DESIGN AND EVALUATION OF DIRECTORY-BASED CACHE COHERENCE SYSTEMS

by

Brian Walter O'Krafka

Memorandum No. UCB/ERL M92/4

6 January 1992

DESIGN AND EVALUATION OF DIRECTORY-BASED CACHE COHERENCE SYSTEMS

by

Brian Walter O'Krafka

Memorandum No. UCB/ERL M92/4

6 January 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

DESIGN AND EVALUATION OF DIRECTORY-BASED CACHE COHERENCE SYSTEMS

by

Brian Walter O'Krafka

Memorandum No. UCB/ERL M92/4

6 January 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Design and Evaluation of Directory-Based Cache Coherence Systems

Brian Walter O'Krafka

Department of Electrical Engineering and Computer Sciences,

University of California, Berkeley.

Abstract

A cache coherence scheme is a critical part of a shared memory multiprocessor because it relieves the programmer of the burden of moving shared data among local and remote memories. In this dissertation several new techniques are described for implementing and evaluating the performance of cache-coherent multiprocessors. The first contribution of this work is a generalization of shared bus stack simulation techniques that supports directory-based cache coherence schemes. This is desirable because stack simulation permits the evaluation of multiple cache sizes in a single simulation run. Results are presented for three benchmark programs, three directory methods, and multiple cache, block and multiprocessor sizes. The results quantify the tradeoffs between network traffic and miss ratio that are possible by varying the number of updates in a competitive directory scheme. These results extend previous studies of shared bus architectures by accounting for point-to-point network traffic and larger numbers of processors. A second contribution is an approximation technique for analyzing interconnection networks as open, acyclic networks of finite queues. The technique combines the algorithms used in the Bell Laboratories Queueing Network Analyzer with a known algorithm for approximating the effect of finite buffers. The combined algorithm permits analysis of queueing networks with hundreds of queues and is applicable to a broad class of interconnection network, including hypercubes, meshes, tori and Delta networks. Using traffic estimates from cache simulations, this analysis technique is applied to a number of alternative networks. Good cache and network performance requires good synchronization support. The latter part of this dissertation describes several efficient implementations of fetch&op synchronization primitives. The implementations are suitable for hardware or software, and can be easily modified to support multiprefix operations.

A. Richard Newton

Professor A. Richard Newton

Dissertation Committee Chairman

Acknowledgements

Writing this dissertation has been an intensely arduous yet intensely rewarding experience. Many of the rewards have been due to interactions with some extremely talented people. My advisor, Professor Richard Newton, gave me the resources and motivation to study multiprocessor memory systems. Richard's advisorship has taught me a lot about independent research and creative thought, for which I am thankful. The other members of my dissertation committee, Professor Abhiram Ranade and Professor Ronald Wolff, provided valuable comments that improved this work. Their participation is gratefully acknowledged. I also received technical support from several students in the Computer Science Division. Mike Carlton was a source of good ideas about cache coherence and simulation. M. T. Raghunath gave me an education in multiprocessor networks. Bob Boothe provided help with benchmark validation. The help of all three is greatly appreciated.

Others provided moral support and friendship. Mark Beardslee, Brian Lee and Gregg Whitcomb endured my griping during many lunch hours and coffee breaks. Wendell Baker, Abhijit Ghosh and Chuck Kring relieved the monotony of focussed research by participating in numerous discussions on cold fusion, Republican politics and other diverse topics. I am deeply grateful for the opportunity to work with these fellow students.

This dissertation would not have been completed without the support of my family. The energy and enthusiasm of my wife, Audrey, were frequent sources of encouragement. My daughters Anne and Catherine added an extra dimension to my life that has relieved much of the tension of graduate study. Their arrival into our family will be the most memorable part of my experiences at Berkeley. Anne, Catherine, Audrey and I all received a great deal of support from our parents in Canada, for which we are grateful.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, Digital Equipment Corporation, and the Defense Advanced Research Projects Agency (under contract N00039-C-87-0182). Their support is gratefully acknowledged.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Goal: Cost-Effective Acceleration of Engineering Applications

In this dissertation techniques are described for the design and evaluation of directory-based cache coherence schemes for large, shared memory multiprocessors. "Large" here implies a machine composed of several hundred uniprocessors, each with the power of a state-of-the-art workstation. The main goal is to show that directory-based coherence strategies are feasible and desirable for machines of this scale, and that their implementation complexity is modest.

Although impressive gains continue to be made in uniprocessor performance, multiprocessors remain an important research area for many reasons. One reason is that faster computers enable new problems to be solved and more design options to be explored. Another reason is that the computing requirements of many engineering problems continue to grow at least as fast as the number of compute cycles that become available. As an example, advances in integrated circuit densities demand computer aided design (CAD) tools that support design problems that grow at the same rate as uniprocessor MIPS ratings. Faster CAD programs also reduce the time to get a product to market, which is crucial for many products.

In this chapter, the state of multiprocessor design is reviewed, with a focus on open problems in scalable cache coherence. It begins with a broad review of the main components of a multiprocessor system: programming paradigms and architectures. The review shows why shared memory is a desirable architectural abstraction for a variety of programming models. Unfortunately, ideal shared memories with unlimited concurrent access must be approximated using collections of single-ported memories interconnected by a network. It is therefore attractive to provide a cache memory at each processor to avoid costly data transfers across the network. If multiple cached copies of shared-writeable data are permitted, some mechanism must ensure that the processors see a coherent view of memory: this is the cache coherence problem. An introduction to multiprocessor cache coherence is provided in Section 1.3, with an emphasis on issues that are not well understood. The specific problems addressed in this work are listed in the final section, along with a summary of the contributions and an outline of the dissertation.

1.2 Multiprocessor Systems

A multiprocessor system is a layered set of abstract machines, with higher level abstractions layered above lower level abstractions (Figure 1.1). The highest level abstractions are im-

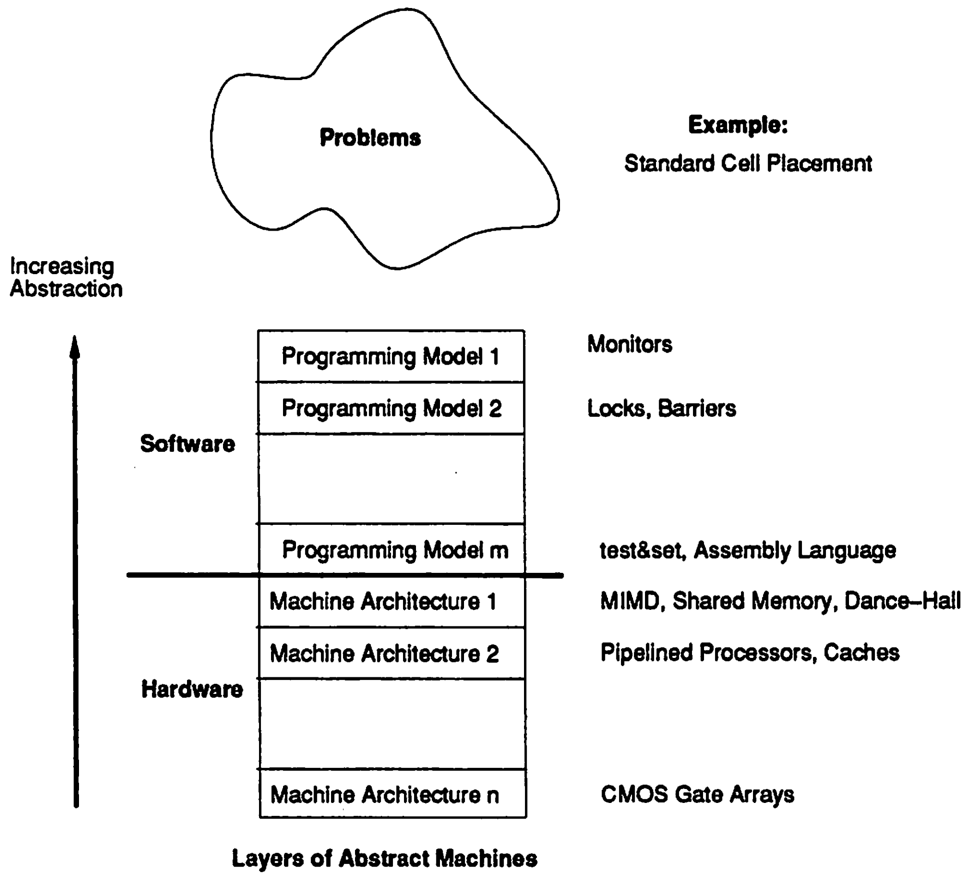


Figure 1.1: Components of a Multiprocessor System

plemented in software, and provide programming models with which a programmer can solve a problem. The lower levels are hardware implementations of more primitive parallel machines; the topmost architecture furnishes a platform on which higher level programming models are built. A typical set of layers would be a monitor-based programming language [AS83] implemented on a cache-coherent "dance-hall" architecture (processors and memories on opposite sides of a multistage network) implemented in CMOS using pipelined network elements. The role of a multiprocessor architect is to devise machine architectures that are sufficiently abstract to simplify implementation of the software layers, and sufficiently simple to permit a fast, cost-effective implementation. The state of multiprocessor design is reviewed in this section by examining programming paradigms and machine architectures in greater detail. The purpose of the review is to show that cache coherent, shared memory architectures make good platforms for implementing a wide variety of programming models.

1.2.1 Programming Paradigms

Good programming paradigms simplify the task of writing parallel programs. The broad variety of computational problems has resulted in a broad variety of programming paradigms, including: functional and logic programming, data-parallel programming, parallel loops, shared memory, and message passing [AS83, BST89, CG89, Hud89, Sha89].

The *functional* and *logic programming* models belong to the larger class of *declarative* programming models, in which computations are described solely via single assignment expressions [Hud89]. Functional languages are based on function application, while logic programs are composed of relations. Prohibiting multiple assignments to variables is equivalent to prohibiting side effects, which makes declarative languages suitable for formal analysis. It also fosters a programming style that exposes more parallelism than imperative languages. In spite of these advantages the use of declarative languages has been minimal. This has been due to the lack of good compilers and the fact that these languages make a significant departure from traditional imperative languages. Recent improvements in compilation techniques and attempts at standardizing a general purpose functional language (Haskell [HW88]) may result in greater interest in this class of language in the future. Declarative programming models are the most abstract in that all notions of a multiprocessor implementation–multiple processes running on distinct processors, an interconnection network, distributed memory–are completely avoided.

In the *data parallel* (or *collection-oriented* [SB91]) model, parallelism is expressed as the application of operations to large collections of data. For example, an addition operation could be applied to a collection of pairs of numbers to add two vectors, or it could be applied in tree fashion to find the sum of a list of numbers. This computational model is surprisingly useful given its simplicity, and is common in many languages for massively parallel computers [SB91]. Collection-oriented constructs are typically embedded in existing languages such as C, FORTRAN, and Lisp. They are also a natural component of functional languages. Like the declarative model, data-parallel programming models abstract the most cumbersome implementation issues. The implementation of multiple processes on distinct processors is usually abstracted in a more restrictive way, however, by assuming a single flow of control (single instruction stream, multiple data stream (SIMD)).

Parallel loops is a programming paradigm that is often used in parallelizing scientific applications (usually written in FORTRAN) [D+88, A+88b]. In this model, loops with few dependencies among iterations are distributed among multiple processors for parallel execution. Typically, most or all of the parallelization is performed automatically, so the notions of multiple processes, interprocess communication, and distributed memory are abstracted.

The *shared memory* programming model provides the programmer with a set of processes that can issue reads and writes to a globally shared memory, such that any number of reads and writes to *distinct* addresses can be performed simultaneously [May90]. Variations in the way in which simultaneous reads and writes to the *same* address are resolved create a family of shared memory models. In the most restrictive model, *exclusive read/exclusive write* (EREW), simultaneous accesses to the same address are not allowed. In the least restrictive model, *concurrent read/concurrent write* (CRCW), the accesses are performed simultaneously, with the effect of the writes being the same as if they were serialized. Basic read and write primitives can be augmented with more sophisticated access instructions such as *fetch&op* [KRS86]. A fetch&op instruction with target address *addr* and value v causes the contents of *addr*, say x, to be replaced with x op v, and returns x. Simultaneous fetch&op instructions with the same target address are satisfied concurrently, with the results corresponding to some arbitrary serialization. Concurrent fetch&op instructions can be viewed as performing a data parallel operation in which a collection of numbers, the v's, are operated on and all partial results are collected. The shared memory model abstracts the implementation of a physical multiprocessor memory. Unlike the preceding programming models, multiple interacting processes are an explicit part of the shared memory paradigm.

The *message passing* programming model [Hoa78] is the least abstract model because it makes explicit the notion of a collection of distinct processes operating on distinct, distributed memories and interacting through a network. Like the data-parallel and shared memory models, message passing primitives are usually embedded in traditional imperative languages.

1.2.2 Architectures

At some level a programming model must be implemented on an abstract machine supported directly in hardware. Virtually all hardware architectures share the basic structure of Figure 1.2 or Figure 1.3: collections of serial processing elements and memories interconnected by a network. The most basic programming model supported by this hardware is the message passing model, in which processes running on different processing elements communicate by messages through the network. This basic architecture can be augmented with additional hardware to support higher level programming abstractions. Shared memory can be approximated with hardware that maps a global shared address space onto multiple physical memories. Data parallel computation can be supported by forcing all processing elements to execute a single instruction stream in lock-step [Hil85]. Data flow computation can be supported by adding complex associative memories [AN87, GKW85].

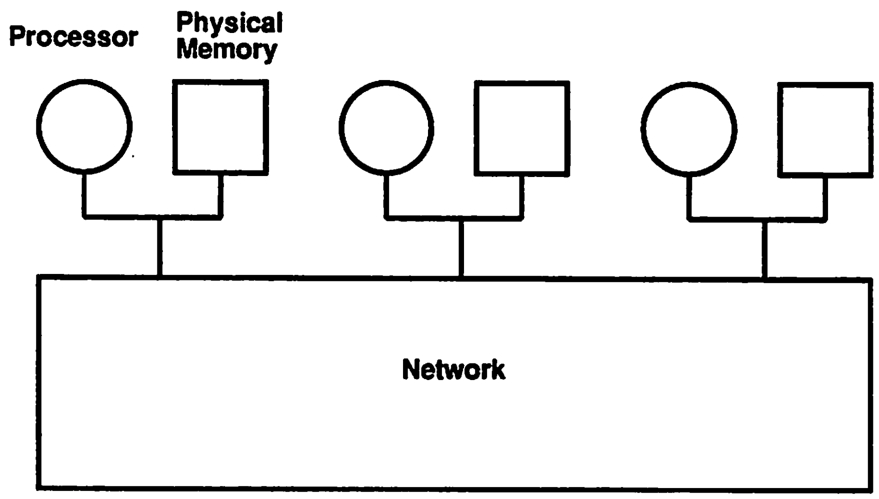


Figure 1.2: Basic Multiprocessor Organization: Distributed Main Memory

Since a particular programming paradigm is best-suited to a particular class of problem, it is desirable to support as many paradigms as possible on a single architecture. Very specialized hardware, however, is often useful for only a single programming model. The most obvious example is the complex hardware provided in traditional data flow computers. Data parallel architectures like the Connection Machine are an extreme example in which the hardware supports only a single instruction stream.

Specialized hardware is often not necessary for good performance. There is growing evidence that higher level programming models can be efficiently emulated on much simpler architec-

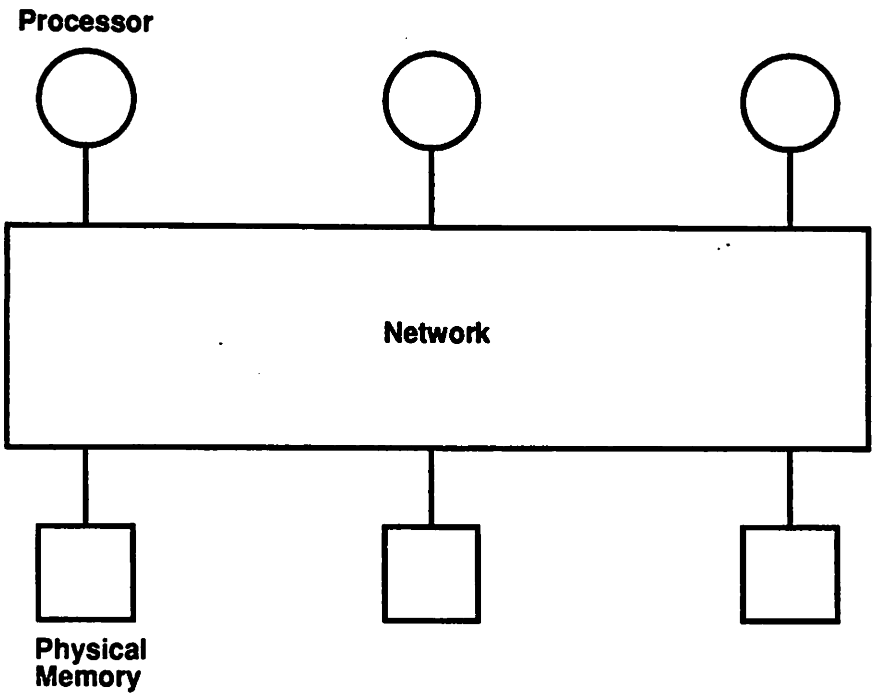


Figure 1.3: Basic Multiprocessor Organization: "Dance-Hall" Arrangement

tures. For example, reasonably efficient techniques have been published for compiling functional, logic programming, data parallel, and parallel loop computations onto shared memory and, in some cases, message passing architectures [Hud89, Sha89, SB91, H+91, D+88]. The Monsoon data-flow architecture is an interesting attempt to support data-flow computation at a much lower level than previous generation data-flow architectures (MIT Tagged Token Machine and Manchester Data-Flow Machine). Instead of providing complex token matching hardware, Monsoon efficiently supports a *multithreaded* abstract machine in which conventional uniprocessors are augmented with hardware to permit efficient context switches of very lightweight processes (or *threads*). Recent work suggests that even this may not be necessary [C+91].

It therefore seems unnecessary to go far beyond a shared memory or message passing architecture to support a broad variety of programming models. There are good reasons, however, to provide hardware support for shared memory. One is that the shared memory *programming model* is much less cumbersome than message passing for many applications. This includes the implementation of operating systems, debuggers, and higher level programming models. Another reason is that emulating shared memory on a message passing architecture has a high cost that cannot be avoided without hardware support. This is because multiple emulation instructions must be used for each shared access in the shared memory program, and many shared memory programs issue frequent shared accesses [DR$^+$87, EK88]. Furthermore, shared memory programs exhibiting good speedups (or, equivalently, high processor utilization) have few free cycles available for emulation.

There are many open research issues concerning the design of a shared memory for hundreds or thousands of processors. One of the most important is the provision of cache memories at each processor. A cache at each processor permits many memory accesses to be satisfied locally by buffering data as it is referenced. Unlike uniprocessor systems, however, a multiprocessor with caches requires a mechanism to ensure that multiple cached copies of the same data remain coherent. The design and evaluation of a particular class of coherence schemes is the topic of this dissertation. The following section reviews previous work in cache coherence in greater detail.

1.3 Multiprocessor Cache Coherence

There are numerous ways to ensure that multiple cached copies of data are consistent. In fact, there are numerous ways in which the notion of "coherent" can be interpreted. The most common interpretations are presented in the following section. Techniques for enforcing these coherence standards are reviewed in Section 1.3.2. The final part of this section summarizes previous work in evaluating multiprocessor cache performance. The discussions in these sections identify problems in directory-based cache coherence that are addressed in this dissertation.

1.3.1 Formal Notions of Cache Coherence

A set of multiprocessor caches is often defined to be coherent if *the value returned on a LOAD instruction is always the value written by the latest STORE instruction with the same address* [CF78]. This definition is unsatisfactory for two reasons. First, it is ambiguous with respect to the temporal occurrence of LOAD's and STORE's. When is a LOAD or a STORE determined to have taken place: when initiated by a processor, processed by a cache controller, or propagated to all cached copies? The second reason is that the definition is based on a notion of processors issuing and completing shared references in lock-step. This is overly restrictive because it requires all processors to observe the same interleaving of all references from all processors. This restrictiveness severely reduces the amount of pipelining and reordering that can be applied to shared references, with a subsequent loss in performance. To illustrate this, consider a processor that issues STORE A followed by STORE B (cpu 1 in Figure 1.4). Assume that STORE's are considered complete once they have been propagated to all copies. If the addressed memory locations reside in different memory banks on the other side of a complex network, they may be propagated to copies out of order if STORE A is delayed. If this occurs, it is possible for another processor to observe STORE A *before* the STORE B, which violates the definition. The most straightforward implementation of this definition, then, requires each processor reference to be completed at the main memory before a new reference can be issued.

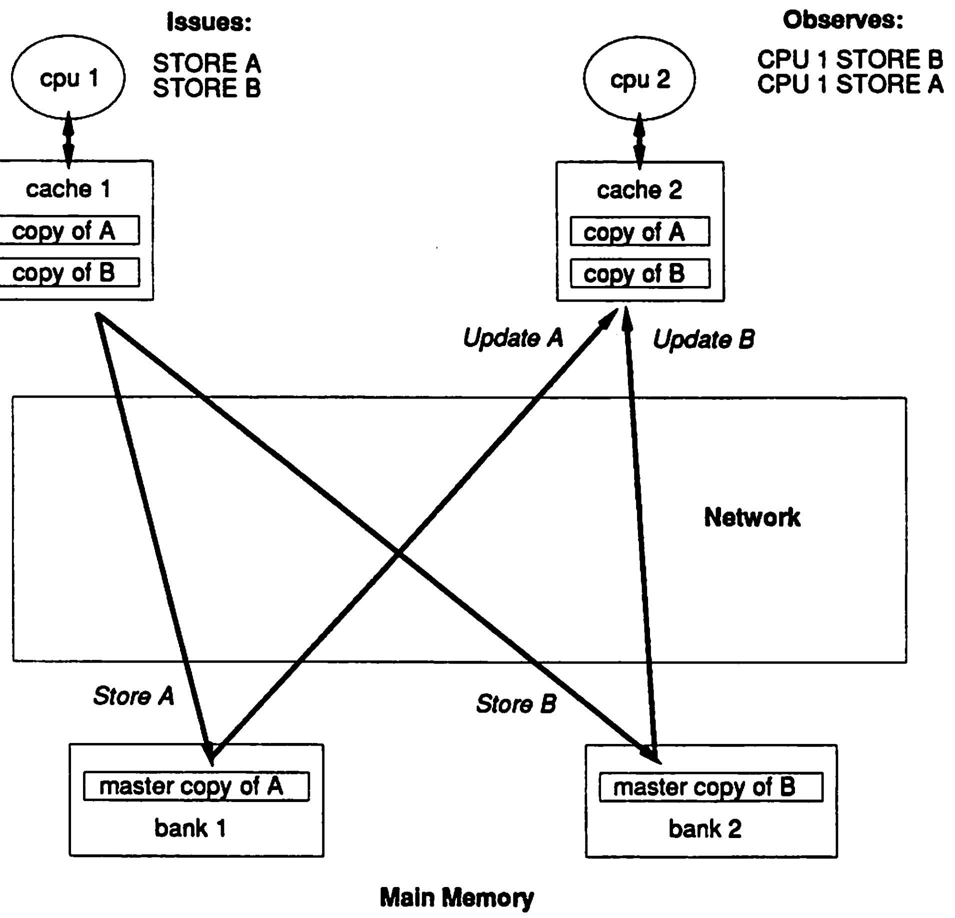


Figure 1.4: The Restrictiveness of Strong Ordering

The above definition of coherence is closely related to Lamport's *sequential consistency*. A multiprocessor is sequentially consistent if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by its program [Lam79]." The simplest way to implement sequential consistency is that described above, in which each processor issues LOAD's and STORE's in program order and ensures that all copies are updated (or invalidated) for each reference before issuing the next [SD87]. Accesses in a multiprocessor that satisfy these restrictions are said to be *strongly ordered.* More complex implementations of sequential consistency have been proposed [AH90, A$^+$89], but it is unclear whether significant improvements in performance are possible.

The restrictiveness of sequential consistency can be relaxed by using a weaker notion of coherence that exploits some knowledge about how synchronization is performed in the programming paradigms of interest. These weaker notions of coherency are, in decreasing restrictiveness:

1. *processor* consistency
2. *weak* consistency
3. *release* consistency

A multiprocessor is *processor consistent* if "the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program [Goo89]." This relaxes the requirement of strong ordering that the same *interleaving* of all processor references is observed by all processors. It exploits the fact most synchronization algorithms designed using sequential consistency still perform correctly under processor consistency. Programs can be constructed, however, that behave differently under sequential and processor consistency.

A multiprocessor is *weakly consistent* if [D$^+$86]:

1. accesses to global synchronizing variables are strongly ordered;
2. no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed;
3. no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

Synchronizing variables are variables that are used in synchronization operations, such as locks, barriers, test-and-set, and fetch-and-add. Weak consistency exploits the fact that most parallel programs do not perform synchronization using LOAD and STORE accesses to the shared memory, but use special synchronization primitives corresponding to the operations just listed. Access to mutually exclusive data can only be acquired by references to synchronization variables, so intervening references can be pipelined as long as they are not initiated until the last synchronization access completes, and as long as they are completed before the next synchronization access is initiated.

Release consistency is an optimization of weak consistency requiring a further classification of synchronization accesses into *acquire* accesses and *release* accesses. These are described in [GGH91] as follows:

> An acquire synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations. A release synchronization access (e.g., an unlock operation or a process setting a flag) grants this permission.

With this differentiation, it is not necessary to impose Condition 2 of weak consistency on an *acquire*, or Condition 3 on a *release*.

The relative performance of these consistency models was examined in [GGH91] using simulations of three parallel applications on a small number of processors (16). The study showed that all three weak consistency models provide substantial benefits over a strongly ordered system. Weak consistency and release consistency performed similarly, and provided performance only marginally better than processor consistency. These results are intuitive for weak and release consistency since synchronization references tend to be infrequent, so the time spent waiting for outstanding accesses to complete at synchronization points should be minimal.

1.3.2 Cache Coherence Strategies

In the previous section several definitions of coherence were described. In this section strategies for enforcing these standards are presented. The strategies fall into three classes:

1. *software* methods;

2. *snooping bus* protocols;

3. *directory* methods;

In *software* methods, a compiler manages the cache using special cache control instructions. The simplest class of software methods requires a programmer to explicitly identify shared-writeable data so that it will not be cached. The penalty for this simplicity is a significant increase in average memory access time and network traffic [ON90]. More sophisticated software methods use compile-time analysis to identify certain parallel programming constructs and insert cache management instructions. Typical instructions include cache-flush, selective invalidation, cache-bypass, and main memory update. Two types of programming construct are typically recognized: critical sections [Smi85] and parallel loops [CV90].

In critical section schemes, the compiler ensures that all data modified within a critical section is reflected in main memory when the critical section is left. This is typically done with cache invalidation or update instructions at every point where a process leaves a critical section. If invalidations are used, all modified cache blocks are written back to main memory.

Most work on software methods has focussed on parallel DO-loops, usually in the context of parallelizing FORTRAN compilers [CV90]. These schemes are similar to the critical section technique described above, except that coherence instructions are placed at the end of loops. The best loop and critical section schemes use sophisticated compiler analysis to perform selective invalidation [CV88, CKM88].

Software schemes have the advantage of requiring minimal hardware support. This is offset by the disadvantages of known techniques. The most significant is that many engineering applications are not easily coded in a FORTRAN-like programming model because they make extensive use of pointer-based data structures; for these applications parallelized loops are not applicable. The critical section schemes are more suitable, but they must be very conservative to deal with arbitrary pointer use. Furthermore, some of the more sophisticated software methods require substantial hardware support for selective invalidation [CV90]. Unfortunately, relatively little is known about the relative performance of software and hardware coherence techniques. Some comparisons have been reported in [OA89, MB90], but all of these focus on parallelized loops.

Snooping bus protocols exploit the broadcast capability of a shared bus, which enables each cache to efficiently monitor and disseminate shared access information. Many snooping bus protocols have been proposed and implemented; a good summary is in [AB86]. Unfortunately, the number of high performance processors that can reside on a bus is limited. Attempts have been made to extend snooping bus protocols to larger numbers of processors using collections of busses interconnected in hierarchies [Wil87, MA89, T+90], and multidimensional meshes [GW88]. These extensions have the disadvantages that they are network-specific and rely on broadcasting for some steps in their protocol. Some are also extremely complex. No results have been published comparing the performance of extended snooping bus schemes with alternative coherence strategies.

Directory-based coherence protocols rely on a (conceptually) centralized set of "bookkeeping" information, the directory, which maintains the status of all cached shared-writeable data. The directory maintains for each block of main memory a list of the caches with copies. Cache accesses that affect directory state and/or other caches are required to consult the directory to ensure that the directory and other cached copies remain consistent.

