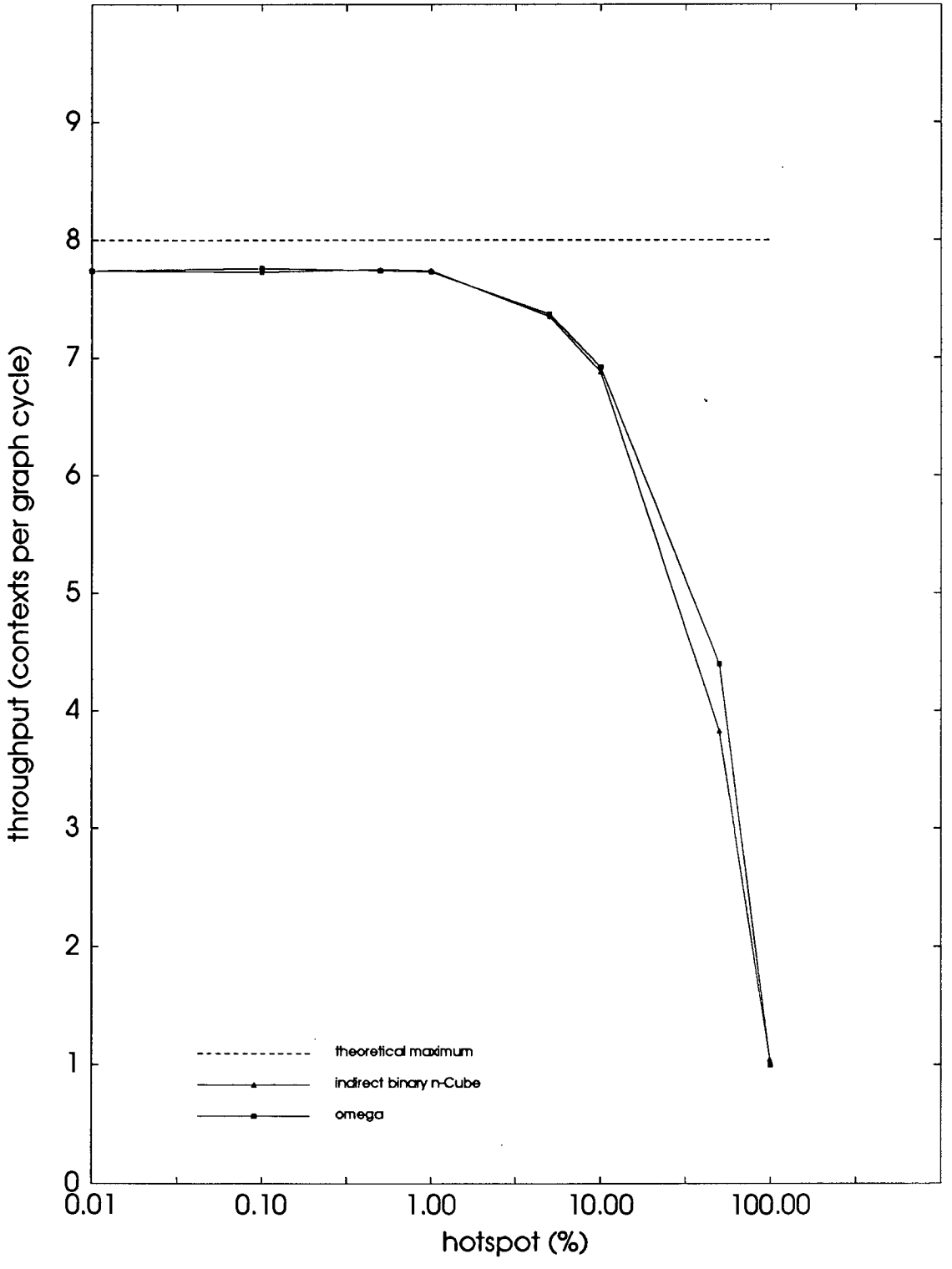


Figure 4–12: Throughput v. hotspot occurrence for CF Omega and binary n-Cube networks with 100% load.

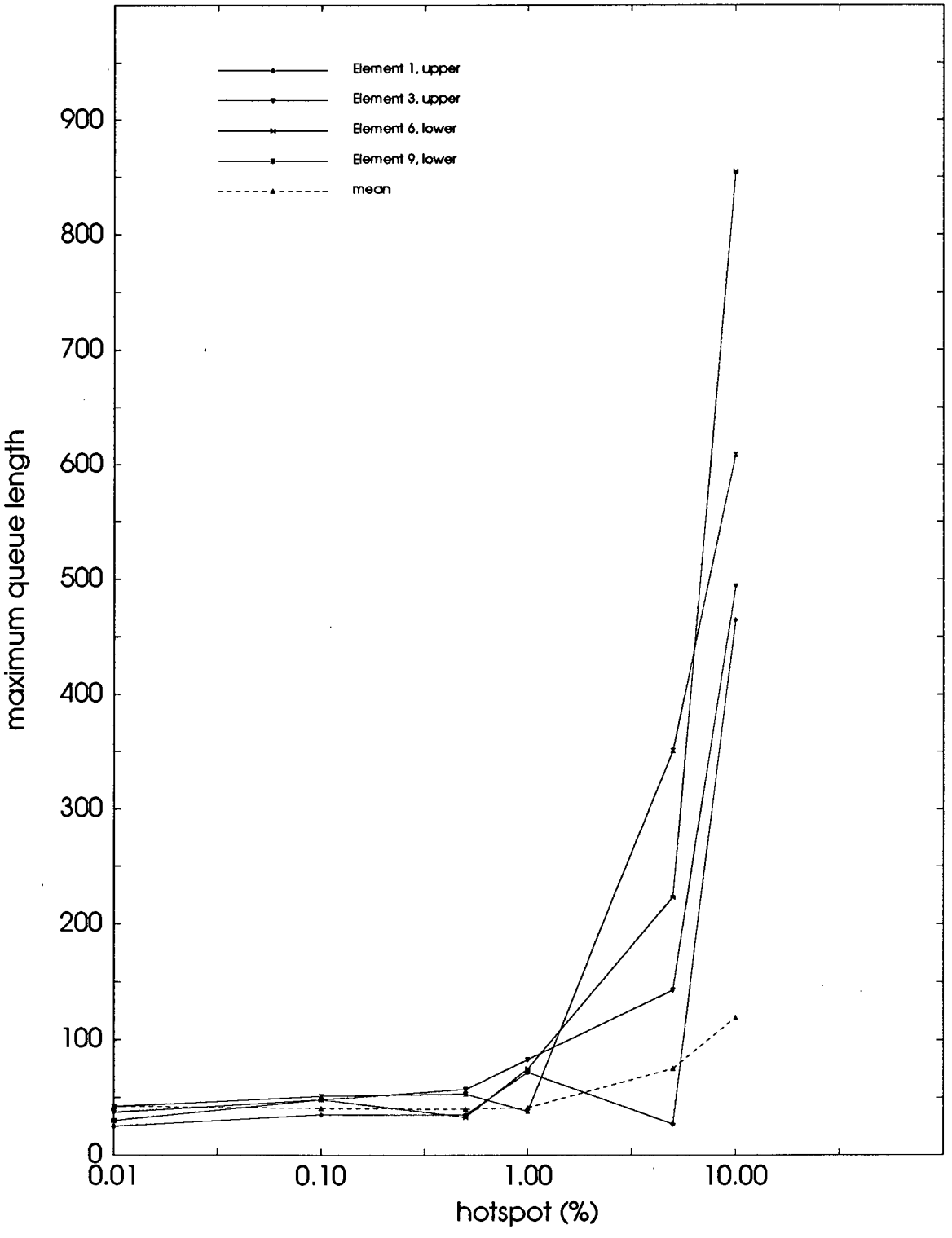


Figure 4–13: Maximum queue length v. hotspot occurrence for CF Omega network with 100% load.

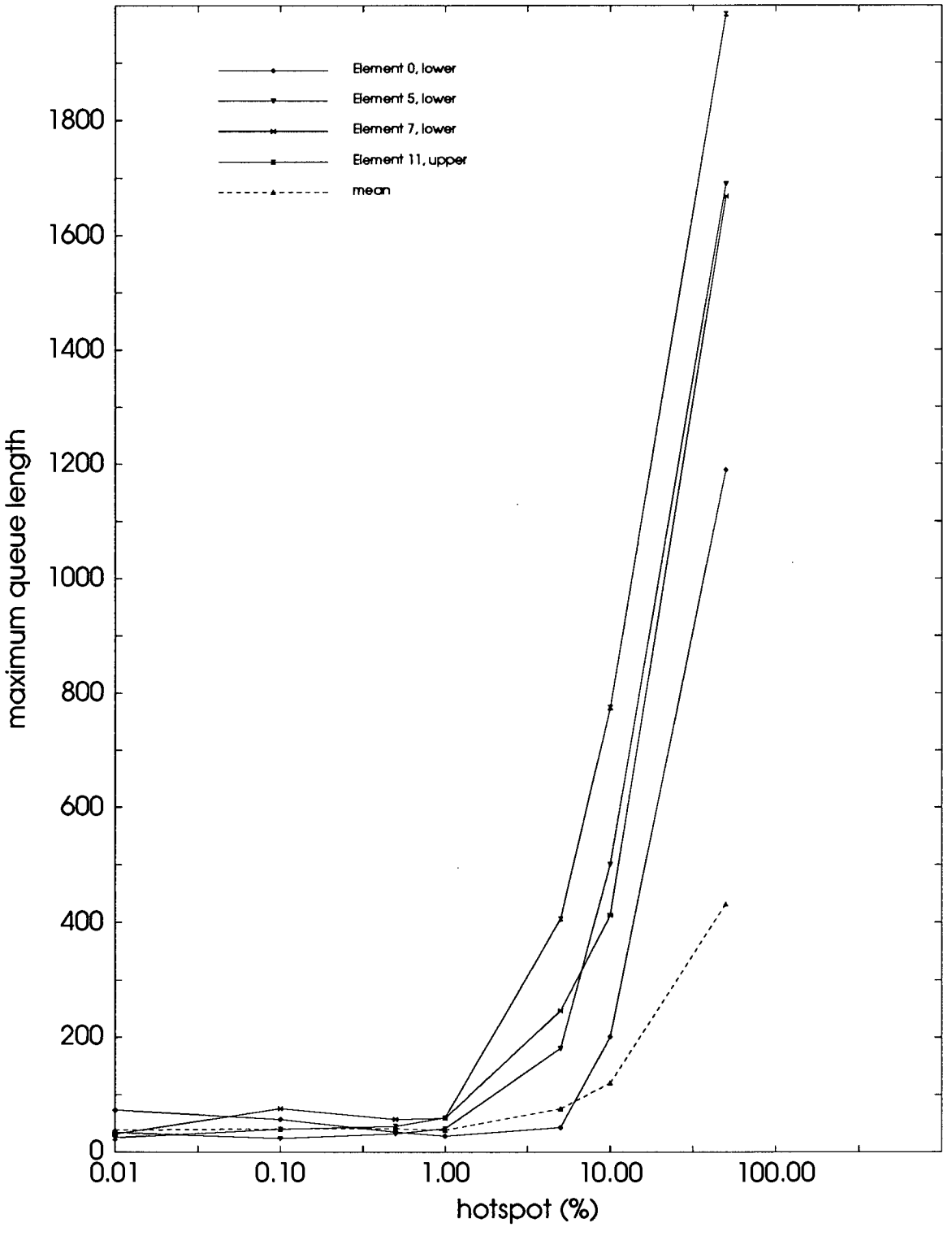


Figure 4–14: Maximum queue length v. hotspot occurrence for CF binary n-Cube network with 100% load.

to common memory locations by multiple contexts is therefore made impossible — as transformation node have a single input and output — thus removing a major source of possible unintentional side-effects.

The formal definition of the temporal properties of the core CF nodes allows graphs containing these nodes to be constructed which exhibit consistent behaviour. This is due to the synchronous nature of the core nodes and the use of null contexts to maintain a regular flow through the graph.

In general, CF structures are formed by expressing the desired functionality in terms of transformation nodes, connected by branch and merge nodes which encode alternative functional sequences.

"In Architecture *as in other* Operative *Arts, the* end *must dictate the* Operation. *The* end *is to build well."*

— HENRY WOTTON (1568–1639), *Elements of Architecture Part I*

Chapter 5

A Context Flow Processor

Context flow implementations of several architectural elements have been presented, and their performance evaluated. The design of a processor provides an opportunity to demonstrate how larger CF structures can be assembled from these simple elements. This chapter outlines the design objectives for a simple, yet quite functional example context flow processor, where traditional architectural criteria are initially tempered by the operational constraints of the context flow model. An instruction set for the processor is presented, together with the operation and structure of the main functional elements. An analysis of the processor performance and of the interaction between streams of contexts in the graph identifies several opportunities for improvement of the original design to allow the processor to meet its original operational goals. The result is a processor capable of sustaining a throughput of one instruction per cycle, with a constant instruction execution time.

5.1 Design Objectives

The context flow processor described in this Chapter is intended to provide high performance for a wide variety of applications by maximal exploitation of hardware resources. The essence of context flow is the formation of highly-pipelined structures. This creates a need for a simple instruction set in which each operation is capable either of being performed in a single cycle, or of being pipelined to allow completion of an operation every cycle. The highly-pipelined nature of the architecture creates the ideal environment for memory to be absorbed into the execution pipeline, thus preventing memory accesses from having an attenuative effect on processor performance. In addition, expansion from a single local memory to a distributed memory, shared among a group of processors by means of an interconnection network, merely lengthens the processor pipeline, without modifying the processor model. This allows an increase in the number of active contexts to compensate for the longer execution time for each instruction due to the greater memory latency. Each contexts accesses a separate register and address space, with seclusion enforced by the architecture. The context flow model, however, makes safe and side-effect free inter-process communication possible by merging streams of contexts requiring concurrent access before the memory unit, as shown in the example of Section 4.3.

