Superscalar Performance in a Multithreaded Microprocessor

by

Bernard Karl Gunther, BE (Hons)

Department of Computer Science

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

UNIVERSITY OF TASMANIA
HOBART

December 1993
Statement of Originality

This thesis does not contain any material which has been accepted for the award of any other higher degree or graduate diploma in any tertiary institution. To the best of my knowledge and belief, this thesis does not contain any copy or paraphrase of material previously published or written by another person, except when due reference is made in the text of the thesis.

[Signature]

B. Smith
Access and Copying

This thesis may be made available for loan and limited copying in accordance with the Australian Copyright Act 1968.

19/10/94
Abstract

Multithreaded processors, having hardware support for the concurrent execution of fine-grained threaded computations, are noted for their latency tolerance and low-cost synchronization. Multithreading is a technique for improving the utilization of processing elements (PEs) in parallel processing systems, thereby reducing cost/performance ratios. With increasing integrated circuit densities it is becoming feasible to integrate several PEs onto a single die, and further diminish the physical dimensions of parallel systems. However, by eliminating the artificial on-chip PE boundaries and sharing expensive resources in a more tightly coupled multithreaded architecture, even greater performance can be achieved from similar hardware.

A multithreaded processor architecture (Concurro) was designed for possible microprocessor implementation with the objective of multiple instruction issues per cycle—sustained superscalar performance—by means of multithreading. This thesis considers the trade-offs necessary for such architectures to achieve high throughput and hardware utilization under scalability and cost constraints. A detailed simulation study was carried out to characterize the architecture and evaluate the impact of implementation decisions. The key to efficiency in Concurro is asynchronous, zero-time context switching among a limited set of contexts, promoting effective use of the storage hierarchy. A 64-bit, register-based, load/store instruction set architecture is augmented with thread manipulation primitives and I-structure synchronization operations. Novel cache architectures and controller algorithms were designed for enhancing latency tolerance in the processor, while maximizing utilization of the most costly resources.

When tested on a variety of numerical and integer workloads, Concurro was able to sustain superscalar instruction issue rates for multithreaded operation, yet showed scalar RISC performance on single-thread code. Even with a simple threading strategy it was frequently possible to extract full utilization from functional units or the instruction cache. The architecture showed size scalability to an order of magnitude while remaining binary
compatible across these configurations. Performance of large configurations was shown to be limited ultimately by the bandwidth available from critical shared resources. With an appropriate memory system Concurro attained supercomputer-level floating point throughput operating out of uncached memory. The hardware requirements for this performance are expected to be comparable with those of VLIW machines with similar datapaths.
Acknowledgments

I thank my supervisor, Arthur Sale, for his continued confidence in me, despite my occasional meanderings, and for teaching me the value of independent and critical enquiry. I am indebted to John Morris for reading my drafts with care and enthusiasm, and returning them with valuable feedback. Andrew Partridge's reviews and insights are also very much appreciated.

I am grateful to Carl Lewis for the initial development of the assembler, which helped in getting the first simulations started. Thanks to John Parry for his efforts with the lcc compiler back-end.

The support of many colleagues from the old GCSB will be long remembered. I thank Ed Kazmierczak, Simon Milton, David Wright, Andrew Partridge, Jo Jordan, Tony Dekker, and Ben Lian for their companionship and advice, and for sharing with me their spirit of discovery. In particular, I wish to thank David Wright for introducing to me some fascinating perspectives of computer science.

I am especially grateful to my mother for her selfless help, patience, and interest. I thank her for instilling in me the courage to pursue my goals.

B.K. Gunther
Hobart, 1993
To my mother,
Eva.
Contents

Statement of Originality ii
Access and Copying iii
Abstract iv
Acknowledgments vi

1 Introduction 1
  1.1 Multithreading .......................................................... 2
  1.2 Related Work .......................................................... 11
  1.3 This Thesis ............................................................. 24

2 Core Architecture 26
  2.1 Instruction Set Architecture ....................................... 27
  2.2 System Architecture .................................................. 36
  2.3 Processor Organization ............................................. 38
  2.4 Instruction Pipeline ................................................... 48

3 Subsystem Architecture 52
  3.1 Context Units ............................................................ 53
  3.2 Instruction Cache ....................................................... 59
  3.3 Functional Units ........................................................ 63
  3.4 Data Cache ............................................................... 67

4 Experimental Method 75
  4.1 Simulation Technique ................................................ 76
  4.2 Benchmark Programs ............................................... 79
  4.3 Performance Evaluation ........................................... 86

5 Organization Alternatives 88
  5.1 Scalability ................................................................. 89
  5.2 Instruction Cache Duplication ..................................... 96
  5.3 Multiprocessing ........................................................ 99
  5.4 Implementation Cost ............................................... 103
  5.5 Observations and Conclusions .................................. 105
6 Instruction Fetch and Dispatch Strategies 107
   6.1 Instruction Fetching and Scheduling ........................................... 108
   6.2 Branching ....................................................................................... 117
   6.3 Instruction Cache Performance .................................................... 122
   6.4 Observations and Conclusions ...................................................... 127

7 Execution Resources 128
   7.1 Utilization ...................................................................................... 129
   7.2 Latency Tolerance ......................................................................... 135
   7.3 Datapath Bandwidth ..................................................................... 141
   7.4 Data Cache Performance .............................................................. 148
   7.5 Observations and Conclusions ...................................................... 155

8 Conclusion 157
   8.1 Major Microarchitectural Features .............................................. 158
   8.2 Reduced-Cost Designs ................................................................... 160
   8.3 Future Work .................................................................................. 162

A Concurro Instruction Set 164

B Synchronization Controller Microprogram 172

C Processor Configurations 176

Bibliography 180
Chapter 1

Introduction

Multithreading is introduced in its traditional context of multiprocessing. The evolution of parallel machines, both coarse and fine-grained, is heading towards implementations placing greater reliance on massive integrated circuits to reduce physical constraints. Multithreading in a microprocessor takes advantage of this opportunity, maximizing hardware utilization and gaining performance from appropriate processing paradigms. This strategy is developed into a new multithreaded architecture, Concurro, which serves as a research vehicle for massively integrated processors. Concurro borrows from superscalar, dataflow, and conventional multithreaded architectures in a design that aims to be implementable despite ambitious performance goals.
1.1 Multithreading

Multithreading was once considered merely an optimization for concurrent processing systems. Now, in an era when computers are approaching hard physical design limits, multithreading is rapidly becoming an essential technique for building viable parallel processors. The primary goal of multithreading is to multiplex concurrent threads of parallel programs onto a shared processor in order to maintain its high utilization under conditions where the processor would otherwise idle waiting for long-latency operations to complete. The realization of this goal is, of course, contingent on the processor's workload containing adequate parallelism.

A broad definition of “thread” will be used here. Essentially, a thread is any schedulable task, larger than a single instruction, with state—also known as a lightweight process. Thus, a thread serves merely as a substantial entity for scheduling. Depending on their implementation of multithreading, various researchers have narrowed the definition of threads by imposing resource, concurrency, re-enterancy, or synchronization constraints. Unlike a conventional process, a thread may coexist with other threads in a common virtual address space, sharing other threads' code and data without restriction. Therefore the architectural support for multithreading is focused on computational state management rather than on storage and exception management.

A multithreaded processor provides hardware support for enhancing the efficiency of multithreading—through maintenance of thread state in registers and automatic scheduling of thread execution. The use of hardware mechanisms is designed to enable multithreaded processors to tolerate the widest possible range of latencies, extending down to the small. Tolerance of even low-latency operations implies the ability to synchronize and switch context cheaply, giving multithreaded processors the significant advantage of executing fine-grained parallel programs with high efficiency.

In recent times, concerns over efficiency have brought about a merging of the dataflow and multithreaded models of computation [PaTr91]. An enduring difference between them is that multithreading permits control-driven scheduling in addition to the data-driven scheduling of dataflow; the more flexible scheduling policy of multithreading offers practical advantages by way of optimizing hardware utilization and allowing extended computational state.

Arvind and Iannucci [ArIa87] argued that traditional von Neumann proces-
sors offer a poor base for scalable general-purpose multiprocessors due to the high synchronization and context switch costs associated with von Neumann processors. Arvind and Iannucci identified long memory latencies and synchronization delays as the fundamental impediments to efficiency in multiprocessors. While von Neumann machines deal with these issues by avoiding them, multithreaded processors—and especially dataflow machines—are specifically designed to overcome efficiency losses caused by latency and synchronization.

Processor utilization is more than just an engineering figure-of-merit. The size of real parallel systems, built from processors and network components having real monetary costs, are constrained to the limits set by generation scalability and economic returns; this principle applies broadly to all scales of parallelism, from massively parallel microprocessor ensembles to small supercomputer clusters, promoting the efficient use of hardware. Ultimately though, utilization affects scalability—the improvement of performance by adding hardware resources to a machine; the inability to arbitrarily scale parallel machines limits the maximum performance attainable from them, and in some pathological cases deprives their users of speed advantages.

1.1.1 Multithreading in the Small

One way to counter the pressures of greater system cost and growing (relative) latencies in multiprocessors is to maintain modest physical network sizes while improving processing element (PE) performance. Fortunately, microprocessors continue to improve their price/performance ratios, thus encouraging this solution. With increasing processor clock speeds locality becomes an important issue, mandating some form of hierarchical organization, such as PE clustering in the Stanford DASH [Leno+92]. While multithreading, as in Alewife [Agar+93] and *T [NiPA92], should allow high-performance PEs to operate at almost normal utilization in a large multiprocessor, it cannot compensate fully for the inexorable lengthening of critical paths that occurs as cycle times diminish relative to network propagation delays and remote memory latencies. Parallel machines are, in terms of the PE's time frame, expanding.

In light of this situation, a number of architects have suggested that the physical shrinkage of multiprocessors is inevitable and, in the near future, feasible because of the increases in integrated circuit densities [Gwen92, MiCh92]. Their proposal is straightforward: place multiple PEs onto a single die, thereby gaining the advantages of on-chip communications and lower component counts in systems. Whether PEs are separate from memories or integrated alongside them, this proposal allows the integration
of PEs to follow the density advances of RAM. Cached PE clusters or multithreaded PEs would still be implemented, albeit on a smaller scale. Bringing a PE cluster on-chip improves hardware utilization both directly and indirectly, through lower average latencies. This approach offers, arguably, more flexibility and better scalability than building "super PEs" of the very long instruction word (VLIW) or superscalar kind.

Let us turn our attention to a system implemented on a single die. In itself, this constitutes a parallel processor implemented, possibly, as an array of RISC macrocells sharing a common bus interface. However, the concerns of utilization apply equally well at this scale of implementation. If we are to minimize PE size and cost/performance of the whole system, utilization must be optimized—regardless of how cheap integrated circuit hardware becomes. The key step in achieving better utilization of the hardware is to dissolve the artificial PE boundaries, both physical and architectural, that remain after integrating PEs together on a die.

Binding the PEs into a closer, more cooperative arrangement implies some measure of resource sharing in the system. The difficulty of optimizing utilization in this case lies partly in the different costs and usages of the various units comprising a processor. For example, if both a data cache and an integer ALU were shared between two processors, it is probable that the ALU would become a bottleneck, resulting in sub-optimal overall utilization by depriving the cache (an expensive resource) of work. Thus, resources must be shared on an individual basis, in a manner appropriate to their cost and usage; ideally the inevitable bottlenecks that do occur should be confined to the most costly components.

Organizing \( n \) PEs to share resources, together with the need for latency tolerance in a multiprocessor environment, leads naturally to a multithreaded architecture for a new machine. However, if this machine is to match the potential performance of the \( n \) PEs that it replaces, it must provide not only multithreading, but also superscalar performance, that is, multiple instruction issues per cycle; most conventional multithreaded processors, by contrast, issue only one instruction per cycle and so are unable to exploit fully inter-thread parallelism. Given the possibility that each of the original PEs could itself be multithreaded conventionally, the requirements of the new processor are that it achieves an aggregate throughput greater than that of \( n \) multithreaded processors while costing no more to implement.

The multiprocessing argument above can be considered a top-down approach to multithreading in the small. Alternatively, the development of high
performance uniprocessors provides a bottom-up perspective. In an effort to escape the limitations of circuit technology, computer architects have resorted to exploiting instruction-level parallelism for improving uniprocessor performance. Techniques such as integrating large first-level caches, superpipelining, long instruction words, and superscalar instruction dispatch leverage hardware complexity against utilization to gain performance. With microprocessors containing upwards of 10 million transistors in the near future, will it remain cost effective to continue in this way?

The current popularity of superscalar implementations can be traced to the requirement of backward-compatibility and the relative ease with which machines to date have been able to break the Flynn bottleneck [HePa90]. Further advances, however, may come at significantly greater costs, as techniques such as branch prediction and speculative execution yield diminishing returns and prolong verification [John89, Uht93]. Compilers, too, will have to cooperate in realizing the potential of such machines, diluting the software advantage superscalar designs hold over VLIW and vector architectures. Even with adequate instruction-level parallelism available from applications, the expression of naturally multithreaded programs as single threads places an unreasonable burden on both hardware and compilers to extract large amounts of parallelism efficiently. Just as the development of RISC arose from engineering concerns over optimality, it would seem appropriate to re-evaluate what architecture/compiler combination for fine-grained parallelism provides best performance at practical system complexity.

While VLIW processors offer scalability more readily than superscalar machines [Colw+87, Fish83, RYYT89], traditional VLIW architectures are beset with several practical problems. The difficulties of code generation for a horizontal architecture [Elli86] along with the single threaded model of computation impede efficient use of wide datapaths, although software pipelining is a particularly successful compilation strategy for vectorizable loops [Lam88, RaGP82]. Less significant, the lack of object code compatibility across different machine configurations hampers software portability and ties each machine to particular compilers. The VLIW approach to fine-grained parallelism has demonstrated merit, but it lays bare the hardware to such an extent that many run-time optimizations become inaccessible.

Regardless of what microarchitectural techniques are employed to exploit instruction-level parallelism, the evolving demands on memory systems and trends in RAM technology are expected to shape future architectural development. For processors capable of issuing several instructions in each cycle, the widening of the processor-memory gap results in cache misses
costing increasing numbers of potential instruction issues [JoWa89]. Such a reduction in processor utilization puts to waste much of the hardware and effort committed to sustain the peak instruction issue rates. With microprocessor-based systems gaining widespread use for large dataset applications—such as numerical modelling, image processing, and computer-aided design—processor performance is frequently becoming limited by main memory characteristics, and is suffering from over-reliance on multi-level caching.

The technological trends outlined above have implications for the development of future high-performance uniprocessors. Firstly, the single threaded model of computation cannot be expected to provide highly scalable architectures at moderate costs. Backward-compatibility and application experience may, eventually, have to be sacrificed in the pursuit of greater performance. Secondly, the growing penalty of accessing off-chip memory as well as the desirability of bypassing cache for scientific applications require future processors to be tolerant of memory latency if they are to fully realize the potential of fine-grained parallelism. By adopting a multithreaded architecture for uniprocessing, we have a practical solution to these problems without departing radically from traditional von Neumann computing. The important requirement for such a multithreaded machine, as in the multiprocessor case, is that it provide superscalar performance in a single-chip design. Multithreading in the small merges the concepts of multiprocessing and uniprocessing to gain advantages of both.

1.1.2 A New Multithreaded Processor Architecture

The notion of multithreading with superscalar performance in a massively integrated microprocessor is embodied in a new processor architecture called Concurro. Concurro was conceived as, principally, a high-performance uniprocessor architecture; its possible role in multiprocessing falls outside the scope of this thesis. In later chapters we will explore the design space for Concurro and use the architecture as the subject of a feasibility and performance evaluation. Let us first consider some fundamental issues relating to the design and use of Concurro and architectures like it.

The previous arguments made a case for multithreaded processors, but many of the same benefits are available from dataflow architectures. Why not adopt dataflow for Concurro? Pure dataflow processors are relatively accessible compilation targets [ArNi89, ArNi90] and offer almost ideal scalability from multiprocessors [HCAA92], but are plagued by practical inefficiencies that prevent them from competing with conventional von Neumann machines. To combat these difficulties, recent dataflow multi-
processors, such as Monsoon [PaCu90] and EM-4 [Saka+89], have intentionally restricted the full generality of the dataflow model by executing aggregated dataflow nodes to gain the locality advantages of implicit sequencing. While this hybrid approach—converging on multithreading—has made dataflow computing feasible, it has not completely solved a persistent and important problem of high-speed computing: maximizing the effectiveness of the storage hierarchy.

In their examination of dataflow and unrestricted multithreaded multiprocessing, Culler, Schauer, and von Eicken [CSvE92] have suggested that the gains in latency hiding and synchronization provided by dataflow are, in real implementations, largely undermined by a scheduling policy that destroys and restricts caching in the storage hierarchy. By allowing unrestricted task creation with a large synchronization namespace and relegating all scheduling to hardware, dataflow processors are prone to poor locality of memory references and must avoid the use of fixed register files. The absence of long-lived state in dataflow computation further limits the exploitation of the upper levels of the storage hierarchy.

RISC processors rely on large, general-purpose register files to achieve high instruction issue rates [Henn84]; superscalar designs, in particular, would be infeasible if their primary source of instruction operands were cache memory. Registers provide, principally, the highest possible level of bandwidth in the storage hierarchy. For high-performance processors this level of bandwidth is uneconomical, if not impossible, to provide in caches with sub-cycle latency. Accordingly, the associative matching stores or cached frame stores of unrestricted multithreaded processors are ruled out for practical reasons in processors capable of sustaining multiple instruction issues per cycle.

In recognition of the data bandwidth requirements for superscalar performance, Concurro is equipped with multiple sets of general-purpose registers—one register file per running thread. The contents of a register file and an instruction pointer define a thread's state, which is termed a context of the processor. The number of register files that can be implemented is limited by cost and performance constraints, therefore the number of contexts resident in Concurro is limited. This undoubtedly places restrictions on the program, since the forking of threads cannot go unchecked. It is the responsibility of software to either manage context switching when more threads are required than can be held resident in the processor, or simply contain the maximum parallelism of the computation to that provided by the hardware; Concurro itself does not maintain a separate, unlimited pool of tasks across which the processor switches automatically.
On the surface it would appear that providing hardware support for the virtualization of multithreading should improve efficiency, if not only programming convenience. Indeed, a hardware mechanism may well be able to transfer contexts to and from an adjacent level of the storage hierarchy faster than software could alone. But this neglects the hidden costs of secondary data movement associated with cache misses resulting from indiscriminate scheduling [CSvE92]; software-controlled virtualization can reduce these costs by not moving data at all or using static information to minimize the amount of context transferred on switches.

Its adverse impact on ease of use notwithstanding, restricted multithreading in Concurro encourages programmers to exploit the top level of the storage hierarchy—the registers—and allows a compiler to apply its high-level knowledge of programs to improve their scheduling and minimize their resource requirements. These represent opportunities for increasing per-thread performance to the levels attainable from conventional RISC uniprocessors. An established compilation framework for obtaining these optimizations is the Threaded Abstract Machine (TAM) of Culler et al. [Cull+91, CGSvE93], which exposes the fixed physical processor resources and defines a scheduling policy that works in harmony with the storage hierarchy.

Even without a general compilation strategy, such as TAM, it is possible to generate efficient code for a practically important class of parallel programs: programs containing vectorizable loops. Using well understood dependency analysis techniques [ZiCh91], parallel loops may be identified and their bodies coded as threads, which are scheduled at run-time by a control thread (see Chapter 4 for an illustration). Code generation involves combining newly compiled code with hand-tuned template code that manages iteration control and thread synchronization. The resulting programs make good use of registers and run in the confines of a fixed-size context set. This approach is applicable to a wide range of data-parallel operations, including scatter/gather, parallel prefix, and reduction operations.

Given finite hardware for processor contexts, the number of register files (or maximum number of active contexts) must be traded-off against the number of registers per file. A small set of large register files allows the most intra-thread optimization, but can offer inadequate latency tolerance, whereas a large set of small register files improves latency tolerance, but sacrifices per-thread performance and incurs a substantial hardware overhead. In Concurro the requirement for a maximum of \( C \) instructions to be issued in parallel from different threads sets the lower bound on the number of register files to \( C \). This minimum number of contexts, however, cannot
be expected to sustain a high instruction issue rate in the presence of long-latency operations, such as main memory loads, therefore the number of register files in a practical machine should be greater than $C$. Providing latency tolerance along with superscalar performance in a fixed-context processor would appear to require a great many contexts.

Consider the simple model of scalar multithreading presented by Culler et al. in [CSvE92]. It is assumed that main memory loads have a mean latency of $L$ cycles, and are executed, on average, every $R$ cycles. Therefore, a single threaded processor operating for $R$ cycles before issuing a load and waiting on it experiences an average utilization of $U_1 = R/(R+L)$. Suppose that a multithreaded processor capable of issuing one instruction per cycle and supporting $v$ contexts incurs a context switch and synchronization cost (for split-phase loads) of $S$ cycles. The processor is operating in saturation when $v$ is sufficiently large for the latency of a load by one thread to be completely hidden by the work of the other $v-1$ threads; thus, $(v-1)R+vS \geq L$ for saturated multithreading, and the processor utilization is $U_{\text{sat}} = R/(R+S)$. An intermediate mode of operation occurs when $v$ is insufficient to hide the entire load latency, reducing scalar utilization to $U(v) = vR/(R+L)$; this is called, appropriately, linear mode multithreading, since utilization is proportional to the number of contexts. More detailed analytical models of multithreading can be found in [SaCE90, Agar92].

In Concurro the minimum number of contexts required for saturation is increased by $C$ times over that required for scalar saturation. Thus in the worst case, setting $S$ to zero, $v \geq C(1+L/R)$ to achieve saturation. With $L/R$ expected to be in the neighbourhood of 10, it is readily apparent that quite unreasonable values for $v$ are necessary to support issue rates of several instructions per cycle. Rather than implementing $v$ register files, which are likely to be greatly restricted in size, we can instead support $v$ virtual contexts by building a small set of relatively large register files and sharing each of them among several subcontexts. This strategy reduces not only hardware overheads, but also software overheads by allowing variables to be shared across related virtual contexts.

Under the constraints of fixed-length instruction encoding, Concurro defines 16 general-purpose registers for each context; another 16 registers are globally accessible by all threads, and are properly part of the register set. The number of subcontexts into which each context can be partitioned is, of course, variable, but 16 registers are expected to allow up to 4 subcontexts per context in many situations. Although subthreads may be introduced into programs on an ad hoc basis, most subthreads occur as by-products of techniques for maximizing instruction-level parallelism, such as trace
scheduling [Fish81], superblock scheduling [Chan+91], and loop unrolling. To tolerate latency it is necessary for threads to be statically scheduled in such a way that long-latency operations of component subthreads overlap. Generally this involves no more than generating code for the subthreads as convenience or dependencies dictate, and list scheduling the instructions as if optimizing for a pipelined processor (average latencies can be assumed by the scheduler).

The thread scheduling policy of a multiple-context processor lies between two extremes: interleaving—that is, context switching on every instruction—as in HEP [Jord83] and P-RISC [NiAr89]; and blocking—executing each thread to completion or suspension before switching to another context, as in Alewife and *T. Interleaving can potentially lower context switch costs and, given enough active contexts, completely hide the latency of even intermediate length operations; the major disadvantages of interleaving are the occurrence of pipeline bubbles or busy waiting, and poor single threaded performance. The blocking scheme performs fewer, but generally more expensive context switches, amortizing their cost over several instructions; this usually leaves only pipelining to hide the cost of intermediate length operations, but ensures optimum single threaded performance. Laudon, Gupta, and Horowitz [LaGH92], in comparing these two schemes directly, found the improved latency tolerance of cycle-by-cycle interleaving gives it somewhat better performance than blocking.

Thread scheduling policies from both ends of the spectrum are desirable in Concurro. With a strictly limited context set it is important that processor utilization be maximized in order to sustain superscalar issue rates. Accordingly, low context switch cost and latency hiding across a large range are necessary, which are strengths of pipeline interleaving. However, a truly general-purpose machine must provide acceptable single threaded or linear multithreaded performance, favouring the blocking approach.

Concurro takes advantage of both schemes by using data-driven asynchronous scheduling that adapts dynamically to the requirements of the workload. Threads with instructions enabled through available data contend for instruction issue whenever possible, with fair arbiters selecting up to $C$ ready threads for instruction issue, forcing the other ready threads to wait their turn; inactive contexts or threads blocked waiting on data do not participate. The processor microarchitecture must, accordingly, be geared to the demands of interleaving. In saturated multithreading, asynchronous scheduling emulates pipeline interleaving, whereas for less parallel workloads blocking behaviour develops. Such flexibility can improve the performance of virtual context multithreading by adjusting for imperfections of
static thread schedules. Naturally, asynchronous scheduling is most feasible
in a fixed-context processor, but even then, as indicated in Chapter 2,
some practical compromises must be made to render it feasible.

With a fixed set of resident contexts comes the opportunity to minimize
synchronization costs. Concurro uses register-based synchronization uni­
formly, treating the synchronization of split-phase loads and the simplest
of instructions identically. Split-phase loads include synchronizing loads
and loads from main memory. Synchronization is enforced—at minimal
cost—through register scoreboards [CDC71], which maintain a presence
flag for each context register, somewhat like in the HEP. Unlike HEP,
though, presence flags in Concurro do not impose busy waiting because
asynchronous scheduling disqualifies blocked threads from arbitration,
allowing the regular pipeline interlock mechanism to be used in synchro­
nizing efficiently on split-phase loads.

1.2 Related Work

The concept and architecture of a Concurro-like processor were first intro­
duced by the author in 1990 [Gunt90a]. Many of the ideas contained in
[Gunt90a] were subsequently developed and adopted in the design of
Concurro before the appearance in the literature of a number of related
architectures. The development of both the macroarchitecture and micro­
architecture presented in this thesis was, however, necessarily influenced
by existing single threaded architectures (notably, microprocessor RISC,
vector, and VLIW) in addition to various multi-stream and multithreaded
architectures.

In the following sections we examine a number of approaches to multi­
threading that relate closely to Concurro and have evolved under many of
the same influences and goals that led to the architecture developed here.

1.2.1 Multithreaded Architectures for Multiprocessing

The traditional niche of multithreaded processors has been in high-
performance multiprocessing. As processing elements, these machines are
designed primarily for latency tolerance and flexibility, with single threaded
performance as a secondary consideration. In those machines where network
access frequency can be minimized—through caching or loop-back to local
memory—context switches need not be especially fast, since the dominant
latency to be hidden is relatively large. The highest performance processors,
however, are designed for frequent remote references, requiring context
switching to be handled in hardware alone so as to achieve the required speed.

**Sparcle—the Alewife Processor (Agarwal et al., 1993)**

Sparcle, a recent multithreaded microprocessor with multiple register files [Agar+93], is the processor of the Alewife machine—a cache-coherent, distributed shared-memory multiprocessor with up to 512 nodes. Each Alewife node consists of a Sparcle processor with floating point coprocessor, a communications and memory management unit (CMMU), a 64 kB cache, a 4 MB portion of the shared memory, a network router, and 4 MB of memory for code and local data. Sparcle was implemented by modifying an existing SPARC RISC processor design. The original 8 register windows of this processor were paired and used as 4 separate register frames, of 32 registers each, holding a trap/message handling context and three contexts for user threads. Instead of four program counters, Sparcle maintains one in hardware and exchanges its value with another inside a trap handler on each context switch. Sparcle's multithreading mode is blocking, with context switches occurring on cache misses and synchronization failures. With minimal hardware support the context switch trap handler can switch to another resident context in 14 cycles.

In addition to normal SPARC instructions, Sparcle implements instructions for expediting context switching, fast message handling, manipulating futures, and fine-grain synchronization through J-structures and L-structures. A J-structure associates a full/empty bit with a memory word for producer-consumer data synchronization; L-structures are similar, but have additional mutual exclusion properties. Synchronization failures registered by the CMMU cause it to interrupt Sparcle, which traps to a short routine that enqueues a load on the structure or completes deferred loads upon a write. Sparcle supports futures through the software convention of using deliberately unaligned pointers to placeholders (values of future expressions). Therefore, accessing a placeholder incurs an address alignment trap, allowing the trap handler to either evaluate the future or suspend the thread. Sparcle's interface to the interconnection network is via the CMMU, which has direct access to the network queues and is responsible for message dispatch, reception and DMA. The performance of a full-scale Alewife system remains to be seen, but even with minimal hardware support for multithreading Sparcle should be able to sustain scalar RISC processing rates on fine-grained parallel workloads.

**The Tera Computer System (Alverson et al., 1990)**

The Tera computer system [Alve+90] is a more ambitious multithreaded multiprocessor. Alverson et al. expect the largest configuration of Tera to
have 256 processors with a cycle time of less than 3 ns. The processors and 512 data memory units are interconnected by a fully pipelined network of 4096 nodes, arranged as a $16 \times 16 \times 16$ toroidal mesh with node links in one dimension removed on alternate planes. A Tera processor can multithread on as many as 128 contexts, all of which are resident in the machine. Each running thread has dedicated to it a program counter, status word, 32 64-bit general-purpose registers and 8 64-bit target registers. A thread executes a single instruction to create a new context, which receives from its parent a new program counter and status word, a trap target, and values for three general-purpose registers. This inheritance of context is often sufficient for newly created threads to begin executing useful work immediately. Threads terminate themselves explicitly.

On each cycle a processor selects for instruction issue one thread from among all of the threads that are ready to execute. Pipelining permits a new instruction from a different thread to be issued in every cycle. Competitive operation rates are obtained through horizontal instructions containing three operations—a memory access, an arithmetic operation, and a control operation or second arithmetic operation. Under usual computational intensities horizontal instructions are expected to provide floating point throughputs of around one operation per cycle per processor. With no data caches and a short cycle time, average latencies of some 70 cycles need to be tolerated by the processor. To avoid relying on 70 or more running threads for latency tolerance, Tera allows up to 8 instructions from each thread to be in execution simultaneously. The architecture uses explicit-dependence lookahead rather than scoreboard to observe instruction dependencies; each instruction contains a 3-bit field, set by the compiler, which specifies how many following instructions in the stream are independent of the current instruction and thus free to issue without waiting.

Each 64-bit datum in Tera’s memory is tagged with 4 access state bits. Two of the tag bits may be used by an application for lightweight traps; the forward bit indicates an access indirection, while the full/empty bit is used in synchronization. Loads and stores can take advantage of full/empty bits to enforce producer-consumer synchronization or mutual exclusion. However, the frequently used atomic fetch-and-add operation is executed at the memory units to improve its efficiency. If synchronization on a memory location fails, the access may be re-tried at most a limited number of times, after which a trap is generated to handle the access in software. With the Tera computer system being still in development, Alverson et al. were not able to provide an indication of its performance.
P-RISC (Nikhil and Arvind, 1989)

In the P-RISC architecture Nikhil and Arvind [NiAr89] combined dataflow and von Neumann features on a RISC base. A P-RISC multiprocessor is a distributed shared-memory machine, containing separate heap memory elements and PEs linked by some interconnection network. Data-level synchronization is provided through I-structure storage at the heap memory elements. Local to each PE is instruction and frame storage, holding executable code and function frames (tree of activation records), respectively. PEs are multithreaded RISC processors, performing pipeline interleaving for maximum utilization. The essential state of a thread is captured by a continuation, \(<FP, IP>\), which pairs the thread's instruction pointer (IP) with its frame pointer (FP). FP, although a genuine memory pointer, is normally aliased to a local register set, turning the FP-relative addresses of P-RISC's three-address instructions into simple register specifiers.

A large number of continuations circulate through the machine in a circular fashion. Continuations pulled out of a token queue pass through the instruction fetch, operand fetch, execution, and (possibly) result write stages of the main pipeline before normally re-entering the token queue with the IP field incremented (or altered on jumps). In the case of loads from remote heap elements, however, the continuation does not re-enter the token queue immediately but leaves the processor and travels with the destination address and read message to the heap element, which responds some time later with the read data and the continuation for restarting the blocked thread. In this way busy waiting is eliminated and remote loads may be freely pipelined and reordered by the network. New threads are created with a fork instruction, which simply inserts an additional, new continuation into the token queue. A thread on a remote node may be passed a frame argument and enabled with the start instruction, whose behaviour mimics the response phase of a remote load. Two threads synchronize (and terminate) through a join instruction; two threads executing join in turn toggle a frame location, its contents determining which thread loses its continuation and which thread continues (the last thread to join). The fork and join instructions are sufficiently inexpensive to make a pure dataflow compilation strategy viable for P-RISC.

Sharma and Nikhil designed and simulated a P-RISC processor, PRISC-1 [ShNi89]. To achieve adequate operand bandwidth, up to 8 frames in PRISC-1 are cached in a set of register windows, each containing 32 registers. Register windows are backed by an external frame cache, which loads registers only on demand. In practice, misses in the register windows were sufficiently low for the average frame cache bandwidth (including register
write-throughs) not to exceed 1 access per cycle. An instruction queue allows instructions to determine the residency status of their operands up to 12 cycles in advance and initiate early register reloads to hide the frame cache latency. Continuations for threads referencing resident frames are cached in a 32-entry active token queue (ATQ), with overflow continuations residing in an external queue. The thread sequencing in PRISC-1 is modified from pure interleaving to increase the utilization of the relatively long pipeline under conditions of low concurrency. Consecutive instructions from a thread are fetched up to and including a potentially suspensive instruction, at which point another thread is dequeued from the ATQ. Under this scheme, Sharma and Nikhil observed pipeline utilizations of greater than 99% on several benchmark programs, for a frame cache latency of 4 cycles.

### 1.2.2 Multi-stream Processors

Multi-stream architectures are specializations of multithreaded architectures where threads participate in few, if any, interactions, and the invocation of threads has been effectively removed from the machine’s instruction repertoire. Although multi-stream processors execute threads, the persistence of these threads makes it more convenient to think of them instead as instruction streams. Multi-stream processors require minimal hardware for synchronization and thread manipulation, but the applicability of these machines is generally restricted to multitasking or coarse-grained parallel processing. These limitations are accepted in multi-streaming in order to fulfill its primary goal of maximizing processor utilization. Given the almost ideal conditions under which multi-stream processors can operate, studies into these architectures are valuable sources of limiting-performance data.

**Multi-stream Microprocessor (Kaminsky and Davidson, 1979)**

One of the early suggestions for multi-streaming in microprocessors was made by Kaminsky and Davidson [KaDa79]. They argued that multi-streaming made effective use of limited die area and reduced system cost by maximizing chip I/O utilization as an alternative to duplicating processors. For a CISC processor, as used for illustration by Kaminsky and Davidson, multi-streaming is particularly beneficial, since it allows relatively complex instructions—making frequent memory references—to be processed at a rate approaching the ideal throughput of a pipelined processor. Microcoded control is assumed for the processor, with a small, fixed number of streams (typically 4) being interleaved on a cycle-by-cycle basis in a circular pipeline containing a stream context in each pipestage.

Kaminsky and Davidson outlined how the resource and precedence require-
ments of the processor's whole instruction set can be used in designing a pipeline of minimal length. The resulting processor structure closely resembles the logical pipeline, with a set of context registers and dedicated functional units or memory interfaces at each stage. On each clock tick a context is simply shifted in its entirety to its adjacent set of registers. This organization maximizes the utilization of the memory and separate functional units, and with the inclusion of additional pipestages, allows modest, but constant, memory latencies to be masked. However, transporting all context registers through the pipeline imposes a practical limit on the number of general-purpose context registers. Kaminsky and Davidson suggested the use of a single accumulator, but also described a scheme for implementing virtual registers backed by common memory. The only performance data given for this machine was based on a highly idealized speedup model (assuming 100% memory utilization) and measured fault rates for the virtual registers on three benchmark programs.

**Multi-stream Cray-1 (Farrens and Pleszkun, 1991)**

Noting that pipeline utilization in the Cray-1 was less than half when executing the Livermore Loops, Farrens and Pleszkun [FaPl91] proposed multi-streaming on two processes as a means of reaching unitary instruction issue rates in the Cray-1. Through one process stealing the pipeline bubbles introduced by the other, it was hoped that both tasks could execute in almost the time taken normally to execute one of them.