One of the earliest directory schemes, due to Censier and Feautrier [CF78], implemented the copy lists using bit vectors. With this organization, the directory can be interleaved with the main memory to get sufficient bandwidth. The main disadvantage of this scheme is the high cost of implementing bit vectors at each main memory block for a large number of processors. Several techniques for reducing the overhead have been proposed. Archibald and Baer [AB84] suggest the use of only a single cache identifier per main memory block, and the use of broadcasts when more than a single copy is required. Agarwal et. al. [A^+88a] generalize this to their Dir_iB and Dir_iNB schemes. In a Dir_iB scheme, i cache identifiers are provided for each block, and broadcasting is used when the number of copies exceeds i. The Dir_iNB scheme is similar to Dir_iB, except that instead of resorting to broadcasting, identifier memory is "recycled" by invalidating older copies. A number of techniques have recently been proposed for reducing directory overhead without the performance penalties of the Dir_iB and Dir_iNB schemes. These techniques, including one developed as part of this research, are described in Chapter 2.

Directory methods are attractive because they apply to a broad class of interconnection network, are conceptually straightforward, and do not require broadcasts. The main disadvantage of early directory methods, efficient implementation of the directory, can now be overcome with the techniques described in Chapter 2. A secondary disadvantage of directory schemes is that a directory controller can become a bottleneck if many copies of a block must be invalidated or updated on a shared write. There is a growing amount of empirical evidence, however, to suggest that this is not a problem in practice [WG89a, ON90, C^+90, SWG91]. Results in Chapter 3 confirm this.

1.3.3 Performance Evaluation of Cache Coherence Strategies

A cache coherence strategy is effective only if its overhead does not outweigh its benefit. Overhead in coherence schemes takes the form of extra network traffic, stolen cache cycles, stolen main memory cycles, and extra cache misses due to invalidations. The overhead of competing schemes can be estimated with several performance evaluation techniques: analytic models, simulation with synthetic workloads, and trace driven simulation.

Analytic models are desirable because of their simplicity, reduced computational requirements, and the insight they provide. Most analytic models of multiprocessor caches take a relatively small number of parameters representing the workload and simulated architecture, and produce estimates of maximum speedup and the frequencies of various coherence operations. Typical workload parameters include miss ratios for shared and local data, the fraction of references to shared data, the fraction of writes to shared data, and the average number of copies of shared data. Common architectural parameters include network path width, cache block size, and memory latencies. The structure of most analytic models is as follows:

1. Estimate the relative frequency of various network transactions (messages that are issued by a processor into the interconnection network).

2. Estimate the arrival rate of network traffic by assuming some network delay.

3. Use the arrival rates to estimate the actual network delay with some congestion model.

4. If the estimated network delay does not match the assumed network delay, return to Step 2, and repeat the process until convergence is reached.

The main differences in models are in the number of workload and architectural parameters, the algorithm used to estimate the frequency of transactions in Step 1, and the congestion model in Step 3. Tables 1.1 and 1.2 categorize some published analytic models according to these criteria.

There are several common techniques for estimating the frequency of various network transactions. The simplest is to simply guess some values, or derive them algebraically from more fundamental parameters. Another technique is to construct a stochastic model of program behavior and either solve it directly or simulate it. Both of these methods must be validated against real programs to verify their accuracy. The most common way to do this is trace-driven simulation. Traces can be acquired during an execution of a program on a particular parallel machine, or they can be generated during the cache simulation by simulating the processors.

There are many ways to estimate network congestion. The simplest is to use a known model for the network of interest. Published network models that could be used here are reviewed in Chapter 4. Unfortunately, most known models do not account for several important behavioral features. Network congestion can also be estimated with an "application specific" model. These are usually constructed as a network of queues or a stochastic Petri net, both of which can be solved using standard techniques [ABC86]. A third way to estimate network congestion is by simulation.

As the tables show, only a few models have been validated against real parallel programs. It is difficult to make believable cache design choices using models that have not been validated against real parallel programs, so most models in Tables 1.1 and 1.2 are of limited use. In particular, recent cache studies using reference traces from real parallel programs indicate that most early

Table 1.1: Analytic Models

Model	Network [a]	Determination of Transaction Frequency	Congestion Model	Notes
[V+88]	bus	algebraic	closed queueing network (MVA [b])	block on write, continuous time, not validated with traces
[VJS89]	hierarchical bus	algebraic	closed queueing network (MVA)	block on write, continuous time, not validated with traces
[VH86]	bus	algebraic	generalized timed Petri nets	block on write, continuous time, not validated with traces
[LV88a]	multicube	algebraic	closed queueing network (MVA)	continuous time, not validated with traces
[NP85]	multistage	algebraic	Kruskal/Snir [KS83], simulation	RP3 model [P+85], software coherence, not validated with traces
[OA89]	multistage, bus, crossbar	algebraic	open and closed queueing networks	continuous time, validation against 4 cpu traces

[a] N/A means not applicable.
[b] MVA denotes Mean Value Analysis [ABC86].

Table 1.2: Analytic Models (continued)

Model	Network [a]	Determination of Transaction Frequency	Congestion Model	Notes
[Dub85]	N/A	algebraic	N/A	not validated with traces
[DB82]	N/A	Markov chain	N/A	not validated with traces
[YBL89]	packet-switched bus	Markov chain	open and closed queueing networks (MVA [b])	not validated with traces
[YF82]	non-blocking	algebraic	open and closed queueing networks (custom solution)	not validated with traces, block on write
[Pat82]	circuit switched delta or crossbar	algebraic	known models	not validated with traces, ignores coherence
[BD81]	crossbar	algebraic	closed queueing network (custom solution)	not validated with traces
[PP84]	bus	algebraic	customized bus model	not validated with traces
[A^+85]	bus	algebraic	stochastic Petri nets	not validated with traces, continuous time
[AB86]	N/A	simulation of synthetic workload	N/A	not validated with traces

[a] N/A means not applicable.
[b] MVA denotes Mean Value Analysis [ABC86].

cache models were overly pessimistic in their assumptions about reference locality and contention for shared data [EK88].

Calibration with real programs can take place at Step 1 as described above, or can be applied to the entire modeling process with detailed simulations of the entire multiprocessor. As mentioned, the most common technique for validating models is trace-driven simulation. A detailed comparison of trace-driven multiprocessor studies is deferred to Chapter 3. The comparison shows, however, that most studies on directory-based coherence focus on invalidation protocols, most consider relatively small numbers of processors (32 or fewer), and few consider the impact of alternative interconnection networks. There is a need to investigate different directory protocols and a wider variety of interconnection networks. There is also a need to consider machines with larger numbers of processors. One of the reasons existing work has had a narrow focus is the high cost of detailed multiprocessor simulation. More efficient techniques are needed to reduce this cost.

1.4 Organization of the Dissertation

This dissertation builds upon existing work on directory-based cache coherence by developing more efficient techniques for performance evaluation and by applying them to a broader class of architectures. The overall performance methodology is based on that described in Section 1.3.3, in which the workload is characterized and iteratively applied to a network model. The dissertation is organized as follows.

In Chapter 2 a more detailed description of cache coherence strategies is provided. The strategies are suitable for multiprocessors with hundreds of processing elements, with a focus on simple and efficient directory-based techniques. A new efficient directory implementation is introduced.

A trace-driven analysis of directory-based coherence schemes is presented in Chapter 3. An efficient stack simulation technique is used that permits multiple cache sizes to be evaluated in a single simulation run. Quantitative data for invalidation, update and competitive directory protocols are presented. In addition, a new technique is introduced that permits efficient evaluation of a spectrum of competitive protocols. These techniques are applied to three benchmark programs for several cache, block and multiprocessor sizes.

The results in Chapter 3 are obtained under the assumption of zero miss penalty. The impact of non-zero miss penalties are considered in Chapter 4, which presents a comparison of a broad class of interconnection network. The comparison is based on new, efficient analytic models.

The results in Chapter 3 are obtained with the additional assumption of ideal synchronization support for locks and barriers. This assumption is justified in Chapter 5, in which known synchronization techniques are reviewed. Several new techniques are presented for the implementation of fetch&op and barrier primitives in a broad class of interconnection network.

Chapter 6 concludes with a summary of important results and a discussion of future research issues.

1.5 Contributions

The contributions of this work are:

1. *A detailed study of directory-based cache coherence schemes using execution driven simulation of several benchmark programs.* This study provides quantitative comparison data for update, invalidation, and competitive directory protocols. An efficient stack simulation technique is used that permits data for multiple cache sizes to be obtained in one simulation run. Efficient ways of storing a global directory are developed and compared using new and previously published results.

2. *Efficient analytic models for evaluating a broad class of packet-switched interconnection networks.* The class includes hypercubes, indirect binary cubes, meshes, and k-ary n-cubes. The techniques are used to compare the most promising interconnection networks using workload data obtained from the cache study in 1.

3. *An implementation technique for incorporating fetch&op primitives in the kxk cross-bar switches used to build meshes, Delta networks, k-ary n-cubes, and many other networks.*

Chapter 2

Directory-based Cache Coherence Strategies

2.1 Overview

This chapter is a review of directory-based coherence schemes that are suitable for machines with hundreds or thousands of processors. It begins with a detailed description of the Censier and Feautrier directory scheme, from which most of the other schemes are derived. As mentioned in the introduction, the main weakness of this scheme is the excessive amount of memory required to implement the directory. A secondary weakness is that directory controllers may be locked out for long periods of time when large numbers of invalidations or updates must be issued. All of the schemes described in this chapter overcome the first weakness, and a few attempt to overcome the second. One of the refined schemes is a contribution of this dissertation.

The goal of the chapter is to show that hardware coherence techniques exist that are of reasonable complexity and require a reasonable amount of hardware for implementation. The contribution of this chapter is the "tag cache" directory implementation.

2.2 The Censier and Feautrier Directory Scheme

In this scheme physical memory is divided into blocks of fixed size. Each block of main memory is associated with a directory entry (or tag) containing N presence bits where N is the number of processors in the system, a single bit indicating whether or not the block is modified, and a lock bit (Figure 2.1). The bit vector implement a list of all cached copies. Using the notation

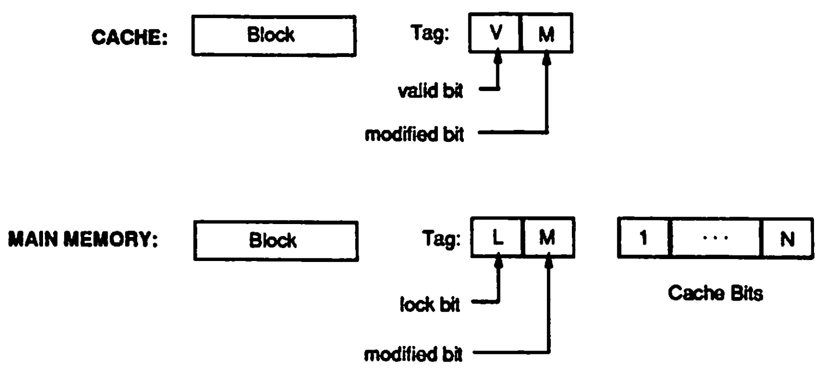


Figure 2.1: Tags for Basic Censier and Feautrier Protocol

of [AB84], a block is always in one of these three states:

1. ABSENT: no cache holds a copy (all cache bits in the directory entry are 0, and the modified bit is 0; lock bit is 0);
2. PRESENT: one or more caches hold copies, and the block is unmodified (one or more cache bits in the directory entry are 1, and the modified bit is 0; lock bit is 0);
3. PRESENTM: exactly one cache has a copy and it is modified (exactly one cache bit is 1 and modified bit is 1; lock bit is 0);
4. LOCKED: an operation on this block is currently in progress (lock bit is 1);

In like manner, each cache block is associated with a cache directory entry consisting of a valid bit and a modified bit (Figure 2.1). Cache blocks may be in one of these states:

1. INVALID: the contents of the cache block are invalid (valid bit is 0);
2. VALID: the contents of the cache block are valid and unmodified (valid bit is 1 and modified bit is 0);
3. VALIDM: the contents of the cache block are valid and modified (valid bit is 1 and modified bit is 1). This state implies that this cache has the only valid copy of the block in the entire multiprocessor.

The coherence protocol is defined by the actions taken by cache and memory controllers for each combination of processor request, cache block state, and main memory state. If a processor issues a read and the local cache block of the data is valid, no main memory access is needed and the data is read from the cache. If a block for the referenced data does not exist, a block must be assigned and its old data displaced to main memory. The missed reference is then handled as if the block was invalid: a read transaction is issued to the main memory. If the main memory block is in an unmodified state, the block contents are returned to the requesting memory controller. If the main memory block is modified, the block contents are read from the single "owning" cache, written to main memory, and forwarded to the requester. In all of these cases the cache and main memory entries have their states updated to VALID and PRESENT, respectively.

When a processor issues a write, it can only be satisfied locally if the local cache block is VALIDM. If the local cache state is VALID, an invalidate transaction is sent to the main memory which, if other caches have copies (ie: main memory state is PRESENT or PRESENTM), issues invalidations to them. If the local cache misses or the block is invalid, the controller issues an invalidate-fetch transaction to the main memory. A fetch is required here so that the portion of the block untouched by the write is made valid. The main memory sends invalidations to caches with copies. If the block is modified, the main memory also fetches the current data, updates itself, and forwards the data to the requester. The states of cache and main memory blocks are updated to VALIDM and PRESENTM, respectively.

The main memory is always notified of block replacements so that the appropriate cache bit is cleared. If a cache replaces a VALIDM block, the block must be written back and the main memory state changed to ABSENT.

The Censier and Feautrier scheme is well-suited to large multiprocessors because it does not depend on the use of broadcasts (and hence does not depend on a particular network), and permits the main memory and its directory to be interleaved. Although the communication overhead could be excessive if many blocks reside in many caches, the scheme's greatest drawback is the severe memory overhead introduced by the large number of cache bits in the main memory tags. As an example, a system with 100 processors requires a 102 bit tag, dictating a block size in excess of 125 bytes for tag overhead to be less than 10%. Systems built using this consistency scheme are not easily expanded because the tag length is dependent on the number of processors.

2.3 Efficient Directory Implementations

The severe memory overhead of the basic Censier and Feautrier scheme can be overcome in many ways. The techniques fall into three categories:

1. Restrict the length of the copy lists.
2. Reduce the granularity of the copy lists: have each cache identifier refer to a group of caches rather than a single cache.
3. Use a hierarchical directory.
4. Store the copy lists per *cache block* rather than per *main memory block*.

2.3.1 Schemes that Restrict List Length

It has been empirically observed for invalidation protocols that a small number of cache identifiers are sufficient most of the time [A+88a]. The most basic list reduction scheme exploits this by providing only a single cache identifier per main memory block. When more than a single copy is required broadcasts are used [AB84]. Agarwal et. al. generalize this to their Dir_iB and Dir_iNB schemes [A+88a]. In a Dir_iB scheme, i cache identifiers are provided for each block, and broadcasting is used when the number of copies exceeds i. The Dir_iNB scheme is similar to Dir_iB, except that instead of resorting to broadcasting, identifier memory is "recycled" by invalidating older copies. Although directory overhead is reduced, for small block sizes it is still considerable. For example, one ten bit cache identifier creates 7.8% overhead for a 16B block size. The performance of these schemes has been evaluated using trace-driven simulation in [C+90]. The results indicate that a full directory permits up to twice the processor utilization than a Dir_iNB scheme with $i \leq 4$. This is largely because of contention for synchronization variables. With various software optimizations, the performance with limited identifiers comes within 10 % of a full directory scheme.

[CKA91] describes a variation of the above schemes called a *limitless* directory. A limitless directory is a Dir_iNB scheme modified so that directory overflows interrupt the local processor, which maintains long directory entries in local main memory.

2.3.2 Schemes that Reduce List Granularity

Weber and Gupta propose the use of directory entries of a single size, but further reduce tag overhead with the use of *coarse vectors* and multicast invalidations [G+90b]. [BH89] describes a similar idea, and discusses a particular implementation of a multistage interconnection network that supports efficient multicast operations; this multicast optimization is similar to one proposed by Stenstrom [Ste89] for his decentralized linked list protocol (described in Section 2.3.3 below).

2.3.3 Schemes that Store Lists Per Cache Block

Another way to reduce directory overhead is to exploit the fact that tags are needed to record the locations of only those blocks residing in caches. Since the total number of cache blocks is typically much smaller than the total number of main memory blocks, tag overhead can be greatly reduced. For example, a 64 processor machine with 32 kilobyte caches and 32 megabytes of distributed main memory requires only 1.6 megabytes of full length tags (assuming they are 100 bits in length), to store directory information for the $64 \times 32K \div 16 = 128K$ cache blocks. Alternatively, 25 megabytes of full length tags are needed if one tag is provided per main memory block.

In one class of these schemes [ON90, LY90, Ste89] copy lists are stored in associative memories accessed by block address. The scheme described in [ON90] was proposed as part of this research. In this scheme, two *tag caches* of different tag sizes are provided at each bank of the distributed main memory: a large cache with small tags capable of holding the identifiers of a small number of cached copies (Figure 2.2), and a small cache with full-sized tags (Figure 2.1).

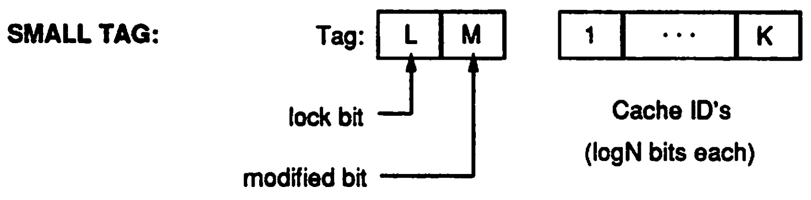


Figure 2.2: Small Tag Fields

The coherence protocol for this scheme is similar to the basic Censier and Feautrier protocol with the exception that tags must be allocated from a tag cache as needed; when no tags are free, a cached block must be invalidated and its tag re-allocated. When a block is first referenced, it is allocated a small tag. When the number of copies of a block exceed the number of copies supported by the small tags, a large tag is allocated and the small tag is freed. A least-recently-used (LRU) replacement strategy may be used to select tags for re-allocation.

Similar schemes were independently published in [G+90b] and [Ste89]. In [G+90b] a tag cache is called a *sparse directory*. Instead of using two tag sizes, the scheme in [G+90b] uses copy lists with coarse granularity, as described in Section 2.3.2. The scheme described in [Ste89] differs from the other two in that full size directory tags are kept at the caches instead of the main memory. The main memory, however, stores a cache identifier with each block to point to the cache with valid directory data.

The complexity of storing lists as bit vectors in associative tables is greatly reduced by using linked lists. These can be organized in a centralized or decentralized fashion. Decentralized linked list schemes include the Scalable Coherent Interface protocol [IEE90] and the Stanford linked list protocol [T+90]. In these schemes tag overhead is reduced by storing single identifier tags at the caches and maintaining copy information in distributed linked lists (Figure 2.3). As in the [Ste89] scheme, the main memory holds a pointer to the head of the copy list for each block. In the basic protocol, invalidations are performed by having each member of the list invalidate their copy and remove themselves from the list (at the main memory). This is done serially, so the time to perform j invalidations takes the time of $2j$ network transactions. In contrast, the centralized directory schemes issue invalidations serially at the controller, but they can traverse the network and be processed in parallel. Optional variations of the SCI protocol have been proposed to reduce serialization, but they require complicated checks to avoid dangling list segments.

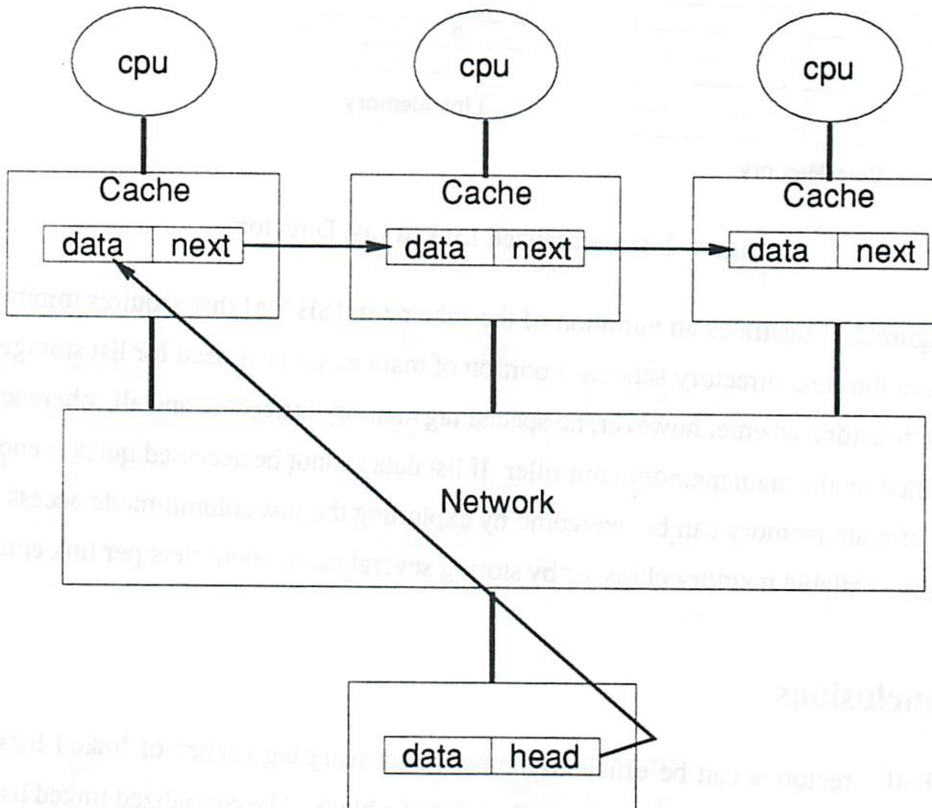


Figure 2.3: Decentralized Linked List Directory

The performance problem of simple decentralized schemes and the complexity of higher performance decentralized linked list schemes can be overcome by centralizing the linked lists at the memory controllers [SH91a]. The scheme in [LY90] stores list entries in an associative table, similar to a tag cache, but relies on compiler support to manage the directory. Figure 2.4 illustrates the centralized linked list scheme due to [SH91b]. Here links are stored in a special memory and managed by the main memory controller. Whenever the supply of links is exhausted, one or more links in use are reclaimed by invalidating the appropriate copies; the link pool is made sufficiently large, however, that this should be rare.

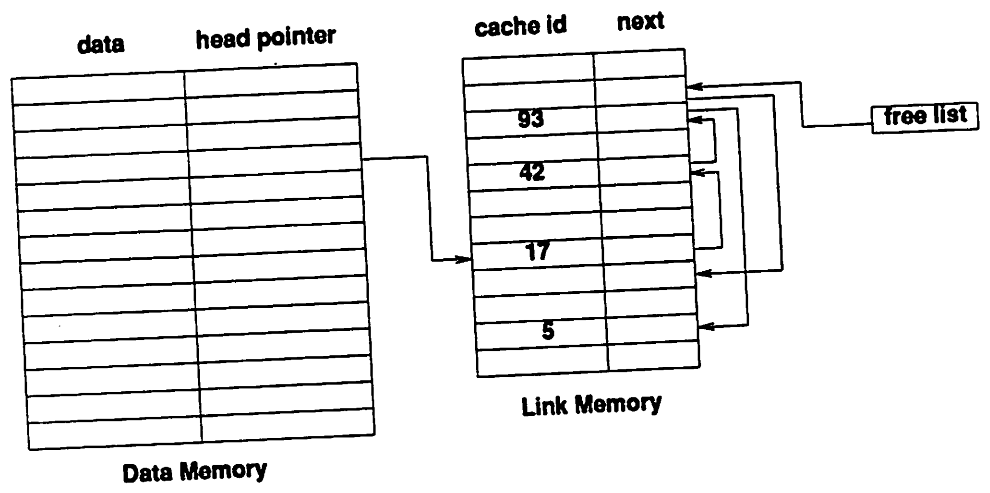


Figure 2.4: Centralized Linked List Directory

Figure 2.5 illustrates an variation of the scheme in [SH91a] that requires minimal hardware. Like the limitless directory scheme a portion of main memory is used for list storage; unlike the limitless directory scheme, however, no special tag memory is needed, and all coherence activities are handled by the main memory controller. If list data cannot be accessed quickly enough, the slow speed of main memory can be overcome by exploiting the fast column-mode access of many commercially available memory chips, or by storing several cache identifiers per link entry.

2.4 Conclusions

Full directories can be efficiently constructed using tag caches or linked lists, without broadcasting or limiting the number of cached copies of a block. The centralized linked list schemes are the most attractive because they do not need hardware to manage large bit vectors, and they avoid the complications of decentralized linked list schemes. The memory-mapped decentralized linked list scheme is especially attractive because of its low implementation cost: a slightly more complex controller and several additional registers.

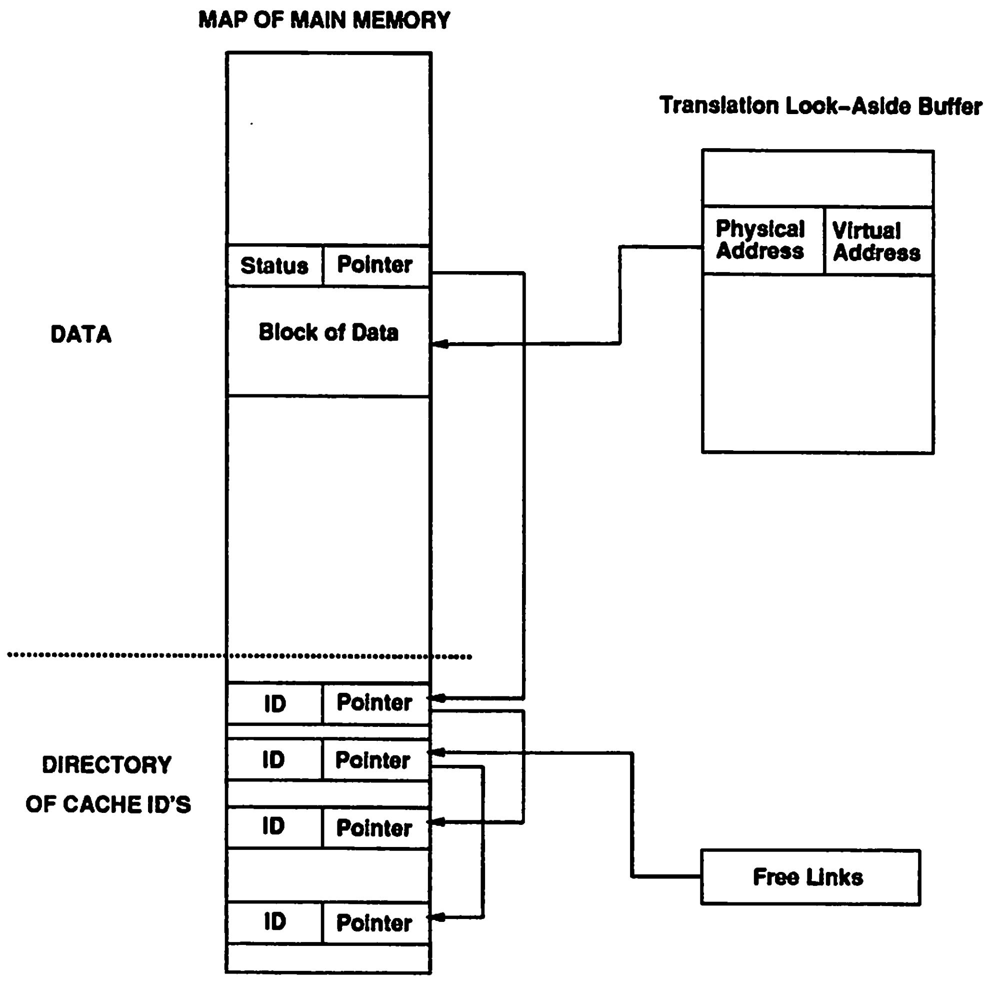


Figure 2.5: Memory-Mapped Linked List Scheme

Chapter 3

Workload Characterization

3.1 Overview

Workload characterization using references from benchmark programs has been extensively used for uniprocessor cache studies [Smi82] and more recently for multiprocessor caching schemes [AG88, SA88, WG89b, ON90]. This chapter begins with a review of previous work. Following this, known stack simulation techniques are extended to support the evaluation of directory methods. These techniques are applied to three benchmark programs to evaluate invalidation, update and competitive directory protocols. An additional technique is described that is used to evaluate numerous competitive schemes from the results of a single simulation run of an update protocol.

The goal of the chapter is to show that directory-based cache coherence is very effective at reducing average memory access time and network traffic. The contributions of this chapter are: extensions to a stack simulation algorithm that support directory schemes, an algorithm for determining competitive protocol performance from an update protocol simulation, and quantitative performance data for three benchmark programs.