The aims of the design are to provide a pipelined processor architecture, capable of sustaining an instruction completion rate of one floating-point, scalar or control operation per clock cycle, for up to 64[1] concurrent processes. It should be noted that the object of this example is to explore some of the pertinent features of context flow system design.

---

[1] Context 0 is used as a null context, thereby reducing the number of possible active user processes to 63.

5.2 Instruction Set

The instruction set for the context flow processor is relatively small, providing a few elemental instructions which may be combined in sequence during compilation to perform functions implemented by single instructions in more complex instruction sets. The instruction set chosen for the CF processor is based on that of the Motorola 88000 processor [Motorola Inc., 1988]. This particular instruction set was chosen for its provision of a core instruction set and the availability of detailed documentation of instructions and processor architecture.

A fixed instruction format simplifies the process of decoding instructions, yielding a reduction in complexity of the circuit required to perform this function. In conjunction with this, the provision of only two addressing modes — *inherent*, specifying that the operands are either constants or data to be extracted from the instruction itself; and *extended*, specifying that operands reside in registers — facilitates fast and easy implementation. A consequence of limiting the number of addressing modes is that all operations are performed on the contents of registers (or on inherent data), with only load and store instructions accessing data in memory.

5.2.1 Instruction Format

As a result of the above decisions, the instruction set of the CF processor can be implemented with 16-bit instructions. The instructions are divided into four groups — *integer*, *floating-point*, *logical* and *control*, with each group containing up to eight instructions. Table 5–1 presents the complete instruction set [1].

Integer Instructions		Control Instructions	
Mnemonic	Operation	Mnemonic	Operation
add	addition	ld	load from memory
addu	unsigned addition	st	store to memory
cmp	compare	mv	move to register
rec	reciprocation	bra	unconditional branch
recu	unsigned reciprocation	bsr	branch to subroutine
mul	multiplication	jmp	unconditional jump
sub	subtraction	jsr	jump to subroutine
subu	unsigned subtraction	bcnd	conditional branch
Floating-Point Instructions		Logical Instructions	
Mnemonic	Operation	Mnemonic	Operation
fadd	addition	and	bitwise and
fcmp	compare	or	bitwise inclusive-or
frec	reciprocation	not	bitwise inversion
fmul	multiplication	xor	bitwise exclusive-or
fsub	subtraction	rot	rotation
inf	convert to FP		
fin	convert to integer		

Table 5–1: CF Processor Instruction Set

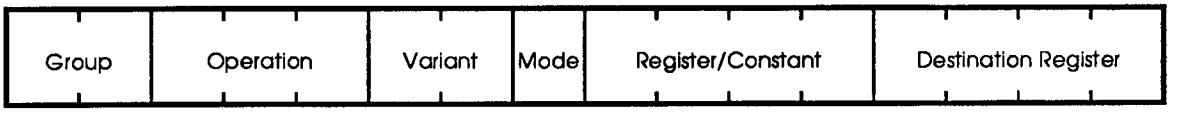


Figure 5–1: Format of CF processor instructions

The instructions are provided in a *two-address* form, with one of the source registers acting as the destination. This is in contrast with the actual Motorola 88000 processor which implements *triadic* register addressing, allowing specification of two sources and a separate destination register. Allocating four bits of each instruction to denote the source registers leaves three bits to describe the addressing mode and other variants of each instruction. Although four bits are sufficient to identify sixteen registers, the nature of the operand, either scalar or floating-point, can be used implicitly to select between banks of sixteen separate scalar and floating-point registers, providing thirty-two general purpose registers for each process. Constant operands are stored in thirty-two read-only scalar and floating-point constant registers, shared between processes. The format of the instructions is shown in Figure 5–1.

Integer Arithmetic Operations

Instructions are provided to perform addition, subtraction, multiplication and reciprocation on integer quantities. With the exception of multiplication, both signed and unsigned forms of the instruction are provided. Separate instructions

[1]Instructions providing support for separate user and supervisor modes, and for handling exceptions and interrupts are not included. These would, however, be required to implement operating system and I/O functions in a processor implementation.

are not provided to perform addition with carry and subtraction with borrow. These functions are encoded in the variant field of the core instruction. This assists in limiting the number of separate instructions thus easing the decoding process. The comparison instruction performs a subtraction of its operands, but does not store the result, merely setting the appropriate flags in the status register. Addition, subtraction and comparison operations are each performed in a single cycle, while multiplication and reciprocation each require three cycles.

Floating-Point Arithmetic Operations

The above arithmetic operations are also provided in floating-point form. Two instructions are provided to convert numbers between integer and floating-point formats. The register fields in floating-point instructions specify floating point registers implicitly as the sources for all operations. This prevents integer operands being used in floating-point operations and vice versa, except after explicit conversion. Double precision quantities are beyond the scope of this example and are not considered further. In any implementation, the IEEE standard 754 for floating point would be used.

Logical Operations

A set of logical operations perform bitwise boolean operations — not, and, inclusive and exclusive or — on the contents of scalar registers. The rotation operation performs a left rotation of the contents of a scalar register, the number of places to be shifted being specified as a 5-bit immediate operand in the instruction. This is another example of instruction set economy, as many more complex instruction sets provide separate instructions to perform rolls, logical shifts and arithmetic shifts in both directions. These can all be accomplished using rotation in conjunction with setting or clearing of appropriate bits.

Control Operations

The only instructions which explicitly reference memory are the load and store operations. Again, the variant field is used to specify the type of the operand, scalar or floating-point, to be referenced. Transfer of data between registers of the same type is performed by the move instruction. The two unconditional control transfer instructions perform the same function, but use their operand from different sources, the unconditional branch using an 11-bit immediate offset, and the unconditional jump, the contents of a register. This is also the case for the branch and jump to subroutine instructions. The conditional branch uses status information from the execution of the previous instruction to determine whether the branch is taken.

5.3 Processor Architecture

The architecture of Rogers and Topham [1990] is used as a starting point for the CF processor design. The context flow processor architecture is divided into three main parts — the execution, control and memory pipelines. The execution pipeline contains the instruction decoding unit, the register file, and the ALU; the control pipeline contains status registers and program counters; and the memory pipeline contains either the memory unit or a network interface, depending on the application of the processor. The overall architecture of the processor is shown in the context flow graph of Figure 5–2.

5.3.1 Execution Pipeline

The execution pipeline consists of three main units — the *thread issue unit* (TIU), the register file (RF) and the arithmetic unit (ALU). The execution pipeline per-

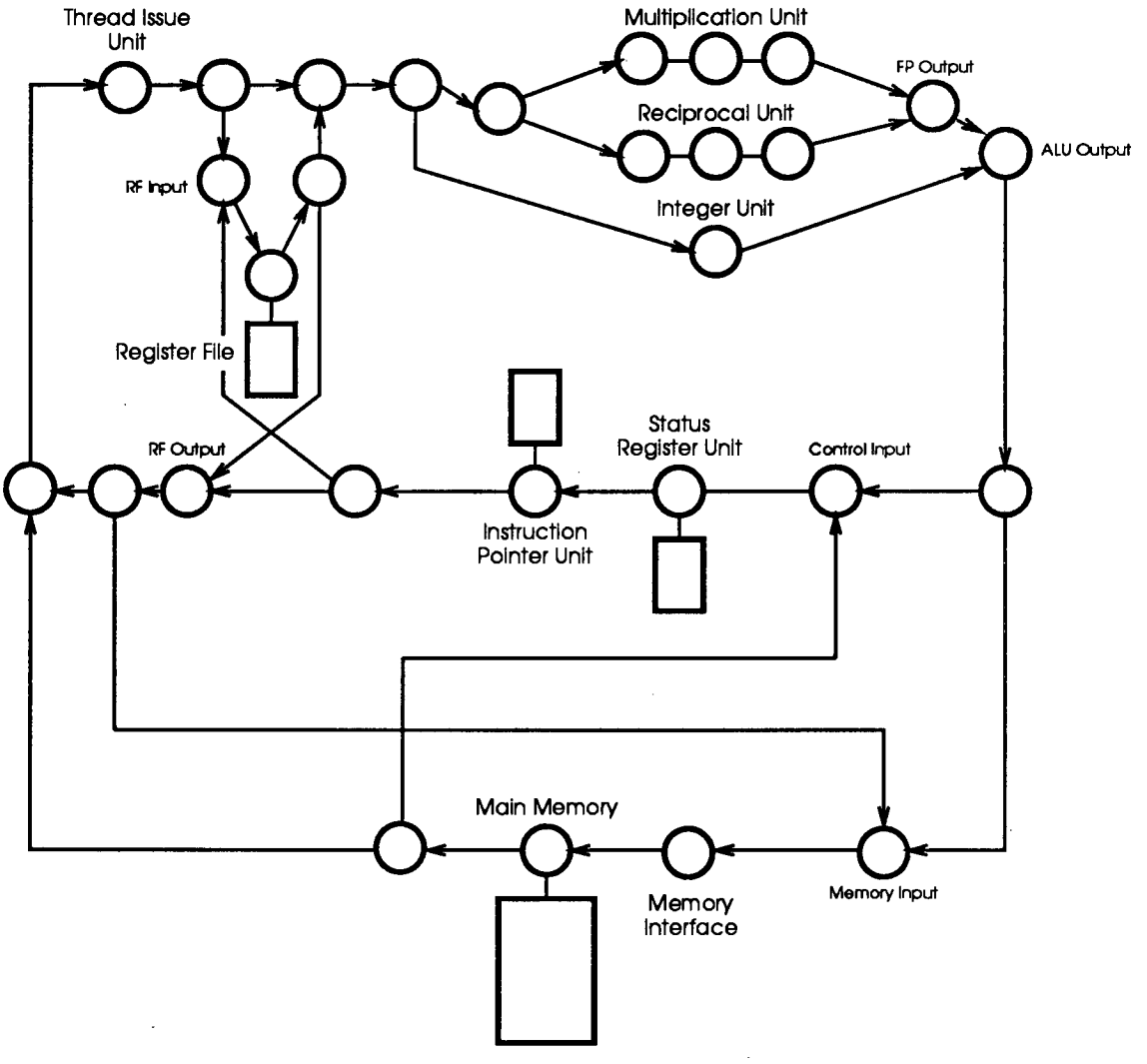


Figure 5–2: Architecture of the CF Processor in context flow graph form

forms instruction decoding and issue, operand fetch and evaluation for each instruction. Thread issue is performed in a single node; the register file is implemented by a single transformation node with a shared access interface, as both operand fetch and result storage context streams require access to the registers; and the ALU is implemented as three separate pipelines, performing multiplication, reciprocation and other floating-point operations, and scalar operations respectively.

Threads and Instruction Decoding

Instruction decoding is performed in the thread issue unit, where instructions are converted from their external format, as shown in Figure 5–1, to an internal format called a *thread.* A thread is an 8-bit quantity in which the value of each bit corresponds to the path which the context containing the instructions is to take at each branch node. Threads, therefore, define the flow of a context through the processor. Given an instruction, the TIU produces a 6-bit function specifier which explicitly encodes any variants, two 4-bit register identifiers and an access mode bit, which are passed along with a 6-bit process identification tag to the register file.

Register File

The register file contains 2016 registers, of which 32 are accessible to any one process. The register space of each process is distinct, preventing one process writing to a register belonging to another process and ensuring no unintended interactions. In addition to the general purpose registers, a set of 32 read-only registers, which contain commonly-used constants such as $0, 1, -1, \pi$ etc., are shared between the processes. These are used to provide the immediate mode constants specified in instructions. The register file uses the signals from the TIU and yields two 32-bit inputs to the ALU.

Arithmetic Unit

The organization of the arithmetic unit is based on that of the Motorola 88100, with three separate evaluation pipelines. The multiplication pipeline performs both floating-point and integer multiplication and requires three stages. The other floating-point operations — addition, subtraction, reciprocation and comparison — together with integer reciprocation, are performed in the arithmetic pipeline which is also three stages long. All other instructions pass through the scalar pipeline which implements integer addition and subtraction and the logical operators. The ALU is connected to the register file by means of a branch node which routes the contexts to the appropriate evaluation pipeline, and provides a 32-bit result and sign, zero and carry status bit to be used in the control and memory pipelines.

5.3.2 Control Pipeline

All instructions commence evaluation by passing through the execution pipeline. The next stage in processing is determined by the nature of the instruction — either progressing directly to the control pipeline, or, in the case of load and store instructions, firstly to the memory and then to the control pipeline. There are three main components of the control pipeline — the *status register unit* (SRU) and the *instruction pointer unit* (IPU), each implemented in a single node, and the result storage unit which connects to the register file.

Status Register Unit

The status register unit contains one register for each process to hold the condition codes generated by the ALU. These are then used during execution of conditional branch instructions to determine the increment required to create the new target address, this calculation also being performed in the SRU.

Instruction Pointer Unit

The instruction pointer unit performs two functions. A pointer to the next pair of instructions to be executed is maintained for each process. In the case of a control transfer instruction, this value is either replaced or incremented by the output from the SRU. The second task is to determine whether a new pair of instructions is to be fetched from memory. If the instruction being evaluated is the first of the current instruction pair, and is not a control transfer instruction, execution can continue without accessing main memory. The appropriate bit in the thread is then set to send the context back to the TIU. If a new instruction pair is required, the context is routed to the memory pipeline.

Result Storage

The control pipeline has a connection to the register file in the execution pipeline to allow a result from the ALU to be stored and the return address from subroutine calls to be saved. The connection to the register file is made via the shared access interface.

5.3.3 Memory Pipeline

The third of the pipelines in the context flow processor is the memory pipeline. Its structure is sufficiently general to allow a variety of memory organizations to be used with the processor in an integrated manner. The memory used in this design is a linear array of 34-bit words, each with a 32-bit data field and a 2-bit tag to identify the datum as either an instruction, a scalar quantity or a floating-point quantity. The memory is attached to a single transformation node which performs a limited amount of processing on the referenced datum — setting the thread bit to route the context to the TIU if the datum is an instruction, or setting the condition code flags in the case of a load instruction. Contexts containing load and store

instructions are directed to the control pipeline, while others are returned to the TIU to execute the next instruction.