Farrens and Pleszkun simulated a scalar processor model, similar to a Cray-1S, containing a set of registers and an issue unit for each of two streams, with the normal pipeline interlock mechanisms retained despite multi-streaming. The processor could be configured for different memory and branch latencies, and one or two result buses, either private or shared by the streams. Three schemes for interleaving the two instruction streams were examined: every-cycle, which toggles between the streams on every clock cycle regardless of the issuability of the instruction at the head of a stream; blocked, which switches to the alternate stream only upon encountering a pipeline bubble; and prioritized, which executes one process in preference, switching to the other only in the dead cycles of the first.

The scheduling schemes were simulated using two random concatenations of the first 14 Livermore Loops as the two processes. A clear performance advantage was observed for the blocking scheme, whereas every-cycle scheduling resulted in consistently poorest performance. It should be noted, however, that Farrens and Pleszkun's blocking scheme is unlike most blocking schemes, which switch context more coarsely or later in the pipeline to reduce implementation costs. Generally, speedups improved for longer
memory and branch latencies, reflecting the greater frequency of pipeline bubbling in these configurations. Performance also benefited from the reduced stream interaction of two result buses. The best speedup over serial execution was 1.89, recorded for blocking scheduling with 11-cycle memory latency and two private result buses; best speedups of 1.61 and 1.52 were obtained from the prioritized and every-cycle schemes, respectively. In terms of instruction issue rates these performances represent no better than 0.85 instructions per cycle, indicating that two streams are still not sufficient to saturate the pipeline on a scientific workload—particularly for more realistic main memory latencies.

**Multi-stream Superscalar Processor (Daddis and Torng, 1991)**

The multi-streaming strategies described above had the goal of attaining best scalar throughputs. In contrast, the objective of Daddis and Torng [DaTo91] was to obtain superscalar issue rates from two streams by interleaving them in a superscalar processor. The processor simulated by Daddis and Torng comprises instruction and data caches, an unlimited number of functional units, a program counter and register file for each stream, an interleaving fetch unit, and a central instruction issue unit that issues instructions out of an instruction window containing instructions interspersed from both streams. To distinguish between the streams, a preprocessor maps all register references into disjoint sets according to which stream instructions belong. The issue unit scans the instruction window for instructions free of data or register dependencies, and issues simultaneously as many independent instructions (from both streams) as possible. Completed instructions are deleted from the instruction window, which is then compressed to make room for instructions brought in from the cache in fixed blocks.

Daddis and Torng evaluated this machine and a single stream equivalent using the first 14 Livermore Loops and Dhrystone as benchmarks for numerical and general streams. With only a single stream the superscalar processor achieved issue rates of up to 0.80 and 1.25 instructions per cycle on the numerical and general workloads, respectively. With two streams in use it was possible to almost double these rates, given a sufficiently large instruction window and fetch sizes above 4 instructions. However, attaining speedups of over 1.8 on the numerical streams required an instruction window larger than 12 entries; implementing single-cycle instruction selection and superscalar issue over such large windows would most likely lose much of the issue unit's apparent advantage. Apart from instruction look-ahead, low datapath latencies (data caching and relatively shallow pipelines) allowed Daddis and Torng to improve on the issue rates obtained
by Farrens and Pleszkun. Since the superscalar processor was idealized, with no structural hazards, Daddis and Torn's results can be considered as indications of upper bound performance, rather than realistic performance, for this kind of architecture.

**Scalable Multi-streaming (Park, Fussell, and Jenevein, 1991)**

A more scalable architecture for superscalar multi-streaming was proposed by Park, Fussell, and Jenevein [PaFJ91]. Their design, not far removed from a single-chip multiprocessor, shares only the floating point functional units and the memory bus interface among all streams. Each stream is provided with private data and instruction caches, a branch target buffer, superscalar instruction decoder/issue unit, and a complete integer ALU. Since scheduling is for the most part localized, streams can execute quite independently with minimal interaction between them. Given the maximum of 4 streams suggested and the provision for parallel instruction issue (up to 3 instructions per cycle) from each stream, it appears that latency tolerance was not an important consideration for this architecture.

Park et al. simulated various configurations of the processor in detail, using sequences of 10 integer and floating point benchmarks for streams. Implementation costs of the configurations were also estimated in terms of die area; the die area of the Intel i860 RISC processor was taken as 1.0 units. As would be expected from the processor's organization, for adequate cache sizes performance followed the number of streams in almost direct proportion. Increasing per stream issue parallelism from 1 to 2 instructions per cycle typically improved throughput by 40%; allowing 3 issues per cycle added only 15% to throughputs, but significantly more to die areas. For the 4-stream, dual-issue configuration performance levelled at around 4.3 instructions per cycle when the die area reached 2.7 units; by contrast, a multiprocessor of the same performance would have occupied 4 units of area. Nevertheless, the substantial costs in providing each stream with private caches and an integer execution unit would suggest that even better cost/performance ratios are possible were these components shared.

### 1.2.3 VLIW/MIMD Multithreaded Architectures

Combining VLIW-style static scheduling with multithreading is a means of capturing instruction-level parallelism without sacrificing utilization of the often expensive resources devoted to VLIW processing. Multiple instruction issues per cycle is a fundamental goal for these architectures, which take on a number of MIMD characteristics in order to achieve it. An implicit assumption of these designs is that conventional latency hiding techniques, such as caching, prefetching, and relatively short pipelines,
will make horizontal parallelism a valuable addition to the vertical parallelism of pipelining. Accordingly, the number of contexts supported by these machines is generally significantly less than the numbers common to multithreaded multiprocessors.

**XIMD (Wolfe and Shen, 1991)**

In the XIMD (variable instruction stream, multiple data stream) architecture, Wolfe and Shen [WoSh91] combined the VLIW and MIMD concepts to achieve the performance and scheduling advantages of VLIW with the flexibility of independent threads in MIMD. Reflecting its VLIW basis, XIMD consists of a shared, multi-ported register file connected to a set of functional units. However, to enable separate threads to be executed simultaneously, all of the functional units are general-purpose with each controlled by its own instruction sequencer. A condition code flag and a synchronization flag associated with each sequencer/functional unit provides threads with a synchronization mechanism. Instruction memory is divided into independently addressable modules, one per sequencer, each of which contains a unique portion of the long instruction word. When every memory module is addressed identically, VLIW behaviour is emulated. To fork threads, one or more of the sequencers branches to different addresses; if several sequencers branch in unison then the threads themselves can execute in VLIW mode. Synchronization is achieved through setting synchronization flags and busy waiting for these to clear.

Wolfe and Shen simulated XIMD-1, a XIMD processor with 8 homogeneous functional units, each of which implements a set of single-cycle RISC operations including loads, stores, and branches. No general performance data for XIMD-1 was given, although detailed examples using code fragments indicate that fine-grained parallel programs could be executed with extremely low overheads. The cost of this flexibility, however, appears in the implementation of what amounts to multiple RISC processor cores, and (for XIMD-1) a 24-port general-purpose register file. Furthermore, the instruction and data memory modules are likely to require separate caches, whose sizes are necessarily restricted because of their multiplicity. These costs offset much of the utilization improvement brought about by multiple threads.

The XIMD architecture does indeed overcome the single-thread limitation of VLIW architectures, thereby broadening the applicability and performance of statically scheduled, synchronous MIMD processors. Nevertheless, with the architecture still exposed to the compiler, non-determinism must be dealt with pessimistically and the problem of object code compatibility remains unsolved. Ignoring implementation costs, the scalability of XIMD
appears limited only by its number of registers; recognizing this, Wolfe and Shen expect to use a global register file of 256 registers for a practical implementation of XIMD.

**Processor Coupling (Keckler and Dally, 1992)**

The XIMD approach to multithreading was expanded by Keckler and Dally [KeDa92] in a mechanism called processor coupling, which they described in the context of a node processor of the MIT M-machine. In its most general form, processor coupling multiplexes several statically scheduled, VLIW threads onto a shared set of functional units. The architecture is still exposed to the compiler, but the compiler is assisted through run-time scheduling. The processor proposed by Keckler and Dally comprises a number of clusters of functional units, a register file (assumed infinite for simulation purposes) and thread state registers associated with each cluster, a set of memory banks accessed via an interconnection network, and a unit interconnection network for transporting data between cluster register files. Instructions are distributed across multiple caches, as in XIMD, with each cluster maintaining pointers to the operation spaces of its resident threads. Intra-thread synchronization is obtained through register scoreboards, while presence bits on all memory locations provide more general synchronized communications (whose implementation was not discussed).

Each cluster buffers an operation from each thread. When an operation receives its operands, as indicated by the scoreboard, it proceeds to the register read stage to read them. Were it not for all threads sharing a common register file in each cluster, the scoreboard check stage could be performed in parallel with register read—as is normally the case. Once an operation enters the register read stage, its successor operation is transferred from a prefetch buffer into the operation buffer, and a new operation is prefetched from cache. To maintain the view of very long instruction words being executed in order, operations are not permitted to issue until clusters signal the issue of all operations of the preceding instruction for the thread in question. The operation is executed in one or more cycles, and its result is written to its destination register file in the final pipestage.

Keckler and Dally simulated this machine under various degrees of intra-thread parallelism and threading, ranging from single threaded mode with one cluster to fully coupled mode with multiple threads and unrestricted cluster use; allocation of threads and operations to clusters was performed statically. On four parallel numerical benchmarks, processor coupling with 4 homogeneous clusters and single-cycle memory provided typical speedups of 1.8 over single threaded, VLIW operation. These speedups, however, were achieved with multiple caches, 5-ported register files, and completely
duplicated functional units. Multiple threads in processor coupled mode also improved tolerance to memory latency, although the greater than 40% increases in run times for a 75% increase in mean cache access latency reveals the principal drawback of VLIW multithreading with a limited set of contexts.

**Dynamic VLIW Interleaving (Prasadh and Wu, 1991)**

Prasadh and Wu [PrWu91] examined VLIW multithreading under conditions of more restricted functional unit duplication. Prasadh and Wu's proposed architecture features a heterogeneous set of functional units, each of which is assigned to a VLIW operation; however, multiple instances of some of the units are provided to satisfy bandwidth demands. Each thread is allocated an instruction cache, instruction queue, branch target buffer, and register file. An (unspecified) interconnection network links the register files with the functional units. Instruction issue is controlled by a dynamic interleaving unit (DIU), which has the task of meshing together long instructions from different threads in order to maximize the utilization of functional units.

The DIU buffers the head of each thread's instruction queue. During the dynamic interleaving stage of the pipeline, an operation for each functional unit is selected independently from the buffered operations. Round robin selection ensures that empty operation fields of one instruction can be replaced by operations of other instructions without starving any particular thread. The vector of operations formed by the DIU is then issued to the functional units for execution. In-order issue of instructions is maintained by not refilling a DIU instruction buffer until all of its operations have been issued, allowing issue times of individual operations to slip backward. The execution latencies of the functional units are required to be fixed to ensure correctness of schedules. For cases such as cache misses, Prasadh and Wu outline a complex scheme for freezing the pipelines, removing the faulted thread, and then restarting the pipelines using saved state.

Prasadh and Wu simulated several configurations of this machine equipped with 8, 9, and 10 functional units, and infinite caches. The processor was multi-streamed, as distinct from multithreaded, on a selection of Livermore Loops and integer benchmarks. An issue rate of 0.5 instructions per cycle was observed on the Livermore Loops with one stream; with 4 streams and 8 functional units the issue rate reached 1.9 instructions per cycle—indicating almost ideal speedup, but poor utilization of the available bandwidth. The high costs associated with the VLIW contexts constrains their more extensive duplication for the purpose of improving utilization. Performance relative to the processor with 8 functional units increased by
less than 10\% for the larger configurations of 9 and 10 functional units, with the greatest gains occurring on 4 streams.

**Multithreaded MIMD Architecture (Hirata et al., 1992)**

Hirata et al. [Hira+92] proposed a multithreaded architecture for achieving parallel instruction issue without VLIW techniques. The processor comprises a number of thread slots that share the caches and functional units. Each thread slot contains a thread program counter, instruction decoder, and instruction queue, which is refilled from a common instruction cache (whose structure was not discussed). At run-time, each thread is allocated a 3-ported bank from a large, central register file. This organization allows thread slots to switch context upon cache misses; shorter and more frequent stalls, however, are simply endured. The penalty of centralizing the register files is reflected in a relatively extended pipeline with two decode stages. Special queues linking adjacent thread slots appear to be the only facility for synchronized communications between threads. Each of 7 heterogeneous functional units receives instructions and operands via its instruction schedule unit; the instruction schedule units, connected to all of the thread slots, are responsible for arbitrating instruction issue.

Thread slots can operate as independent processors, dispatching instructions in parallel to whichever instruction schedule units are free to accept them. Once the head instruction of an instruction queue has valid operands, as indicated by the thread slot’s register scoreboard, it is dispatched to a standby station (latch) associated with the thread slot at a selected instruction schedule unit. Every cycle, each instruction schedule unit selects, in a prioritized fashion, a waiting instruction for issue. The instruction selection priorities normally rotate automatically, but they may be rotated explicitly in software to implement static scheduling. Instruction results are eventually written to their destination registers and the scoreboards are updated.

Hirata et al. simulated the execution of a highly parallel ray-tracing program on this machine. With two load/store units, and 4-cycle, infinite caches, speedup factors of 3.72 and 5.79 relative to single threaded operation were obtained with 4 and 8 thread slots, respectively. Assuming a single-thread issue rate of 0.65 instructions per cycle (based on the superpipeline nature of the pipeline), these multithreaded results represent issue rates of between 47\% and 60\% of peak rates. With nothing but pipelining to hide execution latencies, dispatch voids must have occurred frequently. Experimenting with superscalar thread slots, Hirata et al. concluded that the most cost-effective arrangement restricted thread slots to at most one instruction dispatch per cycle.
1.2.4 Conclusions

Several of the multi-stream and multithreaded VLIW/MIMD architectures described in Sections 1.2.2 and 1.2.3 are capable of superscalar performance. However, despite their speedups over scalar machines, these designs generally fail to reach the utilization, flexibility, and economy of the multithreaded architectures used in multiprocessing.

Latency tolerance is particularly neglected in the multi-stream and multithreaded VLIW/MIMD architectures. Reliance on cache memory makes for high instruction issue rates from a small number of threads, apparently justifying the use of VLIW or superscalar issue units. Simulations involving idealized caches tend to support these designs—even on scientific workloads—as the effects of thread interference and bandwidth limitations in real caches are not experienced. In these machines, the latencies of floating point and other complex operations become significant, reducing actual utilization and allowing impressive speedups. Multithreading so as to treat every instruction as a potentially long-latency operation is advantageous under such conditions.

The difficult problems of sustaining multiple instruction fetches and data references in each cycle, and multiplexing datapath components have not been solved satisfactorily in most cases. Hirata et al. have proposed cost-effective solutions, but stopped short of detailed designs. What is at issue is not whether multiple caches and highly duplicated functional units are feasible in a future microprocessor, but whether we can improve cost/performance ratios by sharing expensive components. However, in optimizing resource allocation, duplication will be inevitable and, indeed, desirable for ensuring scalability. Caches, being the most expensive processor items, are the least attractive to duplicate; by sharing caches, we can afford to implement larger storage arrays and reduce miss rates as well as overhead costs. The chief technical challenge lies in obtaining adequate bandwidth from a shared resource.

To be accessible compilation targets, multithreaded architectures will need to carry thread abstraction as far as efficiency allows. A conventional, scalar interface and flexible synchronization mechanism are lacking in many of the proposed architectures in Sections 1.2.2 and 1.2.3. The multithreaded VLIW architectures benefit from some organizational simplicity, but incur the costs of added compiler complexity, object code incompatibility across machines, inefficient memory usage, and substantially enlarged processor contexts. The published architectures all cater for intra-thread synchronization; facilities for synchronized communications between
threads, however, are in many cases cumbersome or severely limited. In contrast, Concurro provides data-level synchronization in memory to make the exploitation of fine-grained concurrency both convenient and efficient.

The published work confines its evaluations of multithreaded processors to performance on parallel workloads. Performance under these conditions is undoubtedly important, but for a general-purpose computer its performance on less parallel and single threaded workloads is also of interest. Design decisions made to improve performance under saturated multithreading can be detrimental to performance on a general workload. The current study therefore considers a range of workloads when assessing performance.

1.3 This Thesis

This architectural study considers feasible approaches to obtaining superscalar performance from ULSI microprocessors through multithreading. Guided by the goals of scalable MIMD behaviour and the constraints of microprocessor implementation, the candidate Concurro architecture is developed and evaluated by detailed simulation. In simulating a variety of possible designs we gain characterizations of such architectures and insights into the issues involved in their implementation.

1.3.1 Layout

Rather than interspersing the design components with their experimental evaluation, the architecture and simulations are presented in separate chapters to minimize conflict.

The instruction set architecture and core microarchitecture of Concurro are presented in Chapter 2; the core microarchitecture is the common substrate on which alternative designs are based. The programming model, system environment, processor organization, and instruction pipeline are covered in this chapter. Chapter 2 builds on the foundation architecture proposed in this introduction. Moving to a greater level of detail, Chapter 3 elaborates on the design and interaction of Concurro’s key subsystems: context units, instruction and data caches, and functional units. Possible physical implementations of these units are proposed, along with descriptions of their variations for simulation purposes.

Simulation of the Concurro processor and the experimental procedures are described in Chapter 4. The design and features of the simulator and support tools are presented, as well as the experimental configurations of the simulator. This chapter also introduces and characterizes a set of
benchmark programs, and discusses their associated programming strategies.

The first of the experimental chapters, Chapter 5 is concerned with the scalability of Concurro, and the comparative performance of alternative superscalar and multiprocessor organizations. Chapter 6 examines the influence of instruction fetch and dispatch strategies on processor performance and scalability. Instruction fetch methods, instruction scheduling, branch behaviour, and cache performance are studied. In Chapter 7 attention shifts to datapath aspects. This chapter examines hardware utilization, latency tolerance, datapath bandwidth, and memory performance.

Chapter 8 concludes by assessing the strengths and weaknesses of the Concurro architecture, and summarizing the options for reduced-cost implementations. Finally, future work and directions are suggested.

1.3.2 Limitations of the Work

Beyond the usual limitations of a simulation-based study, the scope of this work has, by necessity, been restricted due to lack of local software resources. In particular the number and complexity of benchmark programs are less than ideal. This is principally due to the absence of a suitable compiler, although it is questionable as to whether current (dated) parallelizing compiler technology is more appropriate than hand-compilation in the evaluation of future new architectures. Without any parallel compiler or applications support it was impractical, for a relatively challenging compilation target such as Concurro, to pursue benchmark development beyond an assembly language level.
Chapter 2

Core Architecture

The design space for high-performance processors is extensive. Some of the many axes defining this space are instruction set architecture (ISA), implementation technology, processing throughput, and circuit complexity. To make this study of multithreaded microprocessors practicable, it was necessary to restrict the design space by assuming an implementation style and fixing the basic architecture, while still retaining enough degrees of freedom to allow exploration of the design space.

This chapter develops the core architecture of Concurro—the user-visible architecture and those aspects of the microarchitecture that remained fixed throughout the experimental work. Choices made in the design of the core architecture are not entirely arbitrary, but strongly influenced by likely implementations of the processor. Since integrated circuit technology continues to improve rapidly, feasibility was considered in the context of a mid-1990's implementation.
2.1 Instruction Set Architecture

The design of the ISA was motivated by requirements for size and generation scalability, low-cost implementation, and accessible performance. Providing scalability at low cost placed restrictions on instruction encoding, addressing modes, operational complexity, and control complexity. Rather than employing an existing instruction set that fit these requirements poorly, a new ISA was developed, which had few of the single threaded artifacts found in existing ISAs. Through programming experience with Concurro, the ISA was progressively refined by eliminating infrequently used instructions and adding instructions that decreased path lengths without contributing significantly to the cost of implementation. Appendix A lists the full instruction set of Concurro. The final outcomes of these trade-offs give the Concurro ISA many characteristics that Patterson [Patt85] has ascribed to RISC architectures: fixed-length instructions with predominantly simple semantics, register-register operations facilitating pipelining, and memory accesses through load and store instructions.

2.1.1 Execution Model

Concurro is a multiple context processor, with each user-visible context defined by its state variables—an instruction pointer (IP) and a set of local general-purpose registers. Globally accessible general-purpose registers, system registers, and memory are visible to all contexts. The state of the machine's pipelines and implementation-dependent registers are generally not manifest, nor specific to a visible context. This allows a variety of implementations that can maintain views of more than one thread executing concurrently. The maximum number of physical contexts available is fixed, although the programmer is discouraged from assuming a particular number; the actual maximum and number in use may be obtained from the processor status word register at run-time.

Each context executes instructions fetched from a common instruction memory, in which other contexts may execute the same code. Internal processor hazards are invisible to the programmer, therefore any possible code sequence constitutes a legal program. Threads appear to be executed sequentially. Implementations of the processor, however, may issue and complete instructions out of program order while guaranteeing correctness. A safe multithreaded program must use synchronization instructions to force a temporal ordering between executing threads.
Chapter 2 Core Architecture

The processor starts executing the root thread\(^1\) by default; when opportunities for exploiting parallelism in the program occur, the root thread can explicitly fork child threads, which in turn can fork children of their own. If the processor cannot allocate any more contexts to satisfy a fork, the instruction simply blocks until one of the other contexts terminates. This behaviour may deadlock the processor under the circumstance of a thread being prevented from terminating by a blocked fork. In practice deadlock was not a problem with careful coding, although Section 8.3 considers solutions for effecting recovery. There is no visible record of a thread's parent, and the state of a new thread is undefined apart from its IP, which was set by the parent. It is the responsibility of the program to coordinate the use of global resources and synchronization. Threads communicate with others through memory or register channels in a non-destructive fashion, that is, there is no join operation—threads must terminate themselves. A fork/terminate model was chosen over a fork/join model in order to minimize the frequency of context state destruction.

### 2.1.2 Storage

As indicated in Figure 2.1, all instructions are encoded in 32 bits. Control instructions have short address and long address formats. Instructions performing memory operations or arithmetic with constants use the immediate format, which takes a signed/unsigned 16-bit constant as the second operand. The remaining instructions, operating on integers or floating point numbers, specify a destination register and two source registers, with an extra opcode extension for some instructions. A functional class field is encoded in operational instructions to facilitate the rapid dispatch of instructions to a heterogeneous set of functional units. Programs are assumed to reside in a read-only memory segment where instructions are aligned to word boundaries. Since Concurro has a Harvard memory architecture [Furb89], programs that generate executable code must ensure cache coherency in software.

### Registers

Register files occupy the top level of the data-memory hierarchy. Registers are the basic operand sources and result sinks for instructions. Each context has access to 32 64-bit general-purpose registers, designated \(r_0\) to \(r_{31}\). Registers \(r_0\) to \(r_{15}\) inclusive are globally accessible by every context, while registers \(r_{16}\) to \(r_{31}\) are local to a context. As a convenience for programmers and implementors, \(r_0\) is hard-wired to zero, allowing it to be

---

\(^1\)The terms thread and context are used here interchangeably as concurrently running threads map one-to-one onto physical contexts.
used as a null source or non-destructive destination. Providing 16 local registers is a compromise between software performance and the cost of implementing and swapping the registers.

<table>
<thead>
<tr>
<th>Immediate Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 24 20 16 12 8 4 0</td>
</tr>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short Address Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 24 20 16 12 8 4 0</td>
</tr>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Long Address Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 24 20 16 12 8 4 0</td>
</tr>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Three-Address Register Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 24 20 16 12 8 4 0</td>
</tr>
<tr>
<td>0 0 0 X X X</td>
</tr>
</tbody>
</table>

**Figure 2.1** Concurro instruction formats.

**Register Channels**
High-speed argument passing and synchronization of threads can be performed through a set of 32 64-bit register channels [Gupt90], which is a special register file managed by one of the functional units. Each register channel is simply a register with an associated presence flag indicating whether the register is full or empty. All contexts can examine the channel flags to enforce synchronization on a channel. Register channels give a compiler the same opportunity for optimizing inter-thread communications that it has in optimizing data storage.

**Stored Data**
The ISA recognizes five data types: byte, short word, word, long word, and floating point, which have lengths of 8, 16, 32, 64, and 64 bits, respectively. As a consequence, memory addresses are always byte addresses pointing to the boundary of a datum; quantities larger than one byte are stored in memory in little-endian format—the least significant byte occupies the lowest address. Quantities from bytes to long words are assumed to be signed or unsigned integers, while floating point numbers are in the double-
precision binary format of ANSI/IEEE standard 754-1985 [IEEE85].

Data that is judged by the compiler to be accessed with high locality can be loaded and stored to the data cache. This cache and the instruction cache are backed by the common main memory, where program code and data co-reside. The virtual address space is specified in 32-bit byte addresses. The most significant bit of a virtual address indicates whether it belongs in user space or system space: hexadecimal addresses 0x00000000 to 0x7FFFFFFF, inclusive, belong to the user program, while the system software operates in the range of addresses 0x80000000 to 0xFFFFFFFF.

2.1.3 Memory Instructions

Instructions to access data memory fall into four categories: cached loads and stores, cache control, synchronized loads and stores, and uncached loads and stores. These instructions accept two addressing modes—base plus offset addressing and register indexed addressing. The first of these modes specifies a base register, containing a pointer, and a signed 16-bit byte offset that is added to the base pointer to form the effective address. Indexed mode specifies two registers, containing pointers or integers, that are added to yield the effective address. The 32 least significant bits of an address register are taken as a byte pointer.

Cached Accesses

Cached loads and stores move byte, short word, word, or long word (and, consequently, floating point) quantities to and from the data cache. Cached data must be aligned such that multi-byte quantities do not cross long word alignment boundaries. Loads of quantities smaller than a long word are automatically sign-extended. Cache prefetch is accomplished simply by loading into r0, which forces a non-resident word to be loaded from memory into the cache; the processor will not fault for loads of this kind. Both the instruction and data caches can have lines selectively flushed by software, allowing coherency between main memory and the caches to be maintained. A thread updating memory can ensure that other threads find it in a consistent state by following the last unsynchronized store by a load of the same location.

Synchronizing Accesses

Threads accessing a substantial data structure can synchronize through the use of I-structure [ArNP89, ArNi90] and M-structure [BaNA91] operators. These operations are the only operations implemented in microcode (see Section 3.4.2). An I-structure is conceptually a special kind of memory location on which one or more consumers of data can synchronize with a
producer. Once an I-structure is reset to state “empty,” a read from it cannot be completed by a thread until another thread (usually) writes to the structure, and sets its state to “full.” Following the write, any threads previously waiting on the location receive the data and complete their I-structure reads. Subsequent reads of the “full” I-structure complete immediately, in the fashion of imperative reads. An M-structure is similar, except that it enforces mutually exclusive access to a location by resetting its state to “empty” following a successful read. A read from an “empty” M-structure cannot complete until data is written to the structure. If no threads are waiting on a M-structure it is set to “full” following a write. Otherwise, only one of the waiting threads receives the data before setting the M-structure to “empty” and forcing other threads to wait. Any threads continuing to wait on the structure do so until it is written again, at which point another thread is chosen arbitrarily to receive the data exclusively.

Unlike some previous implementations of structure stores [TrPY92], Concurro employs no special presence bits memory, but rather, uses ordinary long word locations in the data cache. This approach reduces system costs and allows added flexibility of use, although there must also be rules for defining and modifying structure contents. When long words are read under the interpretation of I/M-structures, their high-order bits are checked for the presence of a special bit pattern (see Section 3.4.2); long words containing that pattern are taken to be structures having a “deferred” or “empty” state, while any other long word is assumed to be a valid datum (i.e. the structure was “full”). Use of the destination register of a deferred structure load enforces synchronization on it. Threads blocked on synchronizations are not busy waiting [Iann90], reducing the effective cost of structure synchronization to that of ordinary data synchronizations.

The instruction set defines instructions to perform I-structure loads, M-structure loads, and structure stores. Synchronizing stores distinguish between I- and M-structures by their contents in the deferred state. The Concurro memory model guarantees self-consistency for all accesses, and provides weak ordering consistency for synchronization [AdHi90]. In practice, structure storage has proved to be straightforward to use. A location is cleared by imperatively storing the “empty” value to it. Storing any floating point value (including zero or NaN) or sign-extended word to the location using the synchronizing store instruction fills the location. If an empty location is accidentally read imperatively, floating point operations interpret its contents as NaN. Mutual exclusion algorithms, such as the carries benchmark in Chapter 4, may use M-structure operations to manipulate a token or counter in a semaphore location.
Main Memory Accesses
The main memory is accessible in units of long words through uncached load and store instructions, which require data to be naturally aligned. Direct access to main memory is particularly beneficial in scientific computing involving matrices and vectors, as the bandwidth requirements and access patterns occurring in these codes can often render a cache ineffective or even detrimental [RYYT89]. Since main memory accesses are pipelined, a multithreaded program can enjoy the high throughput of this access mode while tolerating the long latencies. Although memory accesses may be reordered by the processor to improve throughput, the programmer can expect dependencies between accesses to be observed.

2.1.4 Operational Instructions
Operational instructions perform an operation on at most two 64-bit operands, and usually return a 64-bit result. The instruction set divides these instructions into eight functional classes—corresponding to the types of functional units found in the processor. Memory class instructions were described in the previous section.

Logical Class
Logical instructions perform bitwise logical and conversion operations on pairs of 64-bit operands. Conversion operations sign-extend or zero-pad byte, short, or word quantities. Logical AND, OR, NOR, and XOR can also take unsigned 16-bit immediate operands; low-half-word and high-half-word varieties of these allow 32-bit constants to be introduced in only two instructions.

Shift Class
Shift instructions shift a 64-bit value a variable number of bit places, ranging from 0 to 63, inclusive. Left, logical right, and arithmetic right shifts by register or constant amounts are supported.

Integer Arithmetic Class
Integer arithmetic operations perform addition, subtraction, comparison, and test on signed or unsigned short words, words, and long words. Add and subtract results are sign-extended according to the size of the operands. As for logical instructions, two varieties of half-word operations allow 32-bit constants to be added or subtracted in two instructions. Word adds and subtracts may optionally specify one operand to be scaled by a small power of two (from 1 to 8) before use, thereby improving the efficiency of common array indexing operations. Compare (test) instructions test the result of a subtraction (addition) against one of 16 result conditions, returning 1 to
the destination register if the condition holds, 0 otherwise. Returning Boolean values in general-purpose registers, as opposed to using condition code registers, provides greater flexibility and eliminates hidden dependencies that complicate instruction scheduling and reduce performance.

**System Class**

Instructions for reading and writing register channels and system registers fall into the system class. Register channels may be forcibly cleared or filled with a long word. Reading a channel's contents can occur either non-destructively (the channel remains full), or synchronously (the channel empties upon reading). Once filled, the writing thread can wait for a channel to be cleared by another thread, before refilling the channel with new data—either for the same thread or another thread using the same channel. The ISA does not guarantee any exclusion properties for register channels, thus each of several threads waiting on the same channel may or may not gain access to that channel when it becomes available. Channels were designed for passing arguments between threads and to provide fast, ad hoc synchronization. Concurro maintains a set of system registers, containing such items as the processor status word, interrupt vectors, and structure free list pointer. These registers can be read, written or exchanged with a general-purpose register, subject to privilege limitations.

**Floating Point Addition Class**

Floating-point addition class instructions perform addition, subtraction, comparison, test, and conversion on double-precision floating point numbers. The compare and test instructions behave similarly to their integer counterparts. Conversion instructions convert between floating point values and signed 64-bit integers. Exceptions caused by floating point operations do not generate interrupts; however, as required by IEEE standard-754 [IEEE85], exceptional conditions appear as NaN result values. Not trapping on exceptions eliminates the cost of implementing hardware for handling precise exceptions and avoids the ambiguity of trapping anonymous threads. During software development, possible exception sites can be (and were) checked by explicit tests for NaN followed by conditional branches to an exception handler.

**Multiplication Class**

The set of multiplication instructions contains instructions for performing double-precision floating point and fixed-point multiplications. Two varieties of integer multiplies take either two 32-bit operands, yielding a 64-bit product, or two 64-bit operands, yielding the least significant 64 bits of the product.
Division Class
Division class instructions perform floating point division and square root, along with signed and unsigned integer division and modulus. The square root operation can often be implemented at a minor incremental cost, given the presence of a floating point divider [Tayl81].

2.1.5 Control Instructions
Control instructions are responsible for thread management and flow control. Since Concurro threads execute in a MIMD fashion, these instructions appear to be executed by each context individually. The set of control instructions also includes conditional moves, which conditionally update a register dependent on another register’s contents being zero or non-zero. These instructions may be used as efficient alternatives to conditional branches guarding short instruction sequences, and they find use by advanced schedulers for run-time memory disambiguation [Ditz93]. The IP may be read or set indirectly via jump and branch instructions. However, the possibility of instruction reordering by the processor invalidates some uses of the IP value.

Most branch and fork instructions specify their target address in relative form—with respect to the current IP—so that an instruction and its address can be encoded in a single 32-bit word. Specifying relative addresses by additive offsets could be detrimental to cost and performance in high-performance implementations, as address generation hardware occurs in duplicate—up to one address generator per context. Therefore, to reduce cost and address generation latency, a semi-absolute address mechanism [Gunt90b] was developed for Concurro.

A semi-absolute address is composed of two fields: a one-bit direction flag, indicating whether the branch target is forward (flag = 0) or backward of the current IP, and a $k$-bit address part containing the $k$ least significant bits of the target word address (since instructions are word-aligned). Figure 2.2 illustrates the encoding of semi-absolute branch targets. On a forward branch, if the value of the branch address part is greater than or equal to the $k+2$ lower bits of the IP, then the target address is the $30-k$ high bits of the IP concatenated with the branch address part, otherwise the high bits of the IP are incremented by one before concatenation with the branch address. For backward branches the comparison is reversed, with the upper bits of the IP being conditionally decremented by one instead. The comparison avoids the asymmetrical branch range of a simple bit substitution scheme. To reduce the comparison cost, it was recognized that only a small number
of most significant bits of the branch address part need be compared to retain most of the symmetry and range of true relative addressing. Comparing just 3 bits provides a worst-case branch range of 94% of that available from relative addressing [Gunt90b]. Hardware for generating semi-absolute addresses by this means is described in Section 3.1.4.

![Diagram of branch instruction]  
**Figure 2.2** Semi-absolute address specification.

The assembler treats branch addresses as absolute, relying on the linker to relocate them and set branch direction flags. Unconditional branches, subroutine calls, and forks take the long address format (see Figure 2.1), giving them an addressable range of 256 MB; conditional instructions can encode only short addresses, reducing their addressable range to 8 MB (1 M-instructions either side of the IP). Unlike some older RISC architectures, Concurro branch and jump instructions are not delayed [GrHe82]. Delayed branches complicate implementation of out-of-order instruction issue, while providing insignificant performance benefits for multithreaded programs, as the processor is equipped to tolerate instruction latency.

Conditional branch instructions test the lower 32 bits of their source register contents for zero, greater than zero, or the negation of these; conditional fork instructions test for zero or non-zero. These tests serve to reduce instruction counts in some common loop tests. Testing only the lower half of a register allows conditional instructions to act directly on Boolean results from comparison and test instructions.

Subroutine calls are normally made with the branch-and-link instruction, which saves the return or link address in r31, permitting threads to call subroutines independently of each other. Computed jumps, indirect calls,
and subroutine returns make use of the jump-and-link instruction, which takes a register argument containing the jump address, and a destination register for saving the link address. User-mode programs may execute a software trap instruction to enter the kernel for service. A number of interrupt save registers in the system register set enable trap handlers to save some user-mode registers before handling traps in system-mode. Executing a return from interrupt instruction restores the previous processor status word and IP of the interrupted context.

Fork instructions specify—either immediately or via a register—the starting address of a thread to be executed concurrently. The fork operation completes as soon as a spare context becomes available. The newly activated context has only its IP initialized; the contents of local registers are initially undefined. Threads must execute the terminate instruction to relinquish their hardware context; although the root (or first active) thread causes an interrupt if it attempts to terminate in this way.

2.2 System Architecture

In the description of Concurro’s microarchitecture it is assumed that a Concurro CPU occupies the system shown (simplified) in Figure 2.3. Concurro’s memory interface comprises a dedicated bus to the secondary instruction cache, a port to main memory servicing data cache traffic, and at least one direct port to the main memory system for uncached load and store operations. The number and bandwidth of main memory ports are configurable in the simulator, allowing a variety of cost points to be tested.

Although machine performance is measured in terms of processor clock cycles, many latency parameters of the simulator are dependent on the actual clock period, requiring a system clock rate to be assumed. Extrapolating from current design trends, a clock speed of 250 MHz was estimated to be reasonable for an implementation of Concurro in the near-future (a 200 MHz RISC system is already available at the time of writing [Dobb+92]).