3.2 Previous Work

Many recent studies present empirical data on multiprocessor performance based on traces of parallel programs (Tables 3.1 and 3.2); some of the benchmarks used in these studies are described in Table 3.3. The tables show that all studies of directory methods with non-bus networks consider invalidation protocols only, and most consider 32 or fewer processors. The MIT and

Table 3.1: Trace-driven Multiprocessor Cache Studies

Study	Focus	Benchmark Set	Number of Procs.	Notes
[G+83c]	Ultracomputer, software coherence	Fortran programs: parallel loops	up to 256	multistage network, execution-driven
[ASK85]	Cedar, software coherence	Fortran kernels: parallel loops	32	multistage network, execution-driven
[EK88] [EK89b] [EK89a]	snooping bus protocols	Eggers (Table 3.3)	5 to 12	shared bus, trace-driven
[A+88a]	directory methods (invalidation)	Mach (Table 3.3)	4	shared bus, trace-driven
[DR+87]	Cedar, software coherence	Fortran kernels, parallel loops	8	multistage network, execution-driven
[SA88]	directory methods (invalidation)	Mach (Table 3.3)	4	shared bus, trace-driven
[AG88]	ping/cling sharing model	Mach (Table 3.3)	4	ideal network, trace-driven
[C+89]	hierarchical directory scheme	Mach (Table 3.3)	4	tree of busses, trace-driven
[WG89b]	analysis of directory invalidation patterns	Stanford1 (Table 3.3)	4 to 32	ideal network, trace-driven
[G+90b]	sparse directories (invalidation)	Stanford1, Stanford2 (Table 3.3)	4 to 64	ideal network, trace-driven

Table 3.2: Trace-driven Multiprocessor Cache Studies (continued)

Study	Focus	Benchmark Set	Number of Procs.	Notes
[SWG91]	sparse directories (invalidation)	Stanford1, Stanford2 (Table 3.3)	4 to 64	ideal network, trace-driven
[GGH91]	coherence model comparison	Stanford1 (Table 3.3)	16	constant network delay, execution-driven
[C+90]	directory schemes, (invalidation)	MIT (Table 3.3)	32 to 64	multistage network, trace-driven
[CKA91]	Limitless directories (invalidation)	MIT (Table 3.3)	64	multistage network, trace-driven
[LY90]	software coherence with directory (invalidation)	Lilja (Table 3.3)	32	multistage network, execution-driven
[AG90]	ping/cling sharing model	Mach (Table 3.3)	4	ideal network, trace-driven
[Wil87]	hierarchical directory (invalidation)	3 C programs	16	bus hierarchy, trace-driven
[L+87]	software coherence	Fortran kernels, parallel loops	32	multistage network, execution-driven

Table 3.3: Common Multiprocessor Benchmarks

Benchmark Set	Program	Language [a]	Parallelism	Data Refs Per Proc. (thousands)
Mach	POPS: rule-based language	?	?	380
	THOR: logic simulator	C	?	442
	LocusRoute: VLSI router	C	?	419
Eggers	TOPOPT: PLA folder	C	6/8	101
	VERIFY: logic verifier	C	11/12	96
	SPICE: circuit simulator	?	?	114
	CELL: VLSI placement	C	6/8	141
Stanford1	Maxflow: max. flow in graph	C	?	281
	SA-TSP: traveling salesman problem	C	?	238
	MP3D: particle simulator	C	52/64	173
	THOR: logic simulator	C	16/64	223
	LocusRoute: VLSI router	C	48/64	214
Stanford2	Ocean: eddy current simulation	Fortran	84/96	?
	Water: water molecule simulation	C	44/48	?
	Cholesky: cholesky factorization	C	22/64	?
MIT	FFT	Fortran	?	68
	Weather	Fortran	12/32	283
	Simple	Fortran	7.3/16	422
	Speech	Lisp	?	184 [b]
Lilja	arc3d: fluid flow	Fortran	?	206
	flo52: transonic flow past airfoil	Fortran	?	313
	trfd: quantum mechanics	Fortran	?	184
	simple24: heat flow	Fortran	?	133
	pic: electrodynamics	Fortran	?	274
	linpack: 125x125 matrix	Fortran	?	313

[a]"?" denotes unpublished.
[b]does not include instruction references.

Stanford benchmark sets include traces of 64 processor systems, but only three of these are for C programs with reasonable parallelism. The only trace-driven analyses of update and competitive protocols are [EK88, Egg91, A$^+$88a], which study shared bus architectures with 12 or fewer processors. By focussing on shared bus architectures, these studies do not adequately measure the coherence traffic that would occur with point-to-point networks.

There are also differences in the goals of previous studies. Most are comparisons of average memory access time and network traffic for very specific architectures. [WG89b] enumerates classes of coherence patterns and relates them to benchmark programs. [AG90] and [Egg91] consider simpler, more abstract models of program behavior based on what [AG90] calls *processor locality*. Processor locality is the degree to which data is referenced by a processor without intervening references by other processors; data that is referenced many times by a processor before a reference by another processor exhibits high processor locality (which is desirable in a coherent caching scheme). It is quantified with the *ping/cling* [AG90] and *write-run* [Egg91] models. Instead of examining the stream of data references issued by each processor, these models consider the stream of references to each data block, focussing on the identity of the processor associated with each reference. The ping/cling model is illustrated in Figure 3.1. A *ping* is defined as a refer-

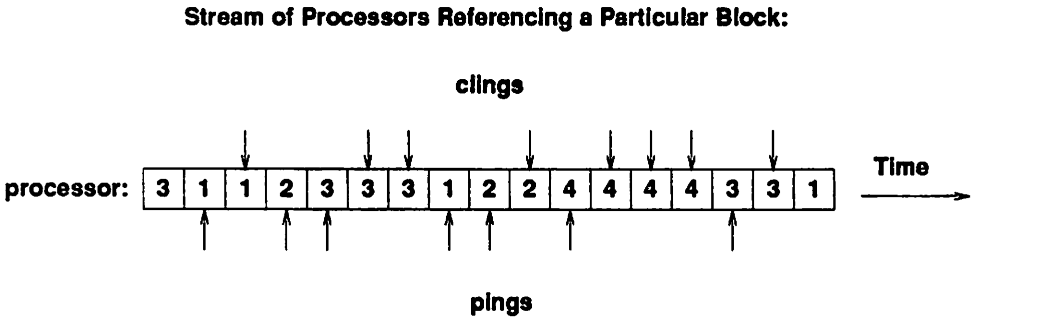


Figure 3.1: The Ping/Cling Locality Model

ence to a block in which the processor differs from that of the previous reference; here the data has "bounced" or *pinged* to another processor, requiring some coherence action. A *cling* occurs when a block is immediately re-referenced by the same processor. Good processor locality is indicated by a large ratio of clings to pings. Programs with good processor locality are said to exhibit *sequential sharing*, and perform well with an invalidation protocol. Alternatively, programs with a large ratio of pings to clings are said to exhibit *fine-grained sharing*, suggesting the use of an update protocol.

The *write-run* sharing model is a special case of the ping/cling model (Figure 3.2). A write-run is defined as a sequence of writes by a particular processor with no intervening references

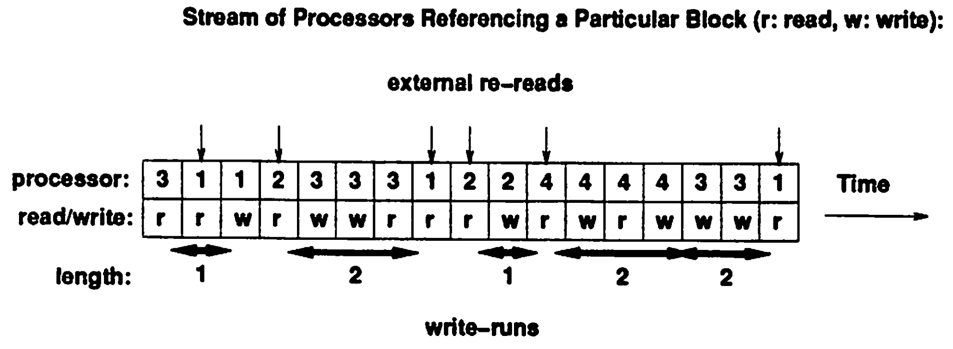

Stream of Processors Referencing a Particular Block (r: read, w: write):
external re-reads
processor: 3 1 1 2 3 3 3 1 2 2 4 4 4 4 3 3 1
read/write: r r w r w w r r r w r w r w w w r
Time
length: 1 2 1 2 2
write-runs

Figure 3.2: The Write-run Model

by another processor; the write-run begins with a write and ends with a read or write by another processor. Write-runs therefore begin and end with particular types of *pings*, and are made up of a particular type of *cling*. A sequence of write-runs may be separated by sequences of reads by a set of processors, called *external reads*. The first read by another processor after a write-run is called an *external re-read*. The behavior of update and invalidation protocols can be measured using components of the write-run model:

- Invalidations in a shared-bus invalidation protocol: number of write-runs (ie. the number of writes that ping).
- Invalidation misses in an invalidation protocol: number of external re-reads (ie. the number of reads that ping).
- Unnecessary updates in a shared-bus update protocol: length of a write-run (ie. the number of writes that cling to a processor).

The ping/cling and write-run models provide a convenient framework for understanding the reference behavior of parallel programs. The write-run model is also sufficiently detailed to permit a comparison of snooping update and invalidation protocols using data from one simulation run [Egg91]. Although the basic ping/cling and write-run models include measures of the number of invalidation or update requests issued from the cache at which a write is made, they do not provide a measure of the number of other cached copies that must be invalidated or updated. This is important in evaluating directory-based coherence schemes, where invalidations and updates are not performed in a single operation.

The results in this chapter build upon previous work in four ways:

1. By investigating a broader class of protocols, including invalidation, update and competitive.
2. By applying efficient stack simulation techniques to the evaluation of directory schemes.
3. By supplementing the write-run and ping/cling performance models with the notion of an *update-run*: a stream of updates received by a *cache* block between local accesses. update-runs quantify point-to-point coherence traffic and permit a comparison of a spectrum of competitive protocols using data from a single simulation run.
4. By considering larger numbers of processors (64 and greater).

This work also avoids some secondary problems with existing studies. First, ideal synchronization support is assumed so that coherence effects are not skewed. This is important because excess traffic and misses due to naive barrier or lock implementations can produce misleading results. For example, results in [$C^{+}90$] show that the use of a tournament barrier instead of a simple counting barrier [1] substantially reduces traffic and average access time. Another secondary problem that is avoided is a lack of parallelism in the benchmarks. As shown in Table 3.3, many of the benchmarks used in previous studies have an unknown or poor amount of parallelism. All of the benchmarks used in this chapter exhibit good processor utilization for at least 64 processors.

3.3 Stack Simulation of Directory Methods

3.3.1 Introduction

Stack simulation is a technique for simulating caches of multiple sizes in a single pass over a reference trace. They significantly reduce the amount of time required to simulate a set of alternative caches. In this section stack simulation algorithms for single processor and shared bus multiprocessors are reviewed . The section begins with a discussion of some cache design policies that must be restricted for stack algorithms to apply.

All cache designs must specify policies for: the selection of a block for *replacement*, the selection of what to *fetch* and when, and what should be done with *write* references [Smi82]. A *replacement* policy specifies how a block is selected for eviction when a newly referenced block is brought in. Common replacement policies include *least recently used (LRU)*, *first-in-first-out (FIFO)* and *random*. *fetch* policies determine when blocks of data are brought into the cache.

[1]These barrier implementations are described in Section 5.2.4.

The most common fetch policy is *demand fetch*, in which blocks are brought into the cache only when they are first referenced. A more aggressive fetch policy is *demand prefetch*, in which one or more blocks following the referenced block are also loaded on a cache miss. A cache *write* policy determines what takes place in response to a write hit or miss. A *write-through* cache updates its copy and main memory on each write; a *copy-back* or *write-back* cache updates main memory only when a written (or *dirty*) block is replaced. When a write miss occurs in a write-through cache, the cache can load the referenced block (*write allocate*) or just update main memory.

With certain restrictions, multiple sizes of a particular cache design can be simulated in a single pass of a reference trace using stack simulation. Stack simulation exploits the *inclusion property* of certain replacement and fetch policies, which ensures that the contents of a cache of a particular size are included in all larger caches. In the basic scheme [MGST70], the hit ratios for all cache sizes can be found in one pass of the reference trace by storing block addresses in a stack such that all blocks in a cache of size C are represented by the top C stack entries (Figure 3.3). To

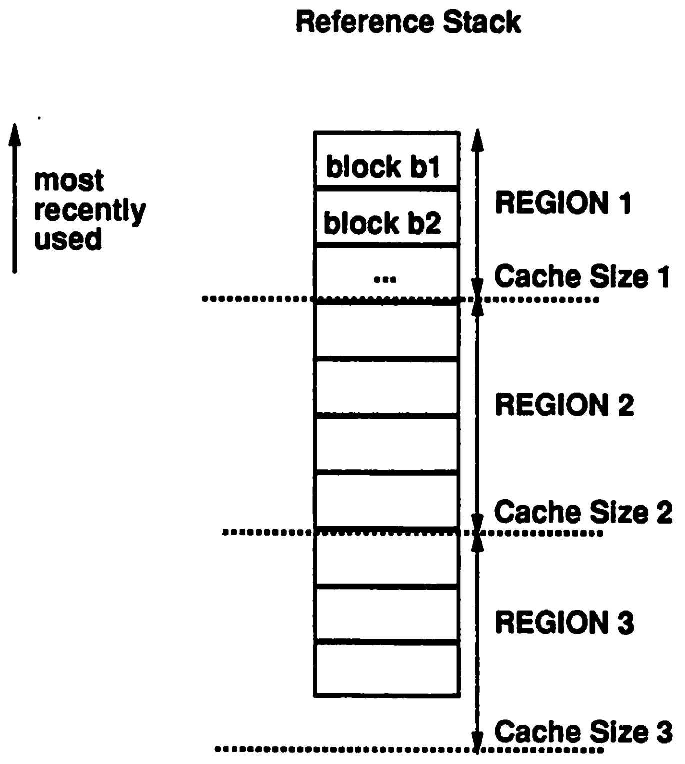


Figure 3.3: Uniprocessor Stack Simulation

keep track of the number of hits, a hit counter is associated with each position in the stack. On each reference, the corresponding block is located in the stack (at level C) and moved to the top (since it must now reside in all cache sizes), and the hit counter is incremented at the position where the block was found. The blocks that were in positions 1 through $C-1$ are now rearranged (in a manner depending on the replacement policy) to make room for the referenced block at position 1. Because of inclusion, a hit at level C represents a hit in all caches of size C or greater. The hit ratio for a cache of size C is simply the sum of hits for all stack positions from 1 to C. The stack algorithm can only be applied to caches with *stack* replacement policies and a restricted set of fetch policies. Stack replacement policies satisfy the constraint that the selection criteria for a replaced block must be independent of cache size. The simplest and most common stack replacement policy is *least recently used (LRU)*; in a stack simulation with an LRU replacement policy, the rearrangement of block addresses is simply a downward shift. Fetch policies that do not violate the inclusion property include demand fetch and demand prefetch.

The hit counters need not be maintained for each block. If the number of cache sizes of interest is much smaller than the stack size, memory can be saved by associating counters with the *regions* of blocks defined by the cache sizes.

The simple scheme just outlined is restricted to a single block size, single processor, and full associativity. Extensions have been published to support multilevel cache hierarchies [Gec74], multiple block sizes and associativities [TS71, ST72], and the collection of other statistics in addition to miss ratios [TS89]. Extensions have also been published to support operations other than read and write, such as deletions and cache flushing [TS89]. Efficient non-stack simulation algorithms have been developed for supporting certain important cache designs for which inclusion does not hold [HS89]. The restriction of a single processor has been relaxed by extensions that support shared bus and file system caching systems [Tho87]. These extensions are the focus of the following section.

3.3.2 Multiprocessor Stack Simulation

The multiprocessor stack simulation algorithm used in Section 3.6 is based on work by Thompson [Tho87], who developed a stack simulation algorithm for the *MOESI* [SS86] class of coherence protocols. The MOESI class of protocols includes most published shared bus coherence schemes, and, with the modifications in Section 3.3.4, is also applicable to the three directory schemes considered in this dissertation.

The MOESI protocol class derives its name from the five states a cached block may take (Figure 3.4):

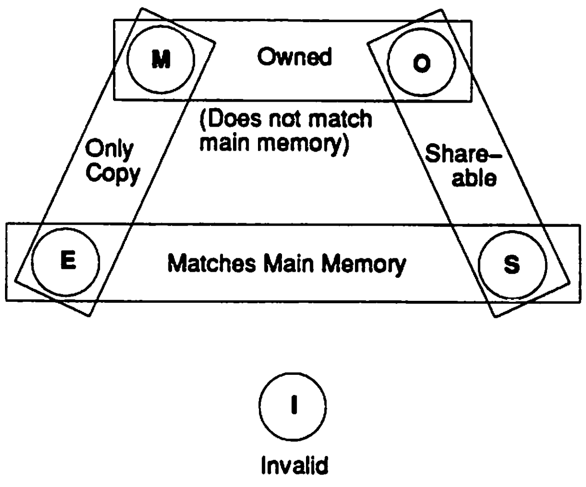


Figure 3.4: MOESI States

- *Modified*: this is the only cached copy and it is dirty.
- *Owned*: there are 2 or more cached copies, and this copy is dirty.
- *Exclusive*: this is the only cached copy and it is not dirty.
- *Shared*: other caches *may* have copies, and this copy is not dirty.
- *Invalid*: this copy is invalid.

The MOESI protocols are described in Tables 3.4 and 3.5 (taken from [Tho87]), which show cache actions in response to processor requests and network (bus) requests, respectively. Responses are shown in one of two forms. "X, action" means change the block state to X and perform "action". "Shared?: X/Y, action" means perform "action" and set the block state to X if the block is shared, and Y if unshared. Transactions of the form Shared?:O/M can be replaced by O, and transactions of the form Shared?:S/E can be replaced by S. The network transactions specified in Table 3.4 are as follows:

1. *nothing*: do nothing.
2. *read block*: fetch a copy of the block (to read).
3. *write block*: write back a copy of the block to main memory. Keep the local copy.
4. *displace block*: write back a modified copy of the block to main memory. Invalidate the local copy.

5. *notify*: notify the main memory of a flush.

6. *inval*: invalidate all other copies of the block.

7. *update*: update all other copies of the block.

8. *read block and inval*: fetch a copy of the block (to write) and invalidate all other copies of the block.

9. *read block and update*: fetch a copy of the block (to write) and update all other copies of the block.

10. *read, write*: perform a separate cpu read (to bring the data into the cache) followed by a separate cpu write.

write-thru denotes a request that is issued by a write-through cache; *no cache* denotes that it is requested by a processor without a cache.

The network transactions specified in Table 3.5 are as follows:

1. *nothing*: do nothing.

2. *forward*: forward a copy of the block to the requesting cache without updating main memory.

3. *forward & update memory*: forward a copy of the block to the requesting cache and update main memory.

4. *update copy*: update the local cached copy.

The original MOESI definition specifies bus signals for each bus transaction. Tables 3.4 and 3.5 abstract this because the protocols are also applicable to directory schemes using general interconnection networks. In the context of a shared bus, all caches observe the network transactions, including those without copies (or, equivalently, copies that are invalid). *sharability* can therefore be determined by a wired-OR sharing line (SL in the MOESI specification), and the bus ensures that all coherence actions for a particular processor reference are satisfied atomically. In the context of a general interconnection network and directory-based coherence, the directory protocol would ensure that only those caches that need to respond to a network transaction are accessed; atomicity would be enforced, and *sharability* determined, via the directory.

Table 3.4: MOESI Cache Responses to Processor Requests

Cache State	Processor Request	
	Read	Write
M (Modified)	M, nothing	M, nothing
O (Owned)	O, nothing	{4} Shared?:O/M, update {5} M, inval
E (Exclusive)	E, nothing	M, nothing
S (Shareable)	S, nothing	{6} Shared?: O/M, update {7} M, inval {8} S, update(write-thru) {9} S, inval(write-thru)
I (Invalid)	{1} Shared?: S/E, read block {2} S, read block (write-thru) {3} I, read block (no cache)	{10} M, read block & inval {11} read,write {12} I, inval (no-cache,write-thru) {13} I, update (no-cache,write-thru) {14} read,write (write-thru)

Cache State	Processor Request	
	Pass (Push & Keep)	Flush (Push & Discard)
M (Modified)	E, write block	I, displace block
O (Owned)	{15} Shared?: S/E, write block {16} S, write block	I, displace block
E (Exclusive)	–	I, notify
S (Shareable)	–	I, notify
I (Invalid)	–	–

Table 3.5: MOESI Cache Responses to Network Requests

Network Request	Cache State	
	M (Modified)	O (Owned)
Read Block or Write Block	{19} O, forward {20} S, forward & update memory	{17} O, forward {18} S, forward & update memory
Read Block & Inval or Inval or Read Block & Inval (Write-thru)	{21} I, forward {22} I, forward & update memory	I, forward
Read Block (No Cache) or Displace Block	M, forward	Shared? O/M, forward
Read Block & Update or Update or Read Block & Update (Write-thru)	–	{23} S, update copy {24} I, nothing
Inval (Write-thru) or Inval (No Cache)	M, forward	O, forward
Update (Write-thru) or Update (No Cache)	M, update copy	O, update copy

Network Request	Cache State		
	E (Exclusive)	S (Sharable)	I (Invalid)
Read Block or Write Block	S, nothing	S, nothing	I, nothing
Read Block & Inval or Inval or Read Block & Inval (Write-thru)	I, nothing	I, nothing	I, nothing
Read Block (No Cache) or Displace Block	E, nothing	S, nothing	I, nothing
Read Block & Update or Update or Read Block & Update (Write-thru)	–	{25} S, update copy {26} I, nothing	I, nothing
Inval (Write-thru) or Inval (No Cache)	I, nothing	I, nothing	I, nothing
Update (Write-thru) or Update (No Cache)	{29} E, update copy {30} I, nothing	{27} S, update copy {28} I, nothing	I, nothing

Table 3.6: A Simple MOESI Invalidation Protocol

Cache State	Processor Request		
	Read	Write	Flush (Push & Discard)
M (Modified)	M, nothing	M, nothing	I, displace block
S (Shareable)	S, nothing	{7} M, inval	I, notify
I (Invalid)	{1} S, read block	{10} M, read block & inval	–

Network Request	Cache State		
	M (Modified)	S (Sharable)	I (Invalid)
Read Block	{20} S, forward & update memory	S, nothing	I, nothing
Read Block & Inval Inval	{22} I, forward & update memory	I, nothing	I, nothing
Displace Block	I, nothing	S, nothing	I, nothing

Table 3.7: A Simple MOESI Update Protocol (Write-thru)

Cache State	Processor Request		
	Read	Write	Flush (Push & Discard)
S (Shareable)	S, nothing	{8} S, update	I, notify
I (Invalid)	{2} S, read block	{14} read,write	–

Network Request	Cache State	
	S (Sharable)	I (Invalid)
Read Block	S, nothing	I, nothing
Read Block & Update Update	{25} S, update copy	I, nothing
Displace Block	S, nothing	I, nothing

Since the MOESI protocols represent a class, there are alternative actions for many states. Tables 3.6 and 3.7 show the alternatives used in simple invalidation and update protocols (these are two of the directory methods examined in Section 3.3.3).

The multiprocessor stack simulation algorithm is based on the following assumptions:

1. Reference streams are *synchronized*: the interleaving references from different processors is independent of cache size. This assumption permits one-pass analysis of a multiprocessor reference stream without considering low-level timing details that cause different temporal behavior for different cache sizes. It is unrealistic because it is equivalent to assuming zero miss penalty. The results should still be useful, however, because the network delays in a symmetric multiprocessor should perturb each reference stream in a similar way, so relative orderings should not change substantially. Furthermore, the simulated ordering is an example of at least one correct execution of the program. This assumption has been made in other simulation studies, including [C+90, OA89, SWG91].

2. Protocol actions are consistently applied on a block basis. If a particular rule from Table 3.5 is used for block i during a read miss, it must always be applied to block i under the same situation. This restriction still permits different protocols to be used for different blocks.

3. The stack position of a block in a particular cache cannot be changed by external cache actions, with the exception of invalidations. This is reasonable because "the fact that another cache is using a block may be reason to discard a block, but never a reason to want to keep it [Tho87]."

4. All caches in the multiprocessor are the same size.

The following description of Thompson's stack simulation algorithm is restricted to the subset of MOESI states required by the directory schemes of interest: the M, S and I states–those required by the directory protocols described in Section 3.3.3. These three states can be determined using the following state variables associated with each cache block: the valid level v_i and, if the block is dirty, its dirty level d_i and the identity of the sole cache containing the dirty copy DC. The *valid* level v_i of block i is the smallest cache size for which block i is valid; this is implicitly maintained as the stack position of the block. If the block is not valid in any cache, $v_i = \infty$. The *dirty* level d_i is the smallest cache size for which block i is dirty. If the block is not dirty in any cache, $d_i = \infty$. The block may be valid in smaller caches, but it is only dirty in caches of size d_i and larger. The situation where $v_i < d_i$ arises in write-back caches when a dirty block is pushed from a small cache and later read again, but not written. In the MOESI protocols a block can be dirty in only one cache at a time, so DC and d_i need not be stored for each copy of a block. The state of block i in a particular cache j of size C is determined by examining v_i, DC and d_i:

- M: $v_i < C$, $j = DC$ and $v_i >= d_i$
- S: $v_i < C$ and $(j \neq DC$ or $(j = DC$ and $v_i < d_i))$.
- I: $v_i > C$.

The stack algorithm for MOESI protocols involving only the M, S and I states is shown in Figures 3.5 and 3.6. It is assumed that the reference streams from all N processors are interleaved into a single trace of length n_{refs} in a manner that is consistent with the synchronization constraints of the program. Furthermore, for each reference the state of all copies of a block is updated atomically. Reference i in the interleaved trace furnishes a block address b_i, the type of reference $action_i$, and the identity of the cache, c_i, at which the access is made.

On each processor reference, the referenced block is located in the stack of the appropriate cache to determine its valid level, v_i. If the block is dirty, the identity of the sole cache containing the copy, DC, is found, along with the dirty level d_i. If the referenced cache happens to be DC, a check is made to see if any write-backs have occurred. This is indicated by the valid level exceeding the dirty level; since a block cannot be dirty in cache sizes for which it is not valid, all cache sizes between d_i and v_i must have flushed the block. These flush operations are recorded by a call to the "Statistics" routine of the form: Statistics(*type, from, to, low_size, high_size*). This routine records a network transaction of type *type* from processor *from* to processor *to* for cache sizes in the range *low_size* to *high_size*. The remaining simulation steps are reference-specific. If the reference is a read, the block must be fetched for all cache sizes for which it is invalid. If the block is dirty in another cache, it must be retrieved from that cache before being forwarded to the referenced cache. *Pass* references require main memory to be updated only if the local copy (if any) is dirty. *Flush* references require main memory to be updated if the local copy is dirty, and the local copy to be invalidated.

If the reference is a write, the simulation steps are protocol-specific. Figure 3.7 shows the steps taken for the simple invalidation protocol of Table 3.6. Here the block must be fetched (for writing) by all cache sizes for which it is invalid. If the block is dirty in another cache, it must be retrieved from that cache, and that copy invalidated. If the block is not dirty in another cache, all other copies are invalidated. The actions for the simple update protocol (Figure 3.8 and Table 3.7)

```
for ($i = 1$ to $n_{refs}$) {

    $b$        = block address for reference $i$
    $action$   = action for reference $i$
    $c$        = cache identifier for reference $i$
    $v_c$      = position of $b_i$ in stack $c$
    $DC$       = sole cache containing dirty copy of $b_i$ (if any)
    $d$        = dirty level for block in cache $DC$ (if applicable)
    $m$        = main memory bank containing $b$.

    if ($c == DC$) { /* if cache holds dirty block */
        if ($d < v_c$) { /* record writebacks */
            Statistics(WRITEBLOCK, $c, m, d, v_c$);
            $d = v_c$;
        }
    }

    switch ($action$) {
        case READ:
            Statistics(READBLOCK, $c, m$, 1, $v_c$);
            if (rules {7} or {10} used) { /* if invalidation protocol */
                if ($DC \neq \infty$ and $DC \neq c$) { /* if block dirty, get copy from owner */
                    Statistics(RETRIEVE_BLOCK_TO_READ, $m, DC, d, \infty$);
                    $d = \infty$;
                }
            }

        case WRITE:
            WriteRoutine(); /* protocol dependent */

        case PASS:
            if ($c == DC$ and $d \neq \infty$) { /* write back if dirty */
                Statistics(WRITEBLOCK, $c, m, d, \infty$);
                $d = \infty$;
            }

        case FLUSH:
            if ($i == DC$ and $d \neq \infty$) { /* write back if dirty */
                Statistics(WRITEBLOCK, $c, m, d, \infty$);
                $d = \infty$;
            }
            $v_c = \infty$; /* invalidate local copy */
    }
    update stack $c$;
}
```

Figure 3.5: Stack Algorithm for MOESI Protocols

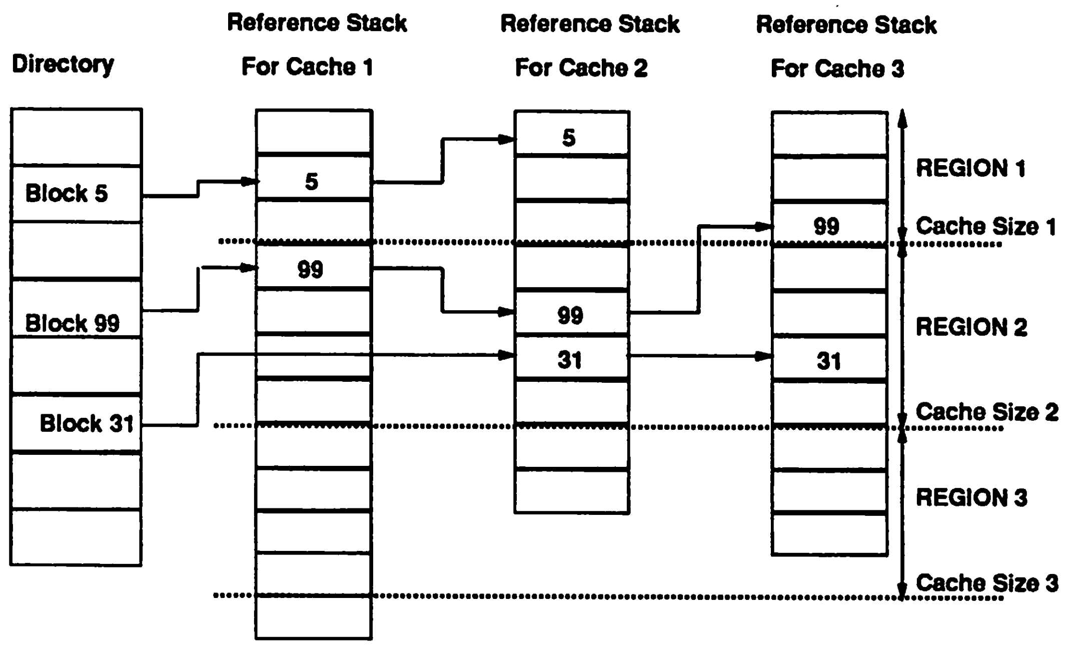


Figure 3.6: Multiprocessor Stack Simulation

are similar, except other cached copies are updated instead of invalidated, and it is not possible for a block to be dirty in any cache.