5.4 Processor Performance

In order to determine the performance characteristics of the context flow processor, an architectural simulator was constructed and used to measure three aspects of performance for a variety of test data. The three performance characteristics of interest are processor utilization, total pipeline latency and instruction throughput. After initialization, the model of the processor executes random instructions whose relative frequencies parameterize the simulation. The proportion of the total number of instructions which initiate control transfers is fixed at 24.2%, the value determined by Lee and Smith [1984] after analysis of a large number of programs. Of the remaining instructions, the proportion of memory reference and floating-point instructions is variable. During each measurement, the operation of the processor was measured over a period of 2500 graph cycles.

5.4.1 Processor Utilization

Processor utilization is measured as the percentage of nodes in the context flow graph representation of the processor which operate on a non-null context during one graph cycle period. As the memory pipeline is an integral part of the processor, the measured values of utilization include that of the memory.

Determining Average Utilization

Before any comparisons can be made between various simulations, a notion of *average utilization* must be established, and an instruction mix to induce this

must be determined. The ratio of integer to floating-point instructions was fixed at 1:1 so as to exercise all parts of the graph, while the percentage of memory reference instructions was varied between 0% and 100%. This was repeated for processor loads of between 1 and 63 contexts. The graph of Figure 5–3 shows an average utilization of approximately 50% for a memory access rate of 50%. The instruction mix was fixed at 24.2% control transfers, 37.9% load/store instructions, 18.95% floating-point instructions and 18.95% integer operations.

As can be seen from the graph of Figure 5–4(a), there is a linear relationship between utilization and load, until the processor contains 18 active contexts. If the load is increased further, utilization remains almost constant at approximately 50%. With more than 18 active contexts, a queue starts to form at the input to the register file as different contexts attempt to read and write from registers. For each write operation that is performed, a null context is sent to the ALU, by virtue of the internal operation of the branch node connected to the output of the register file, which eventually propagates through all eleven nodes of the ALU. Although the same action takes place at all other branch nodes, their effects are mitigated by the fact that they are almost immediately connected to merge nodes which remove the excess null contexts.

5.4.2 Pipeline Latency

Pipeline latency is measured as the average number of graph cycles required to execute an instruction. The graph in Figure 5–4(b) shows pipeline latency for processor loads of between 1 and 63 contexts. Latency is virtually constant at approximately 23 cycles per instruction for loads below 18 contexts. For greater loads, the latency increases linearly, rising to 89.55 cycles for a load of 63 contexts. This, again, is a consequence of context interaction in the queues of the merge nodes.

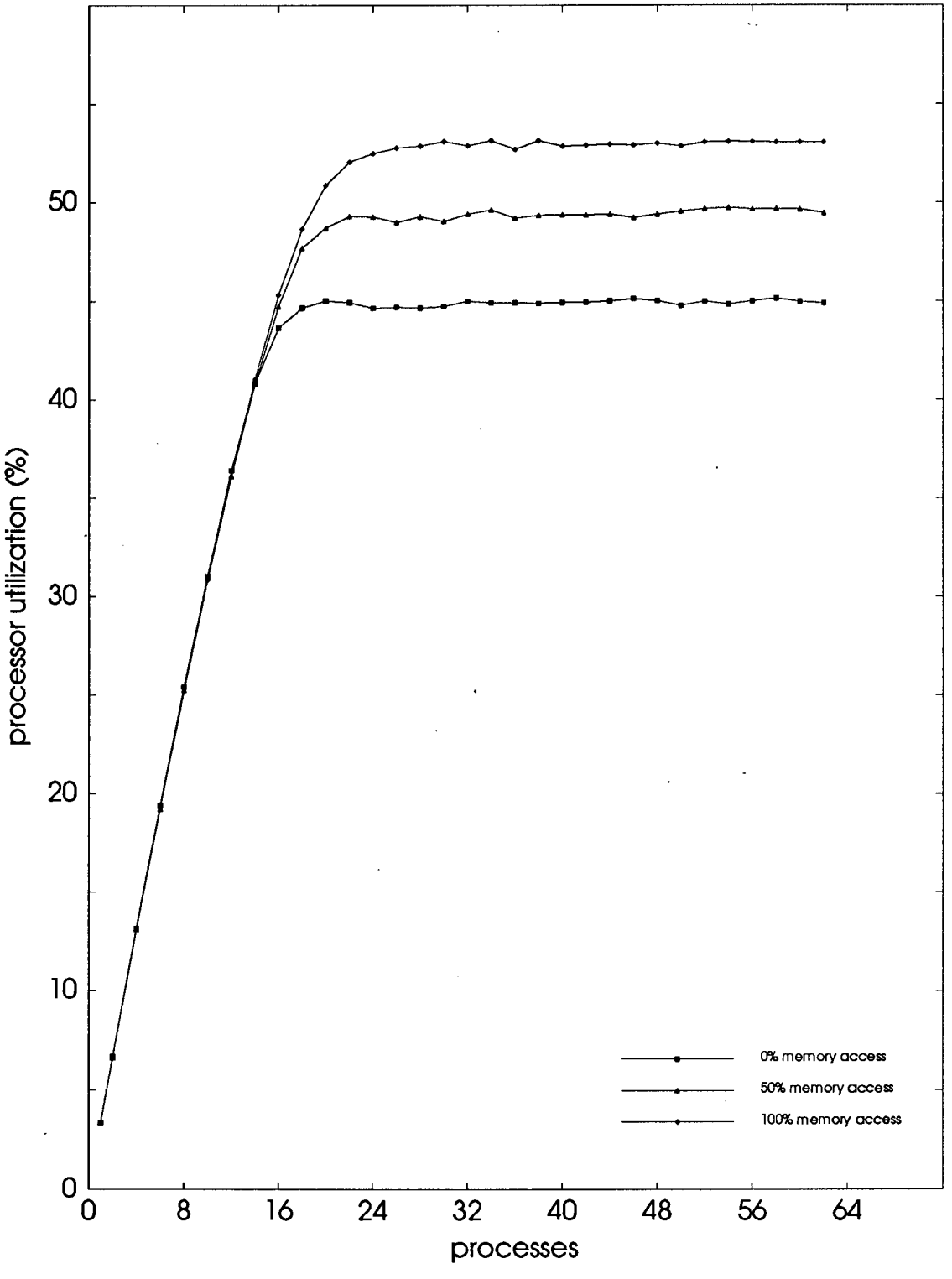


Figure 5–3: Processor utilization v. load for varying memory access rates

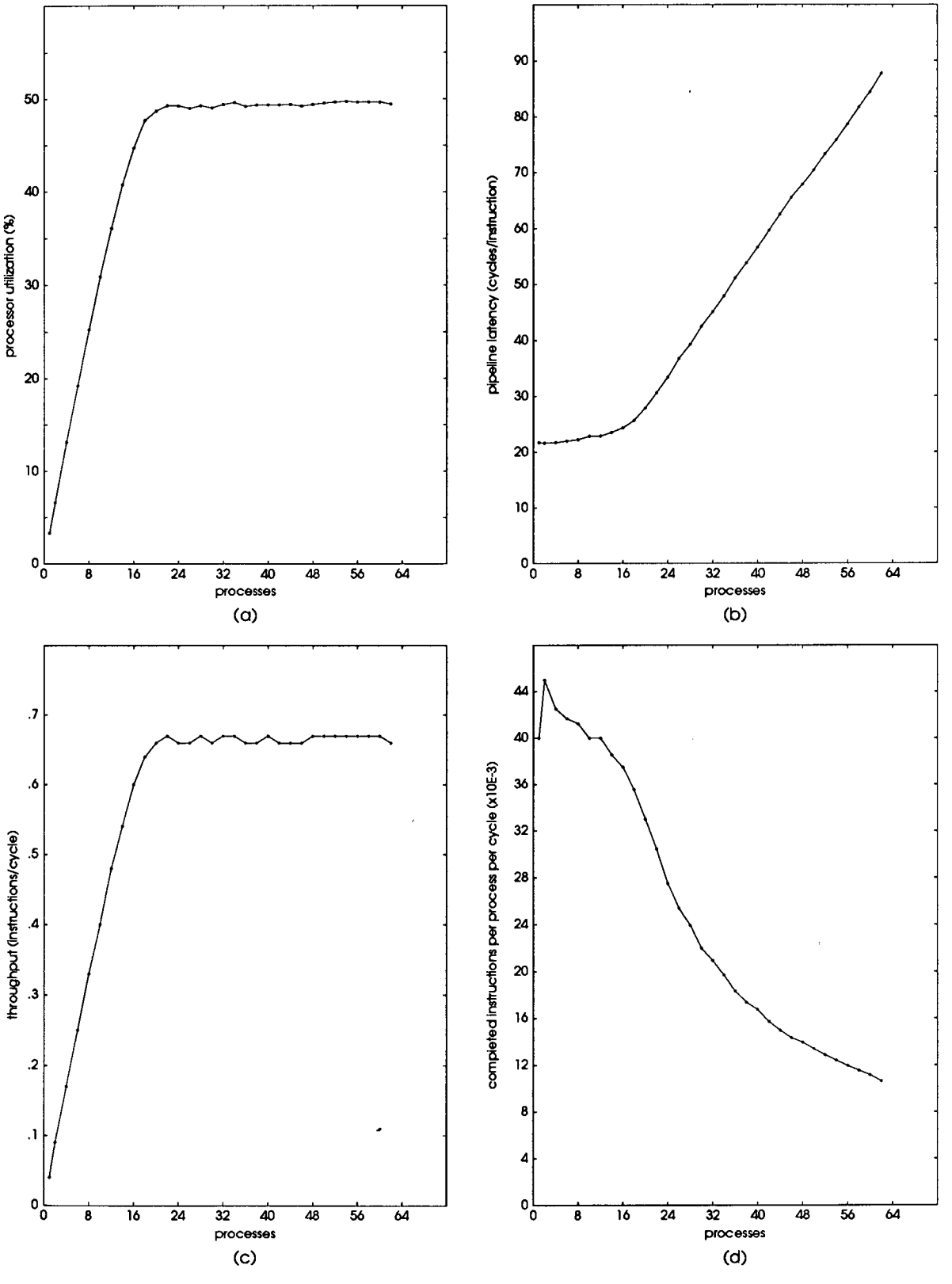


Figure 5–4: Utilization, latency, throughput and instruction completion for increasing processor load

5.4.3 Instruction Throughput

Instruction throughput is measured as the average number of instructions which are completed each graph period. An instruction is deemed to have completed when its context leaves either the control pipeline or the memory pipeline, and is routed to the thread issue unit. Throughput for a range of loads between 1 and 63 contexts is shown in Figure 5–4(c). As with utilization, throughput rises linearly until the number of active processes causes the queue for access to the register file to fill, after which it remains constant at about 0.67 instructions per cycle. Throughput can also be expressed as the number of instructions which are completed by a given process. This relationship is shown in the graph of Figure 5–4(d).

5.4.4 Effects of Memory Latency on Processor Performance

The above results were achieved with a main memory of unit latency, all references, whether read or write, being satisfiable in a single graph cycle. This, however, may not always be the case, for example, if the processor was part of a multiprocessor ensemble, the latency of the interconnection network must be taken into account when deriving memory response time. The effects of increasing memory latency from 1 up to 16 cycles are shown in Figures 5–5 to 5–8.

Processor Utilization

As the memory of the context flow processor is pipelined, the effect of increasing memory latency is to lengthen the processor pipeline. Accordingly, a greater number of active contexts are required to saturate the processor. This is illustrated in the graph of Figure 5–5 by the decreasing gradient in the area of the graph exhibiting linearity. As the memory is considered an integral part of the processor pipeline, the addition of a linear section of pipeline which is kept mostly full results

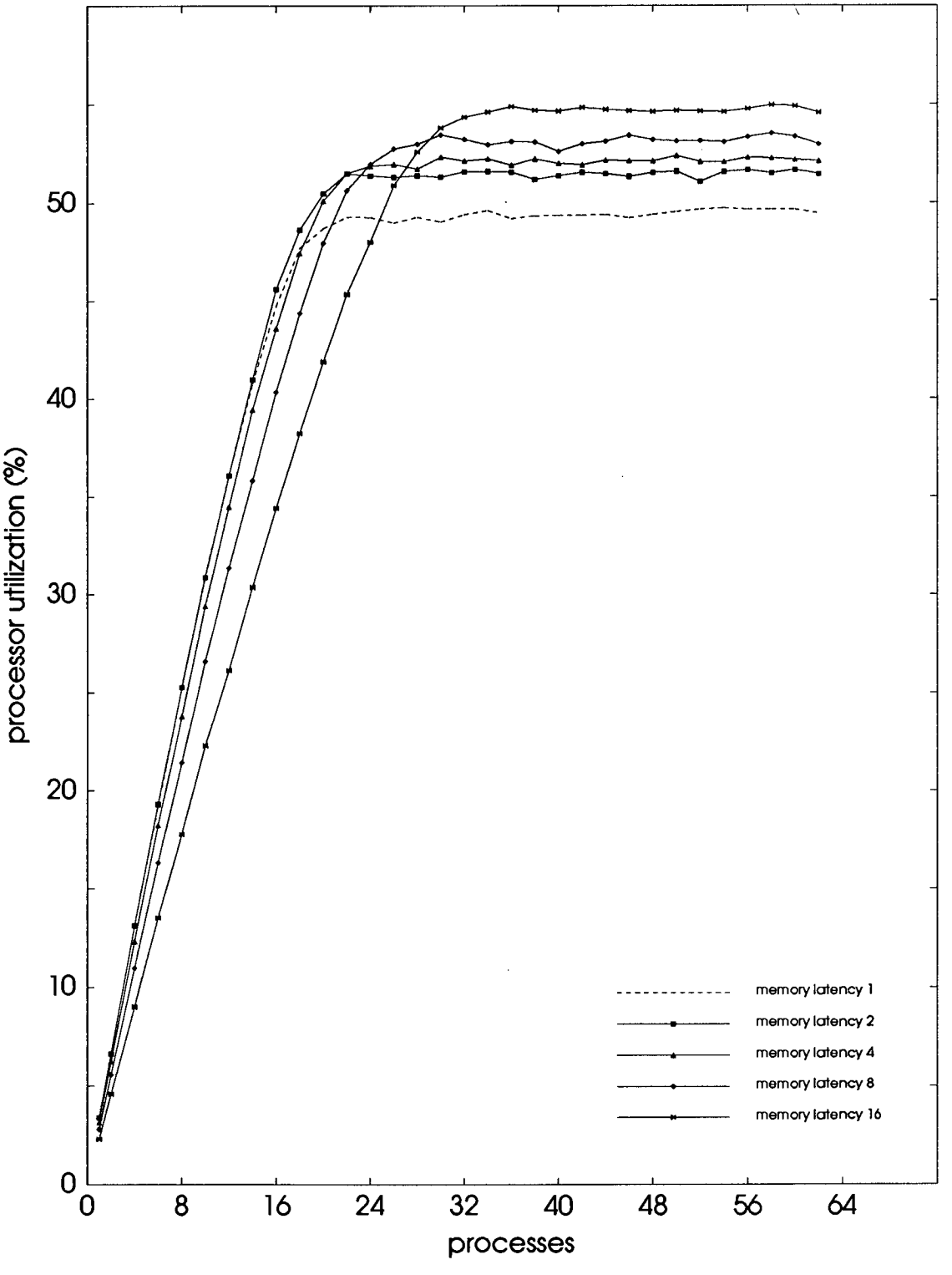


Figure 5–5: Processor utilization v. load for increasing memory latency

in an increase in the percentage processor utilization. This is also confirmed in Figure 5–5.