2.2.1 Main Memory

The main memory system is managed by a controller that attempts to maintain the illusion of a multiported memory by distributing memory accesses across a set of interleaved RAM banks [Ston87]. Each memory port contains a 64-bit bidirectional data bus, an independent 32-bit address bus, and control lines. These buses support pipelined, split-phase transactions [HePa90], with reads and writes using the same data bus. A read
access requires one bus cycle for sending the address, and one cycle (some cycles later) for returning the requested long word. The maximum configurable port throughput is one long word per processor cycle. Memory-mapped I/O requests are channelled by the memory controller to the I/O bus (uncharacterized in simulation).

![Conurro system organization](image)

**Figure 2.3** Conurro system organization.

So as to confine attention to the characteristics of the CPU, second-order effects of the memory system were ignored. It was assumed throughout that main memory is perfectly multiported, with each port behaving as a pipeline delivering reads and writes in strict access order. Similarly, virtual to physical address translation is assumed to take place transparently, ignoring effects of operating system algorithms, page size and TLB design. This is justified for the single-tasking environment in which the benchmarks are run. The effects of finite memory interleave and TLB misses can be approximated by adding an average penalty to the memory latency.
2.2.2 External Cache

With main memory access expected to be more than an order of magnitude slower than an on-chip cache hit, it is necessary to augment the primary instruction cache with a larger, external secondary cache. For example, given a target performance of 75% of peak, and assuming an 8% primary cache miss rate, a miss in the first-level cache must be serviced, on average, in \((1/0.75-1)/0.08\) or 4.2 cycles, requiring a 2 or 3 cycle second-level access time after accounting for overheads.

The secondary cache is assumed to be direct-mapped, and implemented in commodity high-capacity SRAM. The access time of the SRAM and the width of the bus joining the cache with the CPU are configurable in the simulator. Typically, the secondary cache is set for 2-cycle access from a 256 kB data store, with a line size of 64 bytes and a 128-bit bus connecting the CPU.

Upon a primary I-cache (instruction cache) miss, the I-cache controller sends the physical address of the missed instruction to the external cache controller. If the secondary cache hits, it returns an entire primary cache line to the CPU, breaking the line into pieces to accommodate a narrow bus. The secondary cache is unavailable for other requests while servicing an access. A second-level miss causes the secondary controller to request long words from main memory, in a critical-word-first with wraparound pattern. Immediately required data arriving from main memory is also passed on to the primary cache. Data outside the requested block refills the secondary cache in the background. The secondary cache controller may optionally be configured to prefetch the subsequent secondary cache line as well, exploiting the high locality of instruction references to effectively double the line size of the external cache. Explicit line flushes of the first-level cache propagate through to the external cache for flushing a containing line there.

2.3 Processor Organization

An abstracted view of the Concurro processor organization is depicted in Figure 2.4; bold lines represent buses, with heavier lines denoting wider paths. The figure shows a specific configuration of the processor, although all configurations have several features in common, namely, a single shared I-cache, a central thread manager unit, eight classes of functional unit (FU), and a single shared D-cache (data cache) with synchronization controller. More detailed discussions of these subsystems appear in Chapter 3.
**Figure 2.4** Concurro block diagram (8-context, 2-group example).
The hardware context units (labelled C0–C7 in Figure 2.4) are grouped together into one or more groups as a means of reducing cost, by sharing expensive resources such as an operation bus or a register file, and shortening the critical path, by distributing contention for shared resources. Scalability at the lowest cost/performance ratio is obtained by increasing the number of contexts and functional resources. Since the shared data and instruction caches are the single most expensive processor components, their utilization should, ideally, be the limiting factor in Concurro’s scalability. The point where duplicating the caches becomes necessary lies in an area of the design space where genuine multiprocessing could be more effective.

2.3.1 Configuration Designation

For the remainder of the text, Concurro processor configurations will be designated by a three-digit model number: \( abb \), where \( bb \) specifies the number of hardware contexts implemented and \( a \) specifies how many groups are formed by those contexts. Thus, for example, Figure 2.4 indicates a model 208 processor, as it shows 8 contexts arranged into 2 groups.

2.3.2 Contexts

Context units—or simply contexts—sequence the execution of threads. A context comprises a file of local general-purpose registers, instruction fetch and dispatch logic, and a branch unit for executing control instructions. Duplicating contexts to support a number of active threads is not excessively costly, given that several previous processors, including HEP [Jord83], MASA [HaFu88], PRISC-1 [ShNi89], and Sparcle [Agar+93] have also supported multiple register files or register cache and multiple instruction pointers. Concurro has a relatively minor additional burden of a small instruction buffer, simple pre-decoder, and control logic for each context. It is assumed that each context also contains a branch unit, but the semi-absolute address generators in these could be shared by a group.

Duplicated contexts give Concurro two of its most important features, namely, multiple instruction issues in each cycle and asynchronous thread scheduling. By taking advantage of issue opportunities whenever possible, asynchronous scheduling enables Concurro to perform better than conventional time-multiplexed machines in single threaded or linear multithreaded modes. However, the efficiency of time-multiplexing is also retained when program parallelism is sufficient to maintain pipeline saturation.
The activation and suspension of contexts is controlled centrally by the thread manager unit. Contexts or threads wishing to fork or terminate send requests to the thread manager via the shared context control bus. The thread manager maintains a register of active contexts, allowing the unit to distribute work evenly among the context groups. Context control is achieved simply by broadcasting a context number and operation code, which the addressed context responds to; for a fork a context captures its new IP from the bus, whereas for a termination a context receives the acknowledgment of its request before halting its instruction fetch/dispatch machine.

Ideally, \( n \) contexts should choose collectively \( m \) instructions for issue in each cycle. However, the routing complexity and contention delay involved in selecting fairly \( m \) instructions from possibly \( n \) candidates, could well be sufficient to lengthen the cycle time of the machine. By arranging contexts into \( g \) groups, contention for instruction issue can be localized, reducing complexity and delays. Instruction selection is then simplified to choosing from each group one instruction from at most \( n/g \) ready instructions. The complexity of this selection scheme is comparatively minor, and in fact, less than that typically involved in superscalar instruction dispatch [GKTM90, John89].

Grouping contexts can also degrade performance. Load imbalance among groups, the exaggeration of blocking/starvation effects in small groups, and the possibility of missed issue opportunities due to grouping constraints are all factors reducing the instruction issue rate. The two instruction issue strategies, ungrouped and grouped, were simulated in isolation so as to quantify the penalty of context grouping. The simulation assumes \( n \) contexts to be active, with at most \( g \) instructions being issued per cycle; the ungrouped strategy allows instructions to be selected from any of the \( n \) contexts, whereas the grouped strategy restricts selection to one instruction from \( n/g \) contexts per group, with groups sharing load equally. The delay between an instruction issuing and its successor becoming ready for issue is assumed to follow a discretized normal distribution with a mean of \( d \) cycles and standard deviation \( d/2 \). Instructions are selected on a round robin basis.

The simulated issue rates (total instructions per cycle) of the two strategies are shown superimposed in Figure 2.5 for various processor configurations after 1 million cycles, assuming \( d = 3 \). Issue rates for the ungrouped strategy are consistently greater than those of the grouped strategy. For \( d = 3 \), the \( g \times 4 \) configurations represent saturated multithreading, where both issue strategies can achieve almost \( g \) instruction issues per cycle—the disad-
vantages of grouping are mitigated by the abundance of ready instructions. For lower numbers of active contexts per group, the machine is operating in transition and linear modes, where the issue rate depends on the arrival rate of instructions. Higher probabilities of blocking cause the grouped issue rates to suffer as the workload increases. Tested over a wide range of processor configurations and $d$, the grouped issue rate never fell below 90% of the ungrouped rate. Moderate load imbalances of less than 50% among groups had negligible influences on issue rates. Given the possibility of lengthening the processor cycle time to support an ungrouped strategy, these results suggest that grouping contexts is an acceptable design tradeoff.

For comparison, M/M/c/N queuing theory [Taha82] was used to re-evaluate the strategies analytically. The results, although qualitatively reasonable, underestimated performance by up to 20% for the grouped strategy. The discrepancies are due to the breakdown of assumptions about probability distributions and continuous variables in classical queuing theory. FIFO and random instruction selection policies were also simulated; they showed no consistent advantage over round robin selection, which was retained in the final design for its simplicity and fairness in pathological situations.

![Graph showing Instruction Issue Rates for Grouped and Ungrouped Contexts](image)

**Figure 2.5** Instruction Issue Rates for Grouped and Ungrouped Contexts (mean instruction--instruction delay of 3 cycles).
2.3.3 Instruction Cache

The I-cache subsystem must be capable of sustaining an instruction bandwidth exceeding one per cycle, if it is to match the maximum issue rate. Unlike a superscalar processor, Concurro can issue instructions in each cycle that bear no relationship to each other, as they belong to separate threads. This makes the bandwidth requirement of Concurro's I-cache significantly more difficult to achieve. Several solutions are possible:

- duplicate caches—one per context group (since a group issues at most one instruction per cycle)
- a micro-cache for each context, with a larger shared cache refilling the micro-caches
- a multiported cache having one port per context group

On the basis of simulation results, in Section 5.2, for a multiple cache organization, the option of duplicating or multiporting caches was ruled out as being too costly for the less than 3% improvement in performance gained—especially for large configurations. Providing each context with a micro-cache minimizes the size of the main cache and delivers the required performance, but adds significantly to the size of each context. It is expected that micro-caches of sufficient capacity for acceptable hit rates are likely to contain at least as much storage as a local register file, plus additional control logic, thereby multiplying the area of context hardware two-fold or more and becoming the limiting factor in scalability.

Rather than adopting any of the above solutions, a new cache architecture was developed for use with context buffering. Each context buffers several consecutive instructions in a FIFO queue, which feeds the instruction pipeline. When the queue empties, the context requests an instruction fetch from the I-cache. The cache system parallel-loads the empty buffer in one cycle in an attempt to fill it with consecutive instructions from the context's instruction stream. Once the buffer is refilled, the context is able to continue execution for several cycles before needing another cache access, thereby giving other contexts opportunities for instruction fetches from the same I-cache. Instruction buffers are not addressed, therefore a taken branch or interrupt initiates a fetch request to refill the buffer with the target instructions. The buffer size is configuration-dependent, and set so that the fetch request rate approaches unity at full load (see Section 3.1.2).

Ideally, the I-cache returns enough instructions in each fetch to completely fill a buffer; if I-cache returns fall short, the fetch frequency must rise to
satisfy the bandwidth demand until the instruction bus becomes saturated. Even with no cache misses, the return requirement may not be satisfied by a wide bus alone. Consider, for example, the problem of fetching 6 consecutive instructions, designated I0–I5, from address A of a cache 8 instructions wide, illustrated in Figure 2.6. Since A is misaligned with respect to the memory organization, the accessed instructions must be shifted left for alignment with the bus lines, and the last instruction, I5, cannot be read with the others, forcing a no-operation onto bus line Q5. Note that this situation is independent of the cache line size, which may be greater than 8 instructions.

![Diagram of cache memory and instruction bus]

**Figure 2.6** Multiple instruction fetch (6 from 8).

Suppose that fetches request B consecutive instructions from the cache. For programs experiencing branch frequencies comparable with their buffer refill rates, the location of the first fetch instruction is likely to be uniformly distributed over L columns. Hence, the expected return from the cache (for \( L \geq B \)) is:

\[
I_{\text{ret}} = \frac{B(L-B)}{L} + \frac{B(1+B)}{2L} = \frac{B(1+2L)-B^2}{2L} \quad \text{instructions/cycle}
\]
Thus for the given example, where $B = 6$ and $L = 8$, $I_{\text{ret}} = 4.1$ instructions per cycle—only 69% of the 6 instructions per cycle target. Johnson [John89] has verified this shortcoming, in the context of superscalar instruction fetching, for a wide variety of realistic benchmark programs.

Achieving greater returns by increasing the access size results in excessively wide memory organizations if they are to retain power-of-two dimensions. Ultimately, an entire cache line could be accessed and a sequence of instructions extracted from it; however, the necessary memory array width and instruction shift network render this scheme impractical for large fetches. To overcome these difficulties, while retaining single-cycle cache access, a pre-access memory architecture was developed [Gunt92b], and a prototype SRAM of this architecture implemented [Gunt92a].

The pre-access architecture of the I-cache allows it to return a constant number of consecutive instructions per access, regardless of the first instruction's address, with words properly aligned to the instruction bus. An arbitrary number of words can be returned, as there are no power-of-two restrictions. By integrating pre-access at the storage array level, this capability can be obtained at a cost dependent on the required bandwidth rather than the worst-case bandwidth [Gunt92b]. A pre-access architecture is largely immune to pathological fetch patterns that can severely reduce the bandwidth of simple wide-fetch memories. In a simple pre-access cache implementation, fetches cannot straddle cache line boundaries, thus reducing the instruction bandwidth in some cases; more complex implementations allow fetches across adjacent lines, with the line replacement algorithm optimized to maximize the cache return. These issues are explored further in later chapters.

Both the on-chip primary and the external secondary I-caches are physically addressed. The primary cache, being on-chip, is constrained in size and therefore usually set-associative. A page colouring scheme for virtual page mapping enables the instruction TLB (translation lookaside buffer) to be accessed concurrently with the I-cache tags [BrLF90], without significantly reducing the associativity of the main memory in its role as a cache for the virtual memory system.

### 2.3.4 Datapath

Given that Concurro with $g$ groups can issue a maximum of $g$ operational instructions per cycle, a natural datapath structure would comprise $g$ arithmetic and logic units, each accepting operands from a group. Such an arrangement offers ideal throughput, but poor hardware utilization. Instead,
the execution resources in Concurro are split into specialized functional units (FUs) that operate autonomously to increase parallelism and utilization of the hardware. The more expensive FUs, such as the load/store and floating point units, are shared. However, the illusion of integrated FUs is maintained by a set of operand queues decoupling instruction dispatch, by groups, from instruction issue, by physical FUs; operand queues in a minimal implementation can degenerate to single-entry latches. Bottlenecks are avoided by making the number of real FUs in each of the eight functional classes correspond with its average frequency of use. Implementations can range from the extreme of no FU sharing to that of one real FU per functional class. Figure 2.4, for example, shows all but the logical and integer arithmetic FUs as being shared.

Contexts that are granted instruction dispatch read instruction operands from registers—or forward result bus data—and dispatch these with their instructions to selected operand queues via the group operation buses. A group operation bus conveys two 64-bit operands, an 8-bit operation code, and a destination tag, typically 9 or 10 bits, comprising the numbers of the context and register to which the result is to be written. Shared FUs receive their operands via a multiplexer controlled by a fair selector, ensuring an even load distribution. FUs process their operands according to the operation code accompanying them, with most FUs carrying operation codes (or derivatives) in dynamic, multi-function pipelines.

The system FU is special in that it has access to the system registers and register channels. The semaphore flags of the register channels are broadcast to the groups via a small bus. A context wishing to get or put a register channel treats the flags as if they were register scoreboard bits. Privilege checks for system register accesses make use of the destination tags passed to the FU, as tags are sent even with instructions not producing results.

Loads, stores and control operations to both the D-cache and main memory are accepted by the load/store unit. Making up this unit are the address adders, TLBs, and queues shown towards the bottom of Figure 2.4, with each direct access memory port and the D-cache having a dedicated address generation pipe and TLB. The load/store unit is effectively split into two independent units—cached and uncached accesses—that share the group operation buses and result queues. This arrangement allows cached accesses to commence in parallel with uncached accesses. Destination tags passed to the load/store are used not only for routing returned data, but also for facilitating access reordering by the cache controller and main memory.
Since a proportion of instructions do not generate results, the average result bus bandwidth is lower than that of the corresponding operation bus, producing a slight bandwidth difference across the FU results and improving their throughput. Result queues for capturing FU results enable this difference to be exploited, although the queues may be dispensed with in a minimal implementation. The selection of result queues by shared FUs is effected by inferring the destination group from the destination tag. A writeback arbiter for each group selects in round robin fashion at most one result per cycle to be written to its destination via the result bus. The size and complexity of the write back arbiter benefit from context grouping, as result selection is fixed to 1-of-8 for all processor configurations. Results destined for the global register file (GRF in Figure 2.4) may need to update several distributed copies of these registers to maintain coherency among them.

### 2.3.5 Data Memory

Both the main memory ports and cache are physically addressed, with the cache portion of virtual addresses equal to their corresponding physical addresses—as for the I-cache—to allow concurrent TLB and tag access. Although not simulated, TLB misses are envisaged as being handled by dedicated hardware. Techniques for reducing TLB miss rates, such as variable page sizes [ChBJ92], are particularly applicable to the uncached memory TLBs, which must withstand the effects of sweeping array references.

The D-cache is backed directly by the main memory rather than a second-level cache. For a latency-tolerant processor this arrangement is almost as effective as two-level caching, and avoids the overhead of an external cache under high miss rate conditions. The D-cache, as simulated, posts writes to main memory, updating it soon after cache access; however in general, coherency between main memory and the D-cache is not maintained in hardware. Despite the convenience and safety of hardware-implemented coherency, its need was found to be sufficiently infrequent to render it a special case for which the performance degradation and expense of extra checking logic and a snooping directory are unjustified in a uniprocessor.

In practical Concurro programs, data structures can usually use a single storage method, with coherency enforced in the vicinity of a small number of synchronization points. A cached store is guaranteed to appear in main memory if followed by a flush and a load from the same location, while an uncached store can be brought into cache by following the store with a
flush and a cached load. The retiring of writes to either cache or main memory must be accomplished by pushing the writes out of any queues feeding the memory. To this end, a load of data previously stored is sufficient to reach the memory and, as a useful side effect, return confirmation of the write.

### 2.4 Instruction Pipeline

Concurro's context pipeline for three major instruction types is depicted schematically in Figure 2.7. The figure indicates the position of pipe stages relative to the two phases of the system clock, where phase $1 = \Phi_1$ and phase $2 = \Phi_2$. Parallel actions are separated by a dashed line, while sequential actions lie below their predecessors. Arrows indicate bypass and control possibilities between pipe stages. The I-cache, instruction decode and register read stages are common to all instructions, whereas subsequent stages differ according to the instruction. Operational instructions are shown with a latency of $n = 2$ cycles, although the various FUs have different latencies equal to or greater than 1 cycle. The 2-cycle latency for the cache load indicates a typical situation for D-cache hits. Any instruction returning a result passes through the result write back stage. Timing for the branch instruction applies to a predicted conditional or mandatory branch/jump.

![Figure 2.7 Pipeline for operational, cache load, and branch instructions.](image-url)
2.4.1 Instruction Fetch and Decode

Owing to the presence of instruction buffers and a pre-access I-cache, the instruction fetch stages of the pipeline are executed infrequently by each context. With valid instructions buffered, a context can immediately decode the head of the instruction queue, otherwise a fetch request is made in phase 1; the I-cache commences a fetch for the winning fetch contender in phase 2. During the first cycle of the fetch, the cache data store, tag store, and instruction TLB are driven in parallel. The tags are checked against the TLB output after one cycle, but access to the data store is extended for another half cycle during which sensing and selection of cache data and its propagation along the instruction bus occur.

Instruction decode and reading of register operands can take place almost simultaneously due to the regularity of Concurro’s instruction encodings—the results of unnecessary register reads and scoreboard checks are simply discarded. However, Concurro may be optionally configured to separate decode and register read into two pipestages for out-of-order instruction dispatch, where it becomes costly to overlap the decoding of multiple instructions with their operand reads. In this configuration, a context selects from its buffer the “best” candidate instruction for dispatch. Contexts with ready instructions dispatch them at the end of phase 2 in the decode stage, at which time operands have been either read/formed or known to be available as bypasses of the write back stage.

2.4.2 Branches and Jumps

To speedup the execution of branches, the outcome of conditional branches is predicted so that they and unconditional branches can initiate instruction prefetch. Accordingly, branch instructions must be detected early in phase 1 of the decode stage to allow a semi-absolute target address for prefetch to be generated by phase 2. Mandatory branches (including subroutine calls) and conditional branches predicted to be taken contend for fetch in phase 1, in parallel with address generation. Branch prediction fetches are treated as speculative by the contention logic, which assigns them lower priority than buffer refill requests. When lightly loaded, the processor has more opportunities for performing speculative fetches and reducing branch latency. When heavily loaded, however, the increased frequency of mandatory fetches rules out most speculative fetches—but these are precisely the conditions under which instruction latency can be tolerated.

Conditional branches read their source register in the decode cycle. Branch tests are sufficiently simple to be carried out in phase 1 of the execute
cycle that follows. Branches not taken exit the pipeline after the execute stage. Indirect jumps, interrupts, and branches taken, but not predicted to do so, annul and lock all earlier pipestages, and vie for a fetch from the target address in the execute stage. Similarly, predicted, taken branches that failed to speculatively fetch in the previous cycle become mandatory in the execute stage. A mispredicted branch that has erroneously fetched in the decode cycle must abort the pending instruction fetch in phase 2 before it overwrites the currently valid instruction buffer.

If branch prediction were to be carried out one cycle earlier, an abort upon a misprediction would no longer be adequate, as the instruction buffer would already be updated. Possible benefits of earlier branch prediction are almost entirely lost due to two mechanisms: the majority of speculative fetch requests do not succeed and are promoted to mandatory status; and for poor branch prediction accuracy the additional instruction bus bandwidth wasted by unabortable fetches reduces the bandwidth available to necessary computation.

From the pipeline structure it is clear that untaken branches cost 1 cycle; a branch taken with a speculative fetch has a latency of 2 cycles, but taken with a mandatory fetch, its latency increases to 3 cycles (minimum). If an estimated 70% of all branches are taken [Crag92], then the average branch latency varies from \(1 \times 0.3 + 2 \times 0.7 = 1.7\) cycles to \(1 \times 0.3 + 3 \times 0.7 = 2.4\) cycles or more, depending on the branch mix and processor loading.

### 2.4.3 Operation Execution

The majority of instructions enter a variable length operation pipeline. Once dispatched in phase 2 of the decode stage, an instruction may immediately commence its first execution cycle, or it may be queued at a FU for issue in a later cycle. An instruction progresses through the pipeline until impeded by a stalled stage. This may result in pipeline bubbles being squashed. The last execution pipestage will stall if a FU cannot output a value into a result queue.

Write back arbitration and result transmission can occur in phase 2 of the last execution pipestage for instructions encountering empty (or non-existent) result queues. Instructions returning a result to a register other than \(r_0\) perform the result write back stage. An instruction’s destination group is signalled during write back contention in the previous cycle, giving its contexts the opportunity of bypassing write back to a register file by taking an operand off the result bus instead.
2.4.4 Data Cache Access

Although D-cache accesses are another class of operational instruction, the split-phase nature of accesses distort the pipelined view of execution presented by other instruction classes. Data may emerge from the D-cache pipe in an order different from the arrival of corresponding load instructions, whose latencies can exceed the nominal pipeline length on a cache miss. Address components are summed in phase 1 of the access. As in the I-cache, the data store, tag store, and TLB can be accessed simultaneously in phase 2. After one cycle the hit status of the access is known—in time to vie for and signal a write back in phase 2 of the last cycle. With the inclusion of the D-sense stage, the data store is given almost 1.5 cycles to return its data on a hit.

Cache misses disappear temporarily from the pipeline when they are dealt with by the D-cache miss handling mechanism. However, missed loads eventually re-emerge to signal pseudo-hits to permit them the same bypasses as hit loads.
Chapter 3

Subsystem Architecture

A balanced system is essential for realizing the performance potential of a processor architecture. Under-designed subsystems can distort the apparent effectiveness of the microarchitecture, while excessively complex designs provide a sub-optimal global solution and threaten cycle time increases. For the purpose of exploring these issues in Concurro, the simulator allows each subsystem to be configured for varying levels of capability and capacity.

This chapter describes in more detail the subsystems introduced in the previous chapter, and examines the spectrum of design choices available and simulated for each unit. This level of the architecture is generally invisible to machine programmers, and free to be implemented in any way that satisfies performance and cost constraints. The implementation used in the simulator is documented here as a proposal for physical implementations and as a foundation for the experimental work of the following chapters.
3.1 Context Units

The execution of threads in Concurro is controlled by context units. As indicated in Chapter 2, contexts are arranged into groups to reduce complexity and critical path delays. Within each group contexts compete against each other for instruction fetch and instruction dispatch opportunities. These are arbitrated so as to allow up to one fork or instruction fetch per cycle, and up to one operational instruction dispatch per group in each cycle.

![Block diagram of a context unit.](image)

Figure 3.1 Block diagram of a context unit.

The remainder of this section describes the structure and operation of individual context units.
3.1.1 Organization

A block diagram of a context unit is shown in Figure 3.1, with multi-wire data and control paths indicated by single lines. The group global register file (GRF) and various arbiters are the shared resources that tie contexts together in a group; apart from these the contexts are autonomous. All contexts are identical except for the root context, which has interrupt support. The execute control unit is a state machine that controls instruction fetches, instruction dispatch, and the part-execution of thread management functions. Upon reset, all but the root context’s execute control unit are in an inactive state waiting for fork commands from the thread manager. Inactive contexts make no bus accesses, but may receive late register results from previous threads.

The instruction buffer is a FIFO queue that can be parallel loaded from the common instruction bus. Implementing the buffer as an addressible cache (of one line) would not be beneficial for continuous instruction sequences or any but the tightest of loops, besides which, it was estimated that the cache function would more than double the total area devoted to instruction buffering in the machine. An instruction selected from the head of the buffer (or farther back for look-ahead implementations, see Section 3.1.3) is sent to the pre-decoder for determination of instruction type and subsequent handling. Control instructions—such as branches and forks—are diverted to the branch unit where they are executed, whereas operational instructions may vie for dispatch.

The fixed register specifier fields of an instruction are always extracted to address the local and global register files. If the most significant bit of a source specifier is set, all 64 bits of the corresponding local register are read; otherwise a global source register is read. The local register file has normally 2 read ports and 1 write port, with additional read ports provided if speculative instruction decode is implemented. Since each context may read 2 global registers per instruction, a single, shared GRF would require an unwieldy number of ports in a large processor configuration. To maintain global registers at a reasonable size and reduce wire load, the GRF is duplicated, with one copy of the GRF accessible per context group. This approach requires additional write ports on GRFs to maintain their coherency upon register write back.

3.1.2 Instruction Fetch

Active contexts whose instruction buffers are empty or will be empty by the earliest time that a fetch request can be satisfied contend for instruction
fetch in phase 1; thus, to allow for 1.5 cycles of fetch latency, fetches are requested as soon as two instructions remain buffered. However, since taken branches flush the instruction buffer, if the last or penultimate undecoded instructions are control instructions the fetch request is held off until any possible branches have been resolved. Instruction prefetch accounts for the I-cache latency, ensuring optimal single-thread performance and improved multithreaded performance. Predicted branches contend with other predicted branches for a speculative fetch, but the outcome of this contest is ignored in the presence of any buffer refill requests. Arbitration for I-cache access occurs at both the intra-group and inter-group levels to optimize the size and delay of the arbiter logic. Contexts, and groups in turn, are selected for fetches in round robin order. The winning instruction address and context identifier are valid on the instruction bus by the end of phase 1.

Regardless of the outcome of its fetch requests, a context continues to execute any remaining instructions. While several instruction fetches may be pending for different contexts, only one fetch per context can be in progress. Ordinarily, returning instructions and their valid flags are latched simultaneously in the instruction buffer. The abortion of a speculative fetch is achieved simply by disabling latching in the instruction buffer and waiting for the instruction bus to signal the completion of the fetch transaction.

Instruction buffers act as conventional prefetch buffers by reducing the per-context I-cache request rate. If too small, the buffers will empty rapidly and increase the instruction fetch rate—possibly to its limit; conversely, large buffers lower the I-cache request rate, but provide diminishing returns as increasing numbers of instructions become discarded by being branched over. Ideally the combined rate of instruction fetches from all active contexts should approach one per cycle to optimize buffer utilization against I-cache duty cycle. Chapter 6 provides experimental data relating to this issue.

If G context groups dispatch instructions at their maximum rate of one per cycle, then the I-cache must return instructions at an average rate of G instructions per cycle. However, this accounts for buffer refill traffic only. Since the occurrence of branches causes a fraction of fetched instructions to be wasted, fetch quanta greater than G instructions are necessary to compensate for the loss. Assuming that for buffers of the designed length just 80% of fetched instructions are dispatched, and neglecting pre-access inefficiency and cache misses, each fetch must therefore return G/0.8 instructions. Since instruction stalling does not impede buffer refills, it is possible for more instructions to be queued than are pre-accessed, thus
necessitating a buffer capacity of $2+G/0.8$ instructions. While the pre-access rate may readily approach its design limit, the effects of cache misses and instruction wastage are program-dependent and complex to predict. However, given typical run-length distributions [HuFl89], smaller instruction buffers are expected to be better utilized.

### 3.1.3 Instruction Decode and Dispatch

Contexts normally attempt to decode instructions whenever they become available. For comparative purposes, however, the simulator can be configured to share one decoder among a group of contexts, with the instruction buffers and register files switched in strict rotation—thereby emulating a simple time-multiplexed or “barrel” processor. The performance degradation created by such resource sharing is explored in Chapter 6. Although Concurro instruction formats (see Figure 2.1) are regular enough to permit parallel operand reads, some pre-decoding must be carried out by contexts to determine instruction kinds and resource requirements.

Instructions for decoding are selected from a window of the instruction buffer. This window can range in size from the head entry of the instruction queue to the entire buffer. In general, contexts behave as look-ahead processors, detecting instruction-level parallelism according to Keller's precedence relations [Kell75]. For selection windows larger than one, this implies the possibility of out-of-order instruction dispatch: instruction issue and completion are already out-of-order. Look-ahead gives contexts some of the advantages of a dataflow matching store, albeit on a very restricted sub-graph. Concurro implements simple scoreboarding as used in the CRAY-1 [IbTo89] for pipeline interlock. The scoreboard bit of a register is clear if the value of that register is currently valid, otherwise the bit is set if a write to the register is pending. More advanced scheduling and execution schemes of the nature of Tomasulo's algorithm [Toma67, WeSm84, John89] or register renaming [Kell75] provide largely superfluous concurrency to a multi-threaded computation while adding disproportionately to the implementation cost of contexts.

Every instruction in the window is checked for the availability of register or bypass operands, destination register, and execution resources. Those instructions that appear ready participate in a dependency check (absent for a one-instruction window) to determine whether they are free to dispatch ahead of any predecessors that may or may not be ready. All instructions are required to observe register dependencies—including artificial output and anti-dependencies. However, memory and register channel access instructions always dispatch in-order with respect to other instructions of
their kind. The cost of speculative execution hardware [John89] is avoided by not permitting the percolation of instructions through control instructions of any kind. Of those independent instructions that are eligible for dispatch, the first queued instruction whose FU is free is selected. As the dependency check is likely to be time consuming [John89], it may be desirable to read the source operands of every window instruction simultaneously and post-select the operand pair. Where this approach becomes impractical an extra register read stage can be inserted into the pipeline following decode, as in Section 6.1.3.

The context decoder categorizes instructions as: control instructions (fork, terminate, branch, conditional move), immediate instructions (operational instructions taking embedded 16-bit constants), and ordinary operational instructions (arithmetic and memory operations sent to FUs). Control instructions must be seen early in phase 1 of the decode stage in order to form target addresses for possible branch predictions. Since all immediate instructions map directly onto operational instructions, the pre-decoder makes a trivial translation of the operation code to yield the corresponding functional class and operation codes of the equivalent instruction, which is later dispatched with a second source operand extracted from the original instruction word. Operational instructions are decoded to the extent that their register specifiers and functional class are known, but final decode takes place in the FUs.

During phase 2 of the decode stage, contexts within a group contend for instruction dispatch. Forwarded results, following their bypass signals from phase 1, appear on group result buses at this time. Simultaneous writes by multiple threads to the same global register are eliminated by a simple arbiter that grants the global destination to the thread with the lowest context identifier. Dispatch arbitration has most of phase 2 in which to resolve. The adjustment and communication of station priorities in the round robin arbiter can occur during phase 1, when the arbiter would normally be unused. At the end of phase 2 winning contexts commit their incremented instruction pointers, set register scoreboard bits, and place operation data on group operation buses. Contexts that fail to dispatch their chosen instruction re-select anew from the instruction window and re-read registers in the next cycle to improve their choice as circumstances change.

3.1.4 Control Instruction Execution

Fork, terminate, branch, jump, interrupt, and conditional move instructions are executed by the branch execution unit. Once supplied with an instruction,
this unit can operate autonomously, allowing the execution of control instructions to be overlapped with that of other kinds of instructions. Comparison tests with zero and address generation are performed in the branch unit.

![Semi-absolute long address generator](image)

*Figure 3.2* Semi-absolute long address generator.

Fork and branch instructions contain semi-absolute target address specifiers. Figure 3.2 illustrates the function of the branch unit's semi-absolute address generator, which is designed to generate target addresses in under half-cycle latency. The figure shows hardware for generating branch destinations from long address format instructions, which encode the 27 least significant bits of the target address as a 25-bit word address with a direction bit for branch direction. For short address instructions the inc/decrementer size increases to 10 bits, but the comparator remains unchanged (at 3 bits, for example). The output address is a concatenation of the given lower address bits and the upper bits of the IP +0, +1, or -1, depending on the branch direction and position of the target location relative to the IP.

Jump and subroutine call instructions produce link addresses as register results. Rather than suffering the delay of sending these through the normal datapath, link addresses are latched in the link latch, where they wait for a vacant write back slot. The conditional move instructions likewise
make use of the link latch to speed up their execution.

Fork and terminate requests are arbitrated and controlled on the thread management bus, which connects to the global thread manager unit. The simulator implements a round robin arbiter for thread management requests, but a simpler arbiter may be equally effective given the low fork/terminate rates observed in practice. The thread manager asserts an inhibit signal upon overflow of its thread counters; only when a thread terminates may branch units complete blocked forks. Although fork instructions have an execution latency of two cycles, the actual delay before a child thread begins executing will be greater than this on account of contending for and fetching (possibly with a miss) the first buffer of instructions. Thus compilers are encouraged to schedule forks early and away from the critical path.

The accuracy of branch prediction in Concurro is not critical owing to the minor penalties for misprediction and the relative lateness at which prediction occurs. Accordingly, the expense of a dynamic branch prediction device is unwarranted. Lee and Smith [LeSm84] suggest that useful hit rates from branch target buffers require them to be comparable in size to TLBs—which rules out per-context branch target buffers. A centralized approach, such as augmenting the I-cache to store branch history, would suffer the effects of interference between threads and create a bottleneck for updates. Concurro's branch unit avoids these problems by using a simple static prediction strategy: backward conditional branches are assumed to be always taken, while forward conditional branches are assumed not taken. Predicted and unconditional branches and subroutine calls vie for speculative fetches. This strategy is straightforward to implement and captures, in particular, the behaviour of loop-intensive code. Chapter 6 examines experimental results for branch prediction accuracy.

### 3.2 Instruction Cache

A key factor in determining Concurro's performance is the behaviour of the I-cache under heavy load conditions. This subsystem must be designed to deliver the sustained instruction bandwidth required when all contexts are simultaneously active. As was shown in Chapter 2, these demands are complicated by the unrestricted alignment of instructions with respect to cache lines—necessitating a pre-access cache architecture. In its entirety, the I-cache subsystem consists of the external secondary cache, described in Chapter 2, and the internal primary cache, which is the only part to be examined here. The primary cache is typically configured as a small (16 kB), set-associative, non-blocking cache.
Figure 3.3 Instruction cache subsystem.

3.2.1 Organization and Interface

The cache receives instruction fetch requests from the contexts at the maximum rate of one per cycle. A fetching context and the cache participate in a split-phase transaction, with the context sending the word-aligned start address of an instruction sequence and a context identifier for the cache to respond to one or more cycles later. When ready, the cache addresses the destination context and returns in parallel a block of consecutive instructions on the instruction bus, which is no wider than the length of an instruction buffer; valid-flags accompanying instructions indicate the extent to which the request could be satisfied. For Concurro organizations of up to 8 context groups the 1-cache port size is comparable with that of current superscalar processors, such as the 8-instruction cache port in the IBM/Motorola PowerPC™ 601 [Beck+93].
Figure 3.3 shows the components of the I-cache subsystem. Cached instructions are stored in the virtually indexed data store, while physical address tags reside in the tag store. The ITLB (necessary in real implementations, but not simulated) translates high-order virtual address bits for late comparison with the accessed tags. Associativity, although variable between the extremes of direct mapped (one line per set) and fully associative (one set), was configured as set-associative. The data store is a random pre-access memory [Gunt92a, Gunt92b] with a pre-access depth equal to the instruction bus width. The line size of the cache must be at least one instruction less than the pre-access depth if maximum advantage is to be gained from pre-access [Gunt92b]; however, an absolute lower limit of two instructions (8 bytes) is imposed for interfacing with the secondary cache. The tag store pre-accesses either one or two tags per line, depending on the required performance.