In the basic uniprocessor stack algorithm, the number of hits for all cache sizes was efficiently maintained using counters for each stack region. Since the number of hits obeys inclusion, the number of hits for all caches containing the referenced block can be maintained by incrementing just one counter per reference. Other cache statistics can be efficiently collected using counts that obey inclusion. For each of these statistics, the number of events in a cache of size C is the sum of counts for each of the stack regions contained by that cache. These statistics include:

1. block writes (write-backs): count the number of *avoided* write-backs [Tho87]. This is done by incrementing an *avoided write-backs* counter for the stack region containing the dirty level of a written block. Since an avoided write-back in a cache of size C must also be an avoided write-back in all larger caches, the count obeys inclusion.

2. block reads (fetches): these are misses, which are determined by counting the number of *hits*. The number of misses is then the number of reads and writes minus the number of hits. These may be further distinguished by the type of reference causing the fetch: read or write. This is useful because extra actions (invalidations or updates) are required for block fetches caused

```
Statistics(READ_BLOCK_TO_WRITE, $c$, $m$, 1, $v_c$);

if ($d \neq \infty$ and $DC \neq c$) {
    Statistics(RETRIEVE_BLOCK_TO_WRITE, $m$, $DC$, $d$, $\infty$);
    $d = \infty$;
}
for $\{j \mid v_j \neq \infty, j \neq DC, j \neq c\}$ {
    Statistics(INVALIDATE, $m$, $j$, $v_j$, $\infty$);
    $v_j = \infty$;
} $DC = c$;
$d = 0$;
```

Figure 3.7: Write Routine for Invalidation Protocol

```
Statistics(READ_BLOCK_UPDATE, $c$, $m$, 1, $v_c$);

for $\{j \mid j_v \neq \infty, j \neq c\}$ {
    Statistics(UPDATE, $m$, $j$, $v_j$, $\infty$);
}
```

Figure 3.8: Write Routine for Update Protocol

by writes.

3. write-throughs (to main memory): This is just the total number of writes.

Stack simulation is desirable only if the added complexity does not defeat the benefit of analyzing multiple cache sizes simultaneously. Empirical results in [Tho87] show that a stack simulation takes about 15 % more time to run than a non-stack simulation of a single cache size, so the net benefit is substantial. Similar overhead was observed for the simulations used in this work.

3.3.3 Protocols

The directory protocols considered in this chapter are variations of Censier and Feautrier's scheme [CF78]. The first, denoted INVAL, is Censier and Feautrier's invalidation protocol without variation. The second, UPDATE, is Censier and Feautrier's scheme modified so that writes to shared writeable data send updates to all other cached copies. Like INVAL, UPDATE requires the maintenance of copy lists at the main memory. Unlike INVAL, we assume that writes to shared writeable data write through the cache. Update requests are therefore issued to the main memory on each shared write. This is the simplest implementation of an UPDATE protocol, and optimizations may reduce the amount of write-through traffic. Simulation results, however, suggest that update traffic from the main memory to cached copies dominates write-through traffic, so any traffic reduction with a write-back scheme would be minimal. The third coherence scheme, COMPk, is a competitive protocol in which updates are issued to cached copies until an invalidation criterion is satisfied, at which point the copy is invalidated. In COMPk the invalidation criterion is checked at each cache by counting, for each cache block, the number of updates received between references by the local cpu. When the number of updates reaches a threshold, k, the cache invalidates its copy. By varying k from 1 to ∞, a set of protocols with behavior ranging from that of INVAL to that of UPDATE can be constructed. COMPk is similar to the bus-based competitive protocol described in [K$^+$86]. Like UPDATE, we assume write-through caches for shared-writeable data.

3.3.4 Stack Simulation of Directory-based Coherence Protocols

Thompson's stack simulation algorithm assumes an idealized shared bus architecture with zero miss penalty. With the same assumptions, the same algorithm can be used for directory-based coherence protocols that fall into the MOESI class. The invalidation and update protocols described in Section 3.3.3 correspond to the MOESI protocols in Tables 3.6 and 3.7 in the previous section, and hence can be evaluated using stack simulation. In this section it is shown that the competitive protocol can also be evaluated with stack simulation. It is also shown how simulation statistics peculiar to directory schemes can be efficiently collected.

On first inspection, the competitive protocol falls outside of the MOESI class because self-invalidations (flushes) are initiated based on feedback from the coherence system: the number of updates since the last processor reference. It is still as a MOESI protocol, however, because the self-invalidation criterion obeys inclusion: the number of updates observed by block i in a cache of size C is the same as the number observed by block i in a cache of size $C + 1$. This can be shown by induction. At the beginning of a simulation it is trivially true since no blocks are valid. Assume it is valid at time t. Consider the actions that modify the update count for a block: a local reference or an update from the network. On a local reference the update count is set to zero for all cache sizes, so inclusion is preserved. An update from the network increments the update count for all cache sizes in which the block is valid. If the count does not exceed the self-invalidation threshold, inclusion is preserved. If it does exceed the threshold, the block becomes invalid in all cache sizes so inclusion is still preserved. The self-invalidation criteria therefore obeys inclusion, so the competitive protocol can be simulated as a MOESI protocol, with modifications to support update counts. An extra variable, $updates_c$, must be maintained per cached block to store update counts. $updates_c$ for block b must be set to zero for each reference to b at cache c. The write routine for the competitive protocol is shown in Figure 3.9. It is identical to the routine for the update protocol, with the exception that the update counters of cached copies must be incremented as they are updated. When an update count exceeds the self-invalidation threshold, the cached copy is invalidated.

```
Statistics(READ_BLOCK_UPDATE, $c$, $m$, 1, $v_c$);

for $\{j \mid v_j \neq \infty, j \neq c\}$ {
    Statistics(UPDATE, $m$, $j$, $v_j$, $\infty$);
    $updates_j = updates_j + 1$;
    if ($updates_j > k$) {
        invalidate copy;
    }
}
```

Figure 3.9: Write Routine for Competitive Protocol

Because directory schemes issue invalidations or updates on a point-to-point basis, there are a number of of additional traffic statistics that are of interest. These are:

1. invalidations (to main memory): These can be found by counting the number of *write hits to modified blocks*. This count obeys inclusion because validity is inclusive and dirtiness is inclusive. The number of invalidation requests sent to main memory (that do NOT require a block to be fetched) is then the number of unqualified write hits minus the number of write hits to modified blocks.

2. retrievals (from caches with dirty copies): since dirtiness is inclusive, these can be counted directly. The retrieval count corresponding to region d in cache DC is incremented.

3. invalidations (to caches): These can be counted directly since an invalidation in a cache of size C implies an invalidation in all larger caches. When a block is invalidated in the stack for cache j, the invalidation count for that position in stack j is incremented. This statistic only applies to blocks that are not dirty in any cache. If a block is dirty in some cache, it may be possible for the block to be clean for some sizes and dirty for others. This arises when $v_c < d$, which arises in write-back caches when a dirty block is pushed from a small cache and later read again, but not written. Invalidations for sizes v_c to d can be counted using the retrieval count (Item 2 above) plus an additional count of *dirty invalidations* at each stack region i, denoted $dinv_i$. Whenever a retrieval is made from cache DC, the dirty invalidation counter for region v_c is incremented. Unfortunately, the dirty invalidation count does not provide the number of dirty invalidations directly. This is because the count of dirty invalidations does not obey inclusion: sizes v_c to d receive invalidations, while sizes d to ∞ receives retrievals. The following discussion shows that the number of dirty invalidations to cache of size C is just the sum of $dinv_i - r_i$ for all inclusive stack regions i, where r_i is the count of retrievals at region i. To see this, consider Figure 3.10. Assume that separate counters for $dinv_i$ and r_i are maintained for each shared reference j. Denote these counters $dinv_i^j$ and r_i^j, respectively. Figure 3.10 shows, for a particular cache, $dinv_i^j$ and r_i^j for each stack region. Consider a single reference, say 1. It is clear that the number of dirty invalidations for region k is $\sum_{i=1}^{k}(dinv_i^1 - r_i^1)$. The total number of dirty invalidations $dinv$, due to all references, is just

$$dinv = \sum_{j=1}^{n_{refs}} \sum_{i=1}^{k}(dinv_i^j - r_i^j) \tag{3.1}$$

But

$$\sum_{j=1}^{n_{refs}} \sum_{i=1}^{k} dinv_i^j = \sum_{i=1}^{k} dinv_i \tag{3.2}$$

and

$$\sum_{j=1}^{n_{refs}} \sum_{i=1}^{k} r_i^j = \sum_{i=1}^{k} r_i \tag{3.3}$$

therefore:

$$dinv = \sum_{i=1}^{k} (dinv_i - r_i) \tag{3.4}$$

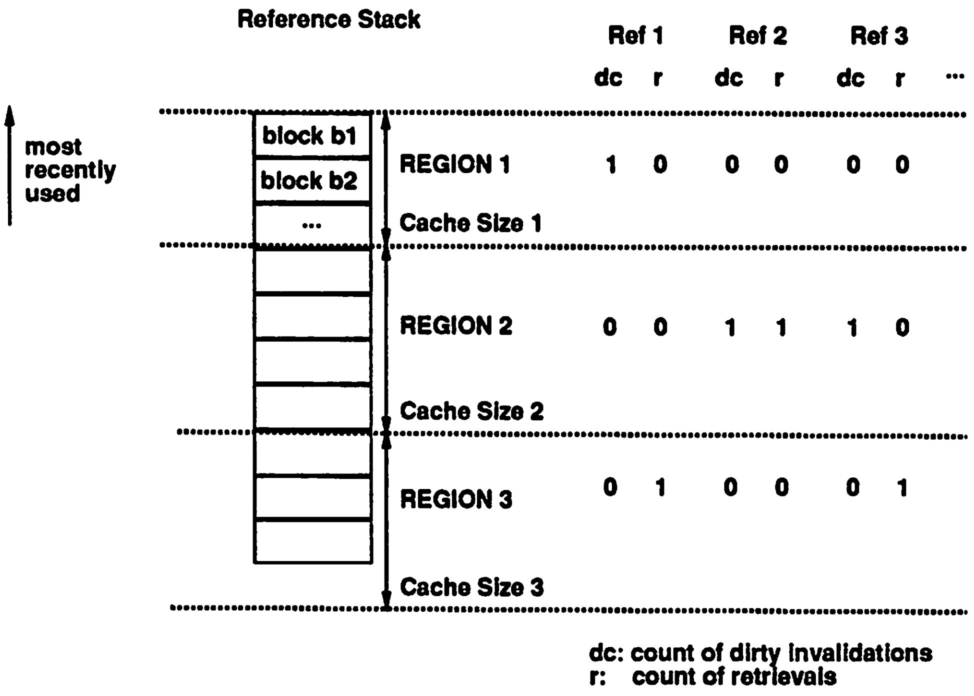


Figure 3.10: Counting Invalidations When a Block is Dirty

4. updates (to caches): These can be counted directly since updates must obey inclusion. An update to a block in stack i causes the update count for the corresponding stack region to be incremented.

One additional metric is collected for the competitive protocol: the number of self-invalidations at each cache. Since update counts are inclusive, the count of self-invalidations is also inclusive, so it can be counted directly.

3.4 Evaluating Competitive Directory Methods

In this section it is shown how the results of a single UPDATE simulation can be used to find the performance of COMPk for different threshold values. The technique is based on the cumulative distribution of the number of updates between references at a cache (Figure 3.11). Because of its similarity to the notion of a write-run, an uninterrupted series of external updates to a cache block is called an *update-run*. Update-runs supplement the ping/cling and write-run models by measuring the number of copies of a block involved in invalidations and updates. Given the distribution of update-run lengths, it is possible to determine the fraction of references that would be misses if cached copies were invalidated every k updates (with no intervening references by the corresponding cpu). Similarly, the fraction of update traffic attributed to updates occurring at the j'th position in an update-run permits a determination of the number of updates that would not take place for a given k.

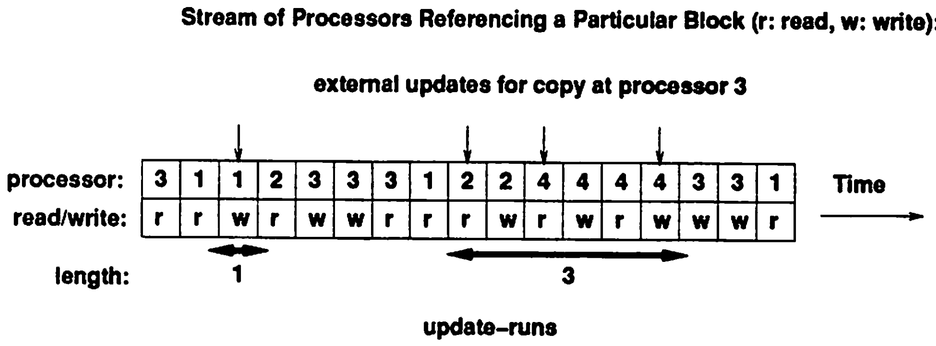

Stream of Processors Referencing a Particular Block (r: read, w: write):
external updates for copy at processor 3
processor: 3 1 1 2 3 3 3 1 2 2 4 4 4 4 3 3 1
read/write: r r w r w w r r r w r w r w w w r
Time
length: 1 3
update-runs

Figure 3.11: The Update-run Model

Figure 3.12 shows a typical plot of the normalized cumulative distribution of the number of references (to shared writeable data) for which k updates were observed since the last reference to the same block by the same cpu. Let the fraction of references finding k intervening updates be $r(k)$. References with zero intervening updates are not included. Let $r_c(k)$ be the cumulative version of $r(k)$: $r_c(k)$ equals the fraction of references finding k or fewer intervening updates. $r(k)$ and $r_c(k)$ are obtained from a single simulation of the UPDATE protocol. Figure 3.13 shows the fraction of update transactions (from main memory to caches) that occur at the k'th position in an update-run. Let the fraction of updates that occur in the k'th position be $u(k)$. $u(k)$ is related to $r(k)$ by the equation:

$$u(k) = \frac{\sum_{i=k}^{\infty} r(i)}{\sum_{i=1}^{\infty} ir(i)} \tag{3.5}$$

Let $u_c(k)$ be the cumulative version of $u(k)$.

The performance of COMPk for threshold k is found by adjusting the miss ratio and traffic of UPDATE to account for the extra misses and fewer update transactions:

$$extra\_misses = refs * f_{nz}(1 - r(k-1)) \tag{3.6}$$

$$updates\_avoided = mupdates * (1 - u(k)) \tag{3.7}$$

where:

- $refs$ is the number of references to shared writeable data.
- $mupdates$ is the number of MUPDATE transactions received by the cache with a full update protocol.
- f_{nz} is the fraction of references that have one or more updates since the last reference to the same block.

The extra misses are the number of data references with update-runs of length k or greater: any cache block that receives more than $k-1$ updates is invalidated. The number of updates avoided is the fraction of updates (in a full update scheme) corresponding to positions $k+1$ or greater in an update-run.

Because they do not account for changes in replacement behavior caused by self-invalidations, these formulae are exact only for infinite caches. Furthermore, if $r(k)$ and $u(k)$ are determined from a subset of a program trace, they must be adjusted to account for the following end effects:

1. Portions of update-runs that occur prior to the start of steady-state. This data can be obtained from the transient portion of the trace during which steady-state is established.
2. Updates to blocks that are not re-referenced in the sampled time interval. The effect of these on $u(k)$ can be found by scanning the caches at the end of the simulation.

Numerical results for this technique are presented in Section 3.6.2.

3.5 Simulation Methodology

The stack simulation techniques of Section 3.3 were applied to three benchmark programs to obtain miss ratios and network traffic for the INVAL, UPDATE and COMPk coherence schemes. In this section the metrics, simulated architecture, and benchmark programs are described.

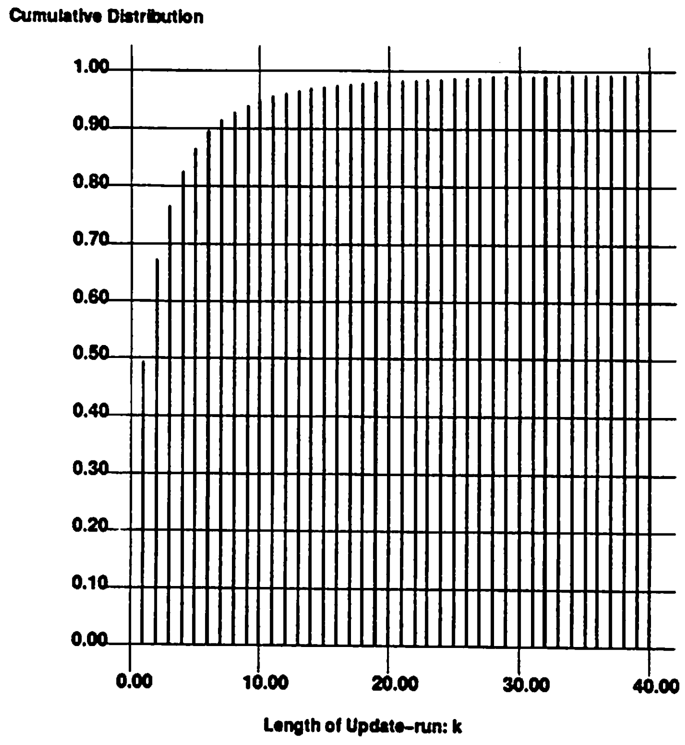


Figure 3.12: Cumulative Distribution of Update-run Length

3.5.1 Metrics

The metrics of the comparison are cache miss ratio and network traffic. Miss ratios include misses due to references to shared writeable and non-shared-writeable data; they do not include instruction misses. Miss ratios reported in Section 3.6.1 are the average of the miss ratios for all simulated processors.

Network traffic is given in bytes of data sent into the network by a cache controller, normalized by the number of instructions referenced at that cache. Cache controllers communicate with main memory banks using *transactions* composed of two point-to-point network messages: a request and a response. Tables 3.8 and 3.9 show the transaction types and their sizes for each coherence scheme. Traffic values in Section 3.6.1 are the average of the traffic values for all simulated processors.

3.5.2 Simulated Architecture

The different coherence schemes were compared using instruction-level simulations of a multiprocessor architecture with separate instruction and data caches, and an idealized main mem-

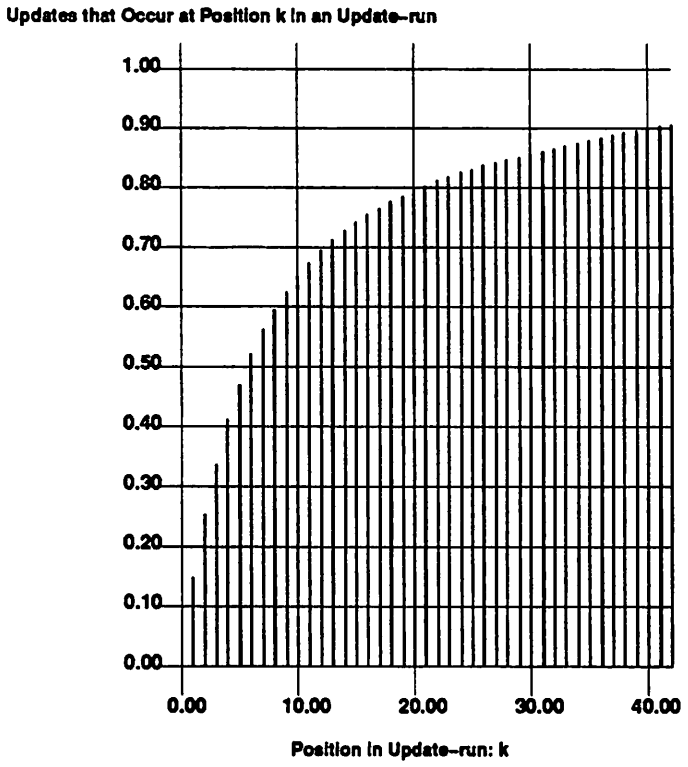


Figure 3.13: Fraction of Updates that Occur at *k*'th Position in an Update-run

Table 3.8: Transaction Types

Name [a]	Description	Format [b]					
		Request			Acknowledge		
		I	U	C	I	U	C
CPUREAD	get shared block to read	1	1	1	5	5	5
CPUWRITE	get shared block to write	1	3	3	5	5	5
CPUUNSHARED	get unshared block	1	1	1	5	5	5
DISPLACE	displace clean block	1	1	1	4	4	4
WRITEBACK	displace dirty block	2	2	2	4	4	4
INVAL	invalidate copies	1	-	-	4	-	-
UPDATE	update copies	-	3	3	-	4	4
LOCK	lock	1	1	1	4	4	4
UNLOCK	unlock	1	1	1	4	4	4
BARRIER	wait at barrier	1	1	1	4	4	4
MREAD	get cache copy to read	4	-	-	5	-	-
MWRITE	get cache copy to write	4	-	-	5	-	-
MINVAL	invalidate cache copy	4	-	-	4	-	-
MUPDATE	update cache copy	-	6	6	-	4	4

[a]transactions beginning with "M" are from main memory to caches; all others are from caches to main memory
[b]I: INVAL, U: UPDATE, C: COMPk; numbers correspond to formats in Table 3.9

Table 3.9: Transaction Formats

Number	Format	Size (bits) [a]
1	(trans)(addr)(src)(dest)	92
2	(trans)(addr)(src)(dest)(block)	124,220,604
3	(trans)(addr)(src)(dest)(word)	124
4	(trans)(addr)(src)	82
5	(trans)(addr)(src)(block)	114,210,594
6	(trans)(addr)(src)(word)	114

[a]assuming: (trans) is 8 bits, (addr) is 64 bits, (src) and (dest) are 10 bits, (word) is 32 bits, (block) is 32, 128 or 512 bits

ory. An accurate model of the MIPS R2000 cpu was used to drive the two level memory hierarchy. The cpu model accurately modeled pipelining and interlocks in the integer and floating point arithmetic units. The main memory was "ideal" in that it was assumed to have zero latency and presented no serialization to simultaneous accesses by multiple processors. This assumption was required by the stack simulation algorithm.

Synchronization operations were restricted to operations on locks and barriers. The ideal main memory provided support for the high-level synchronization operations LOCK(lock_variable), UNLOCK(lock_variable) and WAIT_BARRIER(barrier_variable). Locks and barriers were handled by appropriately stalling processes at the main memory and maintaining queues of stalled process identifiers for each lock and barrier. The main memory incurred no serialization delay for simultaneous queueing operations; delay was incurred, however, for processes waiting in queue for access to a lock or for the completion of a barrier. This particular synchronization model was used so that cache performance would not be biased by poor synchronization support. These ideal assumptions are justified by the detailed discussion of synchronization techniques in Chapter 5.

When measuring network traffic, it was assumed that the main memory was interleaved and distributed among the processors. It was also assumed that one cache controller and one memory bank shared a network port. Any transactions issued from a block of main memory was thus counted with transactions issued from its respective cache controller. Interleaving was performed using bit selection on the lowest order bits of a block address.

Since all memory references were satisfied in one cache cycle, independent of whether there was a hit or miss, the system under study was sequentially consistent. The results should also apply, however, to multiprocessors with weaker coherence standards.

64 processors were used for all simulations. It was assumed that for each benchmark that the entire program could reside in main memory.

The simulated caches were fully associative with LRU replacement, using demand fetch with fetch-on-write, and write-back (except for writes to shared writeable data in UPDATE and COMPk, which were write-through).

3.5.3 Benchmarks

Table 3.10 shows some characteristics of the benchmarks used in the comparison. The three benchmarks are:

1. VERF: a program that checks the equivalence of two Boolean networks;

Table 3.10: Benchmark Characteristics

Benchmark	Instruction[a] References	Data References[b] Shared	Non-shared	% Writes (Shared)	Utilization	Data Memory[c] Touched
verf	2,070,000	332,000	219,000	7.01%	69%	609kB
ugray	3,540,000	359,000	956,000	1.33%	89%	2.27MB
locus	10,600,000	399,000	3,330,000	7.2%	98%	1.76MB

[a]per cpu
[b]per cpu
[c]touched by all processors

2. LOCUS: a router for standard cell layouts;

3. UGRAY: a ray tracer;

The reference counts of Table 3.10 are on a *per child* basis, and only include references after steady-state [2] was reached. Furthermore, all results in Section 3.6.1 are for steady-state behavior. The caches were considered to be in steady-state when one of the following conditions was met:

1. The average working set is loaded. Here the size of the working set is defined as the minimum cache size for which the average steady-state miss ratio is within 10% of that observed for an infinite cache.

2. The average amount of valid data per cache is greater than or equal to the maximum cache size of interest.

Steady-state for VERF was reached using condition 2. Steady-state for UGRAY and LOCUS was reached using condition 1. This is shown in Table 3.11, which shows the average number of valid blocks in a cache when steady-state was established. The results are for 64 processors and the INVAL protocol; results for the other protocols are similar. The table also shows the size of the "working set" for each benchmark. For UGRAY and LOCUS the working set was loaded at the start of steady-state. VERF exhibited much poorer locality, with a working set larger than 256kB. However, the caches in all VERF simulations had at least 64kB of valid data at the start of steady-state.

[2]"Steady-state" cache statistics do not include the effects of loading an empty cache: the cache is loaded at the beginning of the simulation.

Table 3.11: Steady-State Statistics

Benchmark	Block Size	Valid Data (at start of steady state)	"Working Set"[a]	Miss Ratio Cold %	Miss Ratio Warm %
VERF	16B	79kB	∞	7.05	5.47
VERF	32B	115kB	∞	7.17	6.04
VERF	64B	156kB	∞	7.77	6.86
UGRAY	16B	53kB	32kB	2.33	0.586
UGRAY	32B	61kB	32kB	1.36	0.509
UGRAY	64B	76kB	32kB	0.870	0.468
LOCUS	16B	34kB	32kB	1.07	0.486
LOCUS	32B	38kB	32kB	0.789	0.316
LOCUS	64B	43kB	32kB	0.616	0.217

[a]Minimum cache size for which steady-state miss ratio is within 10% of infinite cache miss ratio.

The last two columns of Table 3.11 show cold and warm miss ratios (for all data references). These show the large impact of a cold start[3].

3.6 Numerical Results

The first part of this section compares the UPDATE, INVAL and COMPk coherence schemes for a variety of cache and line sizes. The results are presented in five subsections. The first three examine the three coherence schemes individually. The last two subsections compare their relative performance and examine the effect of varying the invalidation threshold in the competitive protocol. The synchronization behavior of the benchmarks is described in the last part of this section.

3.6.1 Misses and Traffic

Invalidation Protocol

Figures 3.14 and 3.15 show miss ratios and traffic as a function of block and cache size for the invalidation protocol. The UGRAY and LOCUS benchmarks have very good locality of reference at even small cache sizes, so increasing the block size has a more pronounced reduction in misses than increasing cache size. Increasing the block size to 64 bytes for a 16kB cache, however,

[3]"Cold start" cache statistics include the effects of loading an empty cache.

causes a sharp increase in misses due to cache pollution [4]. The VERF benchmark has much less locality of reference and exhibits larger reductions in misses as cache size is increased, but suffers from cache pollution and an increase in invalidation misses when the block size is increased beyond 4 bytes.