Pipeline Latency

For the same reasons that processor utilization increases, so pipeline latency also increases in direct proportion to memory latency, as shown in Figure 5–6. Increased latency is initially compensated by the increased capacity of the pipeline, which allows the, albeit higher, latency to be sustained at a constant level for a greater load before rising linearly. The result of this is that pipeline latency becomes independent of memory latency, providing the processor load is sufficient, making remote memory references in a multiprocessor system no more expensive than local memory references.

Instruction Throughput

The effects of memory latency on instruction throughput are similar to those on processor utilization. For a lightly loaded processor, the reduction in throughput is marked, as shown in Figure 5–7. For 16 active contexts, the throughput is reduced from 0.60 instructions per graph period for a memory latency of 1, to 0.41 for a memory latency of 16 — a reduction of over 30%. Again, the increased capacity of the pipeline compensates, and for a heavily loaded processor, the throughput is independent of memory latency. The number of completed instructions per process follows a similar trend, as shown in Figure 5–8.

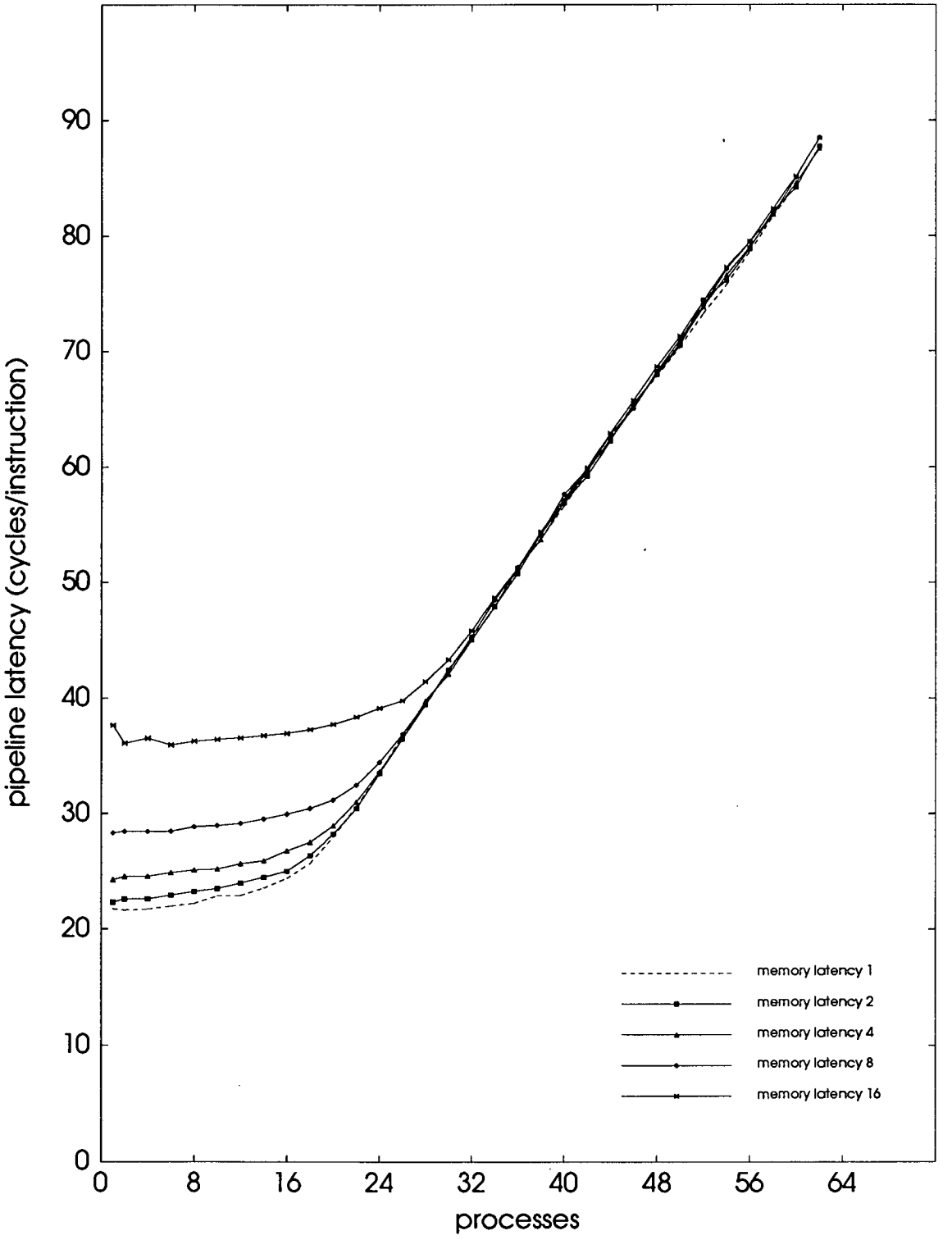


Figure 5–6: Pipeline latency v. load for increasing memory latency

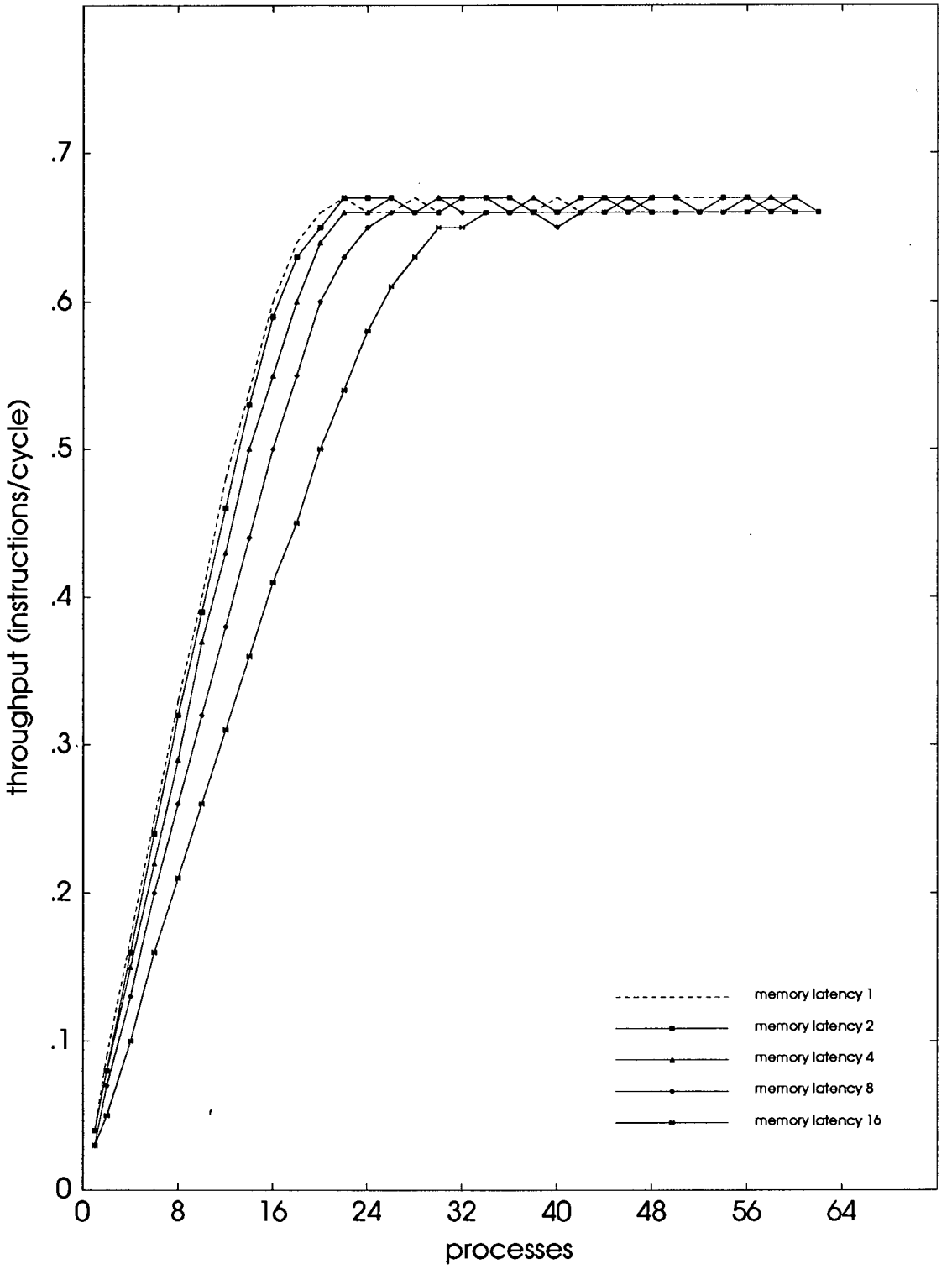


Figure 5–7: Instruction throughput v. load for increased memory latency

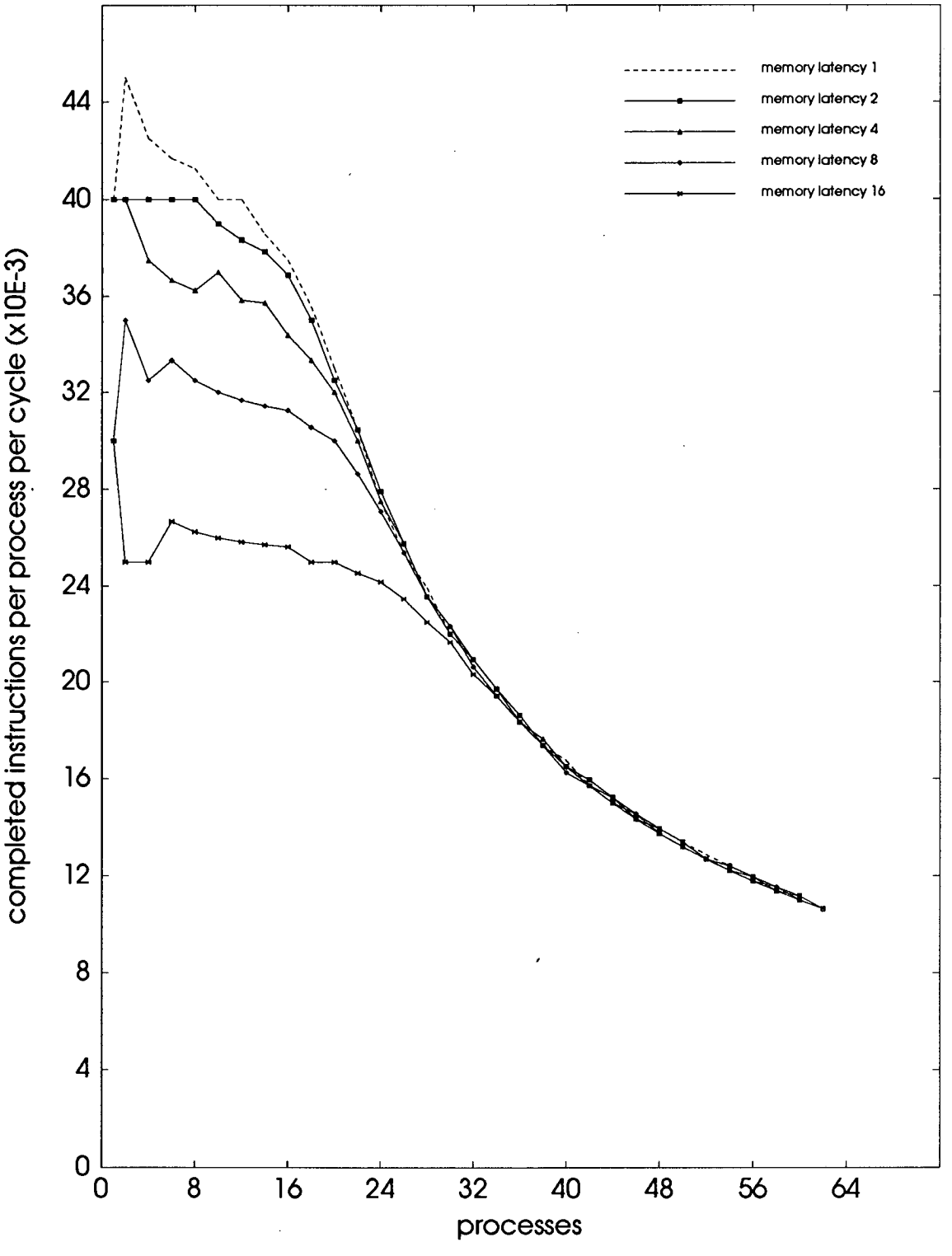


Figure 5–8: Instruction completion v. load for increased memory latency

5.5 Design Alternatives

The aim of context flow is to create architectures which combine maximal pipeline utilization, after initial filling, with a throughput of one instruction per cycle. Individually, the elements of the context flow processor are capable of sustaining such a performance, therefore their interaction must be the source of the measured degradation.