The miss handler is a finite state machine that coordinates access of the secondary cache and reload of the primary stores. I-cache line flushes—whether they hit or miss the primary cache—are always echoed to the secondary cache to ensure coherency of the entire I-cache system. However, the two caches do not maintain the inclusion property [BaWa88]. Flushes and cache misses are recorded in the miss queue. If greater than one entry long, the miss queue avoids blocking of hits under misses [Krof81], while a single-entry queue implements conventional blocking on misses. A non-blocking I-cache brings no performance advantage to a single-threaded machine, but provides a multithreaded machine with the opportunity to tolerate instruction fetch latency in much the same way as datapath latencies are hidden.

The interface to the secondary cache operates synchronously with the processor clock. Instructions returning from the external cache are assembled into a primary cache line in the line reload buffer, prior to being written to the data store.

3.2.2 Hit Access

Provided that sufficient miss queue capacity remains and a line reload is not taking place, a fetch request arriving at the start of phase 2 initiates simultaneous accesses to the data store, tag store, and ITLB. By the end of phase 1 the hit status of the fetch is known. Availability of the cache for a new fetch is signalled if no previous unprocessed request remains.

For an instruction bus width of \( p \) instructions, the data store pre-accesses \( p \) consecutive words, starting at the given fetch address for the first word. Thus for a context that branches to address \( A \), the cache returns instructions
from addresses $A$, $A+1$, $A+2$, ..., $A+p-1$. In some cases pre-access may
wraparound from the end to the beginning of the data store address space.
Instructions are routed by the pre-access network and driven onto the
heavily loaded instruction bus during the third phase of the access.

Assuming a cache hit, the first returned word is always a valid instruction;
if the block of $p$ instructions lies wholly within the addressed cache line,
then all $p$ returned words are valid instructions. However in general, pre-
access may return some words belonging to an adjacent cache line—which
may or may not be contiguous with the addressed line. This situation is
resolved by recording a “lines-run” bit in the tag store entry: if set, the run
bit indicates that all $p$ instructions are valid, otherwise only those instruc-
tions that lie within the first line are flagged as valid. Access across cache
lines could be disabled optionally for experimental purposes; see Section
6.1.1 for results.

![Diagram of cache address, line number 0, line number 1, set no., tag set, pre-access, valid tag, LRU, Data, allocate 0, allocate 1, RUN, LRU, INV, Figure 3.4 Line allocation on a miss in the instruction cache—for a 2-way set-associative organization (results for the example are in parentheses).]
3.2.3 Miss Access

The line-crossing feature described above relies on the miss handler maximizing the frequency of occurrence of contiguous lines. The tag store normally pre-accesses two sets of tags: one for the addressed cache set—used for hit detection—and one from the previous cache set index. Upon a cache miss, the latter set of tags are checked against the current tag to determine the presence of any potential line runs. Additionally the tags from the successor set could be checked, but this requires an extra level of pre-access that is unjustified owing to the tendency for instruction fetches to sweep forward through memory. Given the opportunity to maintain a run of lines, the miss handler chooses the appropriate line number, otherwise an LRU choice is made. Figure 3.4 illustrates the line allocation mechanism for a 2-way set-associative cache. The example given in the figure causes line #1 to be allocated—even though line #0 is marked invalid—so as to continue the run of valid lines with tag value 36.

Apart from the missed fetch address, entries in the miss queue record a hit status, lines-run status, and line number—which are all subject to change in the life of the miss. For each genuine miss arriving at the head of the miss queue, the miss handler requests from the external cache the primary line containing the original fetch address. After an indeterminate delay the filled line reload buffer is written to the data store in one cycle, during which fetch requests are blocked. The tag store is also updated in this time, with the lines-run flag marked in the previous set where contiguous lines would be formed. Since the line allocation of the serviced miss can interfere with as yet unprocessed misses, the remainder of the miss queue is searched associatively for accesses that touch the same set and tag, and matching entries are modified. As a result, queued misses may become congruence hits—satisfied by the just completed miss, have their line allocation and lines-run status changed, or miss again due to a flush. Congruence hits avoid the cost of accessing secondary cache, but the tag and queue updates proceed as normal. The reordering of new fetches with respect to queued, related congruence hits is of no consequence due to the I-cache being read-only.

3.3 Functional Units

All operational instructions execute in the heterogeneous set of functional units. The instruction set architecture explicitly recognizes 8 classes of FU, which implementations of the processor can distinguish for exploiting parallelism cost-effectively. Chapter 2 relates FUs to the architecture of
the datapath. For each functional class, the number of real FUs can be configured from one to at most one per context group. The former end of the range, adequate for configurations containing up to 4 groups (see Chapter 7), is comparable in complexity to that of datapaths in some current superscalar processors [Vege92, DiAl92]; the latter end of the range, however, is expected to be feasible when integration densities improve.

![Anatomy of a pipelined functional unit](image)

**Figure 3.5** Anatomy of a pipelined functional unit (shared by 2 groups).

### 3.3.1 Pipeline Structure

The essential elements of a FU is shown in Figure 3.5. Each physical FU in a functional class may receive operands from one or more context groups (on a corresponding number of operation buses), depending on the level of sharing for that class. Where there exists more than one FU for a functional class each FU services a fixed subset of the context groups. Arriving instructions and data are queued in the operand queues, from which an instruction is selected in each cycle for execution in the pipeline. The vacancy status flags of the operand queues are accessible by each group's contexts to interlock their pipelines. Round robin arbitration among the queues avoids overflows on any particular queue. Results emerging from the pipeline enter a result queue, which drains to the result buses under
control of the write back arbiter.

An instruction selected from the operand queues is delivered to the pipeline early in phase 1. The control pipeline manages pipestage operations and interlocks while completing decoding of instructions; destination tags accompany their instructions in this pipeline. Pipelines in the simulator all have a throughput of one result per cycle, but the result latency is configurable. As an optimization, some FUs allow early completion for fast instruction variants. For example, although all integer add instructions share a common pipeline, additions of word-sized operands require less stages than for long word operands, allowing addw results to bypass the remaining (unused) pipestages. However, if multiple results become available simultaneously then priority is given to the longest latency operation. At the end of phase 1 of any cycle, queued and impending results (whose presence are known) are selected by the write back arbiter for write back. To facilitate bypass of write back, the arbiter transmits the destination tags of the winning results at the start of phase 2. The results themselves develop on the group result buses during phase 2.

The transmission and selection of operands and results all lie on the critical path of execution. Without any optimization of these actions, the instruction pipeline is vulnerable to lengthening in real implementations. The essential role of the arbiters is to ensure fair access to resources when contenders are queued; when contenders arrive at any empty queues a continuation of the scheduling policy is desirable, but unwarranted for maintaining good utilization of the shared resources. Accordingly, for non-empty queues a strict round robin policy can be adopted—giving way to a simpler and faster policy in the case of first arrivals. Arbitration reverts to round robin policy as the queues fill. Such a strategy allows the arbiters to resolve contention by queued requests over a whole cycle, whereas a first-in-line arbiter can be used to bypass its feed queues and select from new arrivals in two gate delays [John89].

In configurations of Concurro with large numbers of contexts the lengths and corresponding propagation delays of buses in the datapath become significant. For low-latency operations the time to transmit operands and results can approach the latency of the operation itself. Simulated and experimental tests of ULSI circuit buses by Nakagome et al. [Naka+93] indicate a lower bound transmission delay of the order of 1 ns for picofarad loads. In the absence of a circuit design for Concurro's datapath, it is estimated that at least half of a 4 ns cycle will be required for the transmission of data during the execution stage of an instruction. This leaves approximately half a cycle for the completion of logic functions and fast
integer adds. Section 7.2.1 explores the possibility of providing an extra cycle for each operation. Although FUs are notionally connected by buses, a sophisticated implementation can be expected to provide dedicated data paths and optimize the location of critical FUs—such as the load/store and integer units—to minimize their transmission delays.

3.3.2 Load/Store Unit

The load/store unit—introduced in Section 2.3.4—presents a regular FU interface to the data cache and main memory subsystems. Memory instructions supply a 32-bit virtual base address and a byte offset to the load/store unit. For stores these address components appear in the low and high words of one of the 64-bit source operands, with the store data as the other operand; other instructions use both operands for the address components. The offset may come from a register or, more frequently, the instruction word's sign-extended immediate field. Since byte offsets are restricted to 16-bit values (sign-extended from bit 15), the size and latency of address adders can be reduced to below that of adders in the integer FU; thus, for example, a conditional-sum strategy [Skla60] may be used to rapidly obtain the upper 16 bits of the 32-bit effective address. Finally, formed effective addresses index the data TLBs to yield physical addresses for the cache and main memory.

Extensive queueing is essential for absorbing irregularities in the request and data traffic of the D-cache and main memory ports. Load data returning from the D-cache passes through a load queue, which, apart from performance reasons, is necessary because the cache controller cannot be blocked at result delivery. The load/store unit (and synchronization controller) reserves entries in the load queue ahead of use, blocking loads when future queue overflow is indicated. Since queued loads may return abruptly from a non-blocking controller, the length of the load queue is set to the depth of the controller's miss queue (see Section 3.4.3) plus 4 entries for producer-consumer decoupling.

Each direct access memory port is equipped with separate load and store queues that receive uncached loads and stores, respectively. This permits independent loads and stores to pass each other at opportune moments when read or write bandwidth becomes available, thereby maximizing the utilization of the bidirectional data bus and independent address bus of each port. However, if data hazards are detected between newly arriving requests and those already buffered, then loads may be redirected to the store queue, and vice versa. Not doing so could violate self-consistency as the queues are able to drain independently. As destination tags cannot be
sent to main memory, load destination tags join a loads-pending queue, where they wait to be united with their load data in FIFO order. To cater for prolonged periods of load activity this queue has sufficient capacity to hide the main memory latency before overflow. New entries appear in the loads-pending queue when loads are dequeued from the load or store queues, as the ordering of accesses is finalized by this time. Potentially better performance, though at greater cost, can be achieved through explicit rather than implicit tagging of main memory references [RaHa93].

3.4 Data Cache

The data cache forms a major subsystem of the load/store unit. An outline of the cache subsystem is depicted in Figure 3.6. Operations on I-structures and M-structures are intercepted by the synchronization controller, whereas other accesses pass directly to the cache controller. In normal use the D-cache was configured as a set-associative, non-blocking cache of a size suitable for on-chip integration. Dual-porting of the cache (indicated in Figure 3.6) was available on the assumption that it was necessary to support the increased load/store traffic of large processor configurations.

3.4.1 Organization and Interface

The data cache comprises the data and tag stores (SRAM), the hit-access logic of the controller, and the miss handler of the controller. The cache is virtually indexed and physically tagged, with size and associativity configurable for experimentation. Dual cache ports are implemented at lower cost by dividing the data store into independently accessible banks, rather than dual-porting the SRAM [AlAv93]. However, for the smaller tag store and data TLB (DTLB) true dual-porting is feasible. Integrating the cache on-chip permits the implementation of a line reload buffer (LRB) for refilling an entire line of the data store in one cycle [Hard+90]. This feature is especially beneficial in a non-blocking cache as it minimizes the time for which the stores are unavailable for other accesses.

The access initiation portion of the cache controller and miss handler operate autonomously with respect to each other, with one unit stalling the other only when a shared resource must be accessed; misses always take priority in this case. The dedicated main memory port is shared for cache write-throughs and reloads.
Figure 3.6 Organization of the data cache subsystem (dual-ported).
Access requests address the cache through a little-endian byte address and an access size specifier—one of 8, 16, 32, or 64 bits. Illegal addresses (incorrectly aligned) are currently ignored and may produce unpredictable results. Cache ports are split into independent channels for incoming requests and cache responses, allowing the cache to return load data while simultaneously accepting store data. The cache may block incoming requests on bank conflicts or during miss servicing. To permit arbitrary reordering of accesses, destination tags passed to the cache controller must be bound to loads for the duration of their processing. Load data returns sign-extended to 64 bits, accompanied by its destination tag for routing by the load/store unit.

### 3.4.2 Synchronization Controller

The synchronization controller (SC) co-ordinates the execution of I-structure and M-structure (structure) operations sent to the cache subsystem. The SC observes the stream of requests entering the cache controller (CC), waiting for a structure access to appear. The initial structure access is always interpreted by the CC as a load of state information from the structure. When the result of that load emerges, a microsequencer in the SC will be ready (executing a wait loop) to branch in accordance with the operation and structure state to a handling routine in microcode for completion of the access. A busy flag is set by the SC during a pending structure access to force atomicity of structure accesses by the load/store unit. Structure data is returned to the load/store unit when the cache result buses would be otherwise idle. The SC is pipelined throughout, and allows independent imperative (unsynchronizing) cache accesses to proceed concurrently with, but at a higher priority than, structure accesses, thus maximizing availability and utilization of the data cache.

The microsequencer implements a vertical, explicit sequence microinstruction set [RaAd80] of 16 instructions. The microinstructions are specialized three-address operations on 8 registers, including computed jumps, cache loads and stores, field manipulations and moves. The registers consist of 5 scratch registers, 2 special-purpose registers set by the SC upon receipt of an operation, and the free-list pointer system register. The entire microprogram for executing I- and M-structure loads and structure stores occupies 48 instruction words, and is listed in high-level form in Appendix B.

An important goal in designing the firmware was to minimize the latency of frequent access situations. Loads from a full structure and stores to an empty structure are executed in effectively one microinstruction each. Deferred loads are split into two categories: single-waiting, where a single
load from the location is deferred; and multiple-waiting, where more than one load is waiting for the store. From benchmarking it was apparent that structure stores satisfy an average of less than one deferred load each. Thus, it is profitable to distinguish single-waiting loads as these can be satisfied with less overhead than required for the other cases. Further optimizations for double-waiting loads, and similar, are possible, but were not implemented.

The state transition diagram for structure accesses is shown in Figure 3.7. As indicated in Figure 3.8, the contents of a 64-bit structure long word determine its state. Structures holding deferred loads indicate whether they are I-structures or M-structures, along with either a destination tag (for single-waiting loads) or a pointer to a linked list (for multiple-waiting loads). Each node of a multiple-waiting list is a long word split into a destination tag and a pointer to the next node in the list, ending with a pointer to ground. List nodes are allocated and released dynamically from a free-list of nodes pointed to by a system register, which initially points to a pre-defined linked list. The free-list may be set up in user memory at run-time, permitting programs to retain a record of their deferred structure accesses, and providing a mechanism for detecting and correcting free-list overflows through memory exceptions.
Chapter 3 Subsystem Architecture

### 3.4.3 Cache Controller

The CC is a collection of state machines that coordinates the use of cache resources. The controller was designed to minimize load latency and maximize access bandwidth—even under heavy miss-rates—to tolerate the demands of multithreading. Both dual- and single-ported configurations were studied—the dual-ported configuration, shown in Figure 3.6, is described here, as it includes the single-ported case.

Cache accesses are either performed upon receipt, or latched for later processing when a miss completion takes priority. If two accesses are available simultaneously they may proceed together provided that they access different cache banks; in the case of bank conflict, one access is blocked while the other is accepted by an oscillating selector updated by the clock. As cache accesses cannot cross long word boundaries, the data store is banked with a long word granularity. A fine granularity is preferred, as it allows two accesses to the same cache line—particularly for long word accesses. Given $B$ cache banks, the probability of two independent, random accesses hitting the same bank is $1/B$, resulting in an average

![Diagram showing structure state encodings.](image)

**Figure 3.8** Structure state encodings.

<table>
<thead>
<tr>
<th>Structure state</th>
<th>State field</th>
<th>Pointer field</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>XXXXXXXXXX</td>
<td>low-order word of datum</td>
</tr>
<tr>
<td>empty</td>
<td>00000000</td>
<td>unspecified</td>
</tr>
<tr>
<td>single I-load</td>
<td>00000001</td>
<td>destination tag</td>
</tr>
<tr>
<td>multiple I-loads</td>
<td>00000100</td>
<td>pointer to deferred list</td>
</tr>
<tr>
<td>single M-load</td>
<td>0000101</td>
<td>destination tag</td>
</tr>
<tr>
<td>multiple M-loads</td>
<td>0000110</td>
<td>pointer to deferred list</td>
</tr>
</tbody>
</table>

**Diagram:**

- 63 40 32 0 aligned long word
- L/M-structure
- 11...1 = not full, else structure full
- State field
- Pointer field
- Deferred list node
- Destination tag
- Pointer to next node; 0 = last

- 63 32 0 aligned long word
bandwidth of $2 \times (1 - 1/B) + 1 \times (1/B)$ or $2 - 1/B$ accesses/cycle. Thus, bandwidths of 1.50, 1.75, and 1.88 are predicted for 2, 4, and 8 banks, respectively. Real access patterns are not random, but we can expect, nevertheless, the delivered bandwidth to approach 2/cycle for a modest number of banks.

The CC receives preliminary accesses at the start of phase 2 of any cycle, with the remaining high-order address bits sent by the load/store unit after translation in the DTLB. A simple check of low-order address bits, the write queue capacity, and the state of the miss handler determine which, if any, of the new accesses can proceed in the current cycle. The data store is indexed by cache set number, while the tag store and miss queue are accessed for miss status. By the end of the next phase, phase 1, the output of the DTLB can be compared with that of the tag store to determine a hit and signal cache results to the load/store unit. Cache busy lines are asserted by the end of phase 1 for unconsumed access requests; SC accesses are serialized in the dual-ported cache to maintain sequential consistency for synchronization operations.

Assuming a hit, load data is selected and returned on phase 2, and store data is latched for delayed write. Stores perform a read-modify-write operation on the data store to insert quantities shorter than a long word, and produce a new long word entry for the write queue; merging of adjacent sub-long word quantities was not implemented. The write queue drains to main memory at the maximum rate of one entry per cycle, however, dual accesses may cause up to two stores to be enqueued in a cycle. The write queue capacity is set to the line size plus a longword entry for each port, reflecting the longer periods of write blocking that occur upon reload of large cache lines.

Misses occur if either no address tags matched the DTLB output, or the access address hits in the miss queue. At this stage misses are considered potential, as the miss handler may be in the process of correcting them. The miss handler (Figure 3.6) implements non-blocking miss recovery [Krog81, StDL91] and pipelined cache reload. These optimizations bring significant advantages to a multithreaded processor, as they allow hit accesses from several threads to occur while, possibly, multiple previous misses from other threads are being serviced. Controller configurations with from $p$ miss queue entries (for a blocking cache with $p$ ports) to $n+p$ miss queue entries (for tolerance of hits under $n$ misses) were studied.

Upon detection of a missed load, store or flush, the cache address, destination tag or store datum, word size, and LRU line allocation read from the tag store are placed in the miss queue. Since dual accesses may cause two
misses in a single cycle, the miss queue must contain at least two spare slots prior to any accesses. The miss queue is split into separate request and reload queues in Figure 3.6 to clarify the operation of the miss handler. The head of the request queue is acted on by the fetch engine, which normally sends to main memory a burst of read requests for filling a cache line in a critical-word-first pattern. If any portion of the requested line appears in the write queue, this must first be allowed to drain. Upon completion of its read burst, the fetch engine passes the head of the request queue to the reload queue, where it will wait while the fetch engine continues with subsequent requests. Without pipelining misses, cache reload makes poor use of main memory, reducing the cache bandwidth under heavy miss rates. For example, given a 20-cycle main memory latency and cache lines of 8 long words, the best unpipelined reload utilization is \((8/(20+8))\times100\%\) or 29%; by contrast, well in excess of double this utilization was achieved with pipelined reload.

Reload data returns in-order from main memory to be buffered in the LRB by the reload engine. For a load, the reload engine returns the requested word, as soon as it arrives, to the destination specified by the head of the reload queue. Once filled, the LRB is written in one cycle to the data store, while the tag store and request queue are updated, and the reload queue is dequeued; no cache requests are accepted in this cycle.

This miss handling mechanism is complicated by three occurrences: congruence hits—hits arising from the completion of preceding misses; line allocations changing due to preceding misses; and new accesses arriving out-of-order with respect to congruence hits. Congruence hits are commonly caused by clusters of accesses to a missed cache line; this devalues the effectiveness of non-blocking under misses by wasting miss queue entries and potential parallelism. To avoid duplicate cache reloads, the fetch engine tests the head of the request queue against all entries in the reload queue; if the request cannot be satisfied by any others in progress (that is, the request is independent), it initiates a fetch sequence, otherwise the fetch engine stalls until the reload queue has drained, at which time the congruence hit is serviced. Cache flushes are treated similarly to congruence hits.

Each entry in the miss queue contains a hit flag and a line number for replacement. New entries are initialized with the hit flag cleared and a recommended line number from the tag store. At the conclusion of each miss service, the request queue is searched associatively (against address tag and set) for entries that will be affected by this miss. This allows the line numbers of unprocessed misses to be changed to avoid duplicate cache tags, and catches late congruence hits, which have their hit flags set.
Subsequent cache flushes or line reuses may reverse a miss entry's hit status.

A non-blocking miss handler can reorder memory references in the cache [StDL91]. Self-consistency, however, must be preserved. Ordinarily hit accesses are retired in the order they arrive, while dependent misses are sequentialized in the miss queue. This simple mechanism fails when an access arrives interspersed between a dependent miss that has been serviced, and a congruence hit yet to be serviced. Correct sequence is enforced by associatively consulting the miss queue, simultaneous with the tag store read, for each access, and placing dependent accesses into the request queue as congruence hits. The various associative look-ups described here are not particularly expensive, since practical miss queues are only several entries long.
Chapter 4

Experimental Method

The experimental evaluation of Concurro is introduced in this chapter, now that its macroarchitecture and microarchitecture have been established. The simulator and its utility software were used to assess Concurro processors at a diverse set of points in the design space. A representative workload, in the form of a benchmark set, provided a basis for comparative studies in addition to obtaining absolute performance data.
4.1 Simulation Technique

Experimental work centred on a detailed behavioural simulator of the Concurro processor and its immediate environment. Figure 4.1 is a schematic of the simulation process. Benchmark programs were written in Concurro assembler and assembled by the cass macro assembler. The object module and any required library modules were linked by the linker, clink, creating a binary file that was directly executable by the concurro processor simulator (CPS). CPS’s internal parameters were set from a configuration file, with the simulation process controlled by a command script.

Figure 4.1 Flow of data in the simulation process.
4.1.1 Concurro Simulator

The Concurro processor was implemented in software at effectively register transfer level of detail. The execution-driven CPS was written in C and designed to be highly portable to enable large sets of simulations to be run in parallel on a heterogeneous workstation farm. Running on a DEC AXP™ 4000/710, CPS processed 16 active contexts at approximately 5000 cycles/second.

CPS provides a virtual memory environment for executing programs similar to that usually available under UNIX® [Bach86]. Two programs may be resident, the supervisor and the user program, with memory management allocating private code, data, and stack regions for each. To preserve the authenticity of this model, executable files are loaded in their native binary format. Concurro is at any one time executing either in system mode (addresses in the supervisor address space) or in user mode. Hardware or software interrupts cause a change in mode. Upon start-up, the supervisor sets registers and system data structures, and allocates a stack for the user program; subsequently the supervisor handles system calls and terminates the simulation when the user program exits. A region of supervisor memory provides a special memory-mapped interface to the simulator, allowing access to CPS's environment and a small set of the host operating system's system calls. In this way, simulated programs can perform input and output to the host file system.

The configuration of the processor microarchitecture is altered from its default with a configuration file listing parameter names and their numeric value. Execution address break-points are used as the basis for controlling the instrumentation of CPS. Simulation scripts contain a series of address labels and associated lists of actions, with the labels being defined as global by the assembler. When the instruction pointer of a chosen context coincides with a label, the associated actions are executed. Actions include starting and stopping traces, logging statistics or comments, dumping sections of main memory to files, and executing host operating system commands. A sample script is shown in Figure 4.2.

Instrumentation variables in CPS are updated every cycle. Scripts have the option of reporting statistics as instantaneous values, difference values, cumulative tallies, or frequency histograms. Traces on any of the statistics may be either unlimited (cycle-by-cycle basis) or aggregated into a specified number of totals over part of the simulation period.
CPS is an ad hoc discrete-event simulator with fixed-time step [HoPe89]. The design of the simulator resembles real hardware—a two-phase global clock controlling synchronous processor logic subdivided into a collection of communicating modules. The two-phase latches separating major modules allow them to execute independently of each other within a clock phase by ensuring that a bus is written only on one phase (phase 1 or phase 2) and read in the other phase. Half of the processor subsystems are exercised in phase 1, while the remainder are exercised in phase 2. In each cycle break-point hits are checked and active traces are updated. The generally high levels of activity throughout the simulator makes this fixed-time step technique competitive with an event tracking approach.

```plaintext
# Sample simulation script
start {
    show general; # time stamp log file
    # show where it stopped
    log "Break-point at symbol start:\n";
    # start 100-point new trace on delta operations
    trace ops.trc new d-operation 100;
}

next_bit {
    log "Break-point at symbol next_bit:\n";
    # show thread activity and register contents
    show threads registers;
    # show cycle and floating operation counts
    log "cycles =" cycle "FLOPs =" flop ":" v-flop;
    # show memory activity
    log "mem reads =" memread "mem writes =" memwrite;
}

drop {
    close ops.trc; # stop trace
    # send notification of end
    shell 'Mail -s concurro gunther' "Finished!\n";
    stop 0; # exit with code true
}

Figure 4.2 A sample Concurro break-point script.

4.1.2 Support Software

Portability was also a design goal for the support software so as to allow the free interchange of source, object, and executable files across hosts. Assembly codes generated manually or by compiler are translated by cass—a relocating, optimizing macro-assembler. Cass supports most usual assem-
bler features, such as address space segmentation, conditional assembly and file inclusion, instruction macros, symbol scoping, register naming, 64-bit arbitrary expressions, data alignment, and floating point constants.

The instruction mnemonics of Concurro assembly code are listed in the instruction set summary of Appendix A. Mnemonics of the base instruction set map one-to-one with machine instructions. However, cass also supports an extended instruction set, which presents a richer ISA, allowing for 32-bit constants and higher level operations. Extended instructions translate to one or more base instructions, with constant usage and instruction selection optimized to minimize instruction count and operation latency. An extended instruction set (such as, for example, that supported by MIPS® assemblers [KaHe92]) eases manual coding and simplifies compiler back-end design without significantly sacrificing code quality, as the assembler has sufficient knowledge of its target architecture to make good code selection choices.

As an aid to both manual programming and compilation, instruction reordering within basic blocks was added to cass to improve utilization of Concurro's instruction pipeline. Only a simple scheduling heuristic was implemented. A critical path method analysis is applied to instructions in a basic block, before instructions are emitted in ascending order of earliest issue time, thus preserving dependencies and eagerly filling pipeline bubbles. The resulting schedules, while not always optimal for fixed pipelines, provide most codes on Concurro with better tolerance of its variable length instruction pipeline. The scheduler does not let synchronization instructions pass each other, but memory operations that can be statically disambiguated may be reordered.

Sets of traces generated by CPS were collated and converted into normalized tables with the traceview utility. This program performed period averaging and collected elementary statistics on its input data streams. A portable dump viewing utility, adump, was used to examine memory array dumps from the simulator in binary form. Floating point, 64-bit decimal, hexadecimal, and octal output formats are available for any of four array element sizes.

4.2 Benchmark Programs

The performance of various processor configurations was determined from their speed in executing a set of six benchmark programs. In the absence of a parallelizing compiler, the benchmarks were coded directly in assembly
language and manually optimized, limiting the size and complexity of the programs. As a consequence, the benchmarks consist of small realistic applications and synthetic benchmark programs. Each benchmark was assembled only once, that is, the same executable files were run in all simulations. Although this limited the scope of some optimizations, it reflects the practical requirement of binary compatibility across multiple platforms spanning a range of cost/performance points.

```c
#include "concurro.h"

double global_a;
int join_cnt;
channel int ready;

daxpy(double a, double *x, double *y, int n)
{
    int i;

    for (i = 0; i < n; ++i)
        y[i] = y[i] + a*x[i];
}

(a) original program

(daxpy(double a, double *x, double *y, int n)
{
    int i;

    global_a = a; join_cnt = n / 4; clear ready;
    /* do 4 iterations per child thread */
    for (i = 0; i < (n & ~3); i += 4) {
        fork daxpy_body(x, y);
        x += 4; y += 4;
    }
    /* do remaining iterations in control thread */
    for (i = 0; i < (n & 3); ++i)
        *y++ = *y + a * *x++;
    if (ready); /* dummy test for ready */
}