For all benchmarks traffic falls gradually as cache size is increased. The trends for different block sizes, however, are different for each benchmark. For VERF, traffic consistently increases with increasing block size, due to increases in miss ratio caused by cache pollution and increased invalidation traffic. For LOCUS, traffic consistently decreases as block size is decreased, indicating that most of the extra data fetched on a miss is eventually used. Traffic for UGRAY falls as block size increases from 4 to 16 bytes, but rises as block size is further increased to 64 bytes. For the 16kB cache this is due to cache pollution, as observed in the plot of miss ratios. For larger cache sizes traffic rises because the reduction in misses is not sufficient to offset the higher cost of fetching 64B blocks.

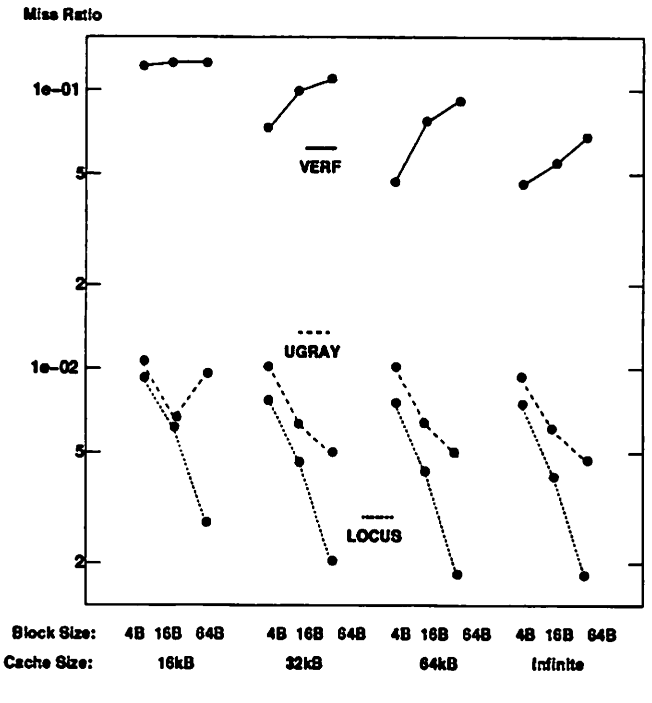


Figure 3.14: Miss Ratios for Invalidation Protocol

[4] At some size, a block becomes so large that more useful data is replaced than is brought in, and the number of misses increases. In such a situation, unused data is said to "pollute" the cache.

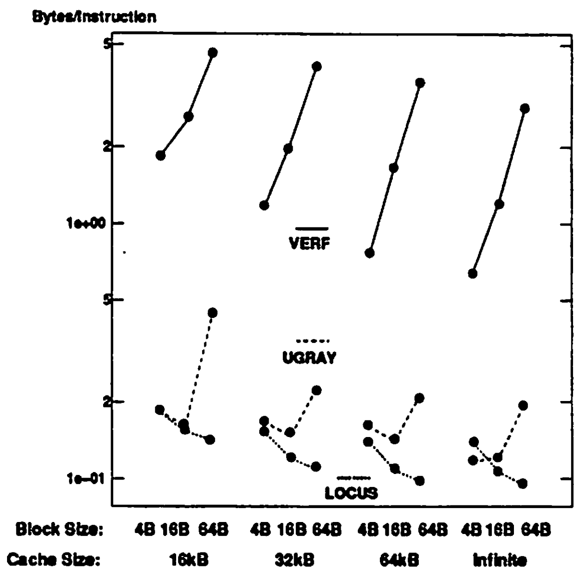


Figure 3.15: Traffic for Invalidation Protocol

Update Protocol

Miss ratios for the update protocol (Figure 3.16) follow trends similar to those of the invalidation protocol, with values up to 5 times smaller. An exception occurs, however, for the VERF benchmark with an infinite cache, in which increasing block size reduces misses substantially. This is expected since cache pollution cannot occur in an infinite cache and misses due to false sharing [5] cannot occur using the update protocol (since no invalidations take place).

Traffic for the update protocol (Figure 3.17) follows a similar trend only for the VERF benchmark. This is because the fraction of update traffic is considerably higher than the fraction of invalidation traffic observed for the invalidation protocol. For the LOCUS benchmark, the relationship between traffic and block size reverses as cache size increases, showing how the contribution of update traffic dominates in large caches. The contribution of update traffic in the UGRAY benchmark remains about the same as cache size increases. As observed for the invalidation scheme, cache pollution causes the sharp increase in traffic for UGRAY with a 16kB cache and 64B block.

[5]False sharing occurs when two or more processors concurrently access unrelated data items that have been put in the same cache block. Although the processors are not sharing data, the coherence system acts as if they are because sharing is determined on a block basis. False sharing causes unnecessary invalidations and update traffic.

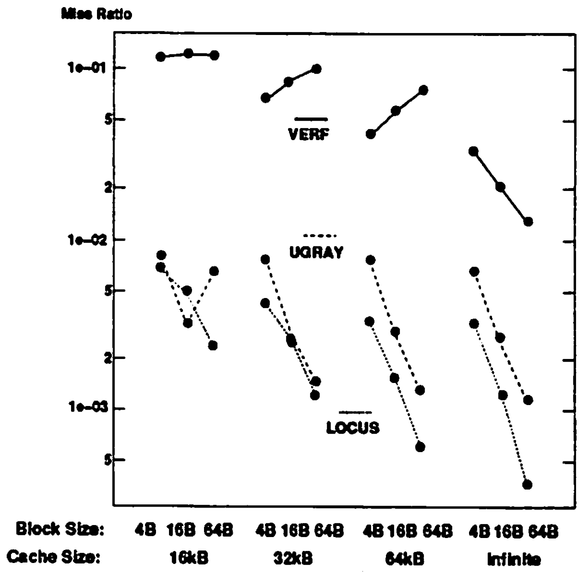


Figure 3.16: Miss Ratios for Update Protocol

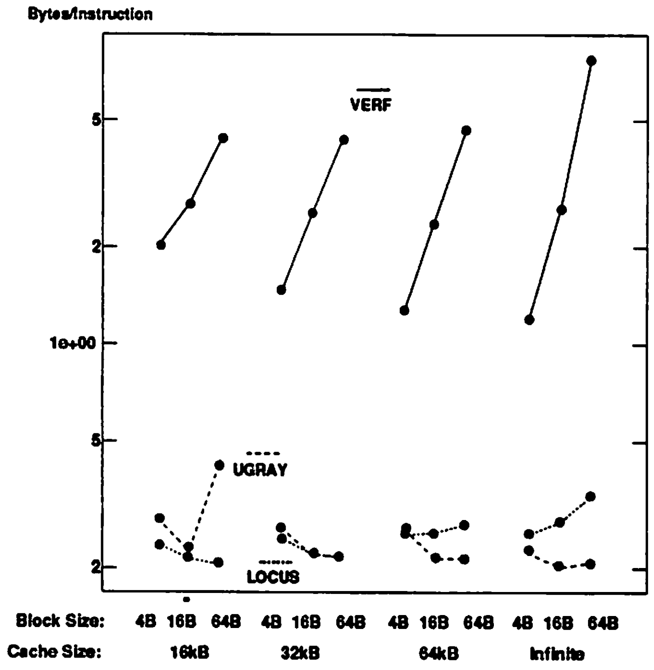


Figure 3.17: Traffic for Update Protocol

Competitive Protocol

As expected, results for the competitive protocol (Figures 3.18 and 3.19) follow trends that are a mix of those observed for the other protocols. These results are for a self-invalidation threshold of 8. Miss ratios and traffic values lie between those of INVAL and UPDATE. Miss ratios follow the trends of the other schemes with the exception of VERF with an infinite cache. Here the miss ratio declines as block size increases from 4B to 16B, as observed for the UPDATE scheme. As block size is increased further to 64B, the number of self-invalidations rises and the miss ratio increases, as observed for the INVAL scheme.

Traffic for COMPk also follows trends that are a composite of the behavior of INVAL and UPDATE. VERF follows the same trends with values between those of INVAL and UPDATE. Traffic for LOCUS decreases from 4B to 16B block sizes, and rises from 16B to 64B block sizes, as observed for INVAL. The presence of updates, however, causes the changes to be less severe since the number of misses is substantially reduced. The results for UGRAY show that, as block size is increased, a reduction in update traffic reverses the increase in traffic observed for UPDATE.

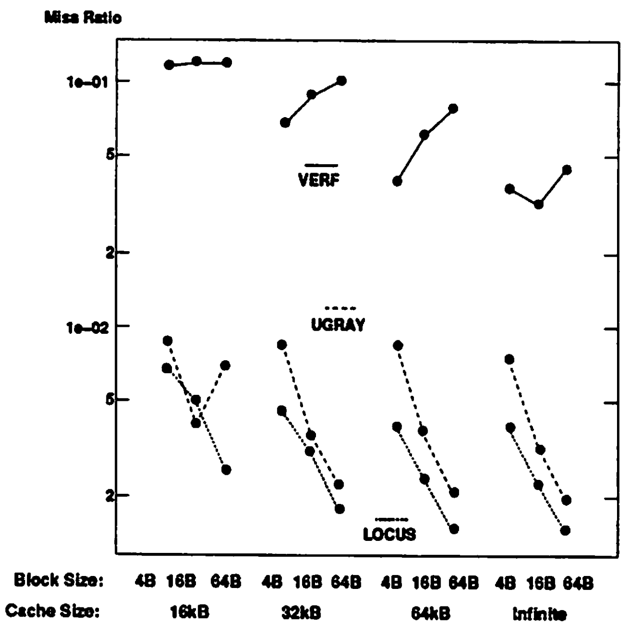


Figure 3.18: Miss Ratios for Competitive Protocol

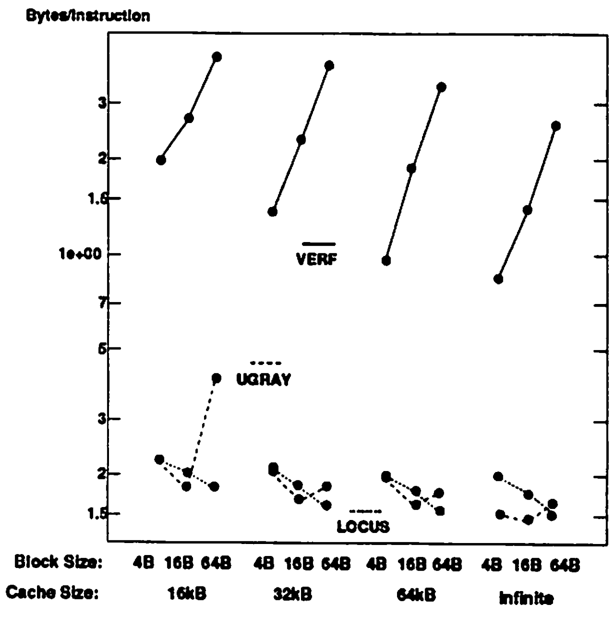


Figure 3.19: Traffic for Competitive Protocol

Comparison

Figures 3.20 and 3.21 show the relative the performance of the schemes assuming a 64kB cache. In almost all cases UPDATE provides a reduction in miss ratio at the expense of an increase in traffic. COMPk results are roughly the average of those for UPDATE and INVAL, although in a couple of examples (VERF, 64B block and UGRAY, 64B block) COMPk has the least amount of traffic. UPDATE and COMPk improve miss ratios the most for large block sizes, which is expected since false sharing increases with block size and causes an increase in invalidation misses for INVAL.

With a 64B block size, which is desirable to minimize directory overhead, the increases in traffic for using UPDATE over INVAL are 31% for VERF, 0% for UGRAY and 282% for LOCUS. The corresponding increases for using COMPk are -5% for VERF, 18% for UGRAY and 56% for LOCUS. By using UPDATE/COMPk in place of INVAL, miss ratios improve by 19%/15%, 74%/57% and 66%/17% for VERF, UGRAY and LOCUS, respectively.

Since all of the miss ratios for UGRAY and LOCUS are less than 1%, reducing them should have little impact on the average number of cycles per instruction. VERF exhibits much higher miss ratios, and the update schemes provide significant reductions for cache sizes larger than

64KB. The benefit of UPDATE and COMPk may therefore be substantial. Absolute multiprocessor performance is considered in detail in Section 4.5, which considers the impact of network delay on processor utilization.

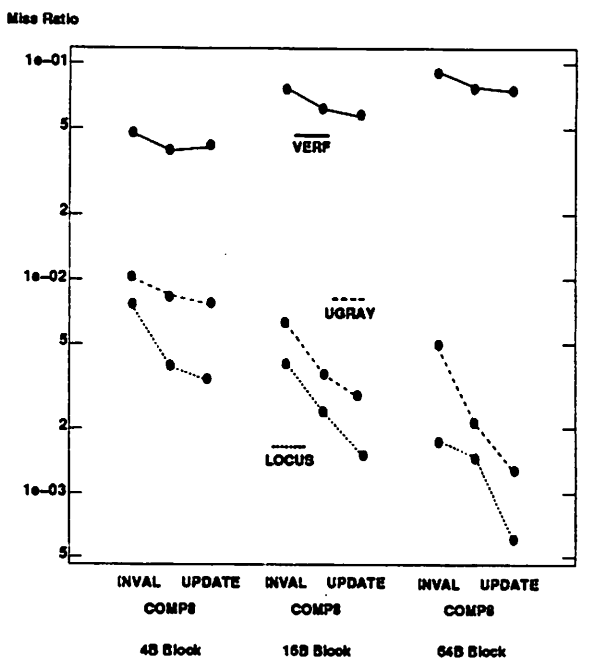


Figure 3.20: Comparison of Miss Ratios for All Protocols

3.6.2 Performance of the Competitive Protocol

Figures 3.22 to 3.27 show miss ratios and traffic for the three benchmarks with infinite caches and three block sizes: 16B, 32B and 64B. Point results for INVAL, COMP8 and UPDATE are shown for reference. As $k \rightarrow \infty$, miss ratios and traffic converge to those measured for UPDATE. As $k \rightarrow 1$, the miss ratios converge to those measured for INVAL. The miss ratios converge because a COMP1 scheme invalidates blocks on the first update. Traffic does not converge, however, because the INVAL protocol sometimes requires blocks to be retrieved from caches with modified copies. The INVAL protocol is also write-back, while COMP1 is write-through.

The figures show that most of the reduction in miss ratio is achieved by $k \approx 10$, at which point traffic has increased to about the average of that for UPDATE and INVAL.

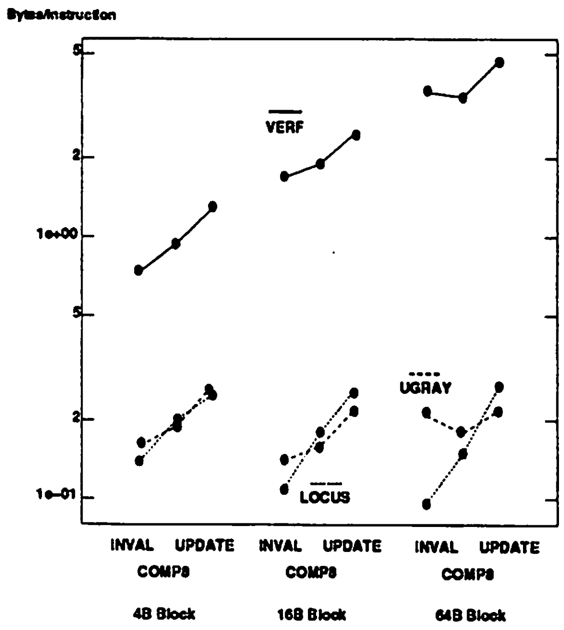


Figure 3.21: Comparison of Traffic (Volume) for All Protocols

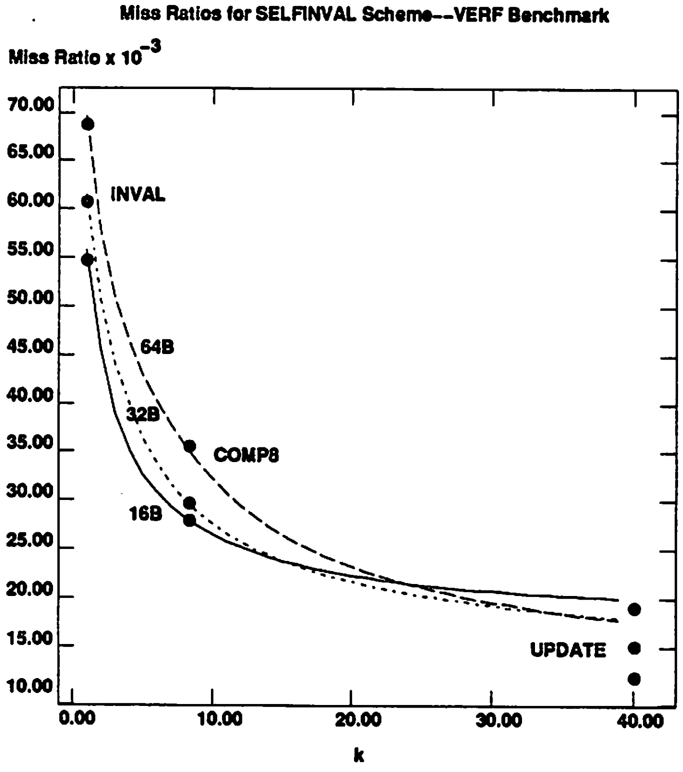


Figure 3.22: Miss Ratios Versus Self-Invalidation Threshold (VERF)

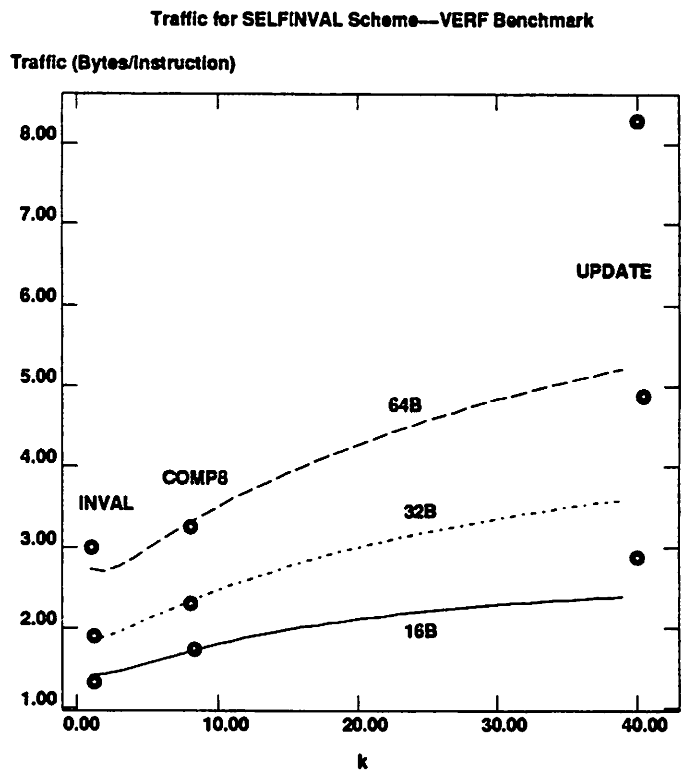


Figure 3.23: Traffic Versus Self-Invalidation Threshold (VERF)

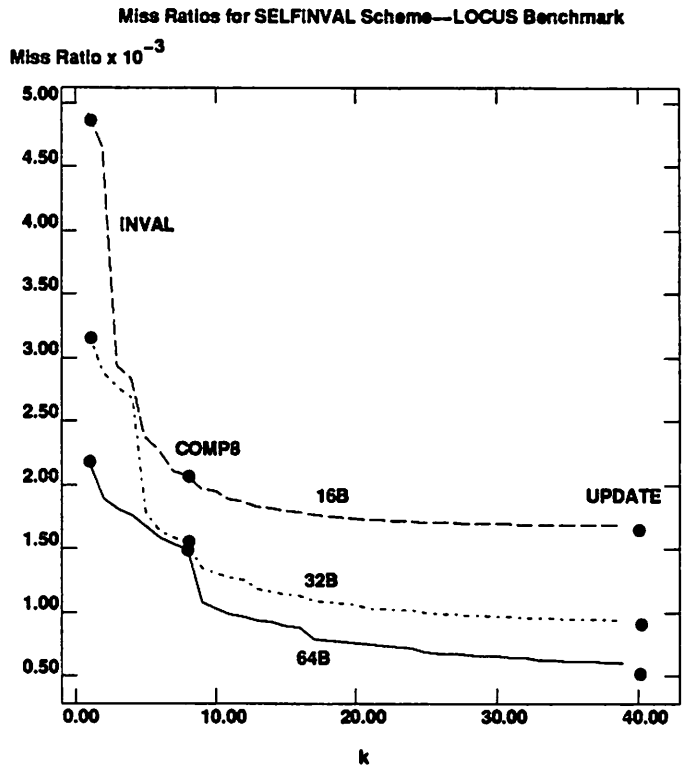


Figure 3.24: Miss Ratios Versus Self-Invalidation Threshold (LOCUS)

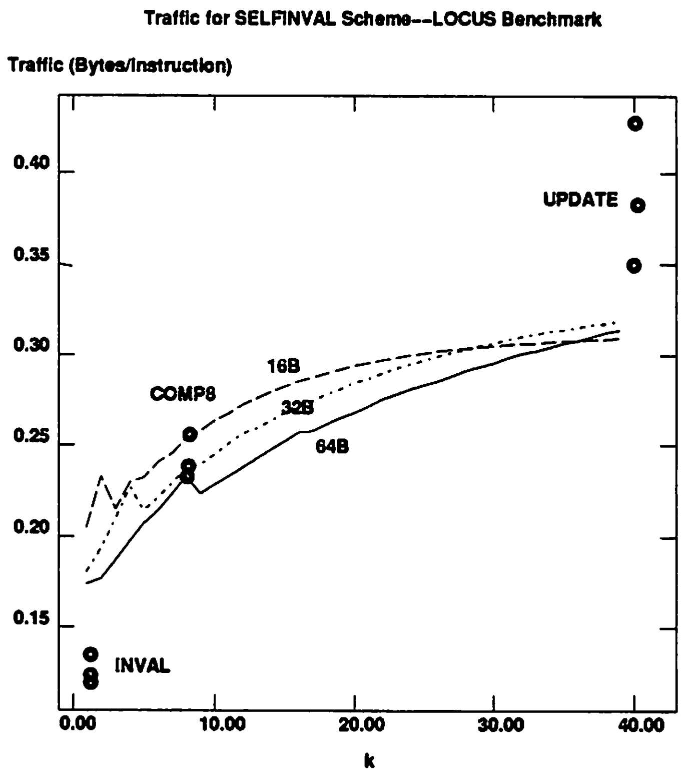


Figure 3.25: Traffic Versus Self-Invalidation Threshold (LOCUS)

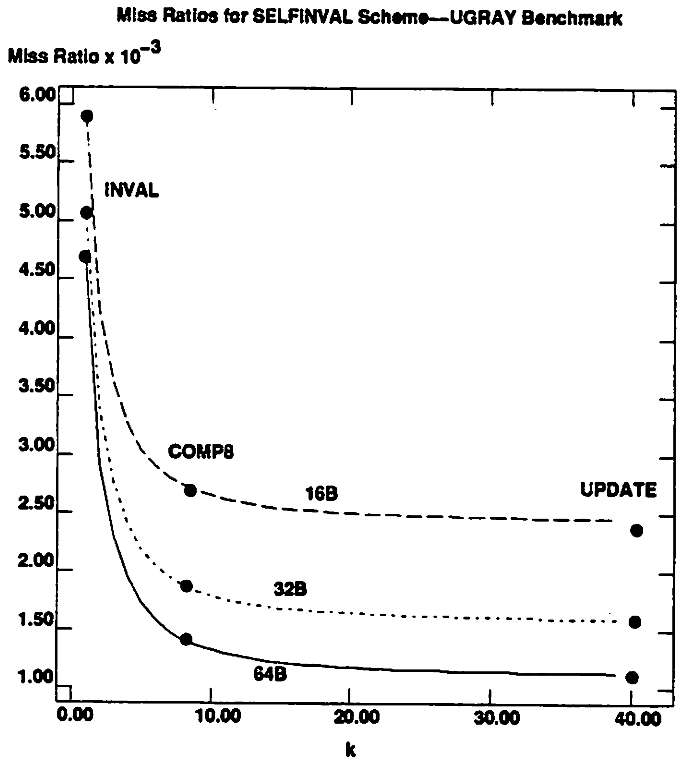


Figure 3.26: Miss Ratios Versus Self-Invalidation Threshold (UGRAY)

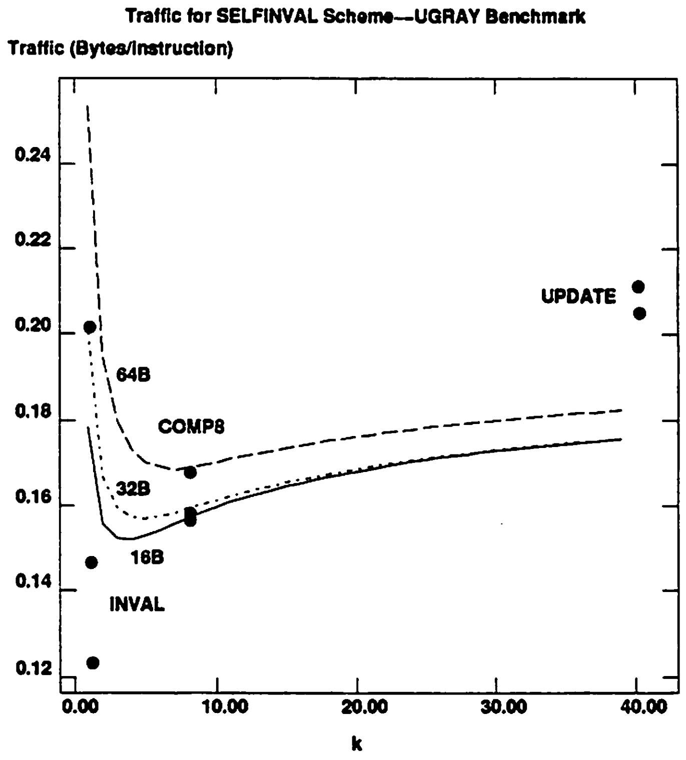


Figure 3.27: Traffic Versus Self-Invalidation Threshold (UGRAY)

3.6.3 Coherence Traffic Versus Multiprocessor Size

If the size of a multiprocessor is increased, the average number of cached copies should also increase if an application has enough parallelism to exploit the processors. This results in an increase in coherence traffic. If the increase is large, the scalability of a directory protocol can be severely limited. Figures 3.28 to 3.30 show the average number of cached copies versus multiprocessor size for the three protocols. The number of copies is the average observed for an invalidation or update request from a cache. The plots show that, to first order, the number of copies grows linearly with the number of processors. As expected, the average number of copies for UPDATE and COMPk are much higher than for INVAL. They are sufficiently high that both UPDATE and COMPk are only appropriate for multiprocessors with 100 or fewer processors. These results were limited to 128 processors, so it is possible that quite different behavior could occur for larger machines.

3.6.4 Synchronization Behavior

The cache simulator was instrumented to collect congestion information for lock and barrier operations. In the simulations no serialization delay was incurred for access to synchronization variables; only serialization due to mutual exclusion was modeled.

A lock was considered to exhibit high contention if the average length of its waiting queue (as observed by an arriving process) exceeded one. The simulations showed that only a few locks experienced heavy congestion, and those that did were readily identifiable because they guarded a frequently used shared data structure. Only 1 of 85 locks in VERF and 2 of 82 locks in LOCUS exhibited mean queue lengths in excess of one (UGRAY did not use locks). In both programs the mutually exclusive operation built with these locks could be more efficiently implemented using a fetch&op operation with combining.

No benchmark made frequent use of barriers, so any barrier congestion was of little consequence.

Since the number of simulated processors was relatively small, and the computation granularity was large, hardware implementations of locks and barriers are not justified for the benchmarks considered here. The congestion observed for a few locks could be avoided by fetch&op with combining, but the improvement in overall performance would probably be minimal. These observations are not surprising because VERF, UGRAY and LOCUS were written to perform efficiently with weak synchronization support.