5.5.1 Register File Access

The ALU, status register unit, instruction pointer unit and memory are individually able to sustain an instruction completion rate of one per graph period, providing their inputs are a continuous stream of non-null contexts. In contrast however, the register file can only sustain a completion rate of 0.5 instructions per cycle. This is due to its outputs being derived from the outputs of a branch node, at least one of which passes a null context every cycle. Although the output which feeds the execution pipeline is connected via a merge node, insufficient contexts arrive at the other input to this merge node to allow a non-null context to enter the ALU each cycle. Therefore, the throughput of the ALU is, on average, halved as a result of being connected to the register file.

The register file also acts as a bottleneck. As approximately 70% of the instructions require both read and write access to the register file, its position in the pipeline exacerbates the problem, with almost complete instructions competing with newly issued instructions for access to the registers. This is illustrated in the graphs of Figures 5–9(a)–(d) showing the maximum queue lengths and theoretically predicted limits at each merge node as the processor load is raised. As the load is increased, the theoretical limit is constant for all queues, except the RF Input queue where one input stream includes the queue in the merge node where

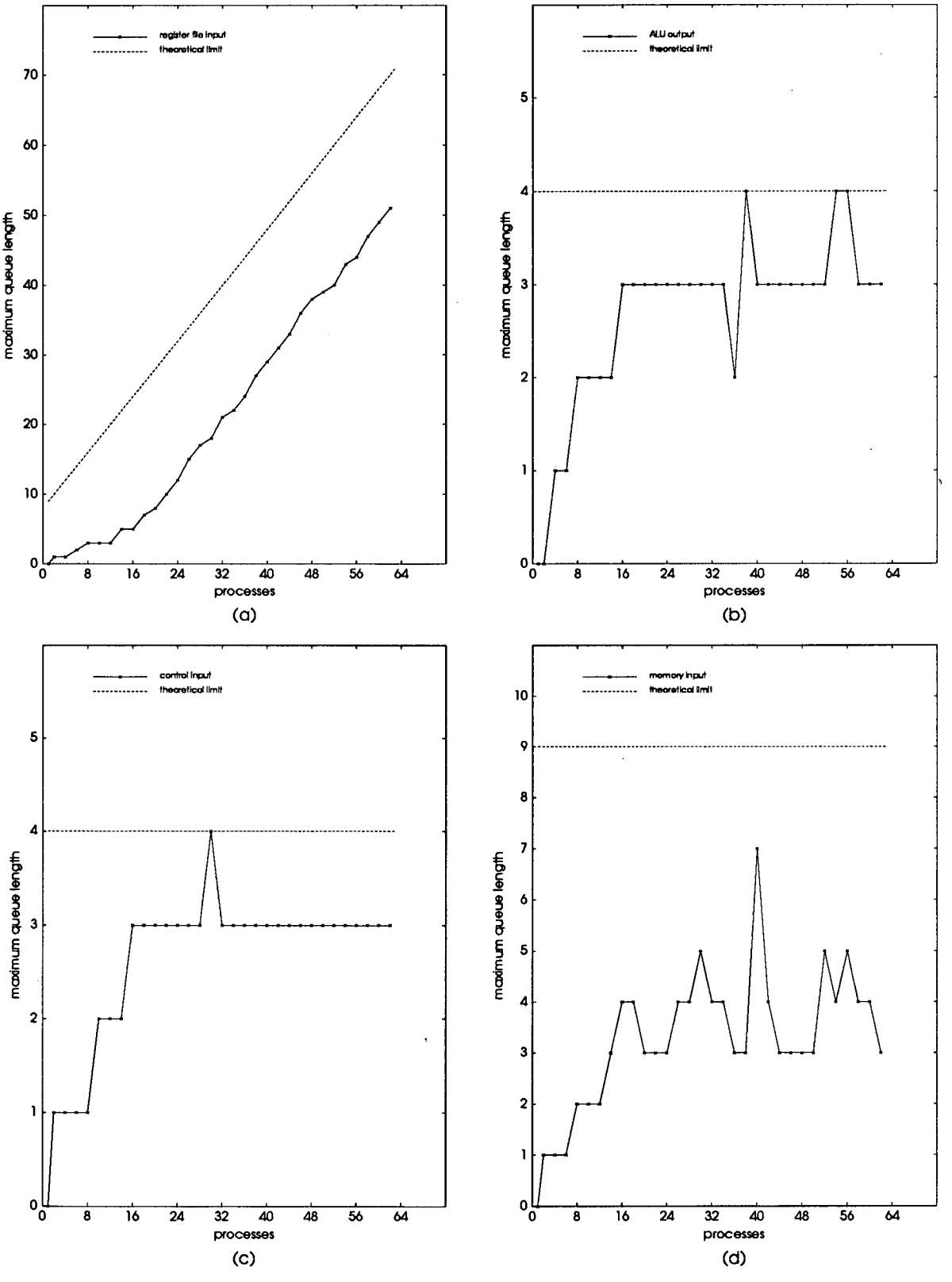


Figure 5–9: Maximum queue lengths v. processor load for the CF processor

initialization occurs. The simulated maximum lengths for all queues are below their predicted theoretical maxima.

The design of the register file is in strict accordance with the rules of context flow which prohibit the sharing of memory between nodes. If this restriction is relaxed, the bottleneck can be removed. As each process accesses a separate register space, and has only one context, the memory containing the register file can safely be shared between a transformation node in each of the execution and control pipelines, without the risk of read/write conflicts. The use of a dual-ported register file shared between the execution and control pipelines results in an improvement in utilization of 20%, as can be seen from the graph of Figure 5–10. More dramatic, however, is the improvement in throughput, which becomes constant at close to one instruction per cycle, as shown in Figure 5–11.

On a macroscopic scale, 100% utilization is both achievable and sustainable, if utilization of functional units rather than individual nodes is measured. Each of the major architectural units of the processor contains at least one active context per cycle, although not all of its constituent nodes may be active concurrently. On a microscopic scale, it is unrealistic to expect sustainable total utilization for anything other than this simplest of linear pipelines. In more complex structures, peak utilization of 100% is possible, but only for short periods. In the context flow processor, total utilization at the node level cannot be achieved, as no more than three non-null contexts may be in the ALU concurrently — a consequence of the operation of the branch nodes which decode the function on entry to the unit.

5.5.2 Instruction Size

The length of instruction is determined by several factors — the size of the instruction set, the number of addressing modes or instruction variants, and whether a 2-address or 3-address format is used. The choice of a small set of 28 instruc-

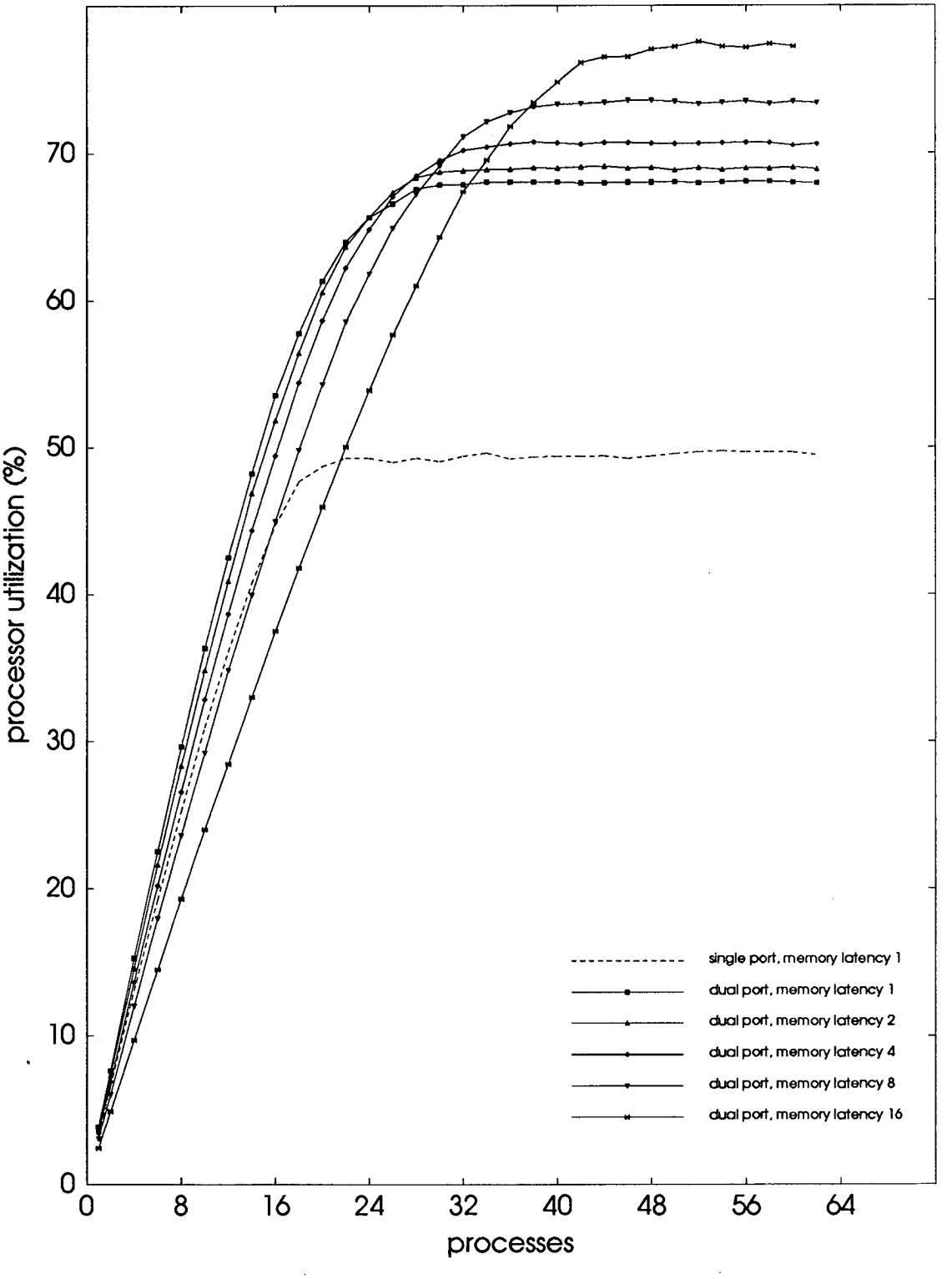


Figure 5–10: Processor utilization v. load for CF processor with dual-port register file

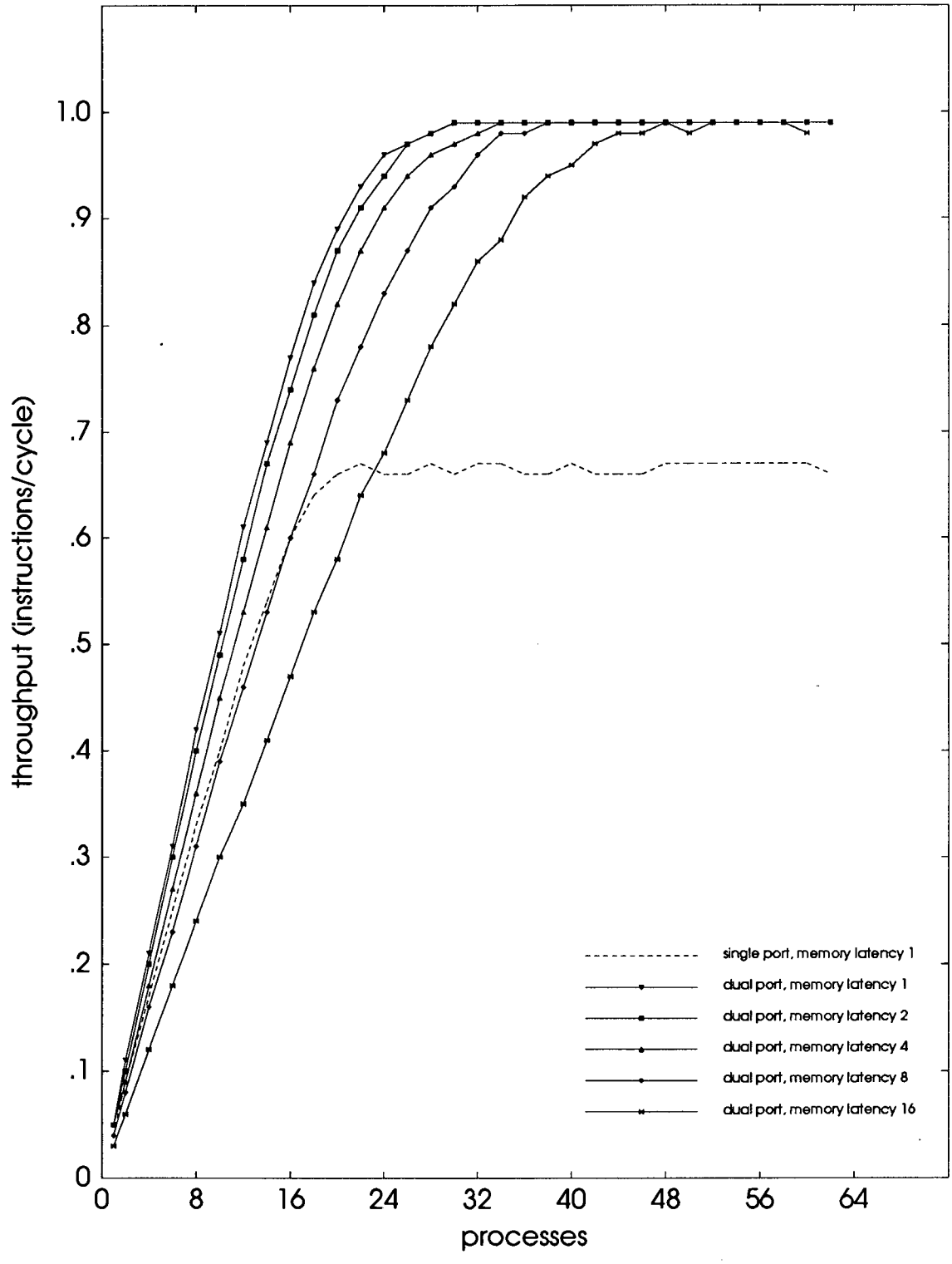


Figure 5–11: Throughput v. load for CF processor with dual-port register file

tions, two addressing modes and a 2-address format allows instructions for the CF processor to be encoded in 16 bits. There are two main benefits to a 16-bit instruction size:

- instructions can be fetched in pairs from memory, reducing the memory access rate,
- instructions can be decoded more easily, and therefore more quickly.