(b) threaded code

thread daxpy_body(double *x, double *y)
{
    y[0] = y[0] + global_a*x[0]; /* unrolled 4 times */
    y[1] = y[1] + global_a*x[1];
    /* barrier synchronization */
    if (--join_cnt <= 0)
        ready = 1;
}
```

Figure 4.3 Multithreading the vectorizable DAXPY loop on Concurro.
Decomposition, data placement, and threads were optimized extensively—emulating parallelizing compilers. All processor configurations benefit from the majority of low-level optimizations, which are thus factored out simply by comparing processors executing the same benchmark codes. Most of the classical scalar optimizations [AhSU86] and standard loop transformations [ZiCh91] were applied to thread code where appropriate. Register allocation restricted local variables and arguments of functions to the set of local registers (r16 to r31), with frequently used, long life-time global variables allocated to the global register set. For leaf threads the conventional stack pointer and link registers were returned to the allocation pool. Benchmarks were expressed in the extended instruction set (Appendix A) as far as possible. In many cases, conditional sequences of three instructions or less could be replaced by their equivalent conditional move sequences to reduce the branch penalty.

Loops provided the major opportunity for exploiting parallelism. For many recurrence relations, modulo variable expansion and software pipelining [Lam88] proved effective in improving the performance of even single-thread code. A simple threading strategy was applied to all vectorizable loops. The root thread is a controlling thread, forking loop bodies and tracking iterations, as illustrated in Figure 4.3 for the DAXPY loop expressed in an augmented C language. Once forked, each child thread reads parameters and induction variables from register channels before executing a fixed number of loop iterations. With only one control thread forking, however, the rate of thread creation on large processor configurations can not match the termination rate unless the thread size is artificially increased. More sophisticated forking strategies permitting finer grained threads were avoided to retain configuration independence. Loop bodies of less than 20 instructions were unrolled a maximum of 4 times, less if register spilling occurred. For Concurro partial loop unrolling is a particularly useful optimization because it increases the effective concurrency available—beyond the fixed context limit—to improve latency hiding.

In the simplest case a barrier synchronization concludes loops. Iteration threads simulate a multi-way join, through either a global register or M-structure counter, with the last thread to finish signalling the control thread. Simple recurrence relations that are recognized vector reductions, such as element sum or greatest value, were threaded as for vectorizable loops, but with the dependent variable expanded to an array of M-structures, which served to synchronize the pipelined threads and, ultimately, the consumer of the result. More general linear recurrences were most often coded as vectorizable, except that the vectors giving rise to the dependencies
were replaced by I-structure vectors. This approach introduces only a minor performance disadvantage with respect to the fully vectorizable case.

The following subsections describe the benchmark programs in more detail.

4.2.1 Carries

Carries tallies the frequencies of the longest runs of non-zero carries for all possible 16-bit binary additions. Carries is derived from a program used by the author for investigating the design of asynchronous adder circuits. This benchmark is a readily parallelized integer search that updates a central M-structure histogram array. Concurro's ability to sustain high instruction issue rates is particularly tested by carries.

The inner loops of this program contain several conditional branches that were optimized into conditional moves; the unoptimized code, using branches, also makes for an interesting workload, and is examined in Chapter 6. Carries consists of 236 lines of assembly code, which was manually scheduled.

4.2.2 Compress

Compress is an implementation of the UNIX compress utility for compressing data using adaptive Lempel-Ziv coding. This integer program is difficult to parallelize and provides a single threaded workload for Concurro. Compress is characterized by short run lengths and poor data cache performance, owing to its use of a hash table as its principal data structure. The benchmark reads and compresses the 22,300-character introduction of the Jargon File, version 2.9.12 [Raym93]; data is read from and written to in-memory "files" to avoid simulating the I/O system, although portions of the read/write routines are executed. The 851-line benchmark was directly translated from original C source, and automatically scheduled by the assembler.

4.2.3 Gauss

Gauss solves a 200×200 dense system of linear equations by Gaussian elimination with partial-pivoting and back-substitution. This program is both a common benchmark (as a reduced form of LINPACK-1000 [Dong88]) and an important supercomputer application. Gauss relies on I-structure and M-structure operations to manage its varying parallelism, testing Concurro on workloads that take it in and out of saturated multithreading. The benchmark also features a rich operation mix, including floating point arithmetic, cached and uncached memory accesses, and address arithmetic. The forward elimination algorithm was modified to use two pivots per pass
of the coefficient matrix in order to increase the computational intensity; the coefficient generation phase is not timed. Gauss is 970 lines long and was scheduled manually.

4.2.4 Loops

Loops is a collection of eight of the Livermore FORTRAN Kernels [McMa86]. Kernels 1 (hydro fragment), 3 (inner product), 5 (tri-diagonal elimination), 6 (general linear recurrence equations), 7 (equation of state fragment), 10 (difference predictors), 11 (first summation), and 12 (first difference) are executed in turn, with a barrier synchronization applied at the end of each loop. These loops are traditional benchmarks and represent common scientific workloads of both a vectorizable (multithreaded) and sequential (single threaded) nature. In combination, the kernels in loops require a balance between bandwidth and latency to minimize total execution time. All imperative accesses to floating point arrays in loops are uncached. Code for this 1097-line benchmark was scheduled manually.

4.2.5 Matmult

Matmult multiplies two 100x100, double-precision matrices, producing a separate product matrix. This benchmark represents a frequently used operation in linear algebra. Matmult is a well-characterized and highly parallel floating point program that stresses Concurro's floating point and memory bandwidth. A blocked algorithm, with each thread computing a 2x2 submatrix of the solution, was used to enhance the demand for floating point throughput. This 128-line benchmark executes no cached loads or stores, and performs a simple barrier synchronization at its conclusion. Manual code scheduling was performed on matmult.

4.2.6 Tf

Tf is a simple text formatter that reads plain text and outputs it as justified paragraphs. This benchmark is an extract from the tf text formatting utility in wide use in the Department of Computer Science, at the University of Tasmania. This integer-only program typifies the workload of character-based system utilities and simple language translators: byte and pointer operations interspersed frequently with conditional branches. Tf is written as a four-stage macropipeline, placing most configurations of Concurro into linear mode multithreading. The benchmark reads a 22,300-character in-memory file (the same used by compress) and writes the formatted output to another file. 1002 lines long, this benchmark had its code automatically scheduled by the assembler.
4.2.7 Benchmark Characteristics

The stack chart of Figure 4.4 shows the dynamic instruction counts and break-downs by type for each benchmark program. Control instructions include branches, conditional moves, forks, and terminates; direct memory instructions comprise the uncached load and store instructions; and miscellaneous instructions cover register channel, and integer multiply/divide
operations. Three of the benchmarks, carries, compress, and tf, fall neatly into an “integer” category, while gauss, loops, and matmult belong to the “floating point” or scientific category. The break-downs suggest that processor performance on the integer set is likely to be highly dependent on the characteristics of the control and integer execution units, whereas the floating point and load/store units are expected to play more important roles for the scientific codes.

![Bar Chart](image)

**Figure 4.5** Overall and floating point issue rates for model 104.

The performance of the base-model Concurro (see Section 4.3.1) running each benchmark is plotted in Figure 4.5 as instructions per cycle (IPC) and FLOPs per cycle (FPC) issue rates. Since the base model consists of only one context group, operational instructions cannot be issued at a rate exceeding 1 IPC. Three of the benchmarks, however, have average issue rates exceeding 1.0 IPC on account of control instructions that can issue in parallel with operational instructions. As expected, the benefits of multithreading in improving utilization are evident for all programs except compress, which contains just a single thread of computation. FLOP rates are well below the 2.0 FPC datapath limit of model 104, reflecting the effects of the instruction issue bottleneck.
4.3 Performance Evaluation

4.3.1 Simulator Configurations

From the many possible configurations of the processor, one was chosen as the baseline model, for scaling comparisons, and three configurations were chosen as reference or standard models, representing realistic configurations at distinct cost and performance points. Since the maximum instruction dispatch rate and datapath bandwidth are determined by the number of context groups—or group count, then this is the chief determinant of processor performance and size. The parameters of the standard models, models 208 (2 groups, 8 contexts total), 416, and 832, are listed fully in Appendix C. A model 104 configuration represents the base model Concurro; this and other standard models are all scaled according to their group count, as specified in the notes of Appendix C. The scaling heuristics are based on observations of Concurro executing the benchmarks and estimates of implementation costs.

The standard models represent attempts at defining microarchitectures at local cost/performance minima. Hence, these configurations do not necessarily maximize performance or minimize complexity within the constraints of their group counts. Context groups are allocated 4 contexts each, a compromise between providing latency tolerance in scientific workloads and maintaining adequate context utilization in less parallel integer codes. FU latencies are set on the assumption that current implementations will scale with reduced feature size. A special case, however, is the latency of uncached memory accesses.

In setting the latency of main memory at 25 cycles—or 100 ns for a 250 MHz clock—and bandwidth at 1 long word per cycle per port, it is assumed that some improvement in DRAM performance will accompany improvements in processor implementation. New DRAM technologies involving caching and high speed interfaces [Przy93] are expected to lower access times and provide high bandwidth at practical interleaving factors [HsSm93]. The 25-cycle figure includes losses due to bank conflict and TLB misses.

Since the throughput of many numerical programs is limited by memory bandwidth, the standard models are supplied with memory systems that allow Concurro to operate in the supercomputing domain. Accordingly, 1 long word/cycle of uncached bandwidth (1 direct port) is provided for every 2 FLOP/cycle of floating point add/multiply bandwidth in the datapath, plus dual-porting of the D-cache is enabled for group counts greater than 4. Less expensive systems, explored in Chapter 7, must also be suitable in
practice, but they would unduly restrict the scope of this study if they were prescribed for the standard models.

4.3.2 Procedure

Apart from being workloads, the benchmarks provided assurances of correctness of the simulator and development tools. The output of each benchmark run on Concurro was checked against its output from either a trivial manual calculation or the execution of an original version, compiled from a high-level language. Concurro had to duplicate either the known correct results exactly or the original results to within an acceptable tolerance, and without any operating exceptions, in order to pass these tests.

The majority of experiments described in the next chapters followed a common procedure. The executable binaries of the benchmarks and a comprehensive simulation script (see Section 4.1.1) were distributed among several hosts for running simulations concurrently. For each experiment, processor configurations were derived from the standard models' master configurations and used in multiple simulations. These new configurations were relatively localized deviations from the standard models, designed to test the influence and effects of particular features. This approach has its limitations, however, as the interaction between architectural features is rarely one of true linear superposition. Since it is not feasible to simulate every mix of parameters, only strongly interacting features were tested together. A more accurate traversal of the cost–performance spectrum is carried out in Chapter 8. Once the simulators' log and error files had been checked, the key statistics of the experimental results were extracted and collated by software for analysis.
Chapter 5

Organization Alternatives

In this chapter the processor organizations defined in the previous chapters are compared against each other, in terms of scalability and implementation cost, and against other significantly different organizations that may be viewed as viable alternatives to a Concurro-like machine. Through the latter comparisons it is possible to gauge the degree of optimality of the Concurro architecture across a larger design space than will be covered in subsequent chapters. Many of the major design choices made in Chapter 2 can thus be assessed.

Comparisons here are on the basis of simulated performance, with all machines running the same benchmark suite described in Chapter 4. Initially, simulation results characterizing the base configurations are presented. These are followed by performance evaluations of superscalar processors (both single and multithreaded) and multiprocessors operating out of coherent cache memory.
5.1 Scalability

Scalability refers to the potential for implementations of an architecture to yield improved performance with greater hardware resources or refined implementation technologies [Bell92]. Size scalability is rarely an important issue in a single-chip processor because integration limits usually dominate the potential complexity. Generation scalability, however, is often more relevant, but difficult to predict in advance. After over a decade of experience with RISC architectures it is becoming clear that simplified instruction set architectures, free of early-implementation artifacts, can be designed specifically to be scalable and extensible [Site93]. Accordingly, Concurro is expected to be readily generation scalable due to its register-oriented, load/store architecture.

To offer performance comparable to that of vector processors and VLIW machines, it is crucial that Concurro be scalable to integration limits. Ideally this requires performance to grow linearly with hardware complexity. Ultimately, provided that applications themselves are adequately scalable, the ability to share centralized resources, such as caches, is a major determinant of the performance range possible.

5.1.1 Configuration Size

The standard processor models have their contexts arranged into groups of 4 contexts each. In a MIMD sense, each group behaves as a latency-tolerant processing element, capable of a peak dispatch rate of one operational instruction per cycle. We can therefore expect in the presence of sufficient program parallelism for best performance to scale with the number of groups. Excessive demands on latency hiding without adequate program parallelism, however, are likely to result in under-utilization of the processor.

To determine Concurro's scalability, the benchmark suite was executed on configurations of varying group counts. The parameters of each model were set as described in the previous chapter. Figure 5.1 plots the speed-up in instruction throughput (instructions per cycle) of each model relative to the 1-group model. The curves clearly separate the benchmarks into categories of highly parallel, moderately parallel, and single-threaded programs. Small performance peaks at 2, 6, and 10 groups were caused by the discrete nature of resource allocation, giving rise to slightly unbalanced datapath bandwidths for these configurations.
The single-threaded *compress* program did not, of course, show any benefit from the additional contexts of larger models, although negligible second-order cache effects were recorded. Similarly, the single thread sections in *loops* rendered it vulnerable to an Amdahl's Law limit at higher group counts. Benchmark *tf* was restricted by being coded to use only 4 contexts simultaneously, thus limiting Concurro's issue rate and, for 10 groups in particular, latency tolerance. The more parallel benchmarks of *carries*, *gauss*, and *matmult* executed with efficiencies (100%×speed-up/group-count) from 80% to 89% on the 8-group model; at 10 groups, however, the scalability of *gauss* and *carries* had begun to deteriorate. *Gauss*, in particular, could have benefited from a finer-grained decomposition to improve efficiency on larger models.

![Figure 5.1](image)

**Figure 5.1** Execution speed-up against number of context groups.

For 2, 4, and 8 groups the harmonic mean speed-ups over the six benchmarks were 1.47, 1.87, and 2.23, respectively. For a 250 MHz clock the mean speed-up of model 832 represents a throughput of 549 MIPS, yet model 832's performance ranged from 165 MIPS for *compress*, through to 2127 MIPS for *carries*, and 862 MFLOPS for *gauss*.

All mean speed-ups reported here were obtained from quotients of unweighted harmonic mean instruction issue rates. The unweighted harmonic mean rate in instructions/cycle is defined as:
where

\[ \frac{1}{\text{CPI}} = \text{mean cycles/instruction} = \frac{\sum \text{cpi}_k}{n} \]

and \( \text{cpi}_k \) is the reciprocal of the instruction rate for program \( k \) from \( n \) programs. The addition of normalized execution times in the calculation of CPI gives the harmonic mean the desirable property of reflecting the total run time of the program set, thereby presenting a fair measure of overall system performance from a user's perspective [Smit88]. However, this property of the harmonic mean also tends to accentuate Amdahl's Law: regardless of how fast the parallel programs or parallel portions of the programs are executed, the CPI value will be bounded below by the normalized execution time of the serial part of the code. Thus harmonic mean measures neglect the common practice of scaling applications to hardware [Gust88].

General performance comparisons between Concurro and existing machines were not drawn, although a meaningful comparison between a model 832 Concurro and a 1-processor Cray Y-MP C90 is possible on account of their approximately equal clock rate, datapath bandwidth, and memory latency [Dong93]. With Concurro executing \texttt{gauss} and the Cray running a hand-tuned equivalent, model 832 achieved for an order-200 problem similar (6% lower) floating point throughput than the C90\(^3\) on a significantly larger, order-1000 problem [Dong92]; moreover, Concurro still attained 500 MFLOPS (half theoretical peak rate) with the problem size reduced to order-60. Such a comparison suggests that for a program with varying vector sizes, Concurro is, at least, comparable in terms of both utilization and overhead with a similarly equipped vector processor—noted for its cost-effectiveness.

### 5.1.2 Concurrency

One of the goals of multithreading is to keep the processor saturated with work so as to make maximum use of its latency hiding abilities. Clearly the success of this endeavour is largely dependent on the parallelism of the workload, but the agility of the processor's task creation mechanisms also influences the utilization of program parallelism. An indication of the concurrency achieved by Concurro is shown in Figure 5.2, which plots the mean numbers of contexts active and total registers in use for each benchmark executing on model 832. Results of similar form, but different scale, were observed for lower group counts. Since most instructions write to a

\(^3\text{Cray Y-MP C90 MFLOPS were scaled upwards to adjust for its 4.166 ns, rather than 4.0 ns, cycle time.}\)
destination register other than \( r_0 \),\(^3\) the number of registers reserved indicates, in effect, the number of operations that are in simultaneous execution and being waited upon.

Context activity rates (mean proportion of all contexts active) in excess of 98% were recorded for the highly parallel benchmarks, \textit{carries} and \textit{matmult}, while \textit{gauss} averaged 91%. The remaining programs were rarely or never able to bring all contexts into use. The ratio of registers reserved to contexts active yields an approximation to the average thread instruction-level parallelism (ILP). Such parallelism is, of course, exposed by virtue of pipelining in the processor. Almost unity ILP (allowing for the execution of control instructions) appeared in the integer codes, as is expected for programs that predominantly execute low-latency integer and cache instructions. In contrast, the effects of uncached memory accesses, floating point operations, and loop unrolling contributed to an average ILP of 3.1 in the numerical codes.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.2}
\caption{Mean context activity and register usage for 8-group model.}
\end{figure}

Context activity and ILP are factors in determining latency tolerance. A first-order parallel speed-up equation, derived by the author in [Gunt90a],

\(^3\)Store instructions are treated as if they wrote \( r_0 \).
relates program parallelism to hardware parallelism:
\[ Cd = sp \]

where \( C \) is the average instruction issue rate, 
\( d \) is the mean instruction latency, 
\( s \) is the mean number of contexts active, and 
\( p \) is the average per-thread ILP.

In Concurro \( s \) is bounded by the number of contexts, \( N_c \), and \( C \) is limited by the group count, \( N_g \), giving an upper bound of tolerable latency as:
\[ d_{\text{max}} = \frac{p N_c}{N_g} \]

for execution with ILP of \( p \); mean latencies below \( d_{\text{max}} \) typically reduce \( p \) (and often \( s \)), while greater latencies reduce the issue rate and extend \( p \) to the program's inherent limit. Letting \( N_c = 32 \), \( N_g = 8 \), and assuming a maximum ILP of 4, the processor can operate at full speed provided that \( d \) remains below 16 cycles. Clearly, tolerance of long latencies is dependent on intra-thread parallelism—present either naturally or derived artificially.

For numerical codes the major contribution to \( d \) comes from memory access time, \( t_{\text{acc}} \); from the instruction breakdowns of such programs in Figure 4.4 and the operation latencies of Appendix C, \( d = 0.30 \times (t_{\text{acc}} + 2) + 0.45 \times 5 + 0.25 \times 2 \), implying, when \( d = 16 \) cycles, \( t_{\text{acc}} = 42 \) cycles. Therefore, we can expect a significant performance reduction when main memory accesses approach this latency. Section 7.2 explores latency tolerance experimentally.

Since the majority of instructions in execution eventually write to registers, product \( sp \) in the speed-up equation is approximated by the number of registers reserved. In the most parallel of the numerical benchmarks, \textit{matmult}, an average of 127 registers were reserved in model 832 (\( N_g = 8 \), \( N_c = 32 \)) at any instant—or almost 4 registers per context. This level of concurrency is presently impossible to achieve in a single threaded processor maintaining just 32 registers. Since registers (or their names, at least) serve as the sources and sinks of instructions, the parallelism accessible to a register-based machine is limited by the number of architectural registers plus their aliases that can be specified simultaneously. Sites [Site93] expects register availability to hamper performance in future superscalar processors, yet this problem may be mitigated somewhat by register renaming [Kell75].

In Concurro the 16 local registers (and 16 global registers) place a per-context restriction on ILP; consequently, thread ILPs are unlikely to exceed 8. However, having registers reusable in different contexts, Concurro does not impede scalability to the same extent that single threaded processor architectures do—with the exception of some VLIW machines, such as the Cydra 5 [RYYT89], which support dynamic register allocation for pipelined loops.
5.1.3 Context Arrangement

The arrangement of contexts into groups trades off peak instruction dispatch rates against latency tolerance. Since resource allocation in Concurro is scaled in accordance with its group count, hardware utilization is likely to be low for groups of few contexts that are unable to hide latency effectively, whereas the high datapath utilization achieved with large groups comes at the price of possibly superfluous context facilities and lengthened critical paths in the processor logic. Hardware utilization is optimized at the point where instruction rates first become immune to instruction latency. Normally, with scientific workloads, this implies that each group must sustain one instruction dispatch per cycle for practical ILPs when making frequent uncached memory accesses; integer workloads are, of course, also adequately catered for under this criterion.

Under the assumption of a worst-case computational intensity of 1 floating point operation for every uncached memory access, at an ILP of 4, and latencies of 25+2 cycles for loads and 5 cycles for arithmetic operations, the parallel speed-up equation of Section 5.1.2 indicates that the group size must reach (0.5×(25+2) + 0.5×5)/4 or 4 contexts to fully hide instruction latency. In many cases this number is more than sufficient, as the presence of operations on indices and pointers will reduce the weight of uncached memory accesses. Since the floating point benchmarks exhibit computational intensities greater than unity, the standard configurations with 4 contexts per group are expected to completely hide the dominant latency of main memory.

The adequacy of various context arrangements against those of the standard configurations were determined by varying the group sizes between 1 and 8 contexts for processors of 4 and 8 groups. Figure 5.3 shows the normalized instruction throughputs of these configurations on each benchmark. In referring to normalized throughput in these and later results we are normalizing instruction throughput of a given processor configuration to the throughput of a standard configuration having the same group count. Thus, models 208, 416, and 832—configured as described in Appendix C—are assigned normalized throughputs of 1.00 when executing the benchmarks.

For programs compress and tf, with their limited context use, the group size was immaterial to performance, but the remaining benchmarks showed varying sensitivities to group size. For the more parallel benchmarks of carries, gauss, and matmult the instruction rate was almost directly proportional to the total number of contexts up to group sizes of 4, as is
expected for linear-mode multithreading. Normalized throughputs of at least 0.7 for 2-context groups suggest that the transition between linear and saturated modes of multithreading occurs in the region of 3 contexts per group for these codes, while the absence of any significant performance improvements above 4 contexts per group demonstrates that full latency hiding was achieved at this group size. The parallel benchmarks maintained a single control thread for forking tasks, thereby effectively reducing by one context the number available for latency hiding. With model 404's task pool thus reduced to 3 threads, or 3/4 of the maximum, as opposed to model 808's 7 threads, or 7/8 of the maximum, configurations of 4 groups were unable to display latency hiding to the same degree as in configurations of 8 groups.

Figure 5.3 Performance relative to context groups of 4 contexts.
In normal operation, involving saturated multithreading or limited parallelism, the number of contexts used bears almost no influence on the performance of Concurro, but in linear-mode multithreading both the number and arrangement of contexts can determine the processor's fitness for various types of workload. With equal context numbers, models 408 and 808 should, ideally, offer identical performance, however model 808 showed a 7% advantage in harmonic mean throughout over model 408. This discrepancy reflects the benefits of greater datapath bandwidth and higher peak dispatch rates in the 8-group configuration. It should be noted that the performance from model 408, although marginally lower, is achieved at close to half of the datapath cost required for model 808, giving the smaller configuration a utilization advantage on floating point codes in particular. Halving context numbers in models 416 and 832 caused losses of 7% and 3% in mean throughput, respectively; performance reductions for individual benchmarks, however, were in some cases considerably larger.

Given, as indicated in Section 5.4, the low cost of contexts relative to FUs and cache, an oversupply rather than shortage of contexts is preferred so as to maximize utilization of the more costly hardware.

5.2 Instruction Cache Duplication

One of the simplest strategies for providing adequate instruction fetch bandwidth is to duplicate the instruction cache, giving each context group its own private cache. This arrangement insulates the processor from inter-thread interference effects and variations in branch behaviour, but can disallow fetch cooperation among threads executing common code. A second, and practically important benefit derived from the abundance of instruction bandwidth is that multiple instruction dispatches per processor context becomes feasible.

The Concurro simulator was modified to accommodate multiple I-caches and superscalar contexts. Each context group in the modified machine has exclusive access to a primary cache of 16 kB, whose characteristics apart from preaccess depth are identical with the standard I-cache for model 832 (see Chapter 3 and Appendix C for details). The primary caches have arbitrated access to the shared secondary I-cache, external to the CPU. The fetch policies described in Chapter 3 were retained. A context contends for an instruction buffer refill at the point when the following cycle's dispatch would empty the buffer.

Each of the group operand buses was duplicated to permit parallel instruction dispatch on a group basis. The instruction look-ahead scheme of Section
3.1.3 is employed to select several instructions for dispatch in each cycle. As in the standard configurations, the dispatch mechanism is not permitted to reorder memory accesses or speculate beyond branch/control instructions. Apart from these restrictions, any combination of operational instructions may be dispatched in parallel provided that operand queue slots are available for the instructions in the next cycle. This approach exploits the natural variation in operation types to minimize the occurrence of structural hazards. For a single superscalar context the instruction pipeline is illustrated in Figure 5.4, where an execution latency of 2 cycles is shown. The write back pipestage may be bypassed, as is conventionally the case, and Concurro's static branch prediction is used in the decode stage to reduce conditional branch latency by one cycle.

![Figure 5.4 Superscalar instruction pipeline.](image)

The benchmarks were executed on three variations of the modified processor, each containing 8 context groups and 8 primary I-caches (preaccessing 8 instructions). Two of the variations had context groups comprising 4 contexts, with the third variation having a single, superscalar context per group. Up to 4 instructions, selected from the first 6 instruction buffer positions, were allowed to be dispatched simultaneously in the superscalar cases. Except for the changes noted above, these processors operated with the datapath and memory configurations defined for model 832 in Appendix C. Figure 5.5 compares the performance of these variations against that of model 832.
Figure 5.5 Effect of instruction cache duplication and superscalar instruction dispatch on performance (normalized to model 832).

Duplication of the I-cache alone produced negligible improvements in performance, raising harmonic mean throughput by 0.2% over that for model 832. The greatest increase in throughput, of 3%, came from carries, whose execution appears to be fetch-limited in model 832 (see Section 6.1.2). The 2% degradation in performance on loops is attributed to the impact of the additional I-cache misses over loops' relatively short run time. In this case identical thread code had to be fetched more than once from the secondary cache rather than just for the initial misses as in model 832. The 8-fold increase in primary caches multiplied secondary I-cache traffic on carries, gauss, and matmult by between 5.6 and 6.8 times, whereas for benchmark tf the lack of code sharing among threads accounted for a traffic increase of only 8%. For highly parallel workloads we can expect some of the bandwidth advantage of the multiple cache organization to be offset at some point by the earlier saturation of the secondary cache.
Providing superscalar capabilities in addition to I-cache duplication in a 32-context processor yielded only a minor performance gain relative to model 832. The bulk of the 7% increase in overall throughput was obtained from the benchmarks possessing least thread-level parallelism—compress and tf. As these programs never placed the processor in saturated multithreading, ordinarily unused dispatch slots were available for exploiting ILP, and boosting throughput. In contrast, the remaining benchmarks tended to be resource limited: superscalar dispatch was unable to alter datapath bandwidth. Additionally, execution of the scientific codes was dominated by the relatively long latency floating point and uncached memory operations for which simple pipelining and scalar dispatch alone were sufficient to fully exploit thread ILP. The anomalous drop in performance for benchmark loops is primarily the consequence of instruction look-ahead, whose effects are studied in detail in Section 6.1.3.

The influence of superscalar dispatch is more clearly revealed in the results for the single-context per group processor. Performance in this case was relatively unaffected by limitations in datapath bandwidth since thread numbers had been cut to one quarter. However, the resulting degradation in latency tolerance reduced harmonic mean throughput by 10% relative to model 832. Normalized throughput of the superscalar processor was comparable with that of a scalar, single-context per group processor, whose performance results appear in Figure 5.3. While the throughputs on compress and tf were raised as in the other superscalar case, performance for the remaining benchmarks improved on the scalar results by less than 4%, typically. This would indicate that pipelining of long latency operations had made superscalar dispatch redundant. In the case of carries, however, superscalar dispatch produced a 13% gain in throughput over the scalar processor because of the limited vertical concurrency possible from integer operations.

5.3 Multiprocessing

The organization of Concurro places an emphasis on resource sharing and gives multithreading the responsibility for providing latency tolerance. An alternative organization for a machine of Concurro's scale is the single-chip multiprocessor, which relies on cache memory for latency tolerance, and resource duplication to sustain high processing throughput. With Concurro the intention is to maximize the utilization of the main memory bandwidth; with the multiprocessor a less pessimistic view is taken, and cache is used in an attempt to amplify the main memory bandwidth.
To examine the trade-offs in these architectural approaches, a heavily modified version of the Concurro simulator was used to simulate three multiprocessors, against which model 832 is compared. By this means it is possible to focus on essential differences, since the instruction set architecture and most of the microarchitecture are common to all machines. Each of the multiprocessor organizations consists of 8 processing elements—to match the sustained instruction throughput rate of the 8-group model 832—sharing a central system control FU and synchronization controller, and the ports to main memory. The arrangement is depicted in Figure 5.6. Each PE accesses private instruction and data caches, and is provided with its own FU set—one physical FU for each of the 7 non-shared FU classes.

![Diagram of multiprocessor organization](image)

**Figure 5.6** Organization of the simulated single-chip multiprocessors.

The multiprocessor D-caches are supplied by three main memory ports, equalling the combined data memory bandwidth available to standard model 832. Every 4 D-caches share one port for reload data, while the third port accepts all write-through data from an enlarged write buffer. In practice the single write port provided adequate bandwidth for all of the benchmarks, and allowed some simplification of the cache-coherency proto-
Coherency across the D-caches is maintained through a write-invalidate snooping protocol [HePa90], with each cache dedicating one port of its dual-ported tag store to snooping a common coherency address/control bus. Since the cache controllers permit 4 misses before blocking, coherency transactions may also alter the status of miss queue entries, and force reloads to be retried in response to invalidate signals. Given the number of caches sharing memory ports in the multiprocessor, it is unnecessary to implement the pipelined cache reload mechanism described in Section 3.4.3. Layered logically on top of the coherent caches is the synchronization controller. The only modification to this unit is the addition of memory barriers following synchronization stores to preserve the existing consistency model. The cost of M-structure loads then becomes similar to the cost of a conventional multiprocessor load-linked and store-conditional spin lock sequence [KaHe92].

The three multiprocessors are designated A, B, and C, while standard Concurro model 832 is designated D. Wherever possible and practical the multiprocessor configurations and behaviours match those of machine D. Machine A is multithreaded, with 4 scalar contexts per processor, taking after Sparcle [Agar+93]. The 2-way set-associative D-caches and I-caches are 32 kB and 16 kB in size, respectively. Machine B contains single threaded, 4-way superscalar processors, following the characteristics described in Section 5.2, and the same caches as in A. The organization of B resembles that of the multi-stream processor proposed in [PaFJ91]. In machine C the D-caches and I-caches are reduced in size to 8 kB each, but associativity of the D-caches is doubled to 4 lines/set. Memory latency is set to 6 cycles, simulating a large, external streaming cache rather than main memory itself. Each processor of C is also 4-way superscalar.

Although the existing executable binaries could be run on all of these machines, it was decided that for a fairer comparison the benchmarks executed on A, B, and C should have their chunking factors increased to reduce the frequency of thread creation and synchronization events in the multiprocessor case. Nonetheless, the multiprocessors enjoyed the efficiency of Concurro's fork and term inate instructions, rather than having to manage task creation completely in software. Uncached loads and stores were simply converted to their cached long word equivalents in machines A, B, and C, whereas machine D continued to access main memory directly.

Figure 5.7 plots the performance of the machines normalized to the performance of D executing the original benchmarks. Multiprocessor performance on all of the parallel benchmarks was significantly below that of machine D. Since compress was executed on one processor in every case, the perfor-
mance on this benchmark simply reflected the changes made to the D-cache controller, memory, and instruction dispatch strategy. The remaining programs can be divided into two categories, according to their cache behaviour. Carries, loops, and tf suffered large increases in miss rate on account of their frequent synchronization operations and the cache line invalidates these regularly caused. For gauss and matmult the frequency of synchronization was relatively low, but these benchmarks’ manipulation of matrices larger than the D-caches produced substantial rises in capacity miss rates, thereby lowering the effective bandwidth of the caches.

![Graph showing performance comparison]

**Figure 5.7** Performance of 8-processor multiprocessors relative to model 832.

Multithreading in machine A enabled it to mitigate the effects of adverse cache performance for benchmarks gauss and matmult. Performance on matmult was reduced 31% relative to D, despite a data miss rate of 12.5% and an average cache access latency of 24.4 cycles. On gauss the access latency fell to less than one-fifth, but this program’s non-uniform parallelism
profile and greater use of synchronization accesses were responsible for the larger performance drop observed. With a miss rate of 40%, carries was the greatest casualty of cache coherency ping-pong effects. M-structure accesses from different threads caused the cache lines containing the structures to migrate from processor to processor as the lines alternated between shared and exclusive states. An average of 0.4 invalidates per cache access were generated with carries, while on loops and tf the invalidation rate was only 0.1 per access—allowing better relative performance for the latter two benchmarks. Overall, harmonic mean throughput of machine A was 43% lower than that achieved by D.

Trading multithreading for superscalar dispatch in machine B improved performance on all benchmarks except gauss and matmult, and brought harmonic mean throughput to within 40% of D. For matmult the lack of multithreading severely impaired the processors' latency tolerance. However, with fewer threads being executed simultaneously, invalidation rates were reduced on gauss and loops, which allowed somewhat better cache performance. The benefits of superscalar processing were most evident on the integer programs—in keeping with the observations of the previous section.

Given that D-cache miss rates were generally high in multiprocessors A and B, it is not surprising that reducing the miss penalty in machine C lead to marked performance improvements. Despite the moderately increased miss rates of the smaller caches, the 6-cycle backing store approximately halved average D-cache access latencies on most benchmarks. Consequently, harmonic mean throughput of machine C came to within 24% of D. Performance increases relative to machine B ranged from 5% for tf to 200% for carries. The substantial improvement on carries is attributed to the enhanced temporal locality of M-structure updates brought about by the reduction in cache access latency. For tf the reduction in cache size from 32 kB to 8 kB had crossed the size threshold for which tf's capacity miss rate rises sharply, thus offsetting the benefit of a reduced miss penalty. Unfortunately the cache bandwidth advantage of machine C was only partly realised as performance gains on the floating point codes. Without multithreading, the latency tolerance of machine C was restricted to such an extent that floating point operations were able to influence processor utilization.

5.4 Implementation Cost
The implementation cost of Concurro is estimated below in terms of transistor counts and die areas. Table 5.1 lists breakdowns of the hardware
budgets estimated for the implementation of models 416 and 832. Those transistor counts that could not be obtained from counting bits of storage (6-transistor SRAM cells assumed) were derived from the transistor counts of components in the Weitek W8701 SPARC processor [Gwen93], which is fabricated in 0.8 μm CMOS, and features a 4-cycle double-precision floating point add operation. Layout densities were also derived from the W8701, and scaled to 0.4 μm (drawn) minimum feature size. Clock circuits are included in the individual component counts, with phase generation as part of miscellaneous control. The values in Table 5.1 are, necessarily, preliminary estimates, given the difficulties of assessing hardware costs at this stage.

<table>
<thead>
<tr>
<th>Processor Component</th>
<th>model 416</th>
<th></th>
<th>model 832</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>transistors</td>
<td>area mm²</td>
<td>transistors</td>
<td>area mm²</td>
</tr>
<tr>
<td>Register files</td>
<td>300k</td>
<td>15</td>
<td>600k</td>
<td>30</td>
</tr>
<tr>
<td>Instruction buffers &amp; context logic</td>
<td>60k</td>
<td>3</td>
<td>160k</td>
<td>8</td>
</tr>
<tr>
<td>16 kB instruction cache</td>
<td>1000k</td>
<td>10</td>
<td>1000k</td>
<td>10</td>
</tr>
<tr>
<td>32 kB data cache</td>
<td>1800k</td>
<td>18</td>
<td>1800k</td>
<td>18</td>
</tr>
<tr>
<td>System unit &amp; thread manager</td>
<td>50k</td>
<td>5</td>
<td>50k</td>
<td>5</td>
</tr>
<tr>
<td>Logical FUs</td>
<td>10k</td>
<td>1</td>
<td>20k</td>
<td>2</td>
</tr>
<tr>
<td>Integer FUs</td>
<td>100k</td>
<td>5</td>
<td>200k</td>
<td>10</td>
</tr>
<tr>
<td>Shift FUs</td>
<td>25k</td>
<td>2</td>
<td>50k</td>
<td>4</td>
</tr>
<tr>
<td>Floating point add/multiply FUs</td>
<td>300k</td>
<td>15</td>
<td>600k</td>
<td>30</td>
</tr>
<tr>
<td>Divide FUs</td>
<td>100k</td>
<td>5</td>
<td>100k</td>
<td>5</td>
</tr>
<tr>
<td>Load/store units with TLBs</td>
<td>200k</td>
<td>10</td>
<td>400k</td>
<td>20</td>
</tr>
<tr>
<td>Synchronization controller</td>
<td>50k</td>
<td>5</td>
<td>50k</td>
<td>5</td>
</tr>
<tr>
<td>Miscellaneous control</td>
<td>200k</td>
<td>20</td>
<td>300k</td>
<td>30</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td>4195k</td>
<td>114</td>
<td>5330k</td>
<td>177</td>
</tr>
</tbody>
</table>

Table 5.1 Estimated transistor counts and die areas for Concurro implementations (assuming a 0.4 μm, 3-metal CMOS process).

The estimated total transistor counts, although substantial, are not unrealistic. Compared with the Intel Pentium™ microprocessor [AIAv93], the transistor counts of Concurro models 416 and 832 are only 35% and 72% greater, respectively. The more numerous interconnects and low-density components in model 832 are responsible for its die size being disproportionately larger than indicated by the difference in transistor numbers. Taking into account the area of bus wiring and the pad ring, it is estimated that model 832 can
be fabricated in under 250 mm\(^2\) of die space. By comparison, an 8-processor, scalar, single-chip multiprocessor, with 8 kB instruction and data caches, is expected to occupy 490 mm\(^2\) (before pads and buses) and use almost 16 million transistors.

5.5 Observations and Conclusions

Superscalar performance—speed-ups beyond the usual instruction issue rate limits of single-pipeline multithreaded processors—was demonstrated to be readily achievable in Concurro. For sufficiently scalable workloads speed-up followed the processor group count, reflecting changes in datapath and instruction fetch bandwidths. Equally importantly, however, the performance on single threaded or inadequately parallelized programs approached typical RISC throughputs per context group, despite some minor penalties of a microarchitecture designed for multithreading. Satisfactory processor efficiencies for the more parallel benchmarks were still available from an 8-group configuration; scalability beyond this in a uniprocessor is unlikely to be important in practice, as unduplicated, costly resources, such as caches, soon become fully utilized.

The provision of a primary I-cache per context group in model 832 returned negligible performance gains for the added cost. Simulation results indicated that an inherent increase in total cache reload traffic can erode the scalability potential of the duplicated I-cache organization. In contrast, a single cache allows more cooperative fetch behaviour among reentrant threads. Superscalar dispatch by context units delivered the most significant performance improvements on poorly parallelized, integer workloads. Under conditions of saturated multithreading and in the presence of long latency operations superscalar dispatch was unable to contribute any useful additional concurrency to aid processor throughput. In light of the these contrasting performance conditions, it may be beneficial to give the root context superscalar capability for improving single threaded performance at least.

Despite fully duplicated caches and functional units, 8-processor multiprocessors were unable to improve on the performance of Concurro model 832. On the basis of harmonic mean throughput, even model 416 offered superior performance. The principal impediments to high performance in the multiprocessor were the deleterious effects of cache coherency on synchronization latency and the poor bandwidth available from caches in scientific applications. Recoding the numerical programs to use blocked algorithms would improve the bandwidth situation, but such efforts are neither universally applicable nor trivial. Substituting a large, multiported
streaming cache for the main memory connections proved effective. This option, however, is also applicable to Concurro, and would allow a comparatively simple threading strategy for vectorizable code to be used to great effect. In scientific applications, in particular, the reliability of latency tolerance provided by multithreading is difficult and costly to match by other means. Assuming that the performance problems of the multiprocessor could be resolved, it would still call for an 8-fold increase in cache numbers over model 832, with little possibility of reducing main memory bandwidth, and a quadrupling in the number of floating point arithmetic units—thus exacerbating the yield problems of single-chip implementations.
Chapter 6

Instruction Fetch and Dispatch Strategies

The stream of instructions consumed by the processor can be considered as being regenerated into an equivalent stream of data in the datapath and across the memory interfaces. The performance of the machine is therefore a function of the throughputs of both the datapath and the instruction processing mechanisms, since the rate of data references is dependent on the instruction dispatch rate, and, conversely, the issue of instructions is affected by the presence of data and structural hazards.

This chapter is concerned primarily with Concurro’s instruction processing behaviour—cache performance, fetching, branching, and instruction scheduling. Simulation results relating to instruction processing behaviour in the base configurations are presented, along with data from experiments assessing implementation trade-offs. A standard datapath, as defined in Appendix C, was configured for this series of simulations.
6.1 Instruction Fetching and Scheduling

Concurro, in common with superscalar and VLIW architectures, is critically dependent on an adequate instruction fetch bandwidth in order to sustain multiple instruction dispatches per cycle. Instruction scheduling is a related issue affecting the rate of instruction throughput in the decode/dispatch stage. This section examines the effectiveness of the designed fetch and buffering techniques, and evaluates the cost/benefit of various scheduling strategies.

6.1.1 Cache Return

Apart from cache miss rate, the other major determinant of instruction fetch bandwidth in Concurro is the efficiency with which the cache can pre-access instruction blocks. The instruction buffer design, detailed in Section 3.1.2, assumed perfect pre-access behaviour. For the low miss rates experienced by the benchmarks this assumption holds well, as indicated by the results of Table 6.1 where line crossing is enabled. The table lists for each model the nominal request size (in instruction words), and for line crossing both enabled and disabled the proportion of the nominal request delivered upon a fetch (average over all benchmarks), alongside the average return for the worst-performing program. Normal configurations operate the I-cache controller with cache line crossing enabled, that is, pre-access may extend over line boundaries to pick up longer instruction runs.

<table>
<thead>
<tr>
<th>Model</th>
<th>Request Size</th>
<th>Line crossing</th>
<th>Mean Return</th>
<th>Worst Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>208</td>
<td>3 words</td>
<td>enabled</td>
<td>99.8%</td>
<td>98.9%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>disabled</td>
<td>87.9%</td>
<td>87.0%</td>
</tr>
<tr>
<td>416</td>
<td>5 words</td>
<td>enabled</td>
<td>99.6%</td>
<td>98.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>disabled</td>
<td>79.0%</td>
<td>75.1%</td>
</tr>
<tr>
<td>832</td>
<td>10 words</td>
<td>enabled</td>
<td>98.5%</td>
<td>95.1%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>disabled</td>
<td>76.5%</td>
<td>70.6%</td>
</tr>
</tbody>
</table>

Table 6.1 Instruction cache return rates.

Mean return rates of 100% are impossible for cold start programs because initial compulsory misses prevent most pre-access opportunities across cache lines. The slightly inferior results for model 832 reflect the coarser granularity of its fetches. From detailed traces of Concurro it was apparent that the line crossing feature allowed initially ordinary return rates to
quickly approach their limits as the cache system adapted to the reference pattern of the program. Without line crossing, the performance of the cache was rendered vulnerable to alignment and branch vagaries, thus invalidating the bandwidth assumption made in designing the instruction buffers. The fetch model of Section 2.3.2 predicts, in the absence of line crossing, return rates of 87.5%, 75.0%, and 71.9% for models 208, 416, and 832, respectively, but in practice these are often bettered on workloads containing large basic blocks.

**Figure 6.1** Performance impact of disabling I-cache line crossing.

The extent to which line crossing effects Concurro’s performance can be gauged from Figure 6.1, which plots the relative change in throughput (referenced against the throughput of the standard configuration) for each benchmark executed on models 416 and 832 with line crossing disabled at two pre-access depths. Pre-accesses of 5 and 10 words represent the normal I-cache widths, whereas the 4 and 8-word depths simulate the lowest-cost organization of power-of-two pre-access and no controller optimization. At normal pre-access depths the harmonic mean performance over all benchmarks decreased by 4% and 2% for models 416 and 832, respectively, while with power-of-two pre-access, the performance of these models fell
by 5% and 4%, respectively.

As is shown below, the execution of carries is fetch-limited for models 416 and 832, making the performance on this benchmark highly susceptible to changes in instruction bandwidth; reduced performance impacts in the other programs correspond with their lower utilization of the I-cache system. The cost of implementing line crossing—pre-access in the tag store and an additional tag-entry bit—is reasonable given the potential for severe performance degradation in fetch-limited workloads.

![Graph showing mean instruction fetch rates for different benchmarks]

**Figure 6.2** Mean instruction fetch rates.

### 6.1.2 Fetch Activity

With only one I-cache to share, Concurro's fetch rate is limited to 1 fetch request per cycle. Ideally the demanded fetch rate for highly parallel codes should be close to, but below, the hard 1/cycle limit, allowing for cache unavailability during misses. Enlarging instruction buffers lowers the fetch rate, but at the expense of degraded cache and instruction bus utilization. For real workloads the fetch rate will be a function of the instruction
Chapter 6  Instruction Fetch and Dispatch Strategies

dispatch rate and the utilization of buffered instructions.

Figure 6.2 plots the fetch rates with each benchmark for standard models 208, 416, and 832; those instruction fetches generated from branch predictions are shown hatched. The graph clearly illustrates two dominant fetch characteristics of Concurro workloads: fetch rates rising slowly with group count—for parallel programs that scale readily, and fetch rates decreasing with group count—for less scalable programs. For the latter class of benchmarks, this behaviour is consistent with the instruction throughput remaining almost invariant with group count, resulting in less frequent refills for larger instruction buffers; the ideal fetch rate varies inversely with the pre-access depth.\(^1\) Fetch rates for the more scalable benchmarks, however, vary little, since the pre-access depth scales to yield higher instruction bandwidths at greater group counts. In practice this additional bandwidth is not quite sufficient to counter the growing instruction discards occurring in contexts as a result of branches. To compensate, fetch rates tend to inflate at higher group counts for all benchmarks.