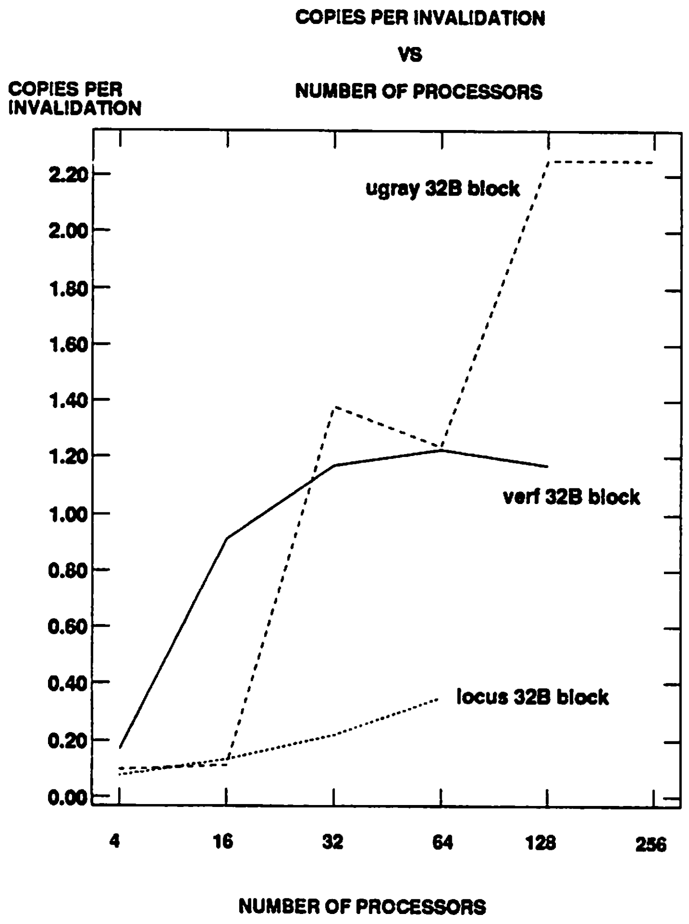


Figure 3.28: Copies Per Inval Versus Multiprocessor Size

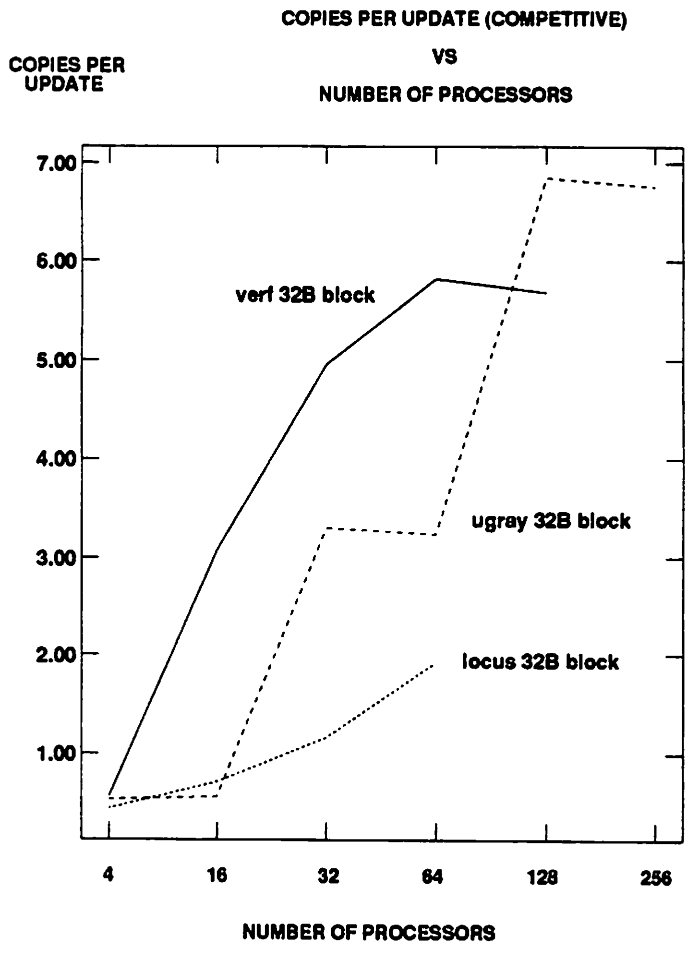


Figure 3.29: Copies Per Update Versus Multiprocessor Size

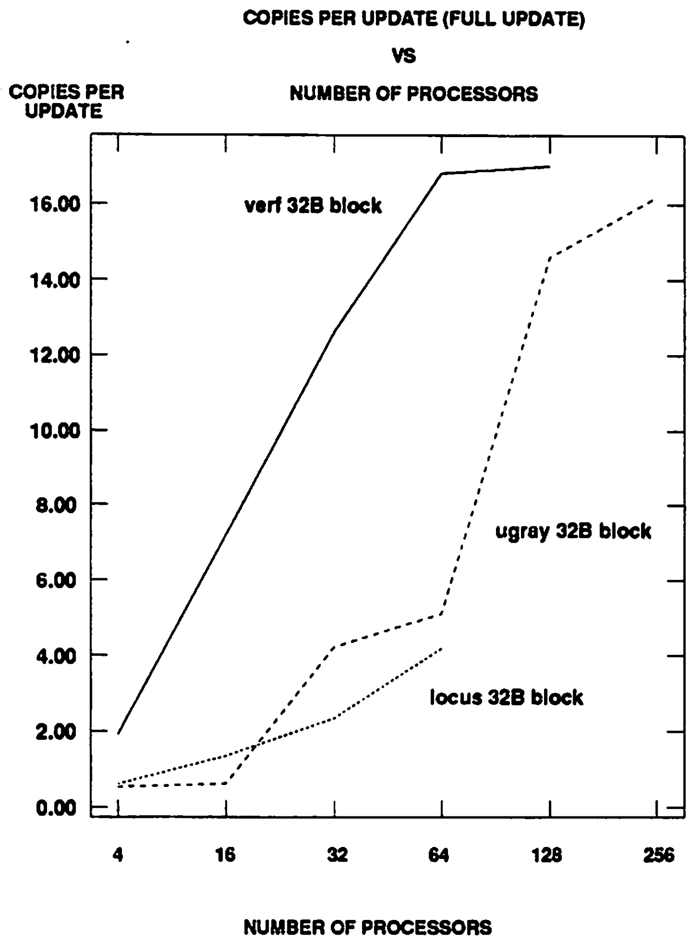


Figure 3.30: Copies Per Update Versus Multiprocessor Size

3.7 Conclusions

In this chapter it has been shown how an efficient stack simulation algorithm for shared-bus coherence protocols can be extended to simulate directory-based coherence schemes. The algorithm substantially reduces the cost of multiprocessor simulation by permitting multiple cache sizes to be analyzed in a single run,

The stack algorithm was applied to update, invalidate and competitive directory protocols for three benchmark programs. The competitive scheme, denoted COMPk, is a simple one in which cache blocks that draw in excess of k updates between references are invalidated. COMPk with k = 8 was simulated, and results for other values of k was found from data obtained from simulations of the update protocol. COMPk performance was determined using a count of *update-runs*: the number of external updates to a cached block between local references. This metric extends the notions of ping/cling and write-run introduced in other locality models.

The simulations show that the update and competitive protocols can reduce overall miss ratios by up to a factor of 3. The reduction in misses comes at the expense of a comparable increase in network traffic. Two of the benchmarks, LOCUS and UGRAY, exhibited overall miss ratios of 1% or less for 64kB or greater cache sizes. Consequently, the improvement in cycles per instruction due to a reduced miss ratio would probably be small. The other benchmark, VERF, had much poorer locality of reference and showed miss ratios between 5 and 10% for 64kB or greater cache sizes. The improvement in cycles per instruction attainable by an update scheme would probably be appreciable only for cache sizes of 128kB or greater.

Estimates of COMPk performance for values of k from 1 to 40 indicate that a broad range of miss ratio/traffic tradeoffs are possible by varying k from 1, in which COMPk is similar to an invalidation protocol, to 10, at which most of the attainable miss ratio reduction is achieved but traffic is still less than that of a full update scheme. This suggests that a good choice for k lies between 4 and 10.

Chapter 4

Network Performance Analysis

4.1 Overview

The stack simulation techniques of Chapter 3 require the assumption that the cache miss penalty (network latency) is zero. This was adequate for estimating miss ratios and traffic, but not for determining average memory access time or, equivalently, processor utilization. In this chapter an analytic modelling technique is presented that permits rapid estimation of the latency and throughput of a broad class of multiprocessor interconnection networks, including k-ary n-cubes, Delta networks, and multidimensional meshes. The relative performance of these networks and the accuracy of the technique are shown via extensive numerical results that are compared to simulation data. In Section 4.5 these results are combined with the cache performance data of Chapter 3 to estimate processor utilization for several protocols on several networks.

In this chapter it is shown that relatively simple interconnection networks can provide good performance for the workloads of interest. The contributions are a powerful analytic modelling technique for multiprocessor networks, and a quantitative comparison of a broad number of alternatives.

4.2 Previous Work

There is a huge literature on the subject of multiprocessor network design and performance analysis, with recent surveys in [RF87, Sie85, DJ81a, Fen81]. This work can be roughly classified into three categories:

1. network design

2. performance analysis via stochastic models and simulation

3. performance analysis via formal properties of specific designs

This section summarizes previous work according to these categories. It concludes with a discussion of the networks considered in this research, and the rationale for their selection.

4.2.1 Network Design

Multiprocessor interconnection networks can be classified according to *topology*, *switching technique*, *flow control scheme* and *routing algorithm*. Virtually all networks of interest are constructed of jxk cross-bar switches, possibly augmented with buffer queues at the inputs and/or outputs (Figure 4.1). A network *topology* is defined by the size of the basic switching elements and

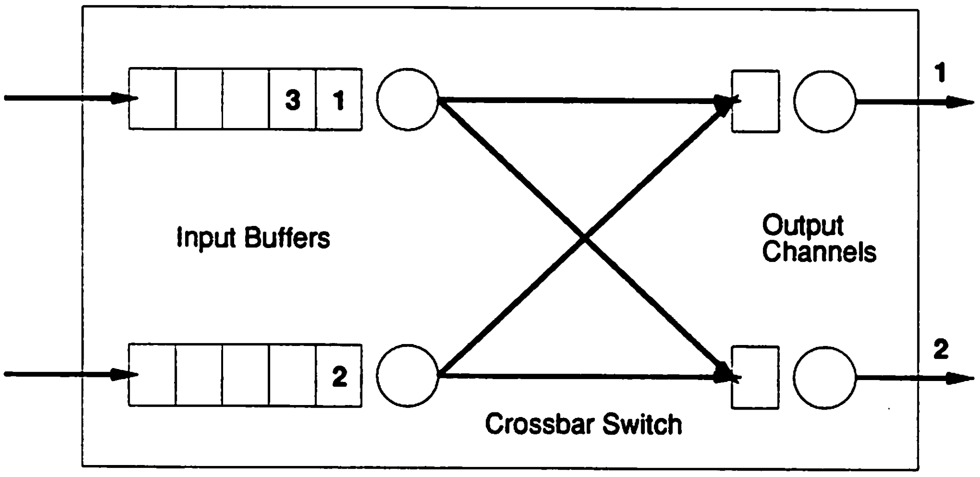


Figure 4.1: Structure of a Network Switch

the way in which they are connected. Much of the literature on interconnection networks concerns different topologies and their properties; surveys are in [Fen81, Sie85, RF87]. Three of the most common topologies are *k-ary n-cubes*, *k-ary n-dimensional meshes*, and *Delta networks*. These topologies are the focus of the numerical results in Section 4.4.

k-ary n-cubes (Figure 4.2) are n-dimensional toroidal meshes with k processors per dimension. Processor locations are denoted by n digit radix k numbers. The neighbors of a processor (the other processors to which it is connected) are those with locations in which one of the n digits differs by one. This class of network includes rings (k-ary 1-cubes) and hypercubes (2-ary n-cubes). k-ary n-cubes for which $k > 2$ can be further distinguished by whether or not two links in opposite directions are provided at each connection (Figure 4.3). Providing two links reduces the minimum latency by about a factor of 2, increases the maximum bandwidth, and supports the combining of synchronization references, as described in Chapter 5.

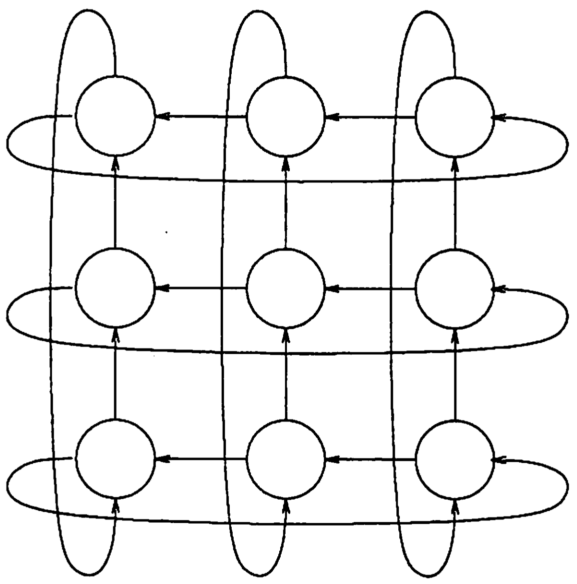


Figure 4.2: Unidirectional k-ary n-cube Network

k-ary n-dimensional meshes (Figure 4.4) are k-ary n-cubes with the "end-around" connections removed. End-around connections are those between processors whose dissimilar location digits are 0 and $k-1$. With end-around connections removed, two links *must* be provided for each connection.

Delta networks [Pat81] are a class of multistage network that includes many common subclasses, including Omega networks and indirect binary cubes [Law75, Pea77]; they are also closely related to the more general class of Banyan networks [GL73]. Delta networks are constructed of n stages of axb crossbar switches, connected in a recursive manner illustrated in Figure 4.5 (taken from [TRH89]). A one stage Delta network is simply a single axb switch. An L-stage Delta network is constructed from a $L-1$ stage Delta networks connected to a new column of b^{L-1} switches. The interconnections are subject to the restriction that all inputs to a particular switch at stage L must come from the same crossbar output at stage $L-1$. An n stage network therefore has a^n inputs and b^n outputs, which are designated by n-digit radix-a and n-digit radix-b numbers, respectively. By construction, Delta networks provide a unique path between each of the inputs and

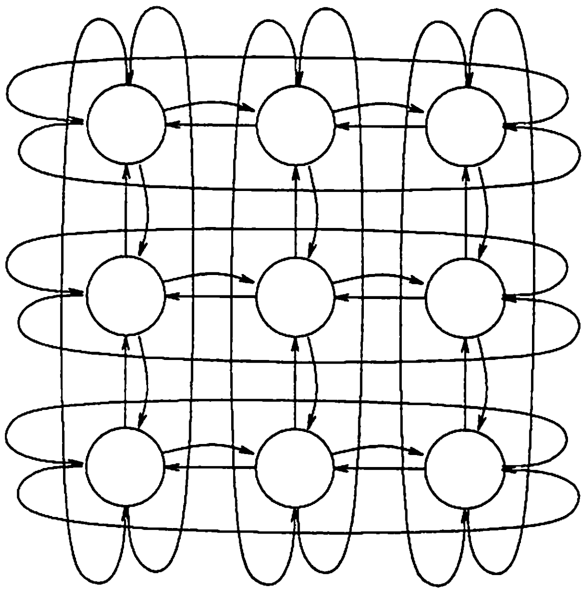


Figure 4.3: Bidirectional k-ary n-cube Network

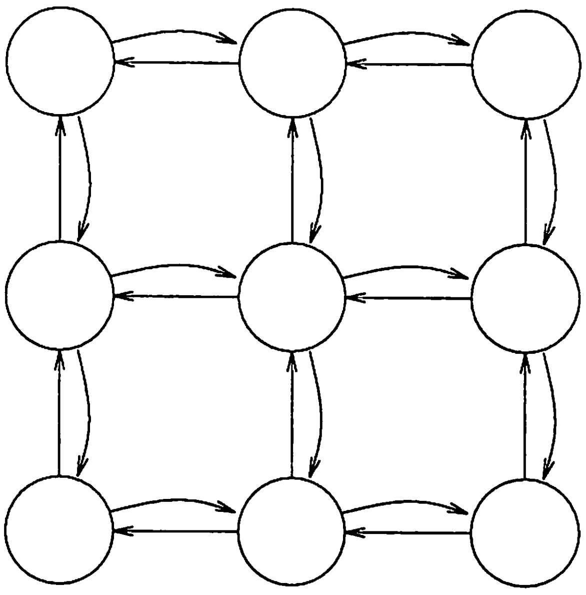
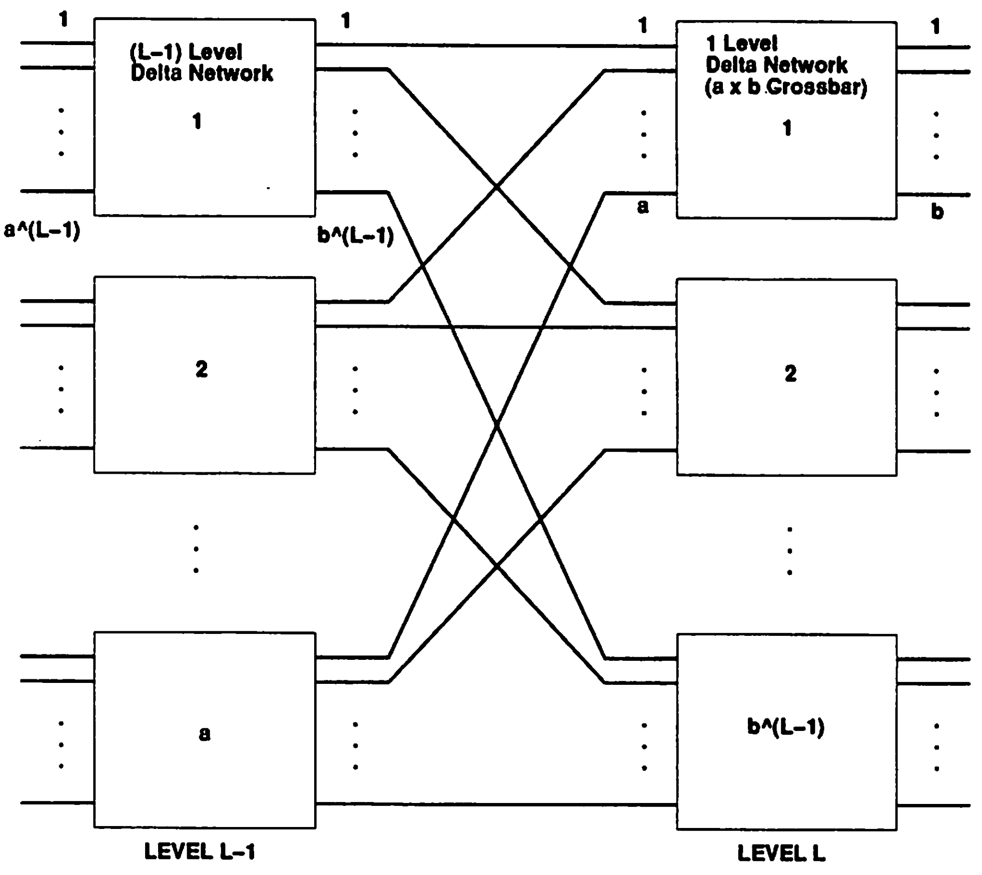


Figure 4.4: k-ary n-dimensional Mesh

outputs, facilitating *digit-controlled* routing. In digit-controlled routing, a packet at stage *i* of the network is sent out the switch output corresponding to the *i*th digit of the destination address. This simple, decentralized routing strategy is an attractive feature of Delta networks.



1
(L-1) Level Delta Network
1
a^(L-1)
b^(L-1)
2
a
LEVEL L-1
1 Level Delta Network (a x b Crossbar)
a
b
2
b^(L-1)
LEVEL L

Figure 4.5: Recursive Structure of Delta Networks

Interconnection schemes are further classified as *single stage* or *multi-stage*. A network topology is *single stage* if each cross-bar switch is directly connected to at least one processing element. A network topology is *multi-stage* if it is not single stage. k-ary n-cubes and k-ary n-dimensional meshes are therefore single stage, and Delta networks are multistage.

There are two common network *switching* techniques: *circuit-switching* and *packet-switching*. In a *circuit-switched* network, a complete path from source to destination is established and held for the entire time during which a packet is transferred. Messages in a *packet-switched* network are buffered at the switches, so only partial paths are established and held. Circuit-switched networks are attractive where low latency is desired and the required bandwidth is low. Packet-switched networks are desirable when high bandwidth is required, because the buffering at switches permits the network to be pipelined. Packet-switched networks are also desirable when the *combining* synchronization technique (described in Section 1.2.2) must be supported.

With a packet-switching scheme, there are three common ways in which packets can be buffered: *store-and-forward, virtual cut-through* and *wormhole.* Figure 4.6 introduces some terminology (due to [DS87]) used to describe the buffering strategies. *Messages* are broken up into *packets,* which are the smallest blocks of data for which routing information is maintained. *packets* are composed of *flits,* which are the smallest blocks of data for which flow control is maintained. *flits* are composed of *phits,* which are the size of the physical channels connecting the switches.

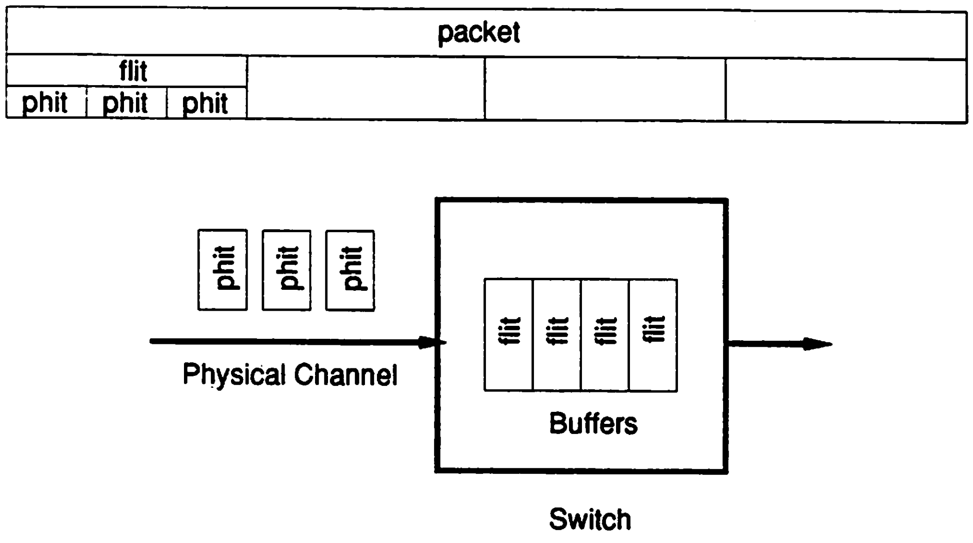


Figure 4.6: Packet-switching Terminology

In a *store-and-forward* packet-switching scheme, complete packets are buffered at switches such that transmission to the next switch cannot begin until the complete packet has been received and stored. In a *virtual cut-through* packet-switching scheme, the latency seen by a packet at a switch is reduced by permitting the packet to be transmitted to the next switch after the first flit is received. In both schemes, a packet can be forwarded only if the destination switch has enough buffers for an entire packet. *wormhole* packet-switching is similar to cut-through, but only requires the allocation of enough buffer space for a single flit before forwarding a packet.

Regardless of the buffer management (*flow control*) scheme that is used, network performance can be substantially improved by breaking buffer queues into multiple *virtual channels* at each physical input or output port in a switch [Dal90c]. This technique is effective at reducing a component of blocking delay illustrated in Figure 4.1. Figure 4.1 shows a typical switch implementation with one buffer per input. If a first-in first-out (FIFO) queueing discipline is used at the input queues, a customer at the head of the queue can block other customers that, if they were at the head, would not be blocked. Figure 4.1 shows an example of such a situation. Here customer 1 (destined for output 1) is blocked because customer 2 at the other queue has been granted access to output 1. Customer 3, however, is destined for output 2, and could be sent to output 2 during this switch cycle if it could bypass customer 1. Simulation studies have shown that this phenomena can significantly reduce the maximum throughput of the switch [Dal90c, TF88]. Figure 4.7 shows how congestion of this nature can be avoided by adding *passing lanes* [Dal90c] at the outputs or inputs. These extra lanes are denoted *virtual channels*. Adding extra lanes reduces the frequency with which the contention illustrated in Figure 4.1 occurs. The cost of extra lanes is extra interconnect within the switch; the size of the cross-bar switch can remain the same, with each input port multiplexed among its virtual channel queues [Dal90c].

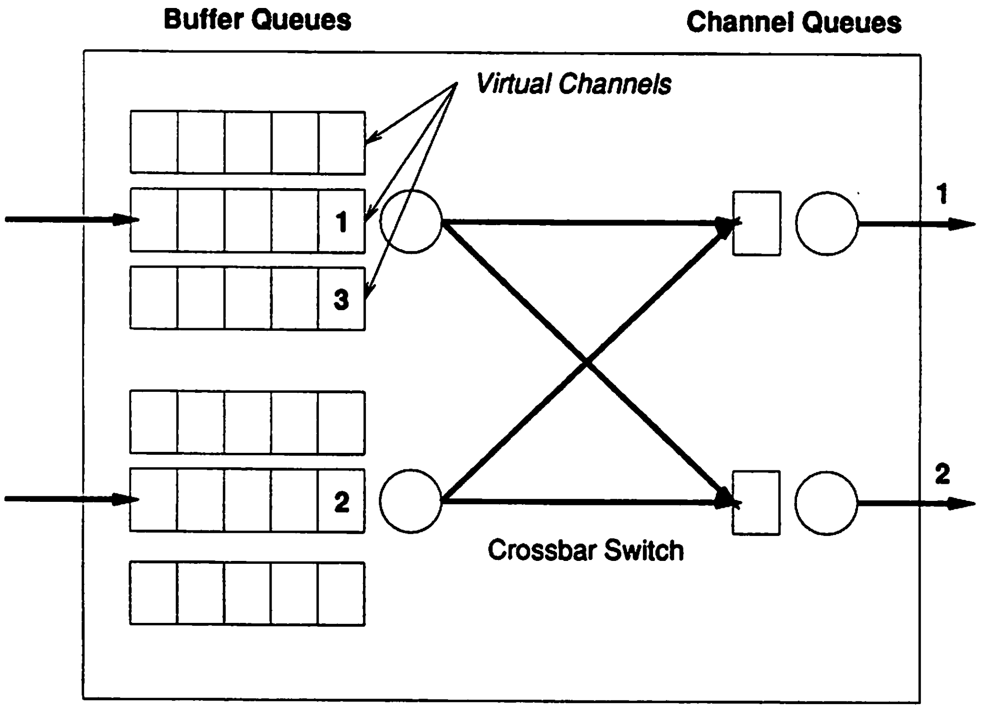


Figure 4.7: A Network Switch with Virtual Channels

A third characteristic of a network is *routing algorithm*, for which there are two classes: *oblivious* and *adaptive*. An *oblivious* routing scheme is one in which the route taken by a packet is determined solely by the packet's source and destination. An *adaptive* routing scheme uses the source and destination plus knowledge about the state of the network, such as the number of packets in buffers. *oblivious* routing schemes are simpler than *adaptive* schemes and ensure in-order transmission of packets between two processing elements. By exploiting extra knowledge about the network, however, *adaptive* schemes offer potentially better performance. The extra performance has a cost: stable, deadlock-free adaptive routing schemes are generally more difficult to design and analyze. It is also more difficult to design combining schemes that work with adaptive routing algorithms.

The network models in this chapter are restricted to oblivious routing. Examples of oblivious routing algorithms include:

1. *digit* routing in Delta networks, as described above.
2. e-cube routing ([SB77]): The e-cube routing algorithm is an oblivious routing algorithm for binary hypercubes. Let N be the number of processing elements in the system. Each of the N switches in a hypercube has $1 + \log N$ output links, where links 0 to $\log N - 1$ go to other switches and link $\log N$ goes to the processing element at that switch. In e-cube routing, a packet with destination d at switch i is sent to the output link corresponding to the most significant bit that differs between i and d; if no bits differ, the packet has reached its destination. Since the algorithm does not depend on the state of the network, it is oblivious.
3. Routing in k-ary n-cubes ([DS87]): Routing in k-ary n-cubes is a generalization of e-cube routing. Here routing is performed in order of decreasing dimension $(k - 1, k - 2, ...1, 0)$ by comparing the k digits of the radix-k addresses of the packet destination and switch. A packet with destination d at switch i is sent along the highest dimension at which d and i differ; when $d = i$, the packet has arrived at its destination. To prevent deadlock, the buffers for each input at a switch must be divided into two virtual channels to prevent cyclic dependencies among buffers at different switches. Virtual channel 0 is taken when $d < i$, and channel 1 otherwise.

k-ary n-cubes require virtual channels for both deadlock avoidance and congestion reduction. For clarity, channels used to prevent deadlock will be referred to as virtual channels, and channels used to reduce congestion will be referred to as *sub-channels* (Figure 4.8). As Figure 4.8 shows, a virtual channel may be made up of several sub-channels.