If the instructions are longer than 16 bits, with 32-bit instructions being the only sensible alternative, a new instruction has to be fetched every cycle. This has an adverse effect on performance, as can be seen from Figures 5–12 and 5–13. Throughput drops by nearly 30%, and pipeline latency is not only higher but also increases more rapidly, if dual-cycle is replaced by single-cycle instruction fetching.

However, a 16-bit instruction length does have disadvantages:

- the ability to specify immediate data within an instruction is severely limited,
- the flexibility in register usage of a 3-address format cannot be achieved,
- maintaining the same format for all instructions is difficult if operand denotation is not to be compromised.

In the CF processor, only the **bra**, **bsr** and **rot** instructions have immediate operands, and, in the case of the branch instructions, the size of encodable operand is small. The constant registers provide an elegant solution in certain circumstances, but if a particular constant value is not available in those registers then it must either be loaded from memory having been placed at a specific address by an assembler, or loaded from the context as the second of the pair of operands. In the latter

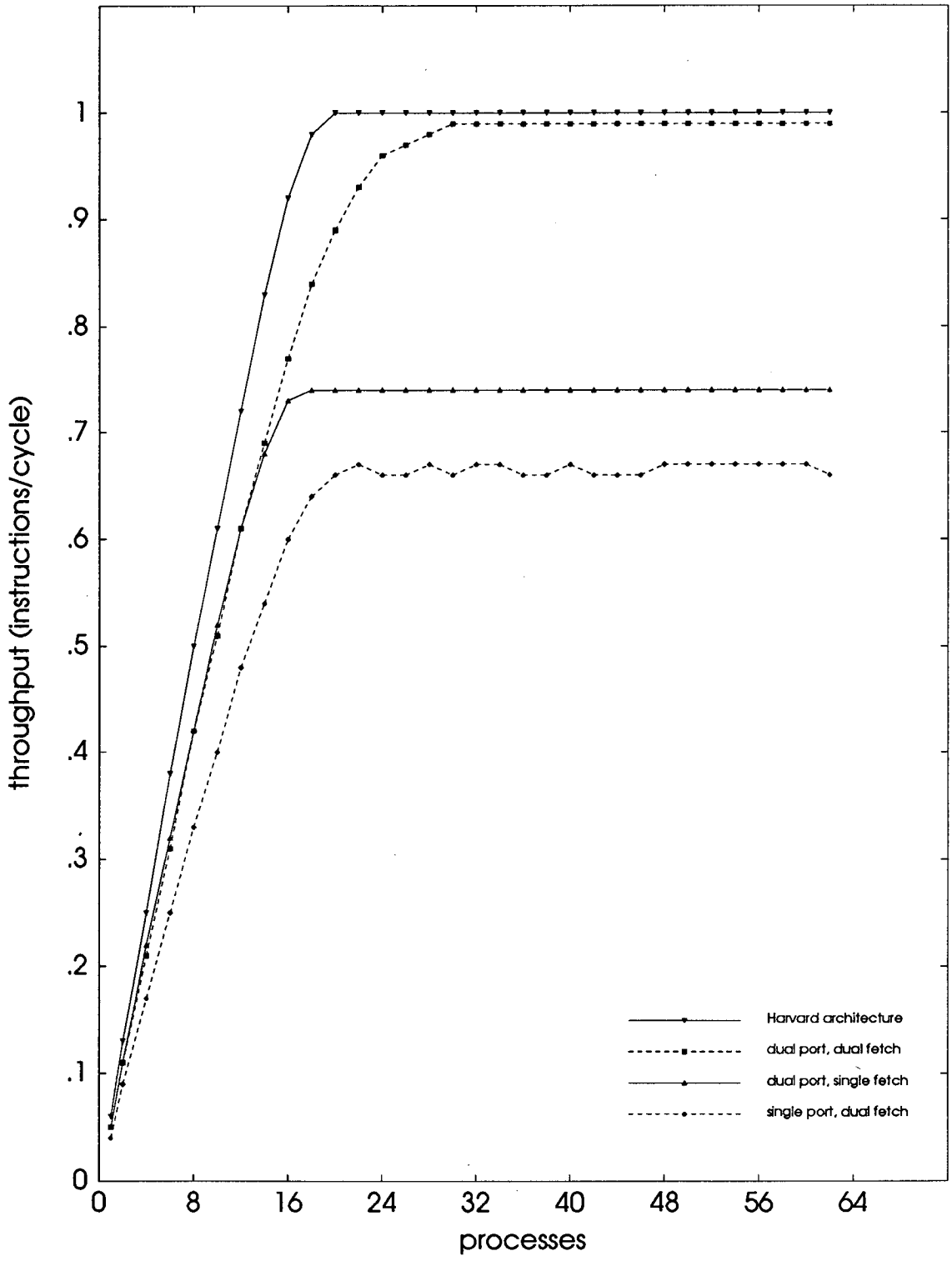


Figure 5–12: Throughput v. load for CF processor with single instruction fetch and Harvard architecture

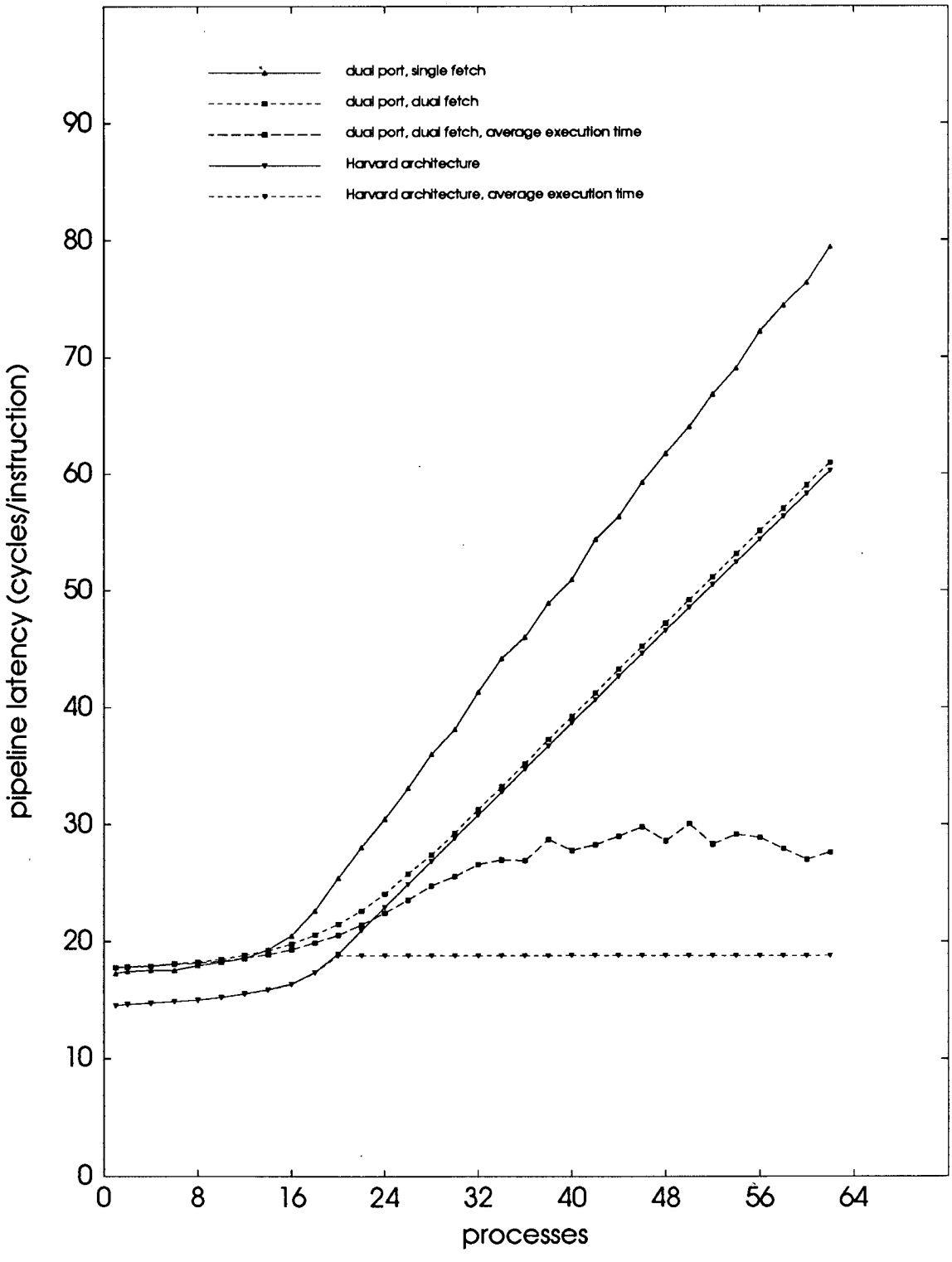


Figure 5–13: Pipeline Latency v. load for CF processor with single instruction fetch and Harvard architecture

case, insertion of noop instructions or delayed branching would be required to ensure correct alignment of an instruction and its immediate operand. Unlike other processors with a 3-address instruction format, for example, the Motorola 88100 and the CDC 6600, a *scoreboard* would not be required in the CF processor, as no data dependencies can arise. With the need to provide operating system and exception support in any practical implementation, a larger instruction set would be required.

By separating the data and instruction memories, thus creating a *Harvard*-type architecture for the context flow processor as shown in Figure 5–14, a substantial improvement in performance can be achieved, even fetching an instruction every cycle. Single instruction-fetching requires a queue between the instruction memory and thread issue units to hold contexts when the processor load is greater than the capacity of the graph. The same function was performed by a merge node connected to the TIU in the dual-cycle instruction fetch architectures. Figures 5–12 and 5–13 provide a comparison between the Harvard architecture and the other architectures with a memory latency of one cycle per reference. The major difference between the Harvard-type and the dual-port, dual fetch architectures is in the average length of time required to execute an instruction. This is measured as the time between a context entering the TIU and leaving the instruction memory unit. Figure 5–13 also shows average execution times for both architectures. In the Harvard architecture, execution time remains constant when processor load is increased beyond the intrinsic capacity of the graph, the corresponding increase in pipeline latency therefore being solely attributable to the additional time spent by each context in the issue queue. Average execution time for the dual-port, dual fetch architecture is not only higher, but also fluctuates, due to interaction between the instruction-fetch and data-fetch streams in the memory pipeline. The constant nature of the pipeline latency illustrates the smoothing effect that the queues in the merge nodes have on system performance. The Harvard architecture

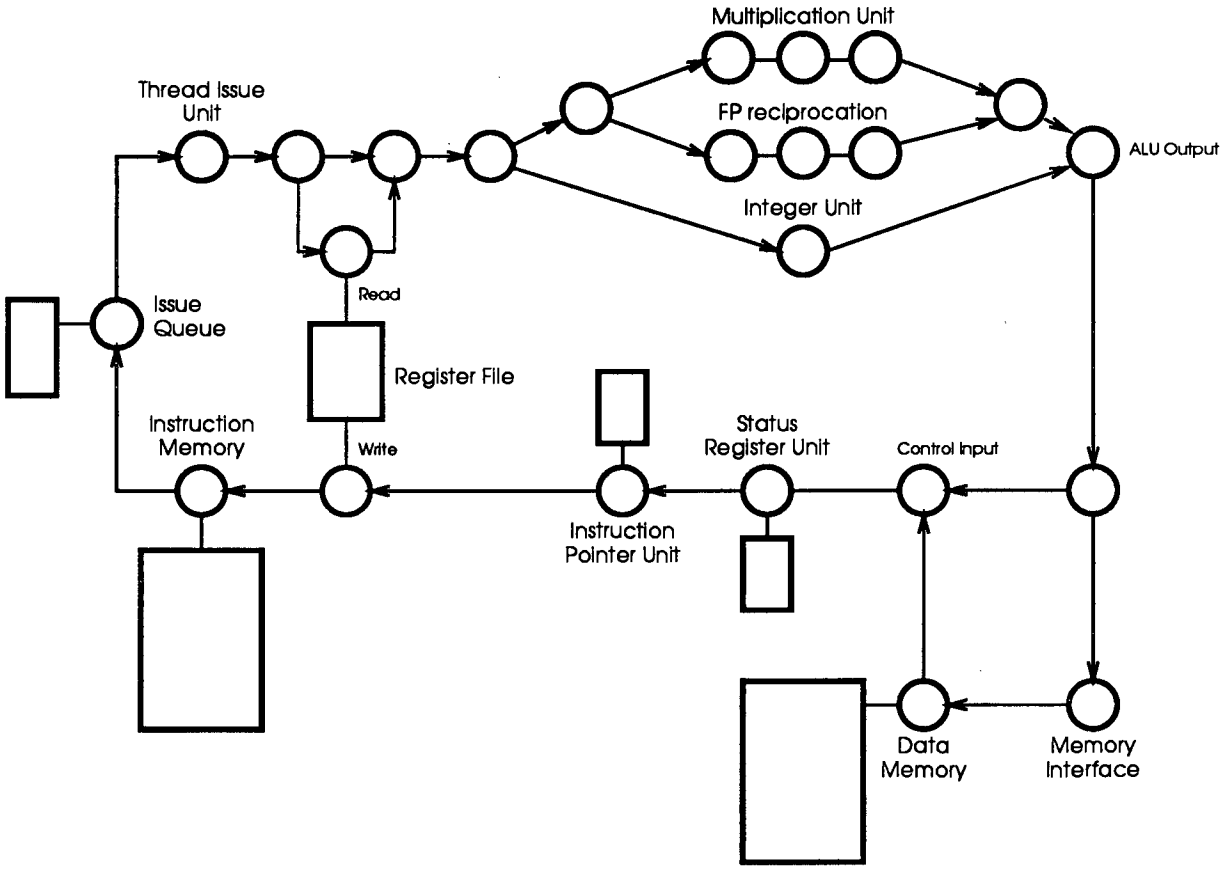


Figure 5–14: CF processor with Harvard architecture and dual-port register file in context flow graph form

is significantly easier to implement as the maximum queue lengths in the merge nodes are small — the longest queue being required at the ALU output which must hold four contexts.