\[\text{model 208} \quad \text{model 416} \quad \text{model 832}\]

\[
\begin{array}{cccc}
\text{Benchmark} & \text{carnes} & \text{compress} & \text{gauss} & \text{loops} & \text{matmult} & \text{u} \\
\text{Percentage Dispatched} & 100 & 80 & 60 & 40 & 20 & 0 \\
\end{array}
\]

\textbf{Figure 6.3} Proportion of fetched instructions dispatched.

With buffer flushing events—such as branches—the utilization of fetched instructions must fall below 100%. Figure 6.3 indicates that actual utilization factors often exceeded the 80% design target set in Section 3.1.2, and, apart from two anomalous cases, declined with greater group counts. We can expect larger portions of the instruction buffers to be annulled as the

\(^1\)From comparisons of pre-access depth to group count in Appendix C.
buffer size extends into the low-probability region of the run length distribution. Such an effect markedly degraded fetch utilization in model 832 for benchmarks *compress* and *tf*, which have run lengths that are relatively short compared to the 10-instruction size of model 832's buffers. Seemingly contradictory fetch behaviour for *carries* provides an example of fetch-limited performance despite adequate utilization. In this case the instruction dispatch rate surpassed Concurro's group count, thereby driving fetch rates into the 1/cycle limit for models 416 and 832.

Speculative fetch requests met with negligible success in being granted amongst even moderate cache traffic. Apart from *tf*, no benchmark experienced a speculative fetch rate above 0.1 fetches per cycle, while the fetch-limited executions of *carries* do not even register speculative fetches in Figure 6.2. With the number of predicted branches being independent of processor model, speculative branches assumed a greater proportion of total fetches as the number of mandatory fetches fell in the larger configurations: this trend was in some cases moderated by reductions in the absolute numbers of speculative fetches, although that is not immediately apparent from Figure 6.2.

### 6.1.3 Instruction Look-ahead

The processor models examined so far were configured for strictly in-order sequencing of instruction dispatch. This strategy, while inexpensive to implement, can delay the dispatch of ready instructions caught behind others waiting on operations of unpredictable latencies. Allowing the ready, but blocked instructions to dispatch ahead of their program predecessors can reduce the critical path of the computation, while the exposure of intra-thread parallelism provides a relatively cheap alternative to increasing the number of processor contexts per group.

To test the merits of this scheme, the benchmarks were executed with instruction look-ahead set above its normal 1-instruction window (that is, the head of the instruction buffer). Figure 6.4 shows the performance impact on models 416 and 832 with look-ahead windows of 2 instructions and the pre-access number of instructions. Somewhat counter-intuitively, out-of-order dispatch yielded almost as many reductions as improvements in performance. For model 416, the harmonic mean performance improved by 0.7% and 0.3% with 2 and pre-access depth windows, respectively, while model 832 gained 1.4% and 0.3% for the same corresponding look-ahead windows. Such minor improvements cannot justify the expense of out-of-order instruction dispatch, particularly in the case of look-ahead to pre-access depth, which requires in model 832 a dependency check over 10 instructions.
Why does look-ahead fail to deliver a significant advantage? For the restricted parallelism benchmarks of *compress* and *tf* out-of-order dispatch achieved the consistent gains that were expected, although *tf*’s improvements were limited chiefly by its inherently low ILP within basic blocks. For the more parallel programs, however, the performance gains were either negligible or negative at greater look-ahead distances. No single cause was found for all losses, but cases of reduced performance commonly suffered from delays in critical datapath resources, such as FU operand or load queues, which became more congested with the added bursts of traffic from look-ahead dispatches. This phenomenon points to a cause for the poor results: a break-down of the original instruction schedules.

Instruction schedules generated at compile-time aim to maximize utilization of the pipeline and reduce the critical path of the computation. With resource sharing amongst parallel threads, their interaction becomes greater as the number of threads increases, resulting in added queue delays as multiple threads are allocated a diminishing proportion of the fixed datapath band-
width. This has the effect of dilating the schedule and constraining throughput to bandwidth limits, but if the increased operation latencies remain similar relatively, then the original schedule continues to be appropriate. Instruction look-ahead in Concurro disrupts schedules by permitting code motion at dispatch time rather than at issue time, preventing contexts from recognizing the changing instruction latencies and appropriate schedule. The greedy scheduling policy of out-of-order dispatch attempts to fill what are perceived as pipeline bubbles, but instead, actually inserts unnecessary work between instructions on the critical path, thereby prolonging run time. The less parallel codes performed well under out-of-order dispatch simply because at reduced utilization most instruction issues followed their dispatches without delay.

To verify the existence of this effect, Concurro’s logical, integer arithmetic, and shift FU’s operand queue lengths were altered, and carries was executed by these configurations with and without instruction look-ahead. With normal queue lengths of 2 entries, 10-instruction look-ahead in model 832 resulted in a slow-down on carries of 0.9%. For queues of 8 entries the possibilities for schedule distortion are greater, and, indeed, performance reduced by 1.2% when instruction look-ahead was enabled. Conversely, forcing a tighter coupling of issue and dispatch by shortening the queues to single-entry length allowed a speed-up of 1.0% under out-of-order dispatch. While short operand queues may improve the behaviour of instruction look-ahead, this gain has to offset a lower absolute performance in this
Chapter 6  Instruction Fetch and Dispatch Strategies

To fully solve the problem of mis-scheduling, out-of-order instruction selection must occur at the time of issue, but this alternative can be expected to involve the complexities and cycle-time hazards of Tomasulo's algorithm [Toma67].

Figure 6.5 indicates that even when allowed 10 instructions of look-ahead in model 832, the buffer positions from where instructions were selected for dispatch rarely deviated from the head entry position. The extent to which average distances exceed 0 reveals the surplus concurrency available from the instruction stream, since look-ahead occurs only when the first instruction is not ready for dispatch. By this measure, the floating point programs showed significant additional ILP, although its exploitation had negligible effects on performance when operating in saturated multithreading.

![Figure 6.6 Performance with out-of-order instruction dispatch (separate register read and instruction decode).](image)

Since the instruction dependency analysis is likely to occupy an entire pipestage, implementations of Concurro allowing out-of-order instruction dispatch would be expected to separate instruction decode and register
read into two consecutive stages to avoid an impractical degree of multiporting in the local register files. As can be seen in Figure 6.6, however, this decision negates all the advantages of instruction look-ahead. Every benchmark showed a drop in performance, although multithreading was able to hide most of the extra pipestage delay for the highly parallel programs of carries, gauss, and matmult. Compress and tf, integer programs with low surplus ILP, were especially vulnerable to the inevitable pipeline bubbles introduced by the separate register read stage. Harmonic mean throughput fell by up to 12% for model 416, and 14% for model 832. These results effectively rule out deep instruction look-ahead, as no performance is gained from an extended pipeline despite out-of-order dispatch.

6.1.4 Barrel Processing

Asynchronous thread interleaving is incorporated in Concurro to enhance its performance on workloads of low parallelism, giving the processor an advantage over conventional multithreaded processors with circular, interleaved pipelines. The drawback of a dynamically responsive instruction scheduler, however, is the added cost of arbiters and duplicated logic. To compare Concurro against the synchronous interleaving or "barrel processor" approach, the processor was reconfigured so that each group shared one instruction decoder and operand dispatch unit among all of its contexts. Contexts were granted instruction dispatch opportunities in strict rotation order, emulating a collection of barrel processors with as many segments as the number of contexts in a group. This arrangement implements busy waiting on registers, since scoreboard blockages introduce pipeline bubbles.

The throughputs of the synchronously interleaved configurations are shown relative to the normal asynchronous models 416 and 832 in Figure 6.7. The harmonic mean throughput dropped by 55% and 59% for models 416 and 832, respectively, being markedly affected by the reduced performance on tf and compress. Note that despite the 4-cycle instruction dispatch period the single threaded compress program achieved a normalized throughput of 0.36 rather than 0.25, because the normal dispatch rate was only 0.65 instructions per cycle. Due to cache misses blocking instruction dispatch for longer than 4 cycles, a dispatch rate of 0.25 instructions per cycle could not be sustained by the barrel processor, thus reducing the normalized throughput from the ideal of 0.25/0.65 to 0.24/0.65. Tf, with only a quarter of all contexts active, presented the barrel processor with insufficient work to prevent it from idling for 75% of the time, giving rise to single threaded performance by each context group.
Although the remaining benchmarks displayed, as expected, more tolerance to time-multiplexed dispatch, performance on these programs was poor in light of the greater parallelism available. The varying parallelism profiles of *gauss* and *loops* exposed the processor to significant periods of low context activity, which allowed barrel processing to magnify Amdahl's Law effects. For *carries* the sharing of dispatch units lowered performance by preventing throughputs beyond 1 instruction per cycle per group. Even on *matmult* performance fell by 5% and 6%, showing synchronous interleaving to be a liability under ideal conditions. Given that moderate proportions of general-purpose workloads are likely to be poorly parallelized, the cost savings of barrel processing are expected to be heavily offset by degraded performance and the reduction of utilization this brings.

### 6.2 Branching

In high-performance single threaded processors the handling of branches, particularly conditional branches, is critical in determining the performance of the machine. In the absence or failure of branch prediction, branches are responsible for annulling part of the pipeline and introducing bubbles into it, causing single threaded processors to suspend instruction dispatch
pending the fetch and decode of the target instructions. For multithreaded processors, however, branch-induced delays in one thread can usually be covered—as for other instruction latencies—by the dispatch of instructions from alternative threads. Although if less parallel programs are to be catered for then some form of branch optimization is desirable. This section examines the behaviour of Concurro’s branch mechanism and its impact on performance.

6.2.1 Branch Behaviour

The stack chart of Figure 6.8 plots breakdowns of conditional branch activity for each benchmark (independent of processor configuration). Jumps and unconditional branches are inherently certain events, and they were treated as such by Concurro’s fetch mechanism. The outcomes of conditional branches, however, were predicted one pipestage in advance, giving conditional branch data added importance. Recall from Section 3.1.4 that static branch prediction based on conditional branch direction was implemented: backward branches were assumed to be always taken, while forward branches were assumed to be never taken.

![Figure 6.8 Conditional branch mix for all models.](image)
The great variation of characteristics across the benchmarks highlights the limitations of this simple branch prediction scheme. Loop-intensive programs, such as gauss and matmult, were dominated by backward, taken branches, with the majority of the remaining branches being untaken, forward. These conditions favour static branch prediction as implemented. Integer benchmarks compress and tf were not so favourable, experiencing majorities of forward branches, of which almost half were taken. Unfortunately the branch prediction scheme performed worst where its need was greatest: limited-parallelism programs unable to hide branch latency. The resulting fractions of branch outcomes predicted correctly ranged from 60% for tf to 98% for matmult, giving an average accuracy of 83% over all six benchmarks.

![Graph](image)

**Figure 6.9** Fraction of predicted branches resulting in speculative fetches.

The conversion of branch predictions to speculative fetches is indicated in Figure 6.9, which plots speculative fetches per backward conditional branch in standard models 416 and 832. The results illustrate that the fetch mechanism was sufficiently adaptive to exploit opportunities for speculation when they arose. In the cases of compress, loops, and tf, where fetch rates were relatively low, a majority of backward branches, particularly in model 832, were able to intersperse speculative fetches among buffer refill requests.
The pressure of normal fetch traffic prevented most speculative fetches for *gauss* and *matmult*, and virtually eliminated speculation on *carries*. For the scalable benchmarks the lowering of speculative fetch activity had the added benefit of reducing the number of unnecessary fetches—resulting from incorrectly predicted backward branches. The amount of wasteful fetch traffic proved negligible though, since only an average of 9% of backward branches were in fact not taken.

With the results of Figures 6.8 and 6.9 we are in a position to determine the execution latencies of conditional branches. It is assumed, ignoring cache misses and contention delays, that untaken branches execute in 1 cycle, taken branches making use of speculative fetches execute in 2 cycles, and taken branches without assistance require no more than 3 cycles; speculative fetches are assumed to favour neither taken nor untaken branches. Thus in model 832 the mean latency of conditional branches ranges from 1.8 cycles for *compress* to 2.6 cycles for *matmult*, averaging 2.1 cycles overall. Speculative fetch contributes only a modest saving, however, for without it conditional branch latencies increase to 2.0 cycles for *compress* and 2.9 cycles for *matmult*, averaging 2.3 cycles.

### 6.2.2 Fetch Bottleneck

A problematic interaction between branching and instruction fetching is evident from the results in Figure 6.3. Extrapolating from the graph, we can expect fetch utilization to approach 0% as the group count or instruction buffer size becomes extremely large. For buffer sizes beyond the point where utilization starts its steady decline no amount of compensatory prefetching can overcome branch-induced losses, growing at the same rate.

This phenomenon was explored through the execution of a troublesome workload: *carries_ncm*, which is the *carries* benchmark with all of its conditional move instructions replaced by equivalent branch sequences. This modification gives *carries_ncm* a 331% increase in dynamic branch count over *carries*, thereby severely reducing its run length. Despite a slightly lower total instruction count, *carries_ncm* ran 2.5% slower than *carries* on model 416, and 29% slower on model 832. Execution of *carries_ncm* was fetch limited on both models, with instruction issue rates down by 17% and 40% for models 416 and 832, respectively, at fetch rates of 0.998 and 0.997 per cycle.

The average distances between (and including) successive taken branches, calls, and jumps of the benchmarks, including *carries_ncm*, are listed in Table 6.2. Note that since run lengths measure instructions between *taken*
branches, they can differ significantly from basic block sizes. As a consequence of loop unrolling up to 4 iterations, the scientific codes gained sizable run lengths. In contrast, the small basic block sizes and lack of loop unrolling in the integer codes were responsible for their lower run lengths, which in three cases were comparable to the size of instruction buffers in model 832. The value of conditional moves is readily apparent in the case of carries; the carries_ncm program averaged run lengths just 24% as long as in its more optimized equivalent, carries.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>carries_ncm</td>
<td>5.4</td>
</tr>
<tr>
<td>carries</td>
<td>22.3</td>
</tr>
<tr>
<td>compress</td>
<td>11.2</td>
</tr>
<tr>
<td>gauss</td>
<td>62.5</td>
</tr>
<tr>
<td>loops</td>
<td>35.1</td>
</tr>
<tr>
<td>matmult</td>
<td>28.5</td>
</tr>
<tr>
<td>tf</td>
<td>6.7</td>
</tr>
</tbody>
</table>

Table 6.2 Mean run lengths (instructions).

In general it is not possible to predict the effects of run length on performance without complete knowledge of the workload’s run length distribution. For low-variance, short run length distributions, however, the mean run length, $R$, can give a useful indication of throughput on large configurations of the processor. Since each branch causes the fetch of a buffer-full or $B$ instructions, with the first instruction being the branch target, then for $B$ greater than the longest run length only $R$ instructions are effectively available for dispatch from each fetch. This implies that the average processor throughput cannot exceed $R$ instructions per cycle, as the fetch rate is limited to unity. For carries_ncm, with $R = 5.4$, this effect is expected to limit performance to 5.4 instructions per cycle on model 832, for which $B = 10$, and reduce fetch utilization to 54%; in actuality, throughput was 5.2 instructions per cycle with utilization of fetched instructions at 52%.

Several solutions to this scalability problem are available. Fortunately, perhaps, the most vulnerable workloads are integer programs, which are difficult to parallelize and thus unlikely to be more scalable than the processor. For the benchmarks the combined use of loop unrolling and conditional move instructions proved effective in maintaining adequate fetch utilization—even in model 832. Software techniques for extending run lengths, such as trace scheduling [Fish81, Elli86, Colw+87], offer the
possibility of improving scalability on integer codes as well. Hardware solutions involving a multi-ported I-cache or addressable instruction buffers are alternatives, of which multi-porting with smaller instruction buffers is likely to be more effective over a wide range of workloads.

6.3 Instruction Cache Performance

As the highest-bandwidth source of instructions in Concurro, the primary I-cache is responsible for maintaining filled instruction processing pipelines. This section examines the effects of cache design parameters and controller algorithm on I-cache bandwidth and machine performance. Experimentally the normal I-cache system showed little variation in behaviour, since even the largest of the benchmarks occupied no more than a few kilobytes of instruction memory, resulting in negligible miss rates—particularly for loop-intensive code. In order to magnify the differences between alternative cache designs, the processor was simulated with primary I-cache storage capacity reduced to unrealistically low levels. The results from doing so give us, at least, a crude assessment of I-cache design trade-offs and their consequences in Concurro.

6.3.1 Miss Rate

Due to the relatively large size of the secondary instruction cache, virtually all of its misses were of the compulsory kind—to use Hennessy and Patterson's nomenclature [HePa90]. With the normal primary cache, secondary cache miss rates averaged, predictably, 26% and 51% for models 416 and 832, respectively, reflecting the ratios of primary to secondary cache line sizes. These rates doubled upon disabling prefetch in the secondary cache. For much reduced primary cache sizes the additional first-level miss traffic allowed secondary miss rates to fall to negligible levels as the compulsory misses assumed less importance.

The variation in average primary I-cache miss rate against cache size is plotted in Figure 6.10 for models 416 and 832 at three controller configurations: non-blocking for up to 2 misses (the standard configuration), blocking after any misses, and non-blocking but with the primary—secondary cache interface width reduced from 128 bits to 64 bits. The I-cache size had to be reduced below 4 kB before capacity showed any influence on miss rate; even then, several benchmarks caused no discernable rise in miss rate until smaller capacities had been reached. At the 256 byte size the cache...
was reduced to holding just 2 sets of lines in model 832, causing thrashing on most programs. Average miss rates for the smallest non-blocking cache peaked at 16% and 22% for models 416 and 832, respectively. The inferior performance from model 832, particularly at lesser capacities, can be attributed to model 832's line size being double that of model 416's.

Of the three cache controller configurations, the blocking controller showed consistently highest miss rates for both processor models. This is to be expected as the blocking controller prohibited accesses, some of which may have been hits on replaced lines, during miss servicing. For the non-blocking controllers the effect of narrowing the bus between the primary and secondary caches was to generally lower miss rates in the primary cache. Indeed, any increase in the miss service time of the primary cache, such as caused by a longer latency secondary cache, had the same effect. This is a thrashing phenomenon, where one thread makes repeated hit accesses to cache space under competition while other threads are blocked waiting on their lines to be reloaded into this space: the slower the reload, the less frequent is the rate of context switching and thus cache misses.

It should be noted that the miss rates plotted in Figure 6.10 are I-cache misses per fetch request, rather than misses per instruction word. On the
latter basis, the miss rates are lowered by a factor equal to the number of instructions returned per fetch. For a single threaded computation the magnification of the scalar (per-word) miss rates is of little consequence as misses per unit time remain unchanged, but for multithreading, sustaining close to one multi-word fetch per cycle, misses occur at greater frequency, making efficient miss handling essential for good performance. At greater miss frequencies fetch latency tolerance deteriorates, since the non-blocking cache controller relies on being able to hide miss delays under substantial sequences of hit accesses. Multi-porting the cache offers no advantage over block fetching, because per-port miss rates are magnified similarly upon multi-word access. Beyond minimizing the primary cache miss rate or reducing miss service time, solutions to this problem must cope with the high miss traffic: multiple miss handlers or even duplicate caches are possible steps in this direction.

![Graph showing instruction fetch return versus primary l-cache configuration.](image)

**Figure 6.11** Instruction fetch return versus primary l-cache configuration.

### 6.3.2 Impact on Throughput

Figure 6.11 shows that greater miss rates of small caches also caused degradation of the average fetch return, indicated on the graph as a percentage of the pre-access depth. For these results the same cache configura-
tions as in Section 6.3.1 were simulated in models 416 and 832. Return rates followed, essentially, negative miss rate trends, with the curves for both models showing the characteristic knee at 4 kB cache size. This is hardly surprising, since for pre-access across cache lines to be efficient, runs of instructions longer than a cache line need to be resident frequently.

![Diagram showing normalized throughput versus primary L-cache configuration.](image)

**Figure 6.12** Normalized throughput versus primary L-cache configuration.

Even with the detrimental effects of cache misses, however, fetch returns never dropped to the levels observed for pre-access without line crossing: line crossing improved the return rate by more than 13% down to cache sizes of 1 kB. Return rates greater than 90% can be considered adequate, given that many of the benchmarks exhibited fetch utilization above the 80% design expectation. The variation in return rate across non-blocking,
blocking, and narrow secondary interface configurations was negligible, although the superior miss performance of the non-blocking controller allowed it marginally greater returns in most cases.

At reduced I-cache sizes the degraded fetch returns forced fetch rates in Concurro to rise in compensation. However, at high miss rates the delays and occasional blocking by the cache imposed a lowered ceiling on the maximum possible fetch rate, depressing instruction throughput. The normalized throughputs of models 416 and 832 for four cache configurations are plotted in Figure 6.12; note that the standard processor configurations have a 16 kB, non-blocking cache. At 1 kB cache size, performance on the majority of benchmarks appeared relatively unaffected for both blocking and non-blocking caches; for \textit{tf}, though, lack of parallelism and miss rates well in excess of 10% were responsible for substantial drops in performance. Over all benchmarks, mean throughput with the 1 kB non-blocking cache was reduced by 4% in model 832, whereas for the blocking cache the fall was 10%. With 512-byte I-caches the miss rates of most programs were sufficient to cause significant changes in performance. Mean throughput of model 832 was reduced by 11% and 15% for non-blocking and blocking 512-byte caches, respectively, while in model 832 performance decreased by 15% and 25%. The results for \textit{gauss}, \textit{loops}, and \textit{tf} clearly indicate the effectiveness of the non-blocking cache controller in Concurro. As an illustration of this we note that model 832 executing \textit{tf} had a performance advantage with the 512-byte non-blocking cache over the 1 kB blocking cache—despite a miss rate some 6% higher for the smaller cache.

From the results of the single threaded \textit{compress} benchmark we can deduce the average miss penalty of the primary I-cache. Under negligible (0.01%) miss rates the execution of \textit{compress} on model 832 averaged 1.52 cycles per instruction (CPI); with any 512-byte cache the miss rate increased to 20% and the CPI value rose to 1.72, at 0.17 fetches per issued instruction. Hence, the average delay for a primary cache miss was \((1.72-1.52)/(0.17\times0.20)\) or 6.0 cycles. Since 4 cycles of this delay can be attributed to the transfer of instructions from the external cache, there is little scope for lowering the miss penalty by simply reducing the latency of the secondary cache; pipelining the servicing of misses—as in the data cache—remains as one of the practical options for improving throughput of the I-cache system under heavy miss conditions.

\footnote{In model 832 16 instruction words are transferred across a 128-bit cache interface in a 2-1-1-1 (cycles) pattern.}
6.4 Observations and Conclusions

The instruction fetch and dispatch mechanisms, as well as the instruction cache, are key influences on the scalability of Concurro. Pre-access across I-cache lines proved successful—and in some cases necessary—in combatting instruction alignment problems and maintaining satisfactory instruction throughputs in the presence of buffer flushes. Despite the negative impact of I-cache misses on fetch return rates, pre-access with line crossing remained sufficiently advantageous in even the smallest caches to justify its implementation. Cache unavailability, one of the causes of degraded processor performance, was noticeably minimized by the non-blocking cache controller, showing that latency hiding techniques can be employed with effect in instruction fetching.

Although unsophisticated, Concurro’s usual in-order instruction dispatch strategy offered performance appropriate for multithreading at moderate hardware cost. Ambitious out-of-order instruction dispatch generally failed to improve performance on multithreaded programs due to adverse effects of thread interaction on scheduling; with an elongated pipeline a look-ahead mechanism actually degraded throughput. In contrast, the added expense of asynchronous thread interleaving gave Concurro important advantages over synchronously interleaved machines. As noted above, the benefits of asynchronous interleaving were greatest for less parallel workloads, although on highly parallel programs, too, the processor showed surprising gains from flexible scheduling.

As with instruction dispatch, the design of the branch mechanism balances single thread performance against the costs and benefits for multithreading. The branch behaviour of the benchmarks generally followed the 90/50 rule-of-thumb for branches [HePa90], giving branch prediction based on static branch direction acceptable accuracy for its simplicity. Instruction buffer flushing upon branches was identified as a potentially serious impediment to scalability. For programs with run lengths comparable to the size of buffers, the instruction throughput was limited by the run length, rather than the group count. Without software techniques for artificially increasing run lengths, this problem is expected to restrict practical instruction buffer sizes, making dual-porting of the I-cache an attractive option for configurations of 8 or more context groups.
Chapter 7

Execution Resources

Given an adequate supply of fetched instructions, responsibility rests with the datapath to sustain processing throughput. The aim in providing execution resources in the datapath is to balance the utilization of functional units against the utilization of other resources while minimizing overall cost. The subject of this chapter is the behaviour of Concurro's execution resources and their interactions with context units. In addition to a characterization of the base models, results are given for experimental explorations of the datapath design. Throughout these simulations the instruction fetch and dispatch machinery remained unaltered from its standard configurations.
7.1 Utilization

An essential requirement of multithreading is the maintenance of processor utilization in the face of large and often unpredictable instruction latencies. Utilization is defined as the proportion of time spent performing useful work. In a practical machine we can expect overall utilization to be compromised in order to support superscalar performance. On account of the varying instruction mixes that a general-purpose computer must deal with it is almost impossible to achieve full utilization from a heterogeneous set of specialized FUs. Indeed, the best we can hope for is that a limiting resource becomes fully utilized. The high utilization factors shown by single pipeline machines [HCAA92, HuGa91, Jord83, LaGH92, ShNi89] are somewhat illusory in the sense that they do not reveal the utilization of the individual components comprising the arithmetic and logic or floating point units. Breaking the functional resources into a collection of shared but autonomous FUs and permitting multiple instruction issues in each cycle allows Concurro to achieve better utilization by bringing functionally dissimilar hardware into operation simultaneously.

7.1.1 Instruction Dispatch and Issue

The utilization of the group operation buses can be regarded as equivalent to the pipeline utilization of single pipeline machines. Under ideal conditions Concurro's operation buses would be saturated with operational instructions, resulting in a dispatch rate of \( N_g \) operational instructions per cycle, where \( N_g \) is the processor group count. However, as can be seen from Figure 7.1, this level of activity was unsustainable over the life times of the benchmarks. Figure 7.1 plots for each program the mean dispatch rates of operational instructions in models 416 and 832: floating point rates are included to indicate cases where the bandwidth of floating point FUs was the limiting factor. The dispatch rate axes of the graphs cover a range of utilizations from 0% to 100%, where full utilization is 4 and 8 dispatches per cycle in models 416 and 832, respectively.

In model 416 the dispatch rate over all benchmarks averaged 2.4 operational instructions per cycle—representing an operation bus utilization of 60%; the influence of the poorly scaling benchmarks reduced utilization to 51% in model 832, which dispatched an average of 4.1 operational instructions per cycle. The modest utilization factors for each of the scalable benchmarks, carries, gauss, and matmul, can be attributed to bottlenecks in other parts of the system. As observed in previous chapters, carries was fetch-
limited with a significant branch/fork component in its instruction mix, making it impossible for operational instruction dispatches to saturate the group operation buses. For *gauss* and *matmult* the throughputs of operational instructions were constrained principally by floating point throughputs, which were approaching their limits in both models.

![Figure 7.1 Mean dispatch rates for operational instructions.](image)

Examining only mean dispatch rates tends to obscure the dynamic characteristics of Concurro’s instruction dispatch and issue behaviour. The parallelism profiles of many workloads vary greatly over their run times, making the capacity for high peak instruction rates desirable in order to compensate for periods of necessarily low activity. Such variation in activity is observable in Figure 7.2, which plots the frequency distributions of instruction dispatch and issue rates for model 832 executing *gauss*. The distributions for other benchmarks showed generally similar characteristics, but with lower variance and greater symmetry. Varying loop bounds in *gauss* give rise to a significant presence of low instruction rates in Figure 7.2.

Note that even though the average throughput for *gauss* was 7.1 instructions per cycle, dispatches of 8 or more instructions occurred in more than 61%
of processor cycles. Dispatch rates beyond 8/cycle were possible when control instructions were dispatched in parallel with a full complement of operational instructions. The dominance of 8-instruction dispatches indicates not only effective latency tolerance, but also lack of blocking from FU operand queues. The decoupling of issue from dispatch and the availability of more than 8 real FUs allowed issue rates to readily exceed 8/cycle and gave rise to a flatter distribution, though still peaked at 8 instructions.

![Graph showing instruction dispatches and issues per cycle](figure7_2.png)

**Figure 7.2** Distribution of instruction dispatches and issues per cycle in model 832 for benchmark *gauss*.

### 7.1.2 Functional Units

The results for operation bus utilization would suggest that poor utilization of the FU set accompanied large configurations of Concurro, but this is not the case. In the absence of a specific circuit design, the amount of FU hardware can be estimated by counting the number of distinct pipestages required to implement the FU set: this assumes that stages of any FU pipeline have similar critical path delays and thus broadly similar gate delays.

---

7Stages of a recirculating pipeline (such as the divider) are counted once only.
counts. Not including the main memory and D-cache, a minimal standard set of eight real FUs (as in models 104 and 208) comprises 17 pipestages; the addition of 2 FUs to model 416 and 10 FUs to model 832 brings the pipestage counts of these configurations to 20 and 38, respectively (please refer to Appendix C). These hardware estimates alone give us an indication of improved utilization in the larger configurations, since the model 832 to model 104 group count ratio is almost four-fold these models' pipestage ratio. With a maximum operational instruction dispatch rate of 1/cycle from model 104, the bulk of its FU allocation can be considered as necessary overhead to support the instruction set; it is not before model 416 that FU numbers scale with dispatch capability.

A more accurate indication of overall FU utilization can be obtained from the utilizations of individual pipestages. For a pipeline the average number of stages in use in any cycle is simply the product of the pipeline's issue rate and latency. An approximation of the actual utilization of FU hardware is then the ratio of all pipestages in use to total pipestages. Table 7.1 lists FU utilization factors derived in this way for various processor configurations. To determine optimality of the FU allocation in model 832, the benchmarks were executed by a reduced configuration, model 832R, which is model 832 with its normal FU set replaced by a minimal 8-FU set (and a single-ported D-cache).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>model 104</th>
<th>model 208</th>
<th>model 416</th>
<th>model 832</th>
<th>model 832R*</th>
</tr>
</thead>
<tbody>
<tr>
<td>carries</td>
<td>6%</td>
<td>11%</td>
<td>17%</td>
<td>16%</td>
<td>11%</td>
</tr>
<tr>
<td>compress</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>2%</td>
<td>3%</td>
</tr>
<tr>
<td>gauss</td>
<td>16%</td>
<td>32%</td>
<td>51%</td>
<td>50%</td>
<td>62%</td>
</tr>
<tr>
<td>loops</td>
<td>13%</td>
<td>23%</td>
<td>24%</td>
<td>16%</td>
<td>28%</td>
</tr>
<tr>
<td>matmult</td>
<td>17%</td>
<td>33%</td>
<td>48%</td>
<td>54%</td>
<td>60%</td>
</tr>
<tr>
<td>tf</td>
<td>6%</td>
<td>8%</td>
<td>8%</td>
<td>5%</td>
<td>9%</td>
</tr>
<tr>
<td><strong>Means:</strong></td>
<td><strong>10%</strong></td>
<td><strong>18%</strong></td>
<td><strong>25%</strong></td>
<td><strong>24%</strong></td>
<td><strong>29%</strong></td>
</tr>
</tbody>
</table>

* model 832 with the functional unit set of model 104.

**Table 7.1** Approximate utilization factors for functional unit hardware.

As in many other utilization results, the FU utilization factors in Table 7.1 display a dependence on software scalability. Namely, scalable benchmarks were able to take advantage of larger configurations while less parallel

---

[^6]: The effective address generators of cache and memory ports count as one pipestage each.
programs wasted the additional hardware. These effects caused the utilization factors averaged over all benchmarks to rise with group count, peak at 4 groups, and fall away slightly at 8 groups; provided that the highly parallel benchmarks remain scalable we can expect utilization to approach an asymptotic limit of around 20% beyond 8 groups. Benchmarks carries, gauss, and matmult were able to achieve up to three times better FU utilization in model 832 than in model 104, proving the value of shared FUs in combination with parallel instruction dispatch. In practice the advantage of sharing is lowered somewhat due to the extra hardware necessary for operand queues and instruction selectors.

The integer/floating point benchmark dichotomy prevented FU utilization factors from ever reaching 100%. Best utilization, then, was dependent on the cost of critical resources, allowing the processor to utilize a greater fraction of the FU hardware on floating point programs than on integer codes. For floating point codes utilization is also of greater concern owing to the high implementation costs of floating point arithmetic units and the memory systems that supply them. Let us examine the behaviour of model 832 executing matmult—the only numerical benchmark to exhibit sustained floating point activity over an extended period. Averaged over the entire run time of matmult, model 832 performed 1.98 main memory access per cycle at a port utilization of 98.9%; this permitted floating point throughput of 3.92 operations per cycle, resulting in mean utilizations of 97.9% for these FUs. Even model 832R, with half the number of floating point units and memory ports, could add only 0.5% to these utilization factors.

Configuring the FU set is essentially an optimization problem. For a given group count, FUs are allocated with the goal of maximizing utilization and performance simultaneously; with small FU sets utilization is maximized to the detriment of performance, whereas with sufficient FUs to prevent sharing performance is maximized at the expense of utilization. Restricting the FU set in model 832R resulted in generally improved FU utilization, although the relatively minor overall rise of 5% in Table 7.1 indicates negligible wastage of the normal FU set in model 832. Creating FU bottlenecks to improve utilization had, of course, disastrous effects on performance (see Section 7.3), with throughput on carries and matmult reduced by 71% and 50%, respectively. The FU allocations of the larger standard configurations of Concurro would appear reasonable, if not optimal, in light of the operation bus utilizations shown in Section 7.1.1 and the near 100% FU utilizations observed here.
7.1.3 Result Buses

Result buses and context groups in Concurro occur in equal numbers in the expectation that the majority of operational instructions will write results back to registers. Stores, and an insignificant number of cache control, system, and no-op instructions, are the only instructions not generating register results. Accordingly, mean result rates are always lower than corresponding instruction issue rates, and the utilization of result buses, plotted in Figure 7.3 as results per cycle for models 416 and 832, follows closely the operation bus utilization shown in Figure 7.1. Clearly then, the result buses are never in danger of over-utilization.

![Figure 7.3 Functional unit result rates.](image)

Figure 7.3 also indicates in addition to the actual result rates the mean demand result rates: the average number of FU results ready for write back in each cycle—regardless of result queue or bus availability. With the number of FUs outstripping the number of result buses in all models, the instantaneous demand for result write back can exceed the available write back bandwidth, causing occasional pipeline stalls when the result queues fill. The differences between the demand and actual result rates reflect the limitations of the synchronous multiplexors and finite result queues.
Demand and actual result rates differed more in model 416 than in model 832 due to the greater total result buffering available from the larger configuration; differences of up to 8% were measured for model 416. However, benchmarks *compress* and *tf* generated insufficient instruction rates to produce any discernable result rate differences in either model. This topic is examined further in Section 7.3.1.

By following the uses and transformations of fetched instructions it becomes apparent that a significant proportion of fetched instructions never go on to write results to registers. Figure 7.4 shows the fate of the typical instruction fetched in model 832. After instruction buffer losses only 79% of fetched instructions reach dispatch and, subsequently, issue. Approximately 87% of the dispatched instructions are of the operational kind, with the remainder being control instructions executed by context branch units. Accounting for store and no-operation instructions leaves 90% of operational instructions—62% of fetched instructions—to generate register results. These statistics bear out the observed pattern of decreasing utilization as we move from the instruction bus to the result buses.

![Figure 7.4 Typical destinations of fetched instructions in model 832.](image)

### 7.2 Latency Tolerance

The utilization results up to this point have demonstrated that complete hiding of datapath latency was possible from standard configurations of Concurro given workloads of adequate parallelism. In this section, processor...
configurations containing the standard context arrangements are tested with FUs and memories of varying latencies in order to gauge Concurro's sensitivity to implementation variations. It should be noted, however, that basing implementation decisions solely on the results of saturated multi-threading can unreasonably bias the performance characteristic against single threaded workloads, thereby limiting the general applicability of the machine. Tolerance of main memory latency is undoubtedly an important requirement in scientific codes, but for integer programs, operating out of cache, the dominant latencies to bear are those of the FUs themselves.

### 7.2.1 Functional Unit Latency

Typical overheads of one half-cycle per instruction for data transmission and resource contention by the FUs are assumed for the standard configurations of Concurro. To ease timing constraints and allow greater execution time, particularly for integer operations, an extra pipestage can be added to each FU pipeline. The effect on performance of so lengthening each FU (including the effective address generators) latency by one cycle is shown in Figure 7.5 in terms of relative throughput differences for models 416 and 832.

![Figure 7.5](image)

**Figure 7.5** Impact of lengthening each functional unit latency by one cycle.

Adding a cycle to each FU latency consistently lowered performance levels for all benchmarks, reducing harmonic mean throughput by 9% in model
416 and 11% in model 832. With excess parallelism to hide the added latency, benchmarks carries, gauss, and matmult allowed processor throughput to approach to within 1% of normal levels for both models. Performance on the remaining benchmarks was significantly affected, however, with tf causing the most losses due to its greater reliance on bypasses of write back in the instruction pipeline. It should be noted that for about 48% of compress and tf’s instructions adding one cycle latency to FUs effectively doubled the instruction latency, yet performance drops of not even a quarter were observed with these benchmarks. Compared with tf, which provided 4 threads for hiding latency, the single threaded compress program suffered relatively low performance drops, illustrating that pipelining and pessimistic scheduling can impart a degree of latency tolerance on their own.

Figure 7.6 Performance variation against functional unit latency.

The exploration of FU latency tolerance was expanded by trialing Concurro with both single-cycle pipelines for each FU and double-length pipelines; cache and memory latencies remained normal, although the address generators were considered as FU pipelines. The single-cycle FUs are intended to reveal best-possible performance, whereas the doubled pipelines represent relatively low-cost implementations. Figure 7.6 plots the relative changes in throughput for these latency settings in models 416 and 832.
Reducing FU latency uniformly to 1 cycle increased harmonic mean throughput by 1% in both models 416 and 832. For the integer benchmarks already making frequent use of single-cycle operations the setting of FU latencies to 1 cycle had no effect. However, in loops a single-thread recurrence relation involving floating point arithmetic was speeded up substantially, improving performance on this benchmark by up to 11%. Doubling latencies produced a more pronounced performance change on all the benchmarks, with mean throughputs reduced by 12% in model 416 and 14% in model 832. The results for compress and tf were essentially identical to those in Figure 7.5, since doubling integer operation latencies was equivalent to adding an extra pipestage. Performance on loops fell victim to Amdahl’s Law. In contrast, multithreading enabled throughputs on the remaining benchmarks to remain within a few percent of normal, typically, despite FU latencies being 100% higher than in the standard configurations.

Of the three FU latency options examined here, only the two options for pipeline extension are expected to be feasible. The 1 cycle results, however, did indicate that close to optimal performance was possible from the standard FU configurations. Extending each FU pipeline by one stage serves to superpipeline Concurro’s instruction pipeline from the perspective of logical and integer execution stages [Joup89]. Given that this extra cycle of latency can be partially hidden through pipelining in sequential codes, and has negligible impact on the performance of multithreaded and floating point workloads, the single-stage extension option offers an acceptable means of alleviating layout constraints in implementation.

### 7.2.2 Memory Latency

Despite it being a multithreaded machine, Concurro provides both cached and uncached access to data memory to cater for a wide range of workloads with varying parallelism and bandwidth characteristics. Caches confer natural tolerance of their backing-store latency; in contrast, direct accesses to main memory are exposed to its full latency, which would severely degrade datapath utilization were it not for multithreading. Main memory accesses are typically the longest-latency operations performed by Concurro, setting the practical upper bound requirement for its latency tolerance. Allowing for cache misses, a load from main memory was assumed to require more than an order of magnitude more latency than a typical load from D-cache in the standard configurations. It is fortunate, then, that the programs most likely to benefit from uncached accesses—scientific codes—also often exhibit significant data parallelism, making multithreading to saturation readily feasible.
To show the division between the cached and uncached classes of workload, benchmark \( tf \), representing a program with low parallelism and cache-only accesses, and benchmark \( gauss \), representing a parallel program making frequent uncached memory accesses, were executed on models 416 and 832 with various main memory latencies. So as to maintain full pipelining of accesses the loads-pending queues were set in each case to 16 entries or one plus the memory latency in cycles number of entries—whichever was larger. The normalized throughputs of models 416 and 832 against memory latency are plotted in Figure 7.7, where throughputs of 1 are assigned to executions with 25-cycle memory.

![Normalized Throughput vs. Memory Latency](image)

**Figure 7.7** Normalized throughput versus main memory latency.

Not surprisingly, the results indicate that caching afforded the processor tolerance of a wider range of main memory latencies than multithreading alone. It is important to note, however, that while cached accesses were appropriate for the integer programs, the scientific codes were forced to make uncached accesses to overcome the bandwidth deficiency of the D-cache.\(^9\) For both \( gauss \) and \( tf \) the relatively gradual slopes of their

\(^9\)For example: with benchmark \( gauss \) modified so that all of its uncached accesses were replaced by cached accesses, the main memory latency set to 25 cycles, and the D-cache configured as dual-ported, execution on model 416 proceeded with a normalized throughput of just 0.61.
performance curves below 35-cycle latency are indicative of effective latency tolerance; some variation in performance is inevitable in this region of the graphs due to less than perfect program parallelism and alteration of the critical path.

The knees in the gauss throughput curves appear in the neighbourhood of 40 cycles latency, corroborating the 42-cycle prediction made in Section 5.1.2. As can be seen from Figure 7.8, increasing memory latency reduced the number of contexts ready to contend for dispatch to the point where an average of only one context per group was ready in each cycle. At 45 cycles latency and beyond the processor was unable to hide the entire main memory latency on gauss, although partial hiding caused performance to drop away more slowly than linearly. The transition to linear-mode multithreading was accompanied by a sharp rise in blocked context numbers in both models; however, ratios of contexts ready to group count remained above one because of nonuniformity in the demand for instruction dispatch.

These results suggest that the memory latency of 25 cycles in the standard configurations may be somewhat conservative, allowing a 35-cycle memory for reducing cost with negligible performance impact. Nonetheless, it is
desirable by the considerations of Section 5.1.2 to retain a latency margin so that adequate latency tolerance can be achieved on workloads with lower instruction-level parallelism.

7.3 Datapath Bandwidth

Without adequate bandwidth from critical components of the datapath it is obviously impossible to attain high instruction processing rates. What may be perceived as a disadvantage for multithreaded processors—their high FU and memory bandwidth requirements—is actually an observation of these machines’ capacity for intensifying utilization. As we have seen in Section 7.1, bandwidth and utilization are often complementary issues, needing balance to optimize the overall cost/performance ratio of the system. This section is concerned with the efficiency of the instruction and result transport systems and the bandwidth provision aspect of the bandwidth-utilization trade-off.

7.3.1 Datapath Elasticity

The elasticity provided by the operand and result queues is intended to maximize the rate of data passing through the FUs by decoupling them from the group operation and result buses. For the operand queues this occurs by maintaining an even supply of instructions and data to the FUs despite the lulls and surges of instruction dispatch. FUs in the standard models of the processor are fed from operand queues whose length is proportional to the number of groups sharing real FUs. To measure the adequacy of this arrangement, Concurro was reconfigured with operand queues of one-entry capacity (i.e. reduced to latches) and effectively infinite capacity (actually, just long enough never to fill). Figure 7.9 shows the impact of these alterations on the performance of models 416 and 832.

The shortened, degenerate operand queues caused decreased throughputs on all benchmarks in both models, with the greatest effect occurring for the bandwidth-sensitive matmul program. Overall performance dropped by 0.5% on both models. With single-entry queue capacities, individual queue lengths were recorded as empty more frequently than in the standard models, thus raising the frequency of FU starvation on multithreaded workloads and reducing instruction throughput. This effect was not applicable to compress, though, since instruction issue rates on this benchmark never lagged behind dispatch rates.
Even though infinite operand queues did indeed eliminate all dispatch blocking by busy operational FUs, performance on many of the benchmarks was slightly reduced—contrary to expectations. Harmonic mean throughputs with infinite queues fell by 0.2% in both models. It is not entirely clear why these minor drops occurred, but a feature common to these runs was an increase in the total number of contexts blocked (waiting for operands) in response to extra issue delays introduced by the extended queues; while multithreading buffers against datapath latency, it cannot do so perfectly.

On most of the benchmarks operand queue lengths rarely exceeded their limits in the standard configurations. For the numerical programs “infinite” queue lengths were in the order of 32 entries for the load/store unit, and 12 entries for the floating point FUs. Given the cost penalty and performance degradation of such long operand queues, there is no justification for implementations of operand queues longer than the group count; the alternative of reducing operand queues to latches, however, may result in improved cost/performance ratios of real implementations.

Figure 7.9 Performance variation with operand queue capacity.

---

10Infinite in the sense of the queue capacity being sufficient that blocking never occurs.
The result queues, by absorbing blockages of the write back arbiters, ensure efficient use of the result buses. However, unlike the operand queues, which must always be present in some form to act as reservation stations for instruction issue, result queues can be eliminated completely with the FUs interlocked by the write back arbiters. Normalized throughputs without result queues are shown in Figure 7.10 for models 416 and 832, along with throughputs relating to three other configurations—one-entry queues or result buffers, infinite result queues, and infinite operand queues with infinite result queues.

Removing result queues from Conurro cut significantly into performance on the parallel benchmarks, and lowered harmonic mean throughput in