4.2.2 Performance Analysis Via Simulation and Stochastic Modelling

Until recently, few examples of large scale multiprocessors or programs existed, so performance analysis of multiprocessor networks was typically done using synthetic workloads. With

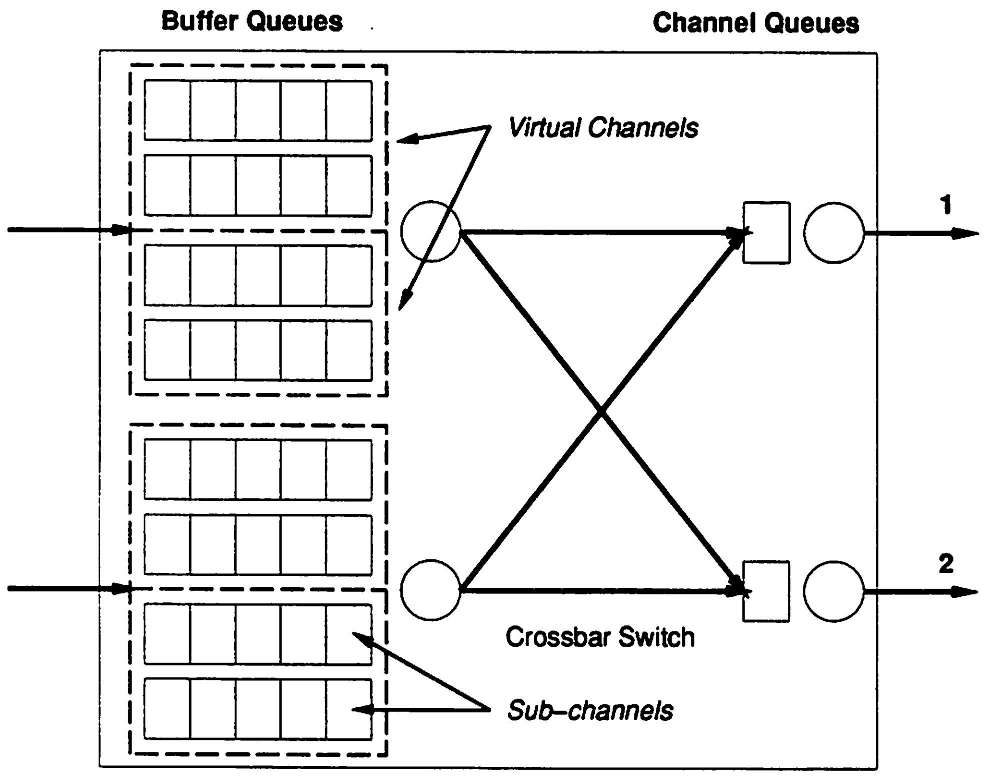


Figure 4.8: Virtual Channels and Sub-channels

Table 4.1: Published Network Performance Studies

Study	Network	Technique	Features
Kruskal/Snir [KS83]	Delta	Probabilistic Analysis	Infinite buffers, Optimistic Switch Model, Store-and-forward
Dias/Jump [DJ81b]	Delta	Probabilistic Analysis	Single buffers, Various Switch Models, Store-and-forward
Jenq [Jen83]	Delta	Probabilistic Analysis	Single buffers, FIFO Buffering Store-and-forward
Yoon et. al. [YLL90]	Delta	Probabilistic Analysis	Multiple buffers, FIFO Buffering, Store-and-forward
Kruskal et. al. [KSW88]	Delta	Probabilistic Analysis	Waiting Time Distribution, Optimistic Switch Model, Infinite Buffers, Store-and-forward
Labarta et. al. [LDC89]	Delta	Open Queueing Network	Multiple buffers, FIFO Buffering, Store-and-forward

a sufficiently simple workload, exact or approximate analysis using stochastic models is possible. Tables 4.1 and 4.2 summarize some commonly referenced network performance studies, indicating analysis technique and special features. Typical workload assumptions are:

1. uniform traffic
2. one flit per packet
3. infinite buffers
4. synchronous switching
5. geometric arrivals

The most common analysis technique is to assume an arrival process independent of network state and use an approximate discrete time Markov model.

Some studies have considered extensions of the basic assumptions. Models of Delta networks, k-ary n-cubes, and hypercubes with finite buffers have been studied in [DJ81b, ABC+89,

Table 4.2: Published Network Performance Studies (continued)

Study	Network	Technique	Features
Marsan et. al. [ABC$^+$89]	Delta	Generalized Stochastic Petri nets	Multiple buffers, FIFO Buffering, Virtual Cut-through
Theimer et. al. [TRH89]	Delta	Probabilistic Analysis	Single buffers, FIFO Buffering, Store-and-forward
Patel/Harrison [PH88]	Delta	Probabilistic Analysis	Hot-spot Traffic, Infinite Buffers, FIFO Queueing, Store-and-forward
Reed et. al. [RF87]	Single Stage	Closed Queueing Networks	Infinite buffers, FIFO Buffering, Store-and-forward
Born et. al. [BK88]	Single Stage	Probabilistic Analysis	Zero delay channels, Infinite Buffers, Store-and-forward
Abraham et. al. [AP89]	Hypercube	Probabilistic Analysis	FIFO Queueing, Finite Buffers, Store-and-forward
Dally [Dal90b]	k-ary n-cubes	Probabilistic Analysis	FIFO Queueing, Wormhole Routing

LDC89, YLL90, Dal90b, AP89]. Hot-spot traffic is considered in [PH88] for cross-bar, multiple-bus, and Delta networks. Multi-flit packets with virtual cut-through are considered in [ABC$^+$89] for Delta networks with finite buffers. As Tables 4.1 and 4.2 show, few results have been published for models considering the simultaneous impact of finite buffers, and multi-flit packets. Another problem with existing work is the use of different timing models for switches. Results in [DJ81b] illustrate the significant variations in network performance that can result from different switch models.

4.2.3 Performance Analysis Via Formal Properties

The stochastic models of the previous section make extremely simplistic assumptions about the temporal and spacial characteristics of packet transmission. With more knowledge about the communication patterns of an intended workload, stronger statements can sometimes be made about network performance. Such results have been published for specific problems in linear algebra, graph theory, sorting and other areas [QD84, B$^+$84, Joh90]. Unfortunately, the engineering applications for which this research is addressed do not typically permit exploitation of these results.

Other formal results have been published concerning efficient emulation of an ideal shared memory on realizable networks. A recent result by Ranade [RBJ88] shows how to efficiently emulate a concurrent read, concurrent write shared memory in time logarithmic in the number of processors. Unfortunately, the emulation algorithm has several implementation problems if coherent caches are provided:

1. It is unclear how combining can be performed on cacheable data. One straightforward solution is to only perform combining for fetch&op accesses, and mark fetch&op variables uncacheable; unfortunately, the formal properties of the algorithm would not necessarily hold. Actual performance of such an scheme could still be good if most contention occurred on fetch&op variables (intuitively, this would be expected).

2. It is assumed that shared memory accesses are issued synchronously. With modern processing elements, however, the time to access local cache memory is one or two orders of magnitude less than the time to access memory across a network; thus the synchronous network "cycle time" is considerably greater than the processor cycle time. Ranade describes a way to emulate asynchronous references on a synchronous network by issuing "dummy" packets at regular intervals, but this consumes extra network bandwidth.

3. Block fetches are generally more time consuming than reads, writes or fetch&op's. This could adversely affect the network "cycle time".

Furthermore, it is theoretically unclear whether ad hoc combining schemes in multiprocessors with asynchronous references is less effective [G^+83c]. These schemes are described in Section 5.2.6.

4.2.4 Focus of this Research

Despite many known network designs and performance studies, it is still difficult to select the "best" network for a cache coherent shared memory multiprocessor. This is because existing results do not adequately deal with the combined effects of multiple flit packets, finite buffers, and sophisticated buffering schemes, or do so only for a very narrow class of network. The rest of this chapter presents a modelling technique that overcomes these problems, permitting a relatively broad class of networks to be studied analytically. The most promising networks can be selected using these efficient analytic methods and subjected to detailed simulation.

As mentioned previously, the networks considered are:

1. k-ary n-cubes

2. k-ary n-dimensional meshes

3. Delta networks

These were chosen because they represent the most common networks addressed in the literature and examples of each class have been implemented in real machines. We further assume packet-switching, oblivious routing and virtual cut-through flow control. Packet-switching is assumed because it increases bandwidth and permits combining. Oblivious routing is assumed because it is easier to analyze and ensures in-order packet transmission between two points. Virtual cut-through flow control is assumed because it is easier to model than wormholing, and requires only a moderate increase in the number of buffers. Despite the focus on these classes of networks, the analysis techniques should be applicable to others.

4.3 An Analytic Modelling Technique

4.3.1 Overview

This section describes a modelling technique that permits any network that can be described as an interconnected set of jxk switches to be analyzed in a uniform way. The technique is based on a combination of algorithms used in the Bell Laboratories Queueing Network Analyzer [Whi83, SW89] and in the finite buffer approximation technique of Altiok and Perros [AP87]. These algorithms are combined and augmented with techniques to model synchronous timing, virtual cut-through flow control, and virtual channels.

The use of queueing models as a basis for network analyis is attractive because there are many published techniques for modelling a wide variety of features. In particular, open networks of GI/G/1/n [1] queues possess many of the modelling features required in the analysis of interconnection networks. Figure 4.9 shows how a Delta network can be modelled as an open network of finite queues. In this model, a customer completing service is held at the server until its destination queue has a free buffer. The queues corresponding to physical channels have queue limit 1 and are necessary to model the timing restriction that only one flit can cross a physical channel per network cycle; they are included in the network only to impose a blocking delay on the buffer queues at the switch inputs. The service delay encountered at the channel queues must be deducted from the final overall latency calculation. This basic queueing model can be extended to model virtual-cut through of multi-flit packets, and the effect of having multiple virtual channels at a switch input (see Section 4.3.4).

4.3.2 The Queueing Network Analyzer

The Bell Laboratories Queueing Network Analyzer (QNA) [Whi83] is a program for analyzing open networks of GI/G/m queues with the following restrictions:

1. first-come, first-served (FCFS) queueing discipline
2. renewal external arrivals
3. no limits on the number of customers in the network or at individual queues
4. continuous time service and interarrival distributions

The following discussion is restricted to single server queues, which are sufficient for the network models of interest.

[1]GI/G/1/n is an example of a commonly used notation for classifying queues. The general form of the notation is "A/S/m/n", where:"A" denotes the interarrival distribution, "S" denotes the service distribution, "m" denotes the number of servers, and "n" denotes the maximum number of customers allowed in the queue. GI denotes renewal arrivals and G denotes a general service distribution. Other common distributions are M and C, denoting exponential and Coxian distributions, respectively.

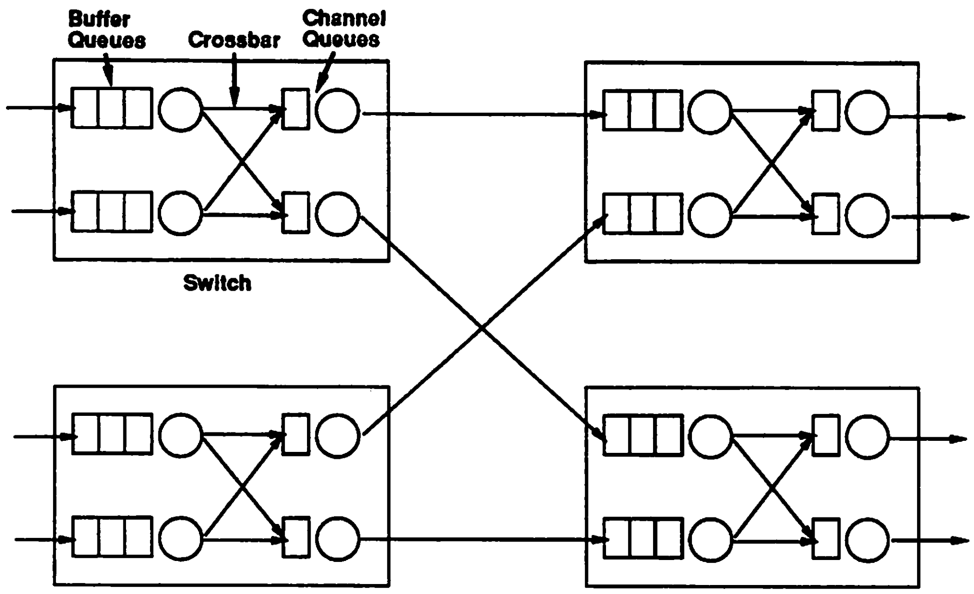


Figure 4.9: Queueing Model of a Delta Network

QNA supports two different routing models: Markovian and deterministic. With Markovian routing, all customers belong to a single class. After completing service at queue i, a customer proceeds to queue j with fixed probability q_{ij}, independent of its previous history. With deterministic routing, each customer belongs to a class with a deterministic route through the network. Customers are not permitted to change class.

Since efficient techniques do not exist for analyzing this class of network, QNA uses approximation techniques. QNA uses a *parametric decomposition* algorithm, in which the arrival processes at each queue are determined globally, and congestion measures determined locally, queue-by-queue. There are three approximations that form the basis of the QNA algorithm. The first is the representation of service and interarrival distributions using only two parameters: the mean and squared coefficient of variation. The second is the approximation of general point arrival and departure processes as renewal processes; in a general point process, the random variables representing interarrivals are not necessarily independent, as they are in a renewal process. The third approximation is a set of linear expressions that determine the two moment representations of the point processes resulting from the superposition and splitting of general point processes, and departures from a queue (Figure 4.10).

Many other approximation techniques have been proposed to analyze open networks of queues with similar features [CS78, Dal90a, Per89, GM88, Per90, KX89, Kue79, Bel82, TMH80,

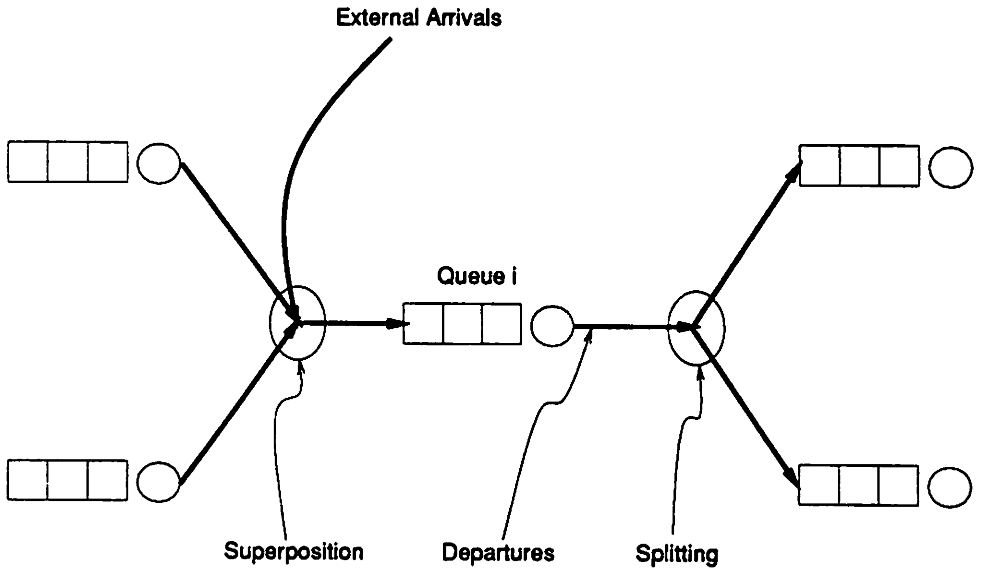


Figure 4.10: Superposition, Departures and Splitting of Point Processes

Aky88, Tak89]. The QNA approach is attractive because it provides a general framework that is easily extended to handle new features and approximations. Since it uses linear equations for its global analysis, it is very efficient and convergence to a single solution is guaranteed.

There are four steps to the QNA algorithm:

1. For deterministic routing, the multiple customer classes are coalesced into a single aggregate customer class with Markovian routing.
2. Two-moment interarrival distributions for each queue in the network are determined using a global analysis.
3. Congestion at each queue is estimated using GI/G/m approximation formulae.
4. Individual congestion measures are combined to estimate overall network performance.

The following terminology is used to describe the algorithm. Let:

- n denote the number of queues in the network.
- λ_i denote the mean arrival rate at queue i.
- c^2_{ai} denote the squared coefficient of variation of the interarrival distribution at queue i.
- λ_{0i} denote the mean rate of external arrivals at queue i.

- q_{ij} denote the probability that a customer leaving queue i proceeds to queue j.
- μ_i denote the mean service rate at queue i. $\tau_i = \frac{1}{\mu_i}$ is the mean service time at queue i.
- $\rho_i = \frac{\lambda_i}{\mu_i}$ denote the utilization at queue i.
- c_{si}^2 denote the squared coefficient of variation of the service distribution at queue i.
- c_{di}^2 denote the squared coefficient of variation of the interdeparture distribution at queue i.

Aggregation of Multiple Customer Classes

Multiple customer classes are treated approximately by transforming the multiple classes and routes into an equivalent single class with Markovian routing; this simplifies steps 2 and 3 of the algorithm. Assume that there are r customer classes with the following parameters for each route k:

- the number of nodes on the route, n_k
- the external arrival rate of the class, $\hat{\lambda}_k$
- the squared coefficient of variation of the external arrival process, c_k^2
- the list of n_k nodes visited on the route, denoted n_{kj}
- the list of n_k service times for each step of the route, denoted τ_{kj}
- the list of n_k service variability parameters for each step of the route, denoted c_{skj}^2

A single aggregate customer class is determined by finding, for each queue j, values of λ_{0j}, c_{0j}^2, τ_j and c_{sj}^2 that account for all routes that pass through the queue. λ_{0j}, the aggregate rate of external arrivals, is the sum of arrival rates of all classes whose routes start at queue j:

$$\lambda_{0j} = \sum_{\{k:n_{k1}=j\}} \hat{\lambda}_k \tag{4.1}$$

The aggregate variability of external arrivals, c_{0j}^2, is determined using a superposition approximation that is described in the discussion of Equations (4.10) to (4.14) below:

$$c_{0j}^2 = (1 - \bar{w}_j) + \bar{w}_j \sum_{\{k:n_{k1}=j\}} \frac{\hat{\lambda}_k c_k^2}{\lambda_{0j}} \tag{4.2}$$

where $\bar{w}_j$ is determined using Equation (4.13).

The mean, aggregate service time for queue j, τ_j, is a weighted average of the service times for all customer classes that pass through queue j:

$$\tau_j = \frac{\sum_{k=1}^{r} \sum_{\{l:n_{kl}=j\}} \hat{\lambda}_k \tau_{kl}}{\lambda_j} \tag{4.3}$$

The weights are the fraction of customers corresponding to each route. λ_j is the sum of the flow rates of all routes passing through the queue:

$$\lambda_j = \lambda_{i0} + \sum_{k=1}^{n} \lambda_{ik} \tag{4.4}$$

λ_{ij} is the flow rate from queue i to queue j:

$$\lambda_{ij} = \sum_{\{k,l:n_{kl}=i,N_{kl+1}=j\}} \hat{\lambda}_k \tag{4.5}$$

and λ_{i0} is the rate at which customers leave queue i.

$$\lambda_{i0} = \sum_{\{k:n_{kn_k}=i\}} \hat{\lambda}_k \tag{4.6}$$

Aggregate service variability, c_{sj}^2, is found using a weighted average of the second moments of the service distributions for each class passing through queue j:

$$c_{sj}^2 = \frac{\sum_{k=1}^{r} \sum_{\{l:n_{kl}=j\}} \hat{\lambda}_k \tau_{kl}^2 (c_{skl}^2 + 1)}{\tau_j^2 \lambda_j} - \tau_j^{-2} \tag{4.7}$$

This corresponds to a mixture of the service distributions for each route passing through the queue. Here $\tau_{kl}^2(c_{skl}^2+1)$ is the second moment corresponding to the squared coefficient of variability c_{skl}^2. The terms τ_j^2 and τ_j^{-2} convert the weighted average of second moments, which is itself a second moment, to a squared coefficient of variability.

Markovian routing parameters q_{ij} for the aggregate class are determined by considering the flow of all customers through each queue i (denoted λ_i), and the flow of customers from queue i to all other queues (denoted λ_{ij}):

$$q_{ij} = \frac{\lambda_{ij}}{\lambda_i} \tag{4.8}$$

Determination of Interarrival Distributions

As mentioned, an interarrival distribution is represented using only a mean and squared coefficient of variation. The mean arrival rates at all queues are related by the rate balance equations:

$$\lambda_j = \lambda_{0j} + \sum_{i=1}^{n} \lambda_i q_{ij} \tag{4.9}$$

The relationship among arrival variabilities are related by linear equations for superposition, departures and splitting (Figure 4.10). The *superposition* formula is derived as the convex combination of two different superposition approximations:

$$c_{aj}^2 = w_j c_{Aj}^2 + (1 - w_j) c_{Pj}^2 \tag{4.10}$$

where c_{Aj}^2 is an *asymptotic* renewal approximation [Whi82] and c_{Pj}^2 is a Poisson approximation.

$$c_{Aj}^2 = \sum_{i=0}^{n} c_i^2 \frac{\lambda_{ij}}{\lambda_j} \tag{4.11}$$

$$c_{Pj}^2 = 1 \tag{4.12}$$

$$w_j = [1 + 4(1 - \rho_j)^2(\nu_j - 1)]^{-1} \tag{4.13}$$

$$\nu_j = [\sum_{i=0}^{n} (\frac{\lambda_{ij}}{\lambda_j})^2]^{-1} \tag{4.14}$$

c_i in Equation (4.11) is the variability of the fanin stream from queue i. c_{Aj}^2 is a weighted sum of the variabilities of the fanin customer streams. c_{Pj}^2 is the variability of a superposition of Poisson fanin streams, which is itself Poisson. The origin of the weighting factor w_j is described in [Whi83].

The *departure* approximation is based on Marshall's formula for interdeparture variability [Mar68] combined with an estimate of waiting time at the queue:

$$c_{di}^2 = 1 + (1 - \rho_i^2)(c_{ai}^2 - 1) + \rho_i^2(max\{c_{si}^2, 0.2\} - 1) \tag{4.15}$$

The *splitting* approximation is the exact formula obtained for random splitting of a renewal process [Whi83]:

$$c_i^2 = p_i c^2 + 1 - p_i \tag{4.16}$$

where c_i^2 is the variability of ith stream after splitting, p_i is the probability that an arrival goes to stream i, and c^2 is the variability of the arrival process before splitting.

Since all of the variability approximations are linear with known coefficients, the resulting system of equations is:

$$c_{aj}^2 = a_j + \sum_{i=1}^{n} c_{ai}^2 b_{ij} \tag{4.17}$$

where:

$$a_j = 1 + w_j\{(p_{0j}c_{oj}^2 - 1) + \sum_{i=1}^{n} p_{ij}[(1 - q_{ij}) + q_{ij}\rho_i^2 x_i]\} \tag{4.18}$$

$$x_i = 1 + (\max\{c_{si}^2, 0.2\} - 1) \tag{4.19}$$

w_j is determined using Eqn. (4.13), and

$$b_{ij} = w_j p_{ij} q_{ij}(1 - \rho_i^2) \tag{4.20}$$

Congestion Approximations

Once the arrival rates and variabilities are known for each queue in the network, the delay at queue i is found using a variation of the Kraemer and Langenbach-Belz approximation [KLB76]:

$$EW_i = \frac{\tau_i\rho_i(c_{ai}^2 + c_{si}^2)g}{2(1 - \rho_i)} \tag{4.21}$$

where:

$$g = \begin{cases} \exp[-\frac{2(1-\rho_i)}{3\rho_i}\frac{(1-c_{ai}^2)^2}{c_{ai}^2+c_{si}^2}] & c_{ai}^2 < 1 \\ 1 & c_{ai}^2 \geq 1 \end{cases} \tag{4.22}$$

EW_i here does not include service time. The average number in queue is found using Little's law. Estimates of other congestion measures, such as the variance of the waiting time and number in queue, can be found using formulae in [Whi83].

Total Network Performance Estimation

Total network performance is computed by combining the congestion measures calculated for each queue. The average number of customers in system is the sum of the average number of customers at each queue:

$$EN = \sum_{i=1}^{n} EN_i \tag{4.23}$$

The average latency for an aggregate customer is the sum of the average latency at each queue, weighted by the probability that an aggregate customer passes through the queue:

$$ET = \sum_{i=1}^{n} ET_i \tag{4.24}$$

where

$$ET_i = (\lambda_i/\lambda_0)(\tau_i + EW_i) \tag{4.25}$$

$$\lambda_0 = \sum_{k=1}^{r} \hat{\lambda}_k \tag{4.26}$$

Expected latency for customers of a particular class k is determined by:

$$ET_k = \sum_{i=1}^{n_k} (EW_{n_{k_i}} + \tau_{n_{k_i}}) \tag{4.27}$$

Overall variabilities and other metrics can estimated as described in [Whi83]. Figure 4.11 summarizes the QNA algorithm.

```
if (multiple customer classes) {
    /* approximate with a single class (Equations (4.1) to (4.8)) */
    ...
}

/* find arrival rates (Eqn. (4.9)) */
solve the set of equations:
    $\lambda_j = \lambda_{0j} + \sum_{i=1}^{n} \lambda_i q_{ij}, \; j = 1, \ldots, n$

/* find arrival variability (Eqns. (4.10) to (4.16)) */
solve the set of equations:
    $c_{Aj}^2 = a_j + \sum_{i=1}^{n} c_{Ai}^2 b_{ij}, \; j = 1, \ldots, n$

/* find local congestion measures (Eqn. (4.21)) */
for ($j = 1, \ldots, n$) {
    $EW_j = \frac{\tau_j \rho_j (c_{aj}^2 + c_{sj}^2) g}{2(1-\rho_j)}$
    $EN_j = \lambda_j (EW_j + \tau_j)$
}

/* find overall network performance */
$EN = \sum_{i=1}^{n} EN_i$
$ET = \sum_{i=1}^{n} ET_i$
```

Figure 4.11: Summary of QNA Algorithm

4.3.3 Altiok and Perros' Finite Buffer Approximation

The QNA algorithm does not model the effect of finite buffers at queues. Finite buffer effects, however, are important in interconnection networks because they determine congestion at physical channels and affect the bandwidth saturation point. This section describes a technique for modelling finite buffer effects due to Altiok and Perros [AP87]; Section 4.3.4 describes how the technique is incorporated into the QNA framework.

The Altiok and Perros algorithm applies to networks of finite queues with these restrictions:

1. A single customer class with Markovian routing.
2. One server per queue.
3. An open queueing network.
4. Exponential, first-come first-served service.
5. The queueing network is acyclic to prevent deadlock. If cycles were permitted, cyclic dependencies would exist on buffers.
6. Poisson external arrivals.
7. *transfer* blocking: in a transfer blocking model customers block *after* they receive service. Blocking occurs if a customer's destination queue is full. When blocked, a customer remains at the server of the source queue and no other customers may be served until it proceeds to its destination.
8. Customers that block for the same destination queue are transferred to the destination in first-come first-served order. Simultaneous blockings are resolved randomly.
9. External arrivals only occur at infinite queues. Modifications of the algorithm to deal with finite input queues are described in [AP87].

Consider an open, acyclic network of finite queues (such as the Delta network in Figure 4.9). Let:

- N_i be the queue limit for queue i, including a customer in service.
- $\pi_{ij}(k)$ be the probability that a customer leaving queue i for queue j finds k customers at queue j.
- $fanin_i$ be the fanin at queue i.

- $\pi_i(k)$ be the probability that a customer from any fanin queue finds k customers at queue i.

The fundamental idea of the Altiok and Perros algorithm is to analyze the acyclic network from outputs to inputs, determining for each queue the distribution of the blocking time it introduces for each of its fanin queues. The service times of the fanin queues are then adjusted to account for this blocking time, and the process is repeated. The precise algorithm is as follows:

1. Determine the values of λ_i by solving the rate equations:

$$\lambda_j = \lambda_{0j} + \sum_{i=1}^{n} \lambda_i q_{ij}, \quad j = 1, \ldots, n \tag{4.28}$$

2. Levelize the acyclic network and perform the following steps on all queues in decreasing order of level (ie. from outputs to inputs).

3. Adjust the service distribution to account for blocking delays at fanout queues. The adjusted distribution is illustrated in Figure 4.12. The branches correspond to the different routes a customer may take upon completing service. The blocking delay boxes correspond to the Coxian [Wol89] representations of blocking delay due to the downstream queues, which have already been calculated. The adjusted distribution is itself a Coxian distribution.

4. For finite queues, determine the distribution of blocking delay that the queue will contribute to its fanin queues:

   (a) Find the *effective* external arrival rate $\bar{\lambda}_i$ at the finite queue, such that the actual rate of customers serviced is λ_i, by solving the equation:

$$\bar{\lambda}_i = \frac{\lambda_i}{1 - \pi_i(N_i)} \tag{4.29}$$

   $\pi_i(N_i)$ is the fraction of customers that find the queue full and are hence refused service. $\pi_i(N_i)$ is a function of the effective arrival rate $\bar{\lambda}_i$, so Equation (4.29) is solved by iteration. $\pi_i(N_i)$ is calculated by analyzing an M/C/1/$N_i + fanin_i$ queue model using matrix geometric techniques [Neu81]. The queue limit is augmented by $fanin_i$ to account for Assumption 8 above that blocked queues are unblocked in first-come first-served order. Poisson arrivals are assumed, and a Coxian distribution is used to model the service distribution (which has been adjusted to account for downstream blocking delays in Step 3).