Summary

An analysis of the performance of the original context flow processor design identified several changes to the design which result in improved throughput and reduced pipeline latency. Although CF theory predicts throughput of one context per graph cycle and 100% graph utilization, it is difficult to achieve this in practice. Throughput can be maximised by reducing interaction of context streams to a minimum. Introduction of branch and corresponding merge nodes results in a two-fold performance penalty being paid — utilization being reduced as a result of the creation of null contexts, and latency being increased, if loading is such that contexts require to be queued in merge nodes.

The processor architecture exhibiting the best performance is the Harvard-type architecture, with separate instruction and data memories. This separation reduces much of the interaction, and hence delay, associated with accessing external memory. It also allows instruction size to be increased to 32 bits, providing a more flexible and powerful instruction set while still retaining a fixed, easily decoded format. Introduction of a dual-ported register file removes the other major bottleneck from the pipeline. The remaining merge nodes have small maximum queue lengths — 1,4 and 2 for the register file read output, ALU output and control input queues respectively — making for compact implementation.

"It is quite a three-pipe problem."

— SHERLOCK HOLMES, *The Red-Handed League*

Chapter 6

Implementation, Application and Development

A new architectural design technique which eliminates the sources of discontinuities from pipelines has been presented, both in a theoretical form, allowing certain properties of resulting designs to be proved; and in the form of practical examples, allowing the performance of these designs to be measured. This Chapter tackles three further aspects of context flow design. Issues concerning the practical implementation of context flow structures are discussed. Using some of the experience gained in the design of context flow structures, some possible applications for context flow are presented — both in general terms, highlighting situations where context flow is particularly suitable and unsuitable; and in real cases where the application of context flow may be especially productive. Some possible directions for future work are outlined.

6.1 Implementation Issues

For a number of reasons, the ideal method for the implementation of context flow structures is using VLSI techniques. The organization of both context flow graphs and VLSI circuits is similar in that they both consist of simple elements — core nodes and leaf cells respectively — connected to give the device the required operation. The design process is also similar, being performed upwards from the simplest level in a hierarchical manner. Many of the nodes in a context flow graph essentially perform the same operation. For example, merge nodes differ only in the size of queue, and branch nodes, in the test condition. Therefore, a VLSI implementation of these nodes could be parameterized and incorporated in an application specific integrated circuit (ASIC) or standard cell design procedure.

A major limiting factor in the implementation of context flow systems is the size of the contexts. Although context flow theory allows contexts to be split into static and dynamic parts, the dynamic part can still be potentially very large. Consider, for example, the processor design in Chapter 5. The process identifier must circulate to each node in order to allow recombination of the static and dynamic contexts — to support 63 processes and the null context requires a 6-bit pathway to each node. The instructions and their operands flow through the pipeline together, increasing the width of the data path to 96 bits in places. Careful analysis of the algorithm being implemented by the graph is required to limit the quantity of data which must transferred between nodes. This aspect of context flow design underlines the desirability of a VLSI solution as implementation using discrete components would necessitate large inter-chip data transfers. Reduction of the number of pins on an integrated circuit package was one of the motivations for Kaminsky and Davidson [1979] in their early work on multiple instruction stream pipelines.

Partitioning the context into static and dynamic segments introduces another problem, when access to statically held data is required at more than one point in the graph. This is illustrated in the design of the register file of the processor in Chapter 5. The original design, which adhered strictly to the rules of the context flow model, proved to be inefficient, creating a large bottleneck in the pipeline. Relaxation of the rule prohibiting the sharing of memory between nodes allowed the bottleneck to be removed. In this particular instance there can be no conflicting access to the memory, but this may not always be the case. Care must therefore be taken if similar decisions are made to share a memory between two or more parts of a pipeline.

6.2 Applications of Context Flow

By maintaining effective use of existing resources as one of its central tenets, context flow achieves improved performance without changing the nature of the problems which may be solved by a given architecture. That is not to say that special purpose context flow machines cannot be designed, merely that context flow maintains generality of purpose.

Context flow structures achieve their improved performance over conventional designs as a consequence of improved pipeline utilization created by sharing the hardware between several concurrent processes. Consider, again, the context flow processor of Chapter 5. Although the time required to execute all the instructions of a given process is liable to be greater than if the process were executed on a single instruction stream pipeline, the total time to execute all instructions from all processes is less as the throughput of the context flow processor is constant. This, however, does require that such a processor be kept as close to being fully-loaded as possible. As with many other parallel architectures, context flow structures do not perform well when executing code with cannot be divided into sufficient parallel

tasks. The existence of a hardware limit on the number of concurrent active processes may place restrictions on the suitability of some types of problems for context flow implementation.

Increased memory latency, whether as a result of the memory being located on the other side of an interconnection network, or simply slower than the graph cycle time, is compensated by a rise in the capacity of the pipeline. If sufficient parallel processes can be generated, the physical location of the memory in a multiprocessor system becomes unimportant, with the latency of all non-local memory access being dependent on topology of the network. The construction of context flow graphs from simple nodes with well defined behaviours, coupled with the similarity between graphs and algorithmic flow charts, provides for the rapid design of context flow systems. The removal of the need for specialized branch prediction hardware greatly simplifies the implementation of context flow designs.

Two particular areas which can make use of the features provided by a context flow device are real-time and graphics systems. In real-time applications, the sustainable and constant throughput offered by context flow is of great importance, offering a guarantee of system performance. For purposes such as process monitoring and data gathering, context flow would be ideal, allowing each sensor to be serviced by its own context. A lightly loaded real-time CF system could also guarantee response times limited only by the size of the graph. In graphics applications, windows, icons and pointers could also be mapped individually on to separate contexts, reducing the software overhead associated with switching tasks.

6.3 Future Research

The next step in the development of context flow is the implementation of a context flow processor. While the designs and simulations described in Chapter 5 will provide a starting point for an implementation, several deficiencies require correction in order to produce a viable design.

- Completion of the instruction set — If the Harvard architecture is to be adopted, a 32-bit instruction length can be used, in which case adaptation of an existing instruction set, such as the Motorola 88100 or the MIPS R3000, would be sensible.

- User and supervisor modes — In order to provide operating system support, some notion of privilege is required, together with the creation of separate user and supervisor modes of operation.

- Support for handling exceptions and an indivisible **test&set** instruction for the implementation of synchronization primitives are required.

Even after inclusion of these features, implementation of the processor on a single VLSI chip should be feasible. If a 32-bit address size is used for both instruction and data memories, up to 64 megabytes of real memory would be available for each of the 63 active processes requiring 128 pins to connect to external memory devices. Allocating up to 36 pins for connections to the power supply would still leave 16 pins in a 180-pin package for interrupts and other control and status signals. The main limiting effect that the implementation will have on the architecture is on the size of the register file, which will depend on the available area of silicon.

One possible area of investigation is the automatic synthesis of designs from descriptions of context flow graphs. This would require the design of a library

of parameterized VLSI cells to implement merge nodes, branch node decision mechanisms and a selection of transformation node functions. These cells could then be composed using automatic routing generation or other silicon compilation or assembly techniques to form VLSI implementations of context flow graphs.

"Il n' existe pas de sciences appliquées, mais seulement des applications de la science."

— LOUIS PASTEUR (1822–1895), *Address, 11th September 1872, Comptes render des travaux du Congrès viticole et séricicole de Lyon*

References

[Barnes *et al.*, 1968] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. "The ILLIAC IV computer". *IEEE Trans. Comput.*, Vol. C-17, No. 8, pp. 746–57, Aug. 1968.

[Bell, 1989] G. Bell. "The future of high performance computing in science and engineering". *Comm. ACM*, Vol. 32, No. 9, pp. 1091–1101, Sep. 1989.

[Brantley and Weiss, 1983] W.C. Brantley and J. Weiss. "Organization and architecture trade-offs in FOM". In *Proc. Int. Workshop on Comp. Sys. Organization*, IEEE,(New Orleans, Mar. 29–31). IEEE Comp. Soc. Press, New York, 1983, pp. 139–145.

[Butner and Staley, 1986] S.E. Butner and C.A. Staley. "A RISC multiprocessor based on circulating context". In *Proc. 5th Ann. Phoenix Conf. Computers and Communication*, IEEE,(Phoenix, Arizona, Feb. 26–28). Mar. 1986, pp. 620–624.

[Butner, 1984] S. Butner. "The circulating context multiprocessor, An architecture for reliable systems". In *Proc. Int. Symp. Mini and Microcomputers.* 1984, pp. 50–52.

[Chen, 1971] T.C. Chen. "Parallelism, pipelining and computer efficiency". *Computer Design*, Vol. 10, No. 1, pp. 69–74, Jan. 1971.

[Christman, 1984] D.P. Christman. "Programming the Connection Machine". *Technical Report ISL-84-3*, Xerox Palo Alto Research Center, Apr. 1984.

[Darlington and Reeve, 1981] J. Darlington and M. Reeve. "ALICE—A multi-processor reduction machine for the parallel evaluation of applicative languages". In *Proc. ACM Conf. Functional Programming Lang. Comp. Arch.*, (Portsmouth, New Hampshire). 1981, pp. 65–75.

[Davis and Keller, 1982] A.L. Davis and R.M. Keller. "Data flow program graphs". *IEEE Computer*, Vol. 15, No. 2, pp. 26–41, Feb. 1982.

[DeRosa and Levy, 1987] J.A. DeRosa and H.M. Levy. "An evaluation of branch architectures". In *Proc. 14th Int. Symp. Comp. Arch.*, ACM/IEEE,(Pittsburgh, PA., Jun. 2–5). IEEE Comp. Soc. Press, Washington D.C., June 1987, pp. 10–16.

[Ditzel and McLellan, 1987] D.R. Ditzel and H.R. McLellan. "Branch folding in the CRISP microprocessor: Reducing branch delay to zero". In *Proc. 14th Int. Symp. Comp. Arch.*, ACM/IEEE,(Pittsburgh, PA., Jun. 2–5). IEEE Comp. Soc. Press, Washington D.C., June 1987, pp. 2–9.

[Ditzel *et al.*, 1987] D.R. Ditzel, H.R. McLellan, and A.D. Berenbaum. "The hardware architecture of the CRISP microprocessor". In *Proc. 14th Int. Symp. Comp. Arch.*, ACM/IEEE,(Pittsburgh, PA., Jun. 2–5). IEEE Comp. Soc. Press, Washington D.C., June 1987, pp. 309–319.

[Farmwald, 1984] P.M. Farmwald. "The S-1 Mark II–A supercomputer". In J.S. Kowalik, Ed., *High Speed Computation*, pp. 145–55. Springer-Verlag, Berlin, 1984. NATO ASI Series F: Comp. and Sys. Sci. Vol. 7.

[Feng, 1981] T-y. Feng. "A survey of interconnection networks". *IEEE Computer*, Vol. 14, No. 12, pp. 12–27, Dec. 1981.

[Flynn and Podvin, 1972] M.J. Flynn and A. Podvin. "Shared resource multiprocessing". *IEEE Computer*, Vol. 5, No. 2, pp. 20–28, Mar./Apr. 1972.

[Flynn, 1972] M.J. Flynn. "Some computer organizations and their effectiveness". *IEEE Trans. Comput.*, Vol. C-21, No. 9, pp. 948–960, Sep. 1972.

[Goodman *et al.*, 1985] J.R. Goodman, J-t. Hsieh, K. Liou, A.R. Pleszkun, P.B. Schechter, and H.C. Young. "PIPE: A VLSI decoupled architecture". In *Proc. 12th Int. Symp. Comp. Arch.*, ACM/IEEE,(Boston, Mass., Jun. 17–19). IEEE Comp. Soc. Press, Washington D.C., June 1985, pp. 20–27.

[Gottlieb *et al.*, 1983] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. "The NYU Ultracomputer—Designing an MIMD shared memory parallel computer". *IEEE Trans. Comput.*, Vol. C-32, No. 2, pp. 175–189, Feb. 1983.

[Gregory and McReynolds, 1963] J. Gregory and R. McReynolds. "The SOLOMON computer". *IEEE Trans. Electronic Comput.*, Vol. 12, No. 6, pp. 774–781, Dec. 1963.

[Hayes *et al.*, 1986] J.P. Hayes, T. Mudge, Q.F. Stout, S. Colley, and J. Palmer. "A microprocessor-based hypercube supercomputer". *IEEE Micro*, Vol. 6, No. 5, pp. 6–17, Oct. 1986.

[Haynes *et al.*, 1982] L.S. Haynes, R.L. Lau, D.P. Siewiorek, and D.W. Mizell. "A survey of highly parallel computing". *IEEE Computer*, Vol. 15, No. 1, pp. 9–24, Jan. 1982.

[Hennessy *et al.*, 1982] J. Hennessy, N. Jouppi, J. Gill, F. Baskett, A. Strong, T. Gross, C. Rowen, and J. Leonard. "The MIPS machine". In *Proc. 9th Int. Symp. Comp. Arch.*, ACM/IEEE,(Austin, Texas, Apr. 26–29). IEEE Comp. Soc. Press, Washington D.C., 1982, pp. 2–7.