models 416 and 832 by 5% and 3%, respectively. The cause of this is evident in Table 7.2, which indicates heavy contention for write back and thus frequent stalls of the FU pipelines. The installation of even simple result buffers greatly reduced the difference between demand and actual result rates, causing mean performance in model 832 to approach to within 0.2% of normal. An important factor in the success of result buffers is the extra buffering capacity provided by their placement on each result bus for each functional class rather than at the output of individual FUs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No queues</th>
<th>1-entry queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>carries</td>
<td>26%</td>
<td>3%</td>
</tr>
<tr>
<td>compress</td>
<td>21%</td>
<td>0%</td>
</tr>
<tr>
<td>gauss</td>
<td>41%</td>
<td>6%</td>
</tr>
<tr>
<td>loops</td>
<td>25%</td>
<td>5%</td>
</tr>
<tr>
<td>matmult</td>
<td>34%</td>
<td>4%</td>
</tr>
<tr>
<td>tf</td>
<td>10%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Means:</strong></td>
<td><strong>26%</strong></td>
<td><strong>3%</strong></td>
</tr>
</tbody>
</table>

Table 7.2 Excess of demand result rates over actual result rates for null and one-entry result queues in model 832.

Progressing to infinite result queues allowed demand and actual result rates to converge. Despite the drops in result blockages, this configuration offered negligible performance improvement on most benchmarks, increasing harmonic mean throughput by 0.6% in model 416 and 0.1% in model 832. Providing infinite operand queues as well produced some slow-downs, in common with previous observations. While result queues are essential for maintaining FU throughput, queues need not be any larger than buffer size to bring the majority of their benefit.

### 7.3.2 Functional Resources

The FU and memory port allocations in the standard configurations are compromises designed to minimize cost while maximizing performance, but delivering less than optimal performance for some workloads. To determine the extent to which performance potential was lost, the benchmarks were executed on processors equipped with unrestricted functional resources. This involved duplicating real FUs so that each functional class except the system class had as many FUs as groups, dual-porting the D-cache, and setting the number of direct access memory ports to the group count. The result queue capacities scaled, as usual, with the number of real FUs. Figure 7.11 plots the relative throughput differences of these altered models.
416 and 832.

As expected from its excellent memory and floating point utilization in the standard models, *matmult* executed with markedly improved throughput when given the extra memory and FU bandwidth of the unrestricted models. In this situation instruction dispatch bandwidth proved to be the limiting factor. Similarly, the memory-bandwidth intensive Livermore Kernels in *loops* showed significant performance gains. For the remaining benchmarks, however, little, if any, of the additional FU bandwidth could be exploited. Overall performance improvement, at 4% in model 416 and 2% in model 832, was modest accordingly. In practice, processors with unrestricted functional resources are likely to be prohibitively expensive—and unjustified—but their simulations show that an appropriate set of shared, heterogeneous FUs can provide performance that approaches best-case for a majority of workloads.

![Figure 7.11 Performance with unrestricted functional resources.](image)

In Section 7.1.2 the utilization benefits for configurations at the other end of the spectrum (with minimal functional resources) were readily apparent. However, as can be seen from Figure 7.12, restricting model 832 to one real FU per functional class (thus, one memory port) severely degraded the machine's performance. Operand and result queue capacities in the restricted configuration were set to 8 entries and 1 entry, respectively, to compensate for the greater FU sharing.
Figure 7.12 Performance of model 832 with one real functional unit per functional class.

The execution bottleneck arising from the minimal FU set reduced model 832's harmonic mean throughput by 21%. For the scalable benchmarks, carries, gauss, and matmult, this configuration created heavy mismatches between the hardware and software, typically causing blocking in more than 27 out of 32 contexts in each cycle, and leading to significant drops in performance. The importance of balance and scalability in the processor's design are demonstrated clearly by these poor, but not unexpected, results. When performance is considered relative to that of model 416, we observe no practical advantage for the 8-group configuration unless the number of functional resources is scaled in accordance with the group count.

7.3.3 Memory Bandwidth

Attempting to lower system cost by simply reducing the number of memory ports, as in the previous subsection, was, unfortunately, highly detrimental to Concurro's performance on the floating point codes. An alternative cost reduction strategy is to alter the data transfer rates of the memory ports. From the standpoint of implementing CPU-memory transmission lines the most favourable option would be to retain the long word port width of the standard configurations, but lower the data frequency to a fraction of the processor clock frequency. To test this option, Concurro was simulated with all main memory ports operated at half-normal frequency. Figure
7.13 plots the resulting throughputs, relative to the standard configurations, of models 416 and 832 with both the usual number (one port for every 4 groups) and double the number of direct access memory ports.

![Figure 7.13 Performance with transfer rates of main memory ports halved.](image)

Halving the memory port bandwidth alone produced in both models effects similar to those observed in Section 7.3.2 for the restricted FU set. Those benchmarks operating exclusively out of cache, such as *carries*, were largely immune to the reduced memory bandwidth, since this merely prolonged the cache miss penalty; the 2% drop in performance on *compress* reflected its relatively high cache miss rate. Performance on the floating point programs, in contrast, was badly limited by a lack of memory bandwidth for these programs' array-based algorithms. (With floating point operation rates so constrained, there is some justification for also halving the number of floating point FUs in model 832.) Overall instruction throughput was reduced by 17% in model 416 and 12% in model 832.

Doubling the number of direct access memory ports compensated almost fully for the halved port bandwidth. Normalized throughputs on the floating point benchmarks rose to within a few percent of unity in all cases, whereas performance on the integer benchmarks remained unchanged. Accordingly,
harmonic mean throughputs were 1% and 2% below normal in models 416 and 832, respectively. From these results we can infer near ideal sharing behaviour in the load/store unit. The chief drawback of these configurations is the large number of memory ports required; for example, fitting two additional non-multiplexed ports to model 832 would increase that processor's pin count by almost 200 pins. It is therefore unlikely, for packaging reasons in particular, that doubling the number of direct access ports would be feasible in configurations larger than model 416. Whatever combination of memory parameters is used, to provide memory bandwidth to the level of the standard configurations or not is largely a choice about catering adequately for scientific workloads.

7.4 Data Cache Performance

Usage patterns of the data cache are generally more diverse than those of the direct access memory. The design of Concurro's D-cache is, accordingly, a compromise between providing low-latency access, for the synchronization controller and workloads able to exploit locality, and sustaining adequate access bandwidth under heavy miss rate conditions. These characteristics and the behaviour of the synchronization controller are the focus of this section. Since some of the benchmarks made negligible use of the cache, our attention will be confined to Concurro executing benchmarks compress, tf, and gauss, which also represent a broad cross-section of general-purpose workloads. As in the I-cache experiments, it was necessary to reduce D-cache sizes to unrealistic levels in order to observe significant miss rates in some cases.

7.4.1 Miss Rate

With two memory access options—cached and uncached—those benchmarks that were expected to make good use of the D-cache were coded to use it, whereas the array-sweeping programs accessed main memory directly, hiding its latency through multithreading. The decision to use the D-cache or direct access memory is rarely difficult, and one that a compiler could be expected to make upon recognizing data-parallel operations; pragmas in the source code could also be used to guide this choice. Of the three benchmarks used in the cache study, tf had consistently lowest miss rates on account of its predominantly byte-sized data accesses. In gauss the D-cache was used primarily for pivot-row storage of long word data and synchronization, resulting in more frequent cache misses than on tf and requiring the greatest cache bandwidth of all the benchmarks. The most
demanding use of the D-cache was from *compress*, which relied on frequent, locality-destroying look-ups of a hash-table for operation.

---

2-way, 64-byte lines, non-blocking to 4 misses

2-way, 64-byte lines, blocking after any misses

2-way, 64-byte lines, non-blocking to 1 miss

2-way, 32-byte lines, non-blocking to 4 misses

4-way, 64-byte lines, non-blocking to 4 misses

---

![Figure 7.14](image.png)

**Figure 7.14** Mean data cache miss rate versus cache size, for various associativities, line sizes, and miss blocking.

The effectiveness of the non-blocking D-cache controller was assessed by simulating Concurro with five cache configurations, including the standard 2-way set-associative configuration with 64-byte lines and non-blocking under misses up to 4 misses. The mean miss rates (over *compress*, *gauss*, and *tf*) for these configurations are plotted against cache size in Figure 7.14 for models 416 and 832; for the dual-ported D-cache in model 832 each port access was counted separately in determining the miss rate. Compared with I-cache miss rates (see Figure 6.13), the D-cache miss rates were invariably higher for a given cache size. This difference is attributed to larger working sets in the data, which also produced a less pronounced knee in the data miss rate curve. Down to cache sizes of 4 kB both models 416 and 832 showed negligible difference in their miss rate, in accordance with these models' identical cache configurations. However, at cache sizes of 2 kB and 1 kB intensified interference effects in model
832 running *gauss* caused mean miss rates in model 832 to rise above those of model 416.

For D-caches of 4 kB size and larger, mean miss rates fell below 5% and differed, in absolute terms, by less than 1% over the five cache configurations. At these low miss rates the frequency of misses was insufficient to cause many missed accesses to queue, and thus reveal any significant differences between configurations with different blocking behaviour. More limited prefetch capability in the cache with 32-byte lines put it at a slight disadvantage against caches composed of larger lines.

With cache sizes below 4 kB came substantial miss rates and differences in cache behaviour. At these miss rates the mean time between misses became short relative to miss service times, giving cache controllers that could overlap miss service with cache accesses a distinct advantage. By allowing accesses to proceed past a miss, or misses, the non-blocking caches favoured hit references over waiting miss references, thereby minimizing interference from competing threads in the multithreaded programs and lowering the miss rate [WeGu89]. Figure 7.14 shows that tolerance to one miss produced significantly improved miss rates, but tolerating further misses yielded diminishing returns—with non-blocking to 4 misses being necessary to double the initial improvement. Combining non-blocking with twice the set associativity or half the line size proved better still, as these configurations directly lowered the conflict miss rates of the smaller caches. For more reasonable cache sizes, however, the improvements obtained by increasing associativity beyond 2 lines/set were insufficient to justify the extra cost and possible cycle time degradation introduced by a more associative cache.

### 7.4.2 Access Latency and Performance Impact

While miss rates reveal much of the cache behaviour under misses, miss rates do not necessarily predict accurately one of the most important cache performance metrics—access latency. To measure cache latency, each request to the D-cache was time stamped on arrival so that at the completion of each access its latency could be determined from the cycle count difference. These measures were performed on all loads, stores, and line flushes; loads were considered complete when the requested data was returned, whereas stores and flushes were considered complete at the time of tag update. Figure 7.15 plots the overall D-cache access latency, averaged over *compress*, *gauss*, and *tf*, against cache size and configuration.
Figure 7.15 Mean data cache access latency (for loads, stores, and flushes) versus cache size, for various associativities, line sizes, and miss blocking.

The less varied miss behaviour for large caches produced access latency characteristics that were broadly in agreement with miss rate characteristics. A notable exception was the blocking cache configuration, which caused access latency to rise sharply with decreasing cache size. This behaviour was partly an artifact of the cache controller design; since all (time stamped) access requests were latched immediately upon receipt, it was possible for hit accesses to be latched and delayed behind miss accesses, thereby artificially enlarging the mean access latency as if the miss rate had doubled. The probability of hit accesses being delayed in this way was far greater in the blocking caches than in the non-blocking caches, which locked the data and tag stores much less frequently. Since a miss is likely to be followed closely by a hit access, the access latency of the blocking cache can be approximated by:

\[ d_{blk} = 1 - 2m + 2mT_{miss} \]

where \( m \) is the cache miss rate and \( T_{miss} \) is the mean access time for
misses. Latency predictions to within 10% of measured values were possible with this equation.

Three main peaks were evident in most of the cache latency distributions. The largest peak appeared, expectedly, at 1 cycle (with a significant presence at 2 cycles due to miss completions) for the majority of hit accesses. Misses on loads and stores introduced peaks at 27 cycles and 35 cycles, respectively, giving an average access latency on a miss of around 32 cycles. Yet setting $T_{\text{miss}}$ to 32 cycles in the familiar access latency equation, $d = 1 - m + mT_{\text{miss}}$ [HePa90], gives values of $d$ that are substantially less than those plotted in Figure 7.15 for the non-blocking caches. Examination of the latency distributions reveals that on gauss and tf a lower proportion of accesses have a latency of 1 cycle than miss rates would indicate. Apart from the effects of additional miss queue delays on $T_{\text{miss}}$, the occurrence of congruence hits (see Section 3.4.3) contributed hit access times comparable to $T_{\text{miss}}$, which increased mean access latencies as if miss rates had grown by 2 to 3 times. This, however, is not actually a problem for the non-blocking caches, since the extra latency is usually both unavoidable and completely overlapped; the lower access latency predicted by the simple equation is, in fact, an underestimate that hides the cost of blocking subsequent hit accesses.

At cache sizes below 4 kB the steep rises in miss rates were responsible for corresponding increases in access latency. A combination of high miss rates and hit delays made latencies of the blocking cache almost double those of the best performing non-blocking configurations. Of the 2-way set-associative, 64-byte line, non-blocking caches, slightly lower latencies were recorded by the cache allowing accesses under 1 miss than allowing accesses under 4 misses, despite the significantly higher miss rates of the former configuration. The effect here was that of increased miss queue delays in the controller with greater miss tolerance; longer miss queues tend to extend the average miss service time and allow a greater accumulation of congruence hits, which increase the apparent access latency. The other non-blocking caches having the lowest miss rates maintained correspondingly low latencies, with the configuration of 32-byte lines compensating for its minor miss rate disadvantage through a lower cache line reload time.

With access latency being but one aspect of cache performance, it is not surprising that processor performance trends deviated from the expectations suggested by the latency results. Figure 7.16 shows the performance obtained
from the test benchmarks, relative to their execution on standard models, when executed under four different configurations of a 4 kB D-cache; the 4 kB size was the smallest for which similar miss rates were obtained from all cache configurations. Reducing the cache size from 32 kB to 4 kB lowered harmonic mean instruction throughput (over the three benchmarks only) by as little as 11%, for the 64-byte line, non-blocking to 4 misses configuration, and as much as 29%, for the blocking cache.

![Figure 7.16 Performance versus cache configuration for a 4 kB data cache.](image)

The differences in performance between models 416 and 832 were significant only for gauss. Since gauss was a scalable benchmark, its requirement for D-cache bandwidth in model 832 was almost double that in model 416. Although both models recorded similar miss rates with the 4 kB D-cache, the more intense cache activity in model 832 raised the time frequency of its misses by 55%, decreasing the availability of the cache relative to that in model 416.

The degraded throughput and miss rate of the blocking cache ensured that

---

11Standard D-caches in models 416 and 832 were 32 kB in size, 2-way set-associative, with 64-byte lines, and allowed accesses under at most 4 misses.
it had the most negative impact of all the caches. Clearly, some form of miss tolerance was necessary for achieving adequate performance out of cache. Despite a somewhat higher miss rate and access latency, the cache with 32-byte lines permitted throughputs similar to those obtained with the other non-blocking caches. Generally, best performance was available from the standard non-blocking cache controller—accesses under 4 misses, 64-byte lines; restricting non-blocking to 1 miss reduced the cache's average bandwidth, particularly on gauss, thereby lowering processor throughput. Nevertheless, reducing the line size or miss tolerance are both reasonable options for lowering the D-cache cost with minimal performance impact, although for larger caches, as in the standard models, the latter option is preferable on the grounds of improved array processing behaviour.

### 7.4.3 Synchronization Controller Behaviour

To ensure efficiency in fine-grained computation, the cost of data synchronization—through I- and M-structures (structures)—must be comparable to the costs of ordinary machine instructions. However, the microcoded synchronization controller, described in Section 3.4, can be very slow in situations involving long queues of deferred loads. Two measures relating to synchronization controller performance, the number of D-cache accesses made in response to each structure operation and the number of deferred loads completed upon each structure store, are listed in Table 7.3 for the benchmarks that used structures. It is desirable to minimize the number of D-cache accesses to avoid monopolizing the cache and suffering, on occasions, its miss. Low numbers of load-data returned per structure store implies minimal wastage of the cache and infrequent traversals of linked-lists holding deferred loads.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cache access/Struct. access</th>
<th>Returns/Structure store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>model 416</td>
<td>model 832</td>
</tr>
<tr>
<td>carries</td>
<td>2.0</td>
<td>2.1</td>
</tr>
<tr>
<td>gauss</td>
<td>1.8</td>
<td>1.9</td>
</tr>
<tr>
<td>loops</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>tf</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td><strong>Means:</strong></td>
<td><strong>1.8</strong></td>
<td><strong>1.8</strong></td>
</tr>
</tbody>
</table>

*Table 7.3 Mean data cache accesses per I/M-structure operation, and average number of longwords returned per I/M-structure store.*

The statistics shown in Table 7.3 indicate that the majority of structure operations were able to take advantage of the single-waiting feature of the
synchronization controller, thereby minimizing the use of linked lists. Cache access rates were as much dependent on the kind of synchronizations used as they were on the synchronization characteristics of the program. Since I-structure loads require a minimum of 1 cache access each while M-structure loads and structure stores require at least 2 accesses each, the best-case (50% loads, 50% stores) average cache activity for a pure I-structure operation stream is 1.5 cache accesses per structure access, whereas with only M-structures the rate rises to 2.0 accesses per load or store. Thus we can account for an average cache activity above 2.0 accesses per operation on carries, which used M-structures only. The other benchmarks employed I- and M-structures in combination, causing their cache access rates to be closer to the best-case M-structure average. However, loops' I-structure loads outnumbered its stores by more than 5-to-1, biasing cache activity heavily towards the low cache usage of this program's I-structure loads. 

Average data returned per structure store remained comfortably below 1 long word per store for all but the loops benchmark. The increases in return rates with processor group count reflect the greater probability of contention from the larger collection of threads in model 832. For average return rates in the vicinity of 1 long word per store, we can expect a significant proportion of deferred loads to be waiting on a linked-list, rather than at a structure location. Note, however, that data returns may not affect loads and stores equally, as shown by loops, which had relatively high return rates, slowing down structure stores, but also low average cache activity, indicating that a majority of loads were satisfied immediately. The above results would suggest that, despite potentially large overheads in maintaining I-structure and M-structure synchronization, a structure store can be managed with acceptable efficiency by a microcoded synchronization controller, provided that some optimization is applied to common access cases.

7.5 Observations and Conclusions

Multithreading in Concurro played a pivotal role in maximizing the utilization of execution resources and tolerating latency. The degree of utilization was, as a consequence, largely dependent on the scalability of the processor workload. Concurro's parallel microarchitecture, despite its provision for high peak instruction rates, was not immune to bottlenecks in the datapath induced by resource sharing. The balance between utilization and performance was difficult to achieve in a general-purpose machine, given the often conflicting needs of integer and floating point codes. Nonetheless,
utilization and performance in the standard configurations running the benchmark suite were short of best-case by small and similar margins. Parallel instruction issue improved the overall utilization of large FU sets, making such configurations feasible and permitting throughputs of several floating point operations per cycle to be sustained indefinitely.

Multithreaded performance was far more influenced by datapath bandwidth than by latency—so much so, that it was preferable to sacrifice latency for greater bandwidth through the widespread use of FIFO queues, which are ideal structures for transforming excesses in bandwidth demand into latency excesses. Sharing execution resources created the problem of combining data flows of different throughput distributions. Queues for operands and results were therefore indispensable for achieving cooperation among context units, FUs, and the result buses. In this application, simple latches were found to be all but as effective as more costly genuine queues. Adequate memory bandwidth was crucial for realizing high throughput on scientific codes. However, supplying memory data at processor clock frequency adds substantially to the cost of the memory system; trading bus frequency for memory ports exploited the stream multiplexing and bandwidth optimization capabilities of Concurro's load/store unit to lower system cost with negligible impact on performance.

Concurro exhibited generally high tolerance of all execution resource latencies. Superpipelining the FUs to simplify implementation was most detrimental to single threaded performance, although pipelining was itself valuable in hiding much of the additional latency. The most significant test of latency tolerance, however, was that presented by the main memory. Employing multithreading to hide memory latency proved as effective as caching—up to a point, beyond which the processor was forced into linear mode operation, exposing it to some portion of the latency. The transition to linear mode occurred at a memory latency of around 40 cycles, giving some scope for a less stringent memory specification and demonstrating that, in this case, 4 contexts per group could be transformed into effectively 16 subcontexts per group with the aid of loop unrolling.

In comparing various D-cache configurations it was readily apparent that the greater bandwidth of non-blocking caches made them ideal for multi-threaded usage. Miss rate, access latency, and processor performance all benefited significantly from tolerance of cache misses. For caches of reasonable size—and correspondingly modest miss rates—the complexity of the non-blocking cache controller can be minimized to tolerance of a single pending miss, which was sufficient to gain almost all of the advantages possible from non-blocking.
Chapter 8

Conclusion

The multithreaded Concurro processor architecture was developed to exploit growing integrated circuit densities in overcoming some of the physical and architectural constraints on processor performance. Superscalar performance with scalability, latency tolerance, and excellent hardware utilization were identified as some of the essential attributes of next-generation machines. Concurro was used to explore and narrow the multithreaded processor design space, and formed the basis of a simulation study into the microarchitecture and behaviour of multithreaded uniprocessors capable of multiple instruction issues per cycle.

The requirement for superscalar performance had far reaching consequences for Concurro's core architecture. The very high data bandwidth necessary to support parallel instruction issue virtually compelled the adoption of a fixed-context design and, along with it, a RISC-style instruction set architecture, augmented with thread-control instructions and I/M-structure operations. The ISA allowed, in practice, binary compatibility across varied processor configurations. With a limited set of processor contexts it was possible to implement asynchronous scheduling—adapting to interleaving or blocking behaviour dynamically—for better than 100% performance improvements over synchronous scheduling on poorly parallelized workloads. To make parallel instruction dispatch viable in this environment it was necessary to arrange context units into groups, each sharing dispatch logic among a small set of contexts. The impact of context grouping on performance was minor; however, having only 4 context units in each group made program optimizations, such as loop unrolling and static scheduling, requisite for adequate latency tolerance.

Supplying the contexts with instructions at a rate matching their execution was accomplished through the instruction cache pre-accessing blocks of
instructions for contexts to buffer in advance. Two important optimizations were found necessary in the primary cache: pre-access was permitted across cache line boundaries, rendering the cache almost immune to alignment problems; and the cache controller did not block accesses under a limited number of misses, thereby conferring latency tolerance to instruction fetching under multithreading. The use of conditional move instructions and optimizations for enlarging basic blocks was generally successful in minimizing the destructive effect of branch-induced buffer flushes. Despite the potential for bottleneck at the shared instruction cache, Concurro was able to obtain parallel efficiencies of more than 80% from configurations of 8 context groups and larger—achieving average throughputs in excess of 8 instructions per cycle in some cases.

Taking advantage of parallel instruction dispatch, the functional units were organized into distinct functional classes, each comprising one or more specialized, pipelined execution units. Operand queues—or latches, at least—decoupled instruction dispatch by context units from instruction issue by functional units, thereby easing the control burden on the contexts and facilitating the sharing of expensive execution resources, such as the load/store unit. This arrangement resulted in improved functional unit utilization, reaching, on scientific workloads, 50% overall with floating point operation throughputs at 98% of peak. Datapath bandwidth proved to be the major determinant of performance in saturated multithreading. Indeed, in many cases the performance of a program could be predicted simply from which of the functional units would be fully utilized based on the dynamic instruction mix. Therefore, devices for optimizing bandwidth, such as queues at subsystem boundaries, and split-phase loads from cache and main memory, were particularly beneficial. Multithreading enabled Concurro to tolerate not only main memory and synchronization latencies, but also the extra latencies of longer functional unit pipelines, demonstrating that the architecture is suitable for future high clock rate implementations.

This thesis has demonstrated that superscalar performance in a multithreaded microprocessor is viable, and an attractive objective given its advantages of readily accessible performance on a variety of workloads, latency tolerance, and all but optimal hardware utilization.

### 8.1 Major Microarchitectural Features

Table 8.1 lists the most significant features of the Concurro microarchitecture, along with their individual contributions to harmonic mean throughput in models 416 and 832. Note that each of the performance gains occurred
with the other features present. In many cases the performance gains on particular programs were considerably higher than listed, but the bias of the harmonic mean towards single threaded performance obscures the importance of some features.

<table>
<thead>
<tr>
<th>Machine Feature</th>
<th>Mean Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>model 416</td>
</tr>
<tr>
<td>&gt;1 Context per Group</td>
<td>46%</td>
</tr>
<tr>
<td>Asynchronous Scheduling</td>
<td>122%</td>
</tr>
<tr>
<td>Pre-access I-cache</td>
<td>5%</td>
</tr>
<tr>
<td>Scalable FU Set</td>
<td>7%</td>
</tr>
<tr>
<td>Result Queues</td>
<td>5%</td>
</tr>
</tbody>
</table>

Table 8.1 Mean performance gains of major microarchitectural features added in isolation.

Populating each group with four rather than one context unit effectively added node multithreading to Concurro, when viewed as a MIMD machine. This afforded the processor the latency tolerance necessary to use the parallel datapath to its fullest extent. Without multiple contexts per group, there would be little real need for the substantial hardware parallelism of models 416 and 832. To exploit the benefits of latency tolerance it was, of course, necessary to scale the functional unit set in accordance with the processor group count. However, the division of execution resources into independent functional classes, each with its own, appropriate level of sharing, makes scaling in Concurro considerably more cost-effective than if monolithic execution units were to be duplicated.

Asynchronous thread scheduling gave Concurro the advantages of pipeline interleaving without significantly compromising single threaded or linear mode multithreaded performance. This is desirable not only from the viewpoint of applicability, but also for portability: programs not containing fork instructions can run with acceptable performance immediately—without recompilation. Implementing asynchronous scheduling incurs the cost of arbitration logic, but this and even more complex hardware is already a feature of superscalar processors.

Result queues and instruction cache pre-access with line crossing had less impact than the other features, although the relatively low costs of these optimizations make their inclusion worthwhile.
8.2 Reduced-Cost Designs

This section summarizes some of the cost reduction measures suggested throughout the thesis. Applying these simplifications together indicates their interactions as much as it does their individual effects. Table 8.2 lists the simplifications applied cumulatively to models 416 and 832, while Figure 8.1 plots the normalized throughputs of these reduced configurations; the mean throughputs in Figure 8.1 (also listed in Table 8.2) are harmonic mean values.

<table>
<thead>
<tr>
<th>Hardware Simplification</th>
<th>Performance Change</th>
<th>model 416</th>
<th>model 832</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1-cache miss tolerance reduced to 1 miss (Sec. 6.3)</td>
<td>0.0%</td>
<td>0.1%</td>
<td></td>
</tr>
<tr>
<td>b a &amp; operand queues reduced to latches (Sec. 7.3.1)</td>
<td>-0.5%</td>
<td>-0.2%</td>
<td></td>
</tr>
<tr>
<td>c b &amp; result queues reduced to latches (Sec. 7.3.1)</td>
<td>-0.5%</td>
<td>-0.5%</td>
<td></td>
</tr>
<tr>
<td>d c &amp; D-cache miss tolerance reduced to 1 miss (Sec. 7.4)</td>
<td>-1.3%</td>
<td>-1.5%</td>
<td></td>
</tr>
<tr>
<td>e d &amp; main memory latency set to 35 cycles (Sec. 7.2.2)</td>
<td>-6.3%</td>
<td>-7.3%</td>
<td></td>
</tr>
<tr>
<td>f e &amp; all FU pipelines extended by one stage (Sec. 7.2.1)</td>
<td>-14.3%</td>
<td>-16.8%</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2 Cumulative effects of hardware simplifications.

Progressing from simplification a to f, representing progressively less complex and costly implementations, produced increasingly significant performance degradations. An exception, however, occurred on reducing the I-cache miss tolerance to a single miss, which improved performance on benchmark loops marginally. The complex interactions between the machine features lessened the individual effects of the simplifications, and in some benchmarks actually produced non-monotonic performance changes.

Since the essential elements of cache non-blocking and data queuing were all still present at simplification d, albeit minimized, its effect on performance was negligible, causing less than a 2% reduction in mean throughput.

These configurations are thus attractive implementation alternatives to the more generalized standard configurations containing large numbers of elastic FIFO queues.

Increasing main memory and pipeline latencies in e and f introduces the most substantial cost reductions (although e is not strictly a processor simplification). These simplifications cut deepest into the performance on compress and tf, leaving saturated multithreaded performance relatively unaffected, indicating that the most significant effect of increasing latency in a latency-tolerant machine is to further unbalance its performance
characteristic. If we view simplifications $e$ and $f$ as opportunities for reducing cycle time rather than cost, then it may be possible to recoup part, if not all, of the performance that would have been otherwise lost.

Figure 8.1 Performance with cumulative hardware simplifications.
8.3 Future Work

The design of Concurro, although quite useable as implemented in the simulator, is not yet complete at either the macroarchitectural or microarchitectural levels. Obviously, the simplified main memory, virtual memory management, and exception handling are candidates for further study and analysis in the context of multithreading. However, much of the remaining design work involves making Concurro more robust and flexible for general-purpose computing.

The inflexibility of multithreading with a fixed context set could be mitigated through additional hardware support for thread control. Unrestricted multithreading with hardware virtualization is not advocated here, but a means to preempt arbitrary threads in software can contribute greatly to the ease and efficiency of software-assisted scheduling [CGSvE93]. Programmable timeout preemption [WeGu89] is a more hardware-oriented approach, and provides the basis for a load-balancing scheme. The implementation and application of these preemption facilities, possibly, with hardware-based virtual thread naming mechanisms, are areas for further research. Research in these areas is needed to yield solutions to the problems of processor state capture associated with exception handling and multitasking.

One of the original motivations for Concurro was the creation of efficient multiprocessors. A number of extensions to the architecture are necessary to make this possible. Currently, the architecture has no provision for distributing work across processors. Rather than asking the hardware to perform load distribution on its own, it is envisaged that handling of fork instructions be modified to optionally interrupt the root thread upon encountering a shortage of free contexts. An interrupt handler would migrate extraneous threads, using non-local knowledge to assist in load balancing. Similarly, maintaining the existing view of I-structures in a shared memory environment involves differentiating between local and remote accesses. Expanded synchronization controllers somewhat like the synchronization coprocessors in *T [NiPA92]—tagging deferred loads with node addresses and interfacing directly to the network—are a possibility for managing global structure storage.

Perhaps the most important and challenging further work concerns software rather than hardware. The strictly limited thread availability in fixed-context multithreaded processors makes them difficult compilation targets; extracting sufficient instruction-level parallelism for superscalar throughput places an additional burden on the compiler. It was, nevertheless, possible to achieve acceptable processor utilization on parallel loops using...
relatively simple—and far from optimal—code generation strategies. In
general, however, more widely applicable compilation frameworks, such as
TAM, are essential and deserve further study. Approaches to compilation
that recognize the hierarchies present in both hardware and software, and
use them cooperatively are the key to exploiting future hardware
technologies.
Appendix A

Concurro Instruction Set

This appendix summarizes the Concurro instruction set architecture for all processor configurations. The instruction set listed here is current as of November 1992 (Revision 12 of the original instruction set). A survey of the ISA and its design is presented in Chapter 2.

A.1 Programmer's Model

A.1.1 Memory

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address range:</td>
<td>32 bits or 4 GB</td>
</tr>
<tr>
<td>Byte size:</td>
<td>8 bits</td>
</tr>
<tr>
<td>Addressability:</td>
<td>byte</td>
</tr>
<tr>
<td>Multibyte quantities:</td>
<td>short word (16b)</td>
</tr>
<tr>
<td></td>
<td>word (32b)</td>
</tr>
<tr>
<td>Multibyte storage format:</td>
<td>long word (64b)</td>
</tr>
<tr>
<td>Alignment restrictions:</td>
<td>I/M-structure (64b)</td>
</tr>
<tr>
<td>Consistency:</td>
<td>floating-point (64b)</td>
</tr>
<tr>
<td>Cache coherency:</td>
<td>little-endian</td>
</tr>
<tr>
<td>Instruction size:</td>
<td>no crossing of long words</td>
</tr>
<tr>
<td>Instruction addressability:</td>
<td>weak ordering</td>
</tr>
<tr>
<td>Instruction alignment:</td>
<td>software managed</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32 bits</td>
</tr>
<tr>
<td></td>
<td>byte</td>
</tr>
<tr>
<td></td>
<td>word aligned</td>
</tr>
</tbody>
</table>
A.1.2 Registers

Global general-purpose registers: R0 (hardwired 0)
R1 – R15, 64 bits

Local general-purpose registers: R16 – R31, 64 bits

Instruction pointer: 32 bits, inaccessible as operand

Reserved registers: R29 for assembler temporary

Stack pointer: R30 by convention

Subroutine linkage: address in R31 by convention

Register channels: 32 × 64 bits

System registers: 16

A.1.3 Execution

Number of active contexts: 1 to implemented maximum

Interlocks: hardware managed

Instruction sequence restrictions: none

Per-context instruction issue sequence: out-of-order

Per-context instruction execution sequence: out-of-order

Instruction completion: out-of-order, continuable

Execution latency: variable

Speculative execution: not implemented

Processing exceptions: page faults

Maximum instruction dispatch rate: 1/context/cycle

Maximum operation dispatch rate: 1/group/cycle

Maximum actual source operands: 2

Maximum destinations: 1

Maximum result rate: 1/group/cycle

Maximum instruction fetch rate: 1/cycle

A.2 Instruction Set

The following symbols and notes apply to the instruction set tables:

Rd = destination register
Ra = first source register
Rb = second source register
R1, R2,... = sequence of registers
C = 32-bit (maximum) constant (< 32 bits may be used)
A = relocatable address
X = opcode extension code (0–15)
### A.2.1 Immediate Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orhi</td>
<td>Rd,Ra,C</td>
<td>logical OR upper half-word</td>
</tr>
<tr>
<td>andhi</td>
<td>Rd,Ra,C</td>
<td>logical AND upper half-word</td>
</tr>
<tr>
<td>xorhi</td>
<td>Rd,Ra,C</td>
<td>logical XOR upper half-word</td>
</tr>
<tr>
<td>ori</td>
<td>Rd,Ra,C</td>
<td>logical OR immediate short word</td>
</tr>
<tr>
<td>andlo</td>
<td>Rd,Ra,C</td>
<td>logical AND lower half-word</td>
</tr>
<tr>
<td>andi</td>
<td>Rd,Ra,C</td>
<td>logical AND immediate short word</td>
</tr>
<tr>
<td>xori</td>
<td>Rd,Ra,C</td>
<td>logical XOR immediate short word</td>
</tr>
<tr>
<td>shli</td>
<td>Rd,Ra,C</td>
<td>shift left logical by immediate</td>
</tr>
<tr>
<td>shri</td>
<td>Rd,Ra,C</td>
<td>shift right logical by immediate</td>
</tr>
<tr>
<td>shai</td>
<td>Rd,Ra,C</td>
<td>shift right arithmetic by immediate</td>
</tr>
<tr>
<td>multi</td>
<td>Rd,Ra,C</td>
<td>multiply by immediate short word</td>
</tr>
<tr>
<td>addhi</td>
<td>Rd,Ra,C</td>
<td>add upper half-word</td>
</tr>
<tr>
<td>addlo</td>
<td>Rd,Ra,C</td>
<td>add lower half-word</td>
</tr>
<tr>
<td>addi</td>
<td>Rd,Ra,C</td>
<td>add immediate short word</td>
</tr>
<tr>
<td>ceq</td>
<td>Rd,Ra,C</td>
<td>compare equal immediate</td>
</tr>
<tr>
<td>cmi</td>
<td>Rd,Ra,C</td>
<td>compare immediate, result minus</td>
</tr>
<tr>
<td>clt</td>
<td>Rd,Ra,C</td>
<td>compare less immediate</td>
</tr>
<tr>
<td>cle</td>
<td>Rd,Ra,C</td>
<td>compare less or equal immediate</td>
</tr>
<tr>
<td>cne</td>
<td>Rd,Ra,C</td>
<td>compare not equal immediate</td>
</tr>
<tr>
<td>cpl</td>
<td>Rd,Ra,C</td>
<td>compare immediate, result plus</td>
</tr>
<tr>
<td>cge</td>
<td>Rd,Ra,C</td>
<td>compare greater or equal immediate</td>
</tr>
<tr>
<td>cgt</td>
<td>Rd,Ra,C</td>
<td>compare greater immediate</td>
</tr>
<tr>
<td>ldb</td>
<td>Rd,Ra,C</td>
<td>load cached byte</td>
</tr>
<tr>
<td>lds</td>
<td>Rd,Ra,C</td>
<td>load cached short word</td>
</tr>
<tr>
<td>ldw</td>
<td>Rd,Ra,C</td>
<td>load cached word</td>
</tr>
<tr>
<td>ldl</td>
<td>Rd,Ra,C</td>
<td>load cached long word</td>
</tr>
<tr>
<td>stb</td>
<td>Rb,Ra,C</td>
<td>store cached byte</td>
</tr>
<tr>
<td>sts</td>
<td>Rb,Ra,C</td>
<td>store cached short word</td>
</tr>
<tr>
<td>stw</td>
<td>Rb,Ra,C</td>
<td>store cached word</td>
</tr>
<tr>
<td>stl</td>
<td>Rb,Ra,C</td>
<td>store cached long word</td>
</tr>
<tr>
<td>isld</td>
<td>Rd,Ra,C</td>
<td>I-structure load</td>
</tr>
<tr>
<td>isst</td>
<td>Rb,Ra,C</td>
<td>I/M-structure store</td>
</tr>
<tr>
<td>iswt</td>
<td>Rd,Ra,C</td>
<td>M-structure load</td>
</tr>
<tr>
<td>ifl</td>
<td>Ra,C</td>
<td>flush instruction cache line</td>
</tr>
<tr>
<td>dfl</td>
<td>Ra,C</td>
<td>flush data cache line</td>
</tr>
<tr>
<td>ldp</td>
<td>Rd,Ra,C</td>
<td>load pipelined (uncached) long word</td>
</tr>
<tr>
<td>stlp</td>
<td>Rb,Ra,C</td>
<td>store pipelined (uncached) long word</td>
</tr>
</tbody>
</table>
## A.2.2 Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>A</td>
<td>fork a thread</td>
</tr>
<tr>
<td>forkr</td>
<td>Ra</td>
<td>fork a thread indirect</td>
</tr>
<tr>
<td>forkz</td>
<td>Ra,A</td>
<td>fork a thread if register word zero</td>
</tr>
<tr>
<td>forknz</td>
<td>Ra,A</td>
<td>fork a thread if register word not zero</td>
</tr>
<tr>
<td>term</td>
<td></td>
<td>terminate thread</td>
</tr>
<tr>
<td>jal</td>
<td>Ra,Rd</td>
<td>jump register indirect and link</td>
</tr>
<tr>
<td>bal</td>
<td>A</td>
<td>branch and link</td>
</tr>
<tr>
<td>trap</td>
<td></td>
<td>trap to supervisor</td>
</tr>
<tr>
<td>rfi</td>
<td></td>
<td>return from interrupt</td>
</tr>
<tr>
<td>br</td>
<td>A</td>
<td>branch unconditionally</td>
</tr>
<tr>
<td>brz</td>
<td>Ra,A</td>
<td>branch if register word zero</td>
</tr>
<tr>
<td>bmnz</td>
<td>Ra,A</td>
<td>branch if register word not zero</td>
</tr>
<tr>
<td>brlz</td>
<td>Ra,A</td>
<td>branch if register word neg. or zero</td>
</tr>
<tr>
<td>brgz</td>
<td>Ra,A</td>
<td>branch if register word strictly positive</td>
</tr>
<tr>
<td>mcz</td>
<td>Rd,Ra,Rb</td>
<td>move if register word zero</td>
</tr>
<tr>
<td>mcnz</td>
<td>Rd,Ra,Rb</td>
<td>move if register word not zero</td>
</tr>
</tbody>
</table>

## A.2.3 Operational Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap</td>
<td>Rd,Ra</td>
<td>swap long word halves</td>
</tr>
<tr>
<td>rvsw</td>
<td>Rd,Ra</td>
<td>bit reverse word</td>
</tr>
<tr>
<td>or</td>
<td>Rd,Ra,Rb</td>
<td>logical OR long words</td>
</tr>
<tr>
<td>nor</td>
<td>Rd,Ra,Rb</td>
<td>logical NOR long words</td>
</tr>
<tr>
<td>and</td>
<td>Rd,Ra,Rb</td>
<td>logical AND long words</td>
</tr>
<tr>
<td>xor</td>
<td>Rd,Ra,Rb</td>
<td>logical XOR long words</td>
</tr>
<tr>
<td>mvub</td>
<td>Rd,Ra</td>
<td>move unsigned byte</td>
</tr>
<tr>
<td>mvus</td>
<td>Rd,Ra</td>
<td>move unsigned short word</td>
</tr>
<tr>
<td>mvuw</td>
<td>Rd,Ra</td>
<td>move unsigned word</td>
</tr>
<tr>
<td>mvul</td>
<td>Rd,Ra</td>
<td>move unsigned long word</td>
</tr>
<tr>
<td>mvsb</td>
<td>Rd,Ra</td>
<td>move sign-extended byte</td>
</tr>
<tr>
<td>mvss</td>
<td>Rd,Ra</td>
<td>move sign-extended short word</td>
</tr>
<tr>
<td>mvsbw</td>
<td>Rd,Ra</td>
<td>move sign-extended word</td>
</tr>
<tr>
<td>mvssl</td>
<td>Rd,Ra</td>
<td>move sign-extended long word</td>
</tr>
<tr>
<td>subq</td>
<td>Rd,Ra,Rb</td>
<td>subtract short word quick</td>
</tr>
<tr>
<td>subw</td>
<td>Rd,Ra,Rb</td>
<td>subtract word</td>
</tr>
<tr>
<td>subws</td>
<td>Rd,Ra,Rb,X</td>
<td>subtract word scaled</td>
</tr>
<tr>
<td>subl</td>
<td>Rd,Ra,Rb</td>
<td>subtract long word</td>
</tr>
</tbody>
</table>
Appendix A  Concurro Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>add short word quick</td>
</tr>
<tr>
<td>addw</td>
<td>add word</td>
</tr>
<tr>
<td>addws</td>
<td>add word scaled</td>
</tr>
<tr>
<td>addl</td>
<td>add long word</td>
</tr>
<tr>
<td>cmpq</td>
<td>compare short word quick</td>
</tr>
<tr>
<td>cmpw</td>
<td>compare word</td>
</tr>
<tr>
<td>cmpl</td>
<td>compare long word</td>
</tr>
<tr>
<td>tstq</td>
<td>test short word quick sum</td>
</tr>
<tr>
<td>tstw</td>
<td>test word sum</td>
</tr>
<tr>
<td>tstl</td>
<td>test long word sum</td>
</tr>
<tr>
<td>ldbi</td>
<td>load cached byte indexed</td>
</tr>
<tr>
<td>ldsi</td>
<td>load cached short word indexed</td>
</tr>
<tr>
<td>ldwi</td>
<td>load cached word indexed</td>
</tr>
<tr>
<td>ldli</td>
<td>load cached long word indexed</td>
</tr>
<tr>
<td>stbi</td>
<td>store cached byte indexed</td>
</tr>
<tr>
<td>stsi</td>
<td>store cached short word indexed</td>
</tr>
<tr>
<td>stwi</td>
<td>store cached word indexed</td>
</tr>
<tr>
<td>stli</td>
<td>store cached long word indexed</td>
</tr>
<tr>
<td>islbi</td>
<td>I/structure load indexed</td>
</tr>
<tr>
<td>islci</td>
<td>I/M-structure store indexed</td>
</tr>
<tr>
<td>iswti</td>
<td>M-structure load indexed</td>
</tr>
<tr>
<td>ifli</td>
<td>instruction cache flush indexed</td>
</tr>
<tr>
<td>dfli</td>
<td>data cache flush indexed</td>
</tr>
<tr>
<td>ldipi</td>
<td>load pipelined long word indexed</td>
</tr>
<tr>
<td>stlpi</td>
<td>store pipelined long word indexed</td>
</tr>
<tr>
<td>get</td>
<td>read channel</td>
</tr>
<tr>
<td>getp</td>
<td>read parent channel</td>
</tr>
<tr>
<td>sget</td>
<td>read synchronous channel</td>
</tr>
<tr>
<td>sgetp</td>
<td>read synchronous parent channel</td>
</tr>
<tr>
<td>put</td>
<td>write channel</td>
</tr>
<tr>
<td>putp</td>
<td>write parent channel</td>
</tr>
<tr>
<td>sput</td>
<td>write synchronous channel</td>
</tr>
<tr>
<td>sputp</td>
<td>write synchronous parent channel</td>
</tr>
<tr>
<td>wt</td>
<td>wait for channel to empty</td>
</tr>
<tr>
<td>wtp</td>
<td>wait for parent channel to empty</td>
</tr>
<tr>
<td>clr</td>
<td>clear channel</td>
</tr>
<tr>
<td>clrp</td>
<td>clear parent channel</td>
</tr>
<tr>
<td>new</td>
<td>create a new channel space</td>
</tr>
<tr>
<td>ldsr</td>
<td>load system register (restricted)</td>
</tr>
<tr>
<td>stsr</td>
<td>store system register (restricted)</td>
</tr>
<tr>
<td>exsr</td>
<td>exchange system register (restricted)</td>
</tr>
<tr>
<td>shl</td>
<td>shift long word left logically</td>
</tr>
<tr>
<td>shr</td>
<td>shift long word right logically</td>
</tr>
<tr>
<td>sha</td>
<td>shift long word right arithmetically</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Arguments</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>lb</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>ls</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>lw</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>li</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>lp</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>sb</td>
<td>Rb,Ra,C</td>
</tr>
<tr>
<td>ss</td>
<td>Rb,Ra,C</td>
</tr>
<tr>
<td>sw</td>
<td>Rb,Ra,C</td>
</tr>
<tr>
<td>sl</td>
<td>Rb,Ra,C</td>
</tr>
<tr>
<td>lc</td>
<td>Rd,C</td>
</tr>
<tr>
<td>andc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>orc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>xorc</td>
<td>Rd,Ra,C</td>
</tr>
</tbody>
</table>

**A.2.4 Extended Instructions**

The following instructions are implemented as built-in macros of the *cass* assembler. Extended instructions expand to small sequences of instructions from the basic instruction set (listed above). The length of an extended instruction can be variable, depending on the operands given, but most instructions occupy a single word. These instructions support full 32-bit constants or addresses, with the assembler optimizing instruction selection in special cases.
### Appendix A Concurro Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp</td>
<td>Rb,Ra,C</td>
</tr>
<tr>
<td>isl</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>sem</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>iss</td>
<td>Rb,Ra,C</td>
</tr>
<tr>
<td>lbm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>lsm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>lwm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>llm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>lpm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>sbm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>ssm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>swm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>slm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>spm</td>
<td>Ra,C,R1,R2,...</td>
</tr>
<tr>
<td>addc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>subc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>subcr</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>mulc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>divc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>divcr</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>modc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>modcr</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>divuc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>divucr</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>moduc</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>moducr</td>
<td>Rd,Ra,C</td>
</tr>
<tr>
<td>inc</td>
<td>Rd [,C]</td>
</tr>
<tr>
<td>dec</td>
<td>Rd [,C]</td>
</tr>
<tr>
<td>com</td>
<td>Rd,Ra</td>
</tr>
<tr>
<td>negw</td>
<td>Rd,Ra</td>
</tr>
<tr>
<td>negl</td>
<td>Rd,Ra</td>
</tr>
<tr>
<td>absbw</td>
<td>Rd,Ra</td>
</tr>
<tr>
<td>absl</td>
<td>Rd,Ra</td>
</tr>
<tr>
<td>jmp</td>
<td>A</td>
</tr>
<tr>
<td>call</td>
<td>A</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>blt</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>ble</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>blo</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bls</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>bne</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bge</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bgt</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bhs</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bhi</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>beqc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bltc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>blec</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bloc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>blsc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bnc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bgc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bgtc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bhsc</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>bhic</td>
<td>Ra,C,A</td>
</tr>
<tr>
<td>beqf</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bltf</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bles</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>blos</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>blesf</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bnef</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bgef</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bgtf</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bhscf</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bhif</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bnnf</td>
<td>Ra,Rb,A</td>
</tr>
<tr>
<td>bvnf</td>
<td>Ra,Rb,A</td>
</tr>
</tbody>
</table>
Appendix B

Synchronization Controller Microprogram

The microcode of the synchronization controller of the load/store unit is listed here in pseudo-code form. The actual microcode has been unpacked and replicated somewhat to render the logic of the code more clearly.

B.1 Definitions

In Section B.2, bold keywords represent micro-operations of the sequencer, while other symbols are constants and variable identifiers.

The microcode operations are:

- **case n of (label: (statement)) endcase**
  Branch to label associated with n.
- **halt**
  Stop the microsequencer and stand-by.
- **if x then (statement1) [else (statement2)] endif**
  If x is non-zero, execute first statements, otherwise execute second statements.
- **while x do (statement) endwhile**
  Iterate while x is non-zero.
- **store x at a**
  Store long word x at cache location a.
- **load x from a**
  Load long word x from cache location a.
- **return x to t**
  Send long word x to destination given by tag t.
- **a := b**
  Assign b to a.
- **(a, b)**
  Form a tuple of a and b in a 64-bit long word.
- **kind(x)**
  Returns the structure state field from x.
- **pointer(x)**
  Returns the next node pointer field from x.
- **tag(x)**
  Returns the destination tag field from x.
Constants in the microprogram have the following meanings:

- **I-structure_load**: Operation code for I-structure loads.
- **M-structure_load**: Operation code for M-structure loads.
- **structure_store**: Operation code for structure stores.
- **null**: Nil pointer.
- **empty**: Structure state: empty.
- **single_I_load**: Structure state: single deferred I-load.
- **multiple_I_load**: Structure state: multiple deferred I-loads.
- **single_M_load**: Structure state: single deferred M-load.
- **multiple_M_load**: Structure state: multiple deferred M-loads.
- **full**: Structure state: valid datum.

The following variables are set independently of the microsequencer:

- **operation**: Operation code sent by the load/store unit.
- **location**: Address of structure (initially read from).
- **cache_return**: Long word returned by the cache from location.
- **given_tag**: Destination tag sent by the load/store unit.
- **given_datum**: Datum sent by the load/store unit for stores.
- **free_list**: Pointer to first node of free list (system register).

Other variables are registers set as needed.

## B.2 Program

Upon detection of a structure operation, the microsequencer is activated to execute the operation and respond to the initial cache return:

```plaintext
case operation of
  I-structure_load:
    case kind(cache_return) of
      empty:
        store (single_I_load, given_tag) at location
      halt
    single_I_load:
      store (multiple_I_load, free_list) at location
      load new_list from free_list
      new_node := (given_tag, pointer(new_list))
      store new_node at free_list
      end_node := (tag(cache_return), null)
      load free_list from pointer(new_list)
      store end_node at pointer(new_list)
      halt
```
multiple_I_load:
    store (multiple_I_load, free_list) at location
    load new_list from free_list
    new_node := (given_tag, pointer(cache_return))
    store new_node at free_list
    free_list := pointer(new_list)
    halt
full:
    return cache_return to given_tag
    halt
endcase
M-structure_load:
    case kind(cache_return) of
    empty:
        store (single_M_load, given_tag) at location
        halt
    single_M_load:
        store (multiple_M_load, free_list) at location
        load new_list from free_list
        new_node := (given_tag, pointer(new_list))
        store new_node at free_list
        end_node := (tag(cache_return), null)
        load free_list from pointer(new_list)
        store end_node at pointer(new_list)
    halt
    multiple_M_load:
        store (multiple_M_load, free_list) at location
        load new_list from free_list
        new_node := (given_tag, pointer(cache_return))
        store new_node at free_list
        free_list := pointer(new_list)
    full:
        return cache_return to given_tag
        store (empty, null) at location
        halt
endcase
structure_store:
    case kind(cache_return) of
    empty:
        store given_datum at location
        halt
    single_I_load:
        return given_datum to tag(cache_return)
        store given_datum at location
        halt

Appendix B  Synchronization Controller Microprogram

single_M_load:
    return given_datum to tag(cache_return)
    store (empty, null) at location
    halt
multiple_I_load:
    if pointer(cache_return) then
        store given_datum at location
        node_pointer := pointer(cache_return)
        load next_node from node_pointer
        return given_datum to tag(next_node)
        while pointer(next_node) do
            node_pointer := pointer(next_node)
            load next_node from node_pointer
            return given_datum to tag(next_node)
        endwhile
        store free_list at node_pointer
        free_list := pointer(cache_return)
    else
        store given_datum at location
    endif
    halt
multiple_M_load:
    if pointer(cache_return) then
        load first_node from pointer(cache_return)
        return given_datum to tag(first_node)
        if pointer(first_node) then
            store (multiple_M_load, pointer(first_node)) at location
        else
            store (empty, null) at location
        endif
        store free_list at pointer(cache_return)
        free_list := pointer(cache_return)
    else
        store given_datum at location
    endif
    halt
full:
    store given_datum at location
    halt
endcase
endcase
Appendix C

Processor Configurations

The values of configuration parameters used for the three standard models of Concurro are listed here in full. The notes accompanying the table expand on the derivation of scalable parameters. Configurations are scaled solely in accordance with their group counts.

C.1 Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Model 208</th>
<th>Model 416</th>
<th>Model 832</th>
<th>Units</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contexts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>groups</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>group size</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>contexts</td>
</tr>
<tr>
<td>decode window</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>instructions</td>
</tr>
<tr>
<td>separate read</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>barrel switch</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main memory system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access latency</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>cycles</td>
<td></td>
</tr>
<tr>
<td>bus freq. divide</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secondary instruction cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>port width</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>bytes</td>
<td></td>
</tr>
<tr>
<td>cache size</td>
<td>256 k</td>
<td>256 k</td>
<td>256 k</td>
<td>bytes</td>
<td></td>
</tr>
<tr>
<td>line size</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>bytes</td>
<td></td>
</tr>
<tr>
<td>access latency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>cycles</td>
<td></td>
</tr>
<tr>
<td>prefetch</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary instruction cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cache size</td>
<td>16 k</td>
<td>16 k</td>
<td>16 k</td>
<td>bytes</td>
<td></td>
</tr>
<tr>
<td>line size</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>bytes</td>
<td>1</td>
</tr>
<tr>
<td>associativity</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>lines/set</td>
<td>2</td>
</tr>
<tr>
<td>preaccess depth</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>words</td>
<td>2</td>
</tr>
<tr>
<td>cross lines</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C  Processor Configurations

<table>
<thead>
<tr>
<th>Allocation Policy</th>
<th>Data Cache</th>
<th>System FU</th>
<th>Logical FU</th>
<th>Integer FU</th>
<th>Shift FU</th>
<th>Load/Store FU</th>
<th>Floating Point Add FU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
</tr>
<tr>
<td><strong>Non-Block</strong></td>
<td><strong>2 misses</strong></td>
<td><strong>2 misses</strong></td>
<td><strong>2 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td><strong>32 k bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>2 lines/set</strong></td>
<td><strong>4 lines/set</strong></td>
<td><strong>2 lines/set</strong></td>
<td><strong>2 lines/set</strong></td>
</tr>
<tr>
<td><strong>Line Size</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
<td><strong>64 bytes</strong></td>
</tr>
<tr>
<td><strong>Assocativity</strong></td>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td><strong>Allocation Policy</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
<td><strong>LRU</strong></td>
</tr>
<tr>
<td><strong>Non-Block</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
<td><strong>4 misses</strong></td>
</tr>
<tr>
<td><strong>Dual-Ported</strong></td>
<td><strong>No</strong></td>
<td><strong>No</strong></td>
<td><strong>Yes</strong></td>
<td><strong>No</strong></td>
<td><strong>Yes</strong></td>
<td><strong>No</strong></td>
<td><strong>No</strong></td>
</tr>
<tr>
<td><strong>Array Banks</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
</tr>
<tr>
<td><strong>Phys. Units</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Operand Queue</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Result Queue</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Instruction Latency</strong></td>
<td><strong>5</strong></td>
<td><strong>5</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>5</strong></td>
<td><strong>5</strong></td>
</tr>
<tr>
<td><strong>Result Latency</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
<td><strong>6</strong></td>
</tr>
<tr>
<td><strong>Cycle Latency</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Load Ports</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Operand Queue</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Result Queue</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Address Delay</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Load Queue</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
<td><strong>26</strong></td>
</tr>
<tr>
<td><strong>Entry Count</strong></td>
<td><strong>10</strong></td>
<td><strong>10</strong></td>
<td><strong>10</strong></td>
<td><strong>10</strong></td>
<td><strong>10</strong></td>
<td><strong>10</strong></td>
<td><strong>10</strong></td>
</tr>
<tr>
<td><strong>F.P. Add Latency</strong></td>
<td><strong>4 cycles</strong></td>
<td><strong>4 cycles</strong></td>
<td><strong>4 cycles</strong></td>
<td><strong>4 cycles</strong></td>
<td><strong>4 cycles</strong></td>
<td><strong>4 cycles</strong></td>
<td><strong>4 cycles</strong></td>
</tr>
<tr>
<td><strong>Int. to F.P. Latency</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
</tr>
<tr>
<td><strong>F.P. to Int. Latency</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
<td><strong>3 cycles</strong></td>
</tr>
</tbody>
</table>
Appendix C  Processor Configurations

Multiply FU

<table>
<thead>
<tr>
<th>Physical units</th>
<th>Operand queue</th>
<th>Result queue</th>
<th>Integer latency</th>
<th>F.P. latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

Divide FU

<table>
<thead>
<tr>
<th>Physical units</th>
<th>Operand queue</th>
<th>Result queue</th>
<th>Word latency</th>
<th>Long latency</th>
<th>F.P. latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>

C.1.1 Notes for Model Parameters

Note 1
Instruction cache line size = \( \lceil (\text{pre-access depth}) / 8 \rceil \times 8 \times 4 \) bytes. To permit a maximum cache return equal to the pre-access depth, the minimum line size cannot fall below \((\text{pre-access depth}) - 1\) words. Allocations occur in multiples of 8 words.

Note 2
Pre-access depth = \( \lceil \text{groups} / 0.8 \rceil \) instruction words. This allows for 20% instruction wastage due to branching.

Note 3
If \( \text{groups} > 4 \), dual-porting of the data cache is enabled, otherwise the cache is single-ported. This measure provides cache bandwidth in proportion to the frequency of load and store instructions (see Note 9).

Note 4
Number of physical functional units for logic operations = \( \lceil \text{groups} \times 0.5 \rceil \), assuming a 50% relative frequency of logical instructions.

Note 5
Length of operand queue = \( \lceil \text{groups} / \text{(number of physical FUs)} \rceil \) entries. This formula is biased towards highly shared FUs, which are more prone to blocking under bursty instruction traffic than highly duplicated FUs are.
Note 6
Result queue length = \( \lceil \frac{(\text{number of physical FUs})}{2} \rceil \) entries. For the load/store unit the number of physical FUs is replaced by the total number of main memory and data cache ports. More numerous FUs are favoured by this formula to account for possible disparities in result and write back bandwidths.

Note 7
Number of physical functional units for integer operations = \( \lceil \text{groups} \times 0.5 \rceil \), assuming a 50% relative frequency of integer add, subtract, compare, and test instructions.

Note 8
Number of physical functional units for shift operations = \( \lceil \text{groups} \times 0.25 \rceil \), assuming a 25% relative frequency of shift instructions.

Note 9
Number of direct ports to the main memory system = \( \lceil \text{groups} \times 0.25 \rceil \), assuming a 25% relative frequency of uncached load and store instructions.

Note 10
Length of pending loads queue = (main memory access latency) + 1 entries to allow full overlap of uncached loads in the worst case activity.

Note 11
Number of physical functional units for floating point add operations = \( \lceil \text{groups} \times 0.25 \rceil \), assuming a 25% relative frequency of floating point add and convert instructions.

Note 12
Number of physical functional units for multiply operations = \( \lceil \text{groups} \times 0.25 \rceil \), assuming a 25% relative frequency of floating point and integer multiply instructions.

Note 13
Integer multiply latency is for long word operands; word operands are pre-formatted to use the same hardware.

Note 14
Number of physical functional units for divide operations = \( \lceil \text{groups} \times 0.1 \rceil \), assuming a 10% relative frequency of floating point divide and integer divide/modulus instructions.
Bibliography