[Hillis, 1985] W.D. Hillis. *The Connection Machine.* Series in Artificial Intelligence. The MIT Press, Cambridge, Mass., 1985. ISBN 0–262–08157–1.

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, London, 1985. ISBN 0–13–153289–8.

[Hockney, 1987] R.W. Hockney. "Classification and evaluation of parallel computer systems". In R. Dierstein, D. Müller-Wichards, and H-M. Wacker, Eds., *Parallel Computing in Science and Engineering. Proc. 4th Int. Seminar on Foundations of Engineering Sciences*, DFVLR,(Bonn, F.R.G., Jun. 25–26). Springer-Verlag, Berlin, 1987, pp. 13–25. Published as Springer-Verlag Lecture Notes in Computer Science Vol. 295.

[Hopfield, 1979] J.J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities". *Proc. Nat. Acad. Sci. U.S.A.*, Vol. 79, pp. 2554–2558, 1979.

[Horowitz and Zorat, 1981] E. Horowitz and A. Zorat. "The binary tree as an interconnection network: Applications to multiprocessor systems and VLSI". *IEEE Trans. Comput.*, Vol. C-30, No. 4, pp. 247–253, Apr. 1981.

[Hwang and Briggs, 1984] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* Series in Computer Organization and Architecture. McGraw-Hill, New York, 1984. ISBN 0–07–031556–6.

[Ianucci, 1988] R.A. Ianucci. "Toward a dataflow/von Neumann hybrid architecture". In *Proc. 15th Int. Symp. Comp. Arch.*, ACM/IEEE,(Honolulu, Hawaii, May 30–Jun. 2). IEEE Comp. Soc. Press, Washington D.C., 1988, pp. 147–153.

[Ibbett, 1982] R.N. Ibbett. *The Architecture of High Performance Computers.* Computer Science Series. Macmillan, 1982. ISBN 0–333–33231–8.

[Intel Corporation, 1989] Intel Corporation. "i860 64-bit microprocessor". *Technical Documentation 240296-001*, 1989.

[Jensen, 1978] C. Jensen. "Taking another approach to supercomputing". *Datamation*, Vol. 24, No. 2, pp. 159–75, 1978.

[Johnson, 1988] E.E. Johnson. "Completing an MIMD multiprocessor taxonomy". *ACM SIGARCH Comp. Arch. News*, Vol. 16, No. 3, pp. 44–47, June 1988.

[Jordan, 1985] H.F. Jordan. "HEP architecture, programming and performance". In J.S. Kowalik, Ed., *Parallel MIMD Computation: HEP Supercomputer and its Applications*, ch. 1, pp. 1–40. The MIT Press, Cambridge, Mass., 1985. ISBN 0–262–11101–2.

[Kaminsky and Davidson, 1979] W.J. Kaminsky and E.S. Davidson. "Developing a multiple-instruction-stream single-chip processor". *IEEE Computer*, Vol. 12, No. 12, pp. 66–76, Dec. 1979.

[Katevenis, 1985] M.G.H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* The MIT Press, Cambridge, Mass., 1985. ACM Doctoral Dissertation Award 1984.

[Kohn and Margulis, 1989] L. Kohn and N. Margulis. "Introducing the Intel i860 64-bit microprocessor". *IEEE Micro*, Vol. 9, No. 4, pp. 15–30, Aug. 1989.

[Krishnamurthy, 1989] E.V. Krishnamurthy. *Parallel Processing — Principles and Practice.* International Computer Science Series. Addison-Wesley, 1989.

[Kuck *et al.*, 1987] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. "Parallel supercomputing today and the Cedar approach". In J.J. Dongarra,

Ed., *Experimental Parallel Computer Architectures*, ch. 1, pp. 1–23. North-Holland, Amsterdam, 1987. Special Topics in Supercomputing, Volume 1. ISBN 0–444–70234–2.

[Lawrie, 1975] D.H. Lawrie. "Access and alignment of data in an array processor". *IEEE Trans. Comput.*, Vol. C-24, No. 12, pp. 1145–1155, Dec. 1975.

[Lee and Smith, 1984] J.K.F. Lee and A.J. Smith. "Branch prediction strategies and branch target buffer design". *IEEE Computer*, Vol. 1, No. 1, pp. 7–22, Jan. 1984.

[Lilja, 1988] D.J. Lilja. "Reducing the branch penalty in pipelined processors". *IEEE Computer*, Vol. 21, No. 7, pp. 47–55, July 1988.

[McFarling and Hennessy, 1986] S. McFarling and J. Hennessy. "Reducing the cost of branches". In *Proc. 13th Ann. Symp. Comp. Arch.*, IEEE/ACM,(Tokyo, Japan). IEEE, 1986, pp. 396–403.

[Melear, 1989] C. Melear. "The design of the 88000 RISC family". *IEEE Micro*, Vol. 9, No. 2, pp. 26–38, Apr. 1989.

[Motorola Inc., 1988] Motorola Inc. "MC88100 RISC microprocessor user's manual". *Technical Documentation MC88100UM/AD*, 1988.

[Nikhil and Arvind, 1989] R.S. Nikhil and Arvind. "Can dataflow subsume von Neumann computing?". In *Proc. 16th Int. Symp. Comp. Arch.*, ACM/IEEE,(Jerusalem, Israel, May 28–Jun. 1). IEEE Comp. Soc. Press, Washington D.C., 1989, pp. 262–272.

[Pease, 1977] M.C. Pease. "The indirect binary n-cube microprocessor array". *IEEE Trans. Comput.*, Vol. C-26, No. 5, pp. 458–473, May 1977.

[Peyton Jones, 1987] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall International, London, 1987. ISBN 0–13–453325–9.

[Pfister *et al.*, 1985] G.F Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and K.P. Weiss. "The IBM research parallel processor prototype (RP3): Introduction and architecture". In *Proc. Int. Conf. Parallel Processing*, IEEE,(Chicago, Ill.). 1985, pp. 764–771.

[Preparata and Vuillemin, 1981] F.P. Preparata and J. Vuillemin. "The cube-connected cycles: A versatile network for parallel computation". *Comm. ACM*, Vol. 24, No. 5, pp. 300–309, May 1981.

[Radin, 1983] G. Radin. "The 801 minicomputer". *IBM J. Res. Dev.*, Vol. 27, No. 3, pp. 237–246, May 1983. Also in *Proc. Symp. for Programming Languages and Operating Systems, ACM SIGARCH Computer Architecture News*, **Vol. 10, No. 2**, Mar. 1982.

[Ramamoorthy and Li, 1977] C.V. Ramamoorthy and H.F. Li. "Pipeline architecture". *ACM Comput. Surv.*, Vol. 9, No. 1, pp. 61–102, Mar. 1977.

[Reisig, 1985] W. Reisig. *Petri Nets — An Introduction*, Vol. 4 of *EATCS Monographs on Theoretical Comp. Sci.* Springer-Verlag, Berlin, F.R.G., 1985.

[Rettberg and Thomas, 1986] R. Rettberg and R. Thomas. "Contention is no obstacle to shared-memory multiprocessing". *Comm. ACM*, Vol. 29, No. 12, pp. 1202–1212, Dec. 1986.

[Rogers and Topham, 1990] D.D. Rogers and N.P. Topham. "Implementing a practical context flow machine". Dept. of Computer Science, University of Edinburgh. Private communication, 1990.

[Shimizu *et al.*, 1986] K Shimizu, E. Goto, and S. Ichikawa. "CPC Cyclic Pipelined Computer—An architecture suited for Josephson and pipelined machines". *Technical Report 86–19*, University of Tokyo, Department of Information Science, Nov. 1986.

[Sietz, 1985] C.L. Sietz. "The cosmic cube". *Comm. ACM*, Vol. 28, No. 1, pp. 22–33, Jan. 1985.

[Skillkorn, 1988] D.B. Skillkorn. "A taxonomy for computer architectures". *IEEE Computer*, Vol. 21, No. 7, pp. 46–57, Nov. 1988.

[Smith, 1984] J.E. Smith. "Decoupled access/execute computer architectures". *ACM Trans. Comput. Syst.*, Vol. 2, No. 4, pp. 289–308, Nov. 1984. also in *Proc. 9th Ann. Symp. Comp. Arch.*, IEEE Comp. Soc./ACM, (Austin, Texas, Apr 26–29), ACM, 1982, pp. 112-119.

[Smith, 1985] B. Smith. "The architecture of HEP". In J.S. Kowalik, Ed., *Parallel MIMD Computation: HEP Supercomputing and Its Applications*, ch. 1, pp. 41–55. The MIT Press, Cambridge, Mass., 1985. ISBN 0–262–11101–2.

[Staley and Butner, 1986] C.A. Staley and S.E. Butner. "A feasability study and simulation of the circulating context multiprocessor CMMP". In K. Hwang, S.M. Jacobs, and E.E. Swartzlander, Eds., *Proc. 1986 Int. Conf. Parallel Processing*, IEEE,. 19–22 Aug. 1986, pp. 455–462.

[Thacker *et al.*, 1982] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs. "Alto: A personal computer". In D.P. Siewiorek, C.G. Bell, and A. Newell, Eds., *Computer Structures: Principles and Examples*, ch. 33, pp. 549–572. McGraw-Hill, Japan, 1982.

[Thornton, 1964] J.E. Thornton. "Parallel operation in the Control Data 6600". In *Proc. Fall Joint Comput. Conf.*, AFIPS,. 1964, pp. 33–40. also in D.P.

Siewiorek, C.G. Bell and A. Newell. *Computer Structures — Principles and Examples.* ch. 43, pp. 730–742. McGraw-Hill, Tokyo. 1982.

[Tomasulo, 1967] R.M. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". *IBM J. Res. Dev.*, Vol. 11, No. 1, pp. 25–33, Jan. 1967.

[Topham *et al.*, 1988] N.P. Topham, A. Omondi, and R.N. Ibbett. "Context flow: An alternative to conventional pipelined architectures". *J. Supercomputing*, Vol. 2, No. 1, pp. 29–53, 1988.

[Watson, 1972] W.J. Watson. "The TI ASC — A highly modular and flexible super computer architecture". In *Proc. AFIPS Fall Joint Comp. Conf.* 1972, pp. 221–228.

[Weber and Gupta, 1989] W-D. Weber and A. Gupta. "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture : Preliminary results". In *Proc. 16th Int. Symp. Comp. Arch.*, ACM/IEEE,(Jerusalem, Israel, May 28–Jun. 1). IEEE Comp. Soc. Press, Washington D.C., 1989, pp. 273–280.

[Wulf and Bell, 1972] W.A. Wulf and G.C. Bell. "C.mmp: A multi-mini-processor". In *Proc. Fall Joint Comput. Conf.*, AFIPS,. 1972, pp. 765–77. Vol. 42, Part 2.

Annotated Bibliography

Architecture

- G.S. Almasi, and A. Gottlieb.
  Highly Parallel Computing.
  Benjamin Cummings, Redwood City, CA., 1989.
  Models and architectures for parallel processing, with detailed examples.

- J.L. Hennessy and D.A. Patterson.
  Computer Architecture — A Quantitive Approach.
  Morgan Kaufman Publishers Inc., San Mateo, CA., 1990.
  Thorough treatment of all aspects of modern computer architecture, with emphasis on RISC architectures.

- R.W. Hockney, and C.R. Jesshope.
  Parallel Computers 2 — Architecture, Programming and Algorithms.
  Adam Hilger, Bristol, 1988.
  Introduction to parallel computer architecture, with particular emphasis on evolution of parallel computers, pipelining and multiprocessors.

- K. Hwang, and F.A. Briggs.
  Computer Architecture and Parallel Processing.
  McGraw-Hill, Singapore, 1985.

Introduction to parallel computer architecture, with particular emphasis in pipelining, SIMD and MIMD architectures.

□ R.N. Ibbett.
The Architecture of High Performance Computers.
Macmillan, London, 1982.
Architectural aspects of high-performance uniprocessors, with extensive description of the Manchester MU5.

□ J.S. Kowalik.
Parallel MIMD Computation.
The MIT Press, Cambridge, Mass., 1985.
Architecture and performance of the Denelcor HEP.

□ E.V. Krishnamurthy.
Parallel Procssing — Principles and Practice.
Addison-Wesley, Sydney, 1989.
Introduction to the principles of parallel processing from a more theorectcal viewpoint.

□ G.J. Lipovski, and M. Malek.
Parallel Computing — Theory and Comparisons.
John Wiley & Sons, New York, 1987.
Theoretical treatment of interconnection networks, includes cases studies of several current novel architecture projects.

□ D.P. Siewiorek, C.G. Bell, and A. Newell.
Computer Structures: Principles and Examples.
McGraw-Hill, Tokyo, 1982.
Extensive source of papers on early high-performance computer systems, although now somewhat outdated.

□ H.S. Stone.

High-Performance Computer Architecture.

Addison-Wesley, Reading, Mass., 1987.

Design aspects of high-performance computer systems, with particular emphasis on memory design, pipelining and interconnection networks.

Theory

□ A. Gibbons.

Algorithmic Graph Theory.

Cambridge University Press, Cambridge, 1985.

Introduction to graph theory and complexity of graph algorithms.

□ C.A.R. Hoare.

Communicating Sequential Processes.

Prentice-Hall, London, 1985.

Mathematical modelling of communication and concurrency.

□ R. Milner.

Communication and Concurrency.

Prentice-Hall, London, 1989.

Mathematical modelling of communication and concurrency using the calculus of communicating systems (CCS).

□ W. Reisig.

Petri Nets — An Introduction.

Springer-Verlag, Berlin, F.R.G. 1982.

The introduction to the theory of Petri nets.

Simulation

□ M.H. MacDougall.

Simulating Computer Systems — Techniques and Tools.

The MIT Press, Cambridge, Mass., 1987.

Introduction to construction of statistical models and simulation of computer systems.

"There must be a beginning of any great matter, but the continuing unto the end until it be thoroughly finished yields the true glory."

— SIR FRANCIS DRAKE (1540?–1596)

Dispatch to Sir Francis Walshingham, 17th May 1587.

Navy Records Society, Vol. XI (1898)