   (b) Assume $\pi_{ij} = \pi_j$ for all fanin queues i.

(c) Approximate the distribution of blocking delay due to this queue as in Figure 4.13. With probability $\pi_i(N_i)$, an arriving customer finds the queue full and no other customers blocked, so it remains blocked for the remaining time of the customer in service. Since exponential service is assumed, the remaining service time has the same distribution as a complete service time. With probability $\pi_i(N_i + m)$, an arriving customer finds the queue full and m customers blocked. The customer now remains blocked for m full service times, plus the remaining service time of the customer in service. With probability $1 - \sum_{m=0}^{fanin_i} \pi_i(N_i + m)$ an arriving customer is not blocked at all.

The size of the Coxian representations of adjusted service distributions grows rapidly, with a corresponding increase in computational complexity. This can be reduced by approximating the complex Coxian representations by simpler ones with a limited number of phases [PS89].

This algorithm is most accurate when the arrival rates at queue fanins are not excessively unbalanced. Unbalanced streams are treated inaccurately because of the approximation of $\pi_{ij}(k)$ by $\pi_j(k)$ (at Step 4(c)). This is inaccurate because customers in a very heavy arrival stream are unlikely to encounter a blocked customer from a light arrival stream, but customers from the light stream are very likely to encounter a blocked customer from the heavy stream.

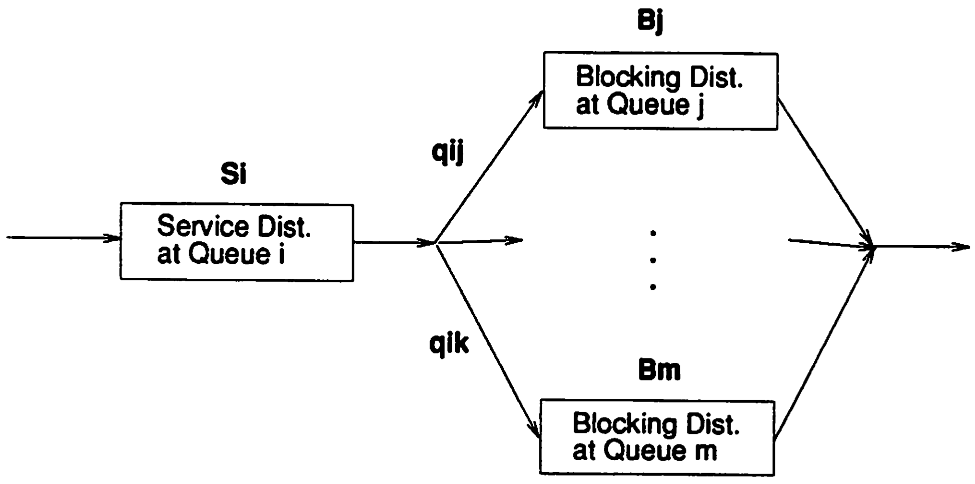


Figure 4.12: Coxian Representation of Adjusted Service Distribution

Figure 4.14 summarizes the finite buffer algorithm.

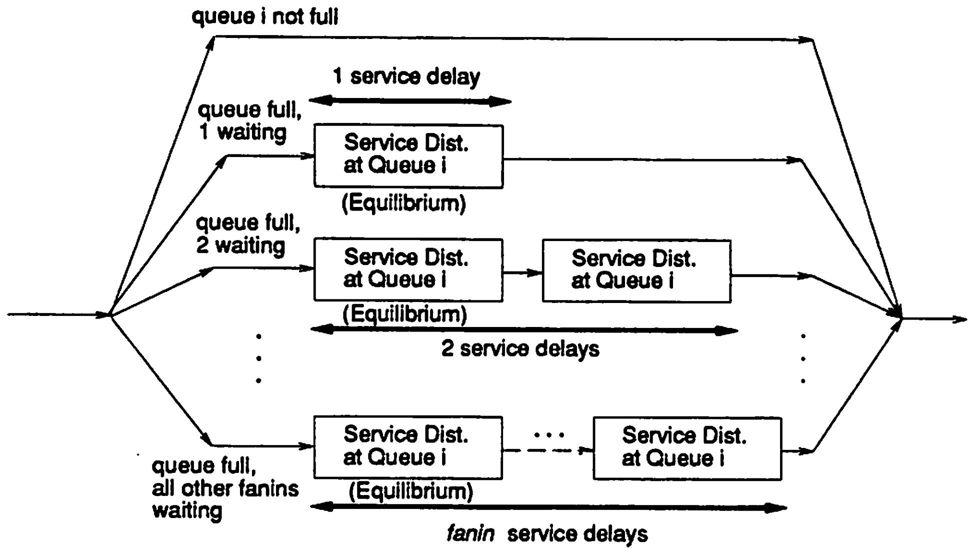


Figure 4.13: Coxian Representation of Blocking Delay

4.3.4 Merging the Algorithms

The finite buffer algorithm of Section 4.3.3 is incompatible with QNA because it uses complex Coxian distributions instead of two-moment approximations, and because it assumes exponential service and Poisson arrivals at all queues. Furthermore, the basic QNA and finite buffer algorithms do not account for virtual cut-through buffering and the use of virtual channels. Section 4.3.5 describes how the incompatibility of Coxian distributions and exponential service is resolved by using two-moment representations for the blocking and service distributions (at Steps 3 and 4); this modification has the benefit of significantly reducing the computational complexity of the algorithm.

The assumption of Poisson arrivals is relaxed by using two moment approximations of interarrival distributions, determined with QNA. QNA and the finite buffer algorithm must be applied in an iterative loop, however, because of a cyclic dependency: QNA requires service parameters to determine arrival parameters, and the finite buffer algorithm uses arrival parameters to determine service parameters. Section 4.3.6 describes the iteration algorithm.

Effects of virtual cut-through buffering and virtual channels can be modelled with techniques described in Sections 4.3.7 and 4.3.8.

```
/* solve rate equations */
$\lambda_j = \lambda_{0j} + \sum_{i=1}^{n} \lambda_i q_{ij}, \;\; j = 1, \ldots, n$

levelize the network;

for each queue $j$ in decreasing order of level {

    adjust service distribution to include blocking
    at downstream queues (Figure 4.12);

    /* determine blocking delay */
    $k = 1$;
    $\bar{\lambda}_i^0 = \lambda_i$;
    repeat {
        find $\pi_i(N_i)$ by analyzing M/C/1/$N_i$ + $fanin_i$ queue with $\bar{\lambda}_i^{k-1}$ arrival rate;
        $\bar{\lambda}_i^{k+1} = \frac{\lambda_i}{1-\pi_i(N_i)}$;
        $k = k + 1$;
    }
    until ($\frac{\bar{\lambda}_i^k - \bar{\lambda}_i^{k-1}}{\bar{\lambda}_i^k}$ < 0.01);

    construct Coxian distribution of blocking delay of this queue
    on fanin queues (Figure 4.13);
}
```

Figure 4.14: Summary of Finite Buffer Algorithm

4.3.5 Two Moment Approximations of Blocking Distributions

Simplification of Coxian blocking distributions to two moment representations is easily done by considering Coxian representations (Figures 4.12 and 4.13) as sums and mixtures of a set of independent random variables. A sum of random variables X and Y with distributions f_x and f_y is a random variable $Z = X + Y$ with distribution $f_z = f_x \otimes f_y$, where $\otimes$ denotes convolution [Wol89]. A mixture of X and Y has distribution $sf_x + (1-s)f_y$, where s is a mixing parameter between 0 and 1. If a mixture of X and Y is denoted $M(s, X, Y)$, the distributions of Figures 4.12 and 4.13 can be represented as:

$$S_i + M(q_{ij}, B_j,\ M(q_{ik}, B_m, \ldots) \ldots) \tag{4.30}$$

and

$$M(\pi_i(N_i), S_i^e, M(\pi_i(N_i+1), S_i^e, M(\pi_i(N_i+2), S_i^e + 2S_i, \ldots) \ldots) \tag{4.31}$$

S_i is a random variable with the service distribution at queue i. S_i^e is a random variable with the distribution of a remaining service time, as viewed by a blocked packet.

It is straightforward to find the mean and second moment of the sum and mixture of two or more independent random variables:

$$E(X+Y) = E(X) + E(Y) \tag{4.32}$$

$$E(M(X,Y,s)) = sE(X) + (1-s)E(Y) \tag{4.33}$$

$$E((X+Y)^2) = Var(X+Y) + E^2(X+Y) = Var(X) + Var(Y) + E^2(X+Y) \tag{4.34}$$

$$E((M(s,X,Y))^2) = sE(X^2) + (1-s)E(Y^2) \tag{4.35}$$

The second moment can then be used to determine the squared coefficient of variation: $c_z^2 = E(Z^2) - 1$. Equations (4.32) through (4.35) can be repeatedly applied to service and blocking distributions to obtain two moment approximations.

In Section 4.3.3 blocking probabilities at a queue were found directly from a matrix geometric analysis of M/C/1/N queues. With two moment approximations of service and interarrival distributions a simpler technique can be used: blocking probabilities can be determined by fitting simple distributions to the moments and solving the resulting queue model. The distributions used here are:

- GEO+1(p): the distribution corresponding to the random variable $1+X$, where X is a random variable with a geometric distribution with parameter p (used when the squared coefficient of variability is less than or equal to 1).

- GEO2+1(p_1, p_2, s): the distribution corresponding to the random variable $1 + X$, where X is distributed as a mixture of two geometric distributions, with parameters p_1, p_2 and mixing parameter s (when the squared coefficient of variability is greater than 1).

In both cases one is added to ensure a minimum interarrival time of one. Technically, GEO+1(p) is a degenerate case of GEO2+1 (GEO+1(p) is equivalent to GEO2+1(p, p, 0.5), GEO2+1(p, 0, 1.0), GEO2+1(0, p, 0), etc.). Discrete time distributions are used because they are more appropriate for synchronous interconnection networks.

A random variable X with distribution GEO+1(p) has first and second moments:

$$E(X) = \frac{1}{p} \tag{4.36}$$

$$E^2(X) = \frac{2-p}{p^2} \tag{4.37}$$

A random variable X with distribution GEO2+1(p_1, p_2, s) has first and second moments:

$$E(X) = 1 + E(Z) \tag{4.38}$$

$$E(X^2) = \frac{E(Z^2) - E^2(Z)}{1 + 2E(Z) + E^2(Z)} \tag{4.39}$$

where

$$E(Z) = 1 + s_1\frac{1-p_1}{p_1} + s_2\frac{1-p_2}{p_2} \tag{4.40}$$

$$E^2(Z) = s_1^2\frac{(1-p_1)^2}{p_1^2} + 2s_1s_2\frac{(1-p_1)(1-p_2)}{p_1p_2} + s_2^2\frac{(1-p_2)^2}{p_2^2} \tag{4.41}$$

A random variable X with moments $E(X)$ and $E(X^2)$ can be fitted to a GEO+1 or GEO2+1 distribution by substituting $E(X)$ and $E(X^2)$ into Equations (4.36) and (4.37) or Equations (4.38) through (4.41) and solving for the appropriate parameters.

A GEO2+1^k/GEO+1/1/N queue is therefore used in place of the M/C/1/N queue in Figure 4.14. The GEO2+1^k/GEO+1/1/N queue has k GEO2+1 arrival streams, a single GEO+1 server, and queue limit N. Multiple input streams are modelled because it is possible for multiple customers to arrive simultaneously in a discrete time system; this is not the case in the continuous time models of the QNA and finite buffer algorithms. By accounting for the arrival rates of each stream, these discrete time superposition techniques overcome the weakness of the original Altiok and Perros algorithm when modeling the superposition of streams with very different rates. A GEO+1 distribution is used for the service distribution because it was observed that service distributions, even after adjustment for blocking delay, exhibit squared coefficients of variation close to 1; using a GEO+1 distribution instead of a GEO2+1 distribution reduces the number of states in the discrete time Markov chain by a factor of 2.

The GEO2+1^k/GEO+1/1/N queue is analyzed by constructing a discrete time Markov chain and solving for steady-state transition rates [Wol89]. The steady state transition rates are then used to determine blocking probabilities and congestion at the queue. Appendix A describes this analysis.

Unfortunately, the number of states in the discrete time Markov chain ($n_s = 2^k(N+1)$) grows exponentially with k. Computation time can be reduced by using the GEO2+1^k/GEO+1/1/N model only when k is small (less than or equal to three). For larger values of k, arrivals are modeled using GEO+1 distributions, requiring a Markov chain with only $n_s = N + 1$ states. This should not impact accuracy significantly because the regularity (coefficient of variation) of the aggregate arrival process should increase with the number of streams [Wol89], making the impact of interarrival variability less important. Appendix A describes the analysis of the GEO+1^k/GEO+1/1/N queue.

With discrete time models, the distribution of remaining service time S_i^e, as observed by a blocked packet, can be approximated by the following two moments:

$$E(S_i^e) = 1 + \frac{E(S_i) - 1}{2} \tag{4.42}$$

$$E(S_i^{e2}) = E(S_i^2) \tag{4.43}$$

This assumes that adjusted service times are almost deterministic (ie. they have low variability). Equation (4.42) accounts for the fact that in a discrete time system a customer that blocks will block for at least one time unit.

4.3.6 Iterating QNA and the Finite Buffer Algorithm

The cyclic dependency between QNA and the finite buffer algorithms can be resolved by combining them in an iterative manner:

1. Using unadjusted service distributions, apply QNA to the network to get rates and variabilities of interarrivals at each queue.

2. Apply the modified finite buffer algorithm to adjust the service distributions.
3. Re-apply QNA to find the interarrival rates and variabilities for the adjusted service distributions.
4. Repeat steps 2 and 3 until the service and interarrival distributions converge.
5. Calculate network performance measures.

More formally, let:

- τ_i^0 and c_{si0}^2 be the mean and squared coefficient of variation of the initial, unadjusted service distribution at queue i.
- τ_i^k and c_{sik}^2 be the mean and squared coefficient of variation of the adjusted service distribution at queue i for iteration k of the algorithm.
- λ_i^k and c_{aik}^2 be the mean and squared coefficient of variation of the interarrival distribution at queue i for iteration k of the algorithm.
- $\Delta_\tau^k = \max\{\frac{\tau_i^k - \tau_i^{k-1}}{\tau_i^k} : i = 1,\ldots N,\ k = 1,\ldots\}$,
- $\Delta_{c_s^2}^k = \max\{\frac{c_{sik}^2 - c_{si(k-1)}^2}{c_{sik}^2} : i = 1,\ldots N,\ k = 1,\ldots\}$,
- $\Delta_\lambda^k = \max\{\frac{\lambda_i^k - \lambda_i^{k-1}}{\lambda_i^k} : i = 1,\ldots N,\ k = 1,\ldots\}$,
- $\Delta_{c_a^2}^k = \max\{\frac{c_{aik}^2 - c_{ai(k-1)}^2}{c_{aik}^2} : i = 1,\ldots N,\ k = 1,\ldots\}$.

Δ_τ^k, $\Delta_{c_s^2}^k$, Δ_λ^k and $\Delta_{c_a^2}^2$ are the maximum relative changes, between iterations, of their respective service and interarrival parameters; they are used as the convergence criteria. Figure 4.15 shows the algorithm.

4.3.7 Virtual Cut-through Flow Control

In QNA and the finite buffer algorithm it is assumed that customers do not proceed to the next queue until service is completed. With virtual cut-through, however, a packet is forwarded after the first flit is served, assuming its destination has sufficient buffer space and the required physical channel is free. The servêr is still occupied for the time corresponding to a full packet. This effect can be modelled by altering the way in which the delay in queue ET_j is calculated:

$$ET_j^{vc} = ET_j - (b - 1) \tag{4.44}$$

```
k = 0
repeat {

    $k = k + 1$

    apply QNA (Figure 4.11) to find $\lambda_i^k$ and $c^2_{aik}$ for $i = 1, \ldots, N$

    apply finite buffer algorithm (Figure 4.14) to find
    $\tau_i^k$ and $c^2_{sik}$ for $i = 1, \ldots, N$

    $\Delta_\tau^k = \max \frac{\tau_i^k - \tau_i^{k-1}}{\tau_i^k} : i = 1, \ldots N,\ k = 1, \ldots$
    $\Delta_{c_s^2}^k = \max \frac{c^2_{sik} - c^2_{si(k-1)}}{c^2_{sik}} : i = 1, \ldots N,\ k = 1, \ldots$
    $\Delta_\lambda^k = \max \frac{\lambda_i^k - \lambda_i^{k-1}}{\lambda_i^k} : i = 1, \ldots N,\ k = 1, \ldots$
    $\Delta_{c_a^2}^k = \max \frac{c^2_{aik} - c^2_{ai(k-1)}}{c^2_{aik}} : i = 1, \ldots N,\ k = 1, \ldots$
}
until (($\Delta_\tau^k < 0.01$) and ($\Delta_{c_s^2}^k < 0.01$) and ($\Delta_\lambda^k < 0.01$) and ($\Delta_{c_a^2}^k < 0.01$))

calculate network performance measures (Eqns. (4.23) to (4.27))
```

Figure 4.15: Combining QNA and the Finite Buffer Algorithm

where ET_j^{vc} and ET_j are the queueing delays (including service) with and without virtual cut-through buffering, and b is the number of flits per packet.

With virtual cut-through, blocking behavior is slightly different from that of the Altiok and Perros model. This is because channel queues never block: buffer queues ensure that destination buffer queues have sufficient buffer space (for an entire packet) *before* sending a packet to a channel queue. Blocking delay caused by full buffer queues is incurred at upstream buffer queues, not channel queues. The blocking calculation for buffer queues is therefore performed using arrival processes that come from the upstream buffer queues (Figure 4.16). The determination of congestion measures (L and w) at buffer queues is also made using arrival processes from upstream buffer queues. It was found that using the single arrival process from a channel queue was not accurate.

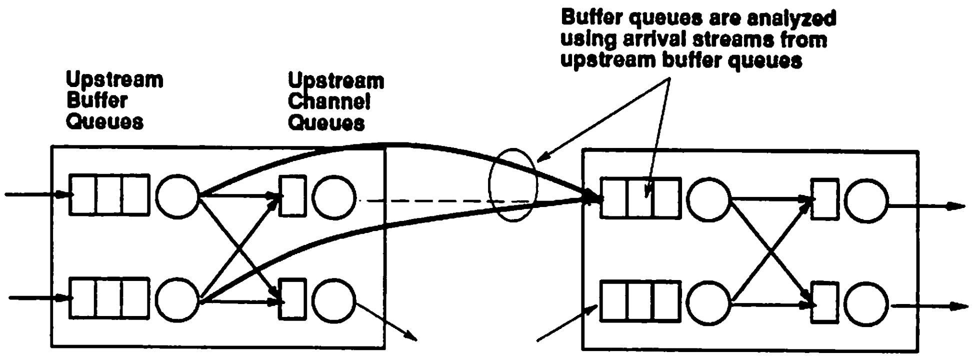

Buffer queues are analyzed using arrival streams from upstream buffer queues
Upstream Buffer Queues
Upstream Channel Queues

Figure 4.16: Arrivals in Buffer Queue Analysis Bypass the Upstream Channel Queue

4.3.8 Virtual Channels

Adding virtual channels to input buffers (Figure 4.7) improves performance by permitting unblocked customers in the middle of a queue to pass blocked customers at the front. Since this is a non-FIFO queueing discipline, the estimate for blocking delay at a server in Section 4.3.3 (Figures 4.12 and 4.13) no longer applies.

Figure 4.17 illustrates a blocking delay distribution that accounts for virtual channel buffering. Like Figure 4.12, Figure 4.17 enumerates all possible situations that can arise when a customer leaves a buffer queue with n_{vc} virtual channels. The different situations are determined by the number of customers in queue at the time of departure (denoted k), the number of distinct destinations of the k customers (denoted i), the particular set of i destinations that are chosen, and by the probability that all i of these destinations block. The rectangular boxes depict the delay experienced by a set of customers blocking at a set of destinations. With virtual channels, if any of the

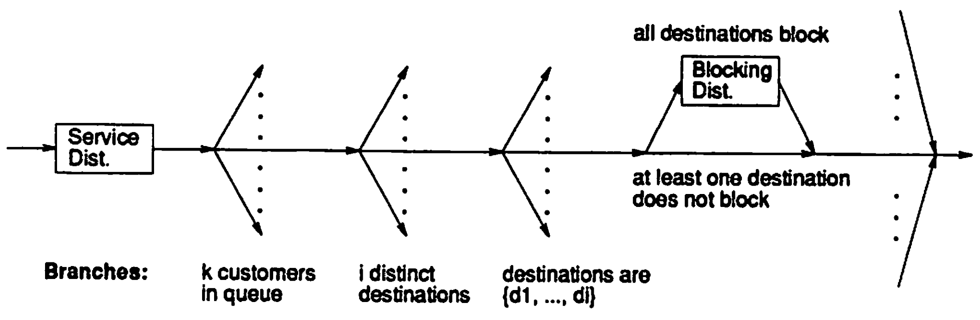


Figure 4.17: Blocking Delay Distribution with Virtual Channels

k customers in queue is destined for an unblocked destination, no blocking occurs– *all* customers block only if *all* destinations are full. All destinations are full with probability

$$P(all\_dests\_full) = \prod_{j=1}^{i} pb_{d_j} \tag{4.45}$$

where pb_{d_j} is the probability that an arrival finds destination d_j full (see Figure 4.13):

$$pb_{d_j} = \sum_{j=0}^{fanin_{d_j}-1} \pi_{d_j}(N_i + j) \tag{4.46}$$

One of the k customers can proceed when one or more of the destinations becomes free; the blocking delay B is therefore the minimum of the blocking delays experienced at the i distinct destinations:

$$B = \min\{B_{d_1}, \ldots, B_{d_k}\} \tag{4.47}$$

B_{d_j} has the blocking distribution seen at destination d_j (determined as in Section 4.3.3 and Figure 4.13). The distribution of B is found by fitting $B_{d_1}, \ldots, B_{d_k}$ to a mixture of geometrics and finding the two moments of their minimum (see Appendix A).

The branching probabilities in Figure 4.17 are calculated as follows. P(k customers) is calculated directly from the Markov chain analysis of the buffer queue in the finite buffer algorithm (Figure 4.14). This, however, introduces a cyclic dependency because the Markov analysis requires that the service time be adjusted for downstream blocking delay. Fortunately, since the algorithm is iterative, P(k customers) can be estimated using the Markov analysis of the previous iteration. For the initial iteration, it is assumed (optimistically) that P(k customers) = 1 for $k = 1$, and zero for $k > 1$.

P(i destinations) and P(dests = $\{d_1, \ldots, d_k\}$) are calculated together as:

$$P(i\ dests)P(dests = \{d_1, \ldots, d_k\}) = \sum_{(i_1,\ldots,i_k)} \binom{k}{i_1, \ldots, i_k} \prod_{j=1}^{k} q_{d_j}^{i_j} \tag{4.48}$$

where the summation is for all tuples $(i_1, \ldots, i_k)$ such that $\sum_j^k i_j = k$ and $i_j > 0$ for $j = 1, \ldots, k$. The summation tuples account for the different ways that i distinct destinations can be assigned to k customers. The total number of branches $n_{branches}$ for which all destinations block is:

$$n_{branches} = \sum_{k=1}^{n_{vc}} \sum_{(i_1,\ldots,i_m)} \begin{pmatrix} k \\ i_1, \ldots, i_m \end{pmatrix} \tag{4.49}$$

$$= \sum_{k=1} n_{vc} m^k \tag{4.50}$$

$\begin{pmatrix} k \\ i_1, \ldots, i_m \end{pmatrix} = \frac{k!}{i_1! i_2! \ldots, i_m!}$ is a multinomial coefficient, corresponding to a term in the expression $(x_1 + x_2 + \ldots + x_m)^k$. The summation in 4.49 is for all m-tuples $(i_1, \ldots, i_m)$ such that $\sum_j^m i_j = k$ and $i_j \geq 0$ for $j = 1, \ldots, m$. Equation (4.49) is a sum of the number of ways that k customers can be sent to m destinations, for all possible values of k. Equation (4.50) follows from the fact that the second summation in Equation (4.49) is equivalent to the multinomial $(1 + 1 + \ldots + 1)^k$ with m ones. Clearly, $n_{branches}$ grows quickly as k increases. Fortunately, k is small for most multiprocessor network models.

Like Figure 4.12, Figure 4.17 corresponds to a mixture of blocking delays, each of which is approximated by two moments. As before, this mixture is reduced to a two moment approximation by repeatedly applying Equations (4.32) to (4.35). The result is an approximation of the service time at the switch input, augmented by an estimate of blocking delay, and the rest of the queueing analysis in Figure 4.15 remains unchanged.

As discussed in Section 4.2.1, some networks require virtual channels to prevent deadlock, and have the buffer organization of Figure 4.8. This buffer organization is approximated as in Figure 4.18. Here two or more virtual channels (with a uniform number of sub-channels) is modeled as one virtual channel with the same number of sub-channels.

4.3.9 Convergence and Computational Complexity

There are two iterations in the algorithm (Figure 4.15), for which convergence is not necessarily guaranteed. Furthermore, convergence does not necessarily guarantee a unique solution. Unfortunately, the combined complexity of finite buffers, two-moment approximations, and the other network features makes a formal analysis of convergence properties difficult. For the networks analyzed in Section 4.4, however, convergence of both inner and outer loops typically occurred in ten or fewer iterations, unless the network was heavily saturated. This was true for network sizes

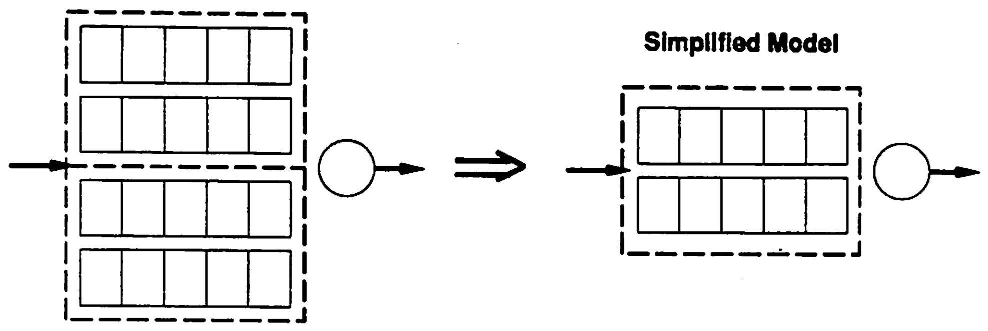


Figure 4.18: Multiple Virtual Channels Approximated as One

ranging from 10 to 10000 queues. When the algorithm did converge, the result was always close to a simulation result.

Since the number of iterations in both loops is only weakly dependent on network size. the approximate computational complexity is the cost of performing the outer loop, multiplied by a constant. The Markov analysis in the inner loop (the finite buffer algorithm) requires the solution of a sparse system of linear equations for each queue. The number of variables is $N + k + 1$ for the GEO+1^k/GEO/1/N queue model, and $2^k(N + k + 1)$ for the GEO2+1^k/GEO/1/N queue model (see Appendix A); since the GEO2+^{k}/GEO/1/N model is only used for k less than four, the number of equations is approximately $O(N)$. The cost of solving a sparse system of N equations is dependent on the pattern of non-zero coefficients in each problem instance. A rule-of-thumb, however, is that a typical sparse system of N equations can be solved in $O(N^{1.3})$ operations [SV81]. The number of blocking cases considered in the virtual channel model is exponential in the number of fanouts. This is usually a small integer, however, so the Markov analysis should dominate. Each outer iteration requires the solution of an additional two sets linear equations of size n for the QNA algorithm, where n is the number of queues. The total complexity of the algorithm is therefore $O(nN^{1.3} + n^{1.3})$, in which the first term is usually dominant.

4.4 Numerical Results

4.4.1 Methodology

In this section the modelling technique of Section 4.3 is validated against simulation data for k-ary n-cubes, Delta networks, and meshes. The following assumptions are made:

1. All switches operate synchronously.

2. The switch model of Figure 4.7 is assumed for all networks. In this model packets are buffered at the inputs by one or more virtual channels per input, and virtual cut-through flow control is used. Virtual channel queues for the same input share a single port to the crossbar network. With virtual cut-through, once a physical channel is assigned to a packet, it is dedicated to the packet for a number of network cycles corresponding to the length of the packet in flits. Only one flit may be passed over a physical channel per network cycle. A physical channel can be assigned to a packet only if the receiving end has space for the entire packet.

3. Buffer queues without external arrivals have a queue limit of 4 packets. Buffer queues with external arrivals have no queue limit.

4. Cycle times are identical for all networks.

5. Processing elements issue packets into the network using GEO+1 interarrival distributions. In the GEO+1 distribution, a packet is injected into the network with probability p on each clock cycle. This corresponds to an input rate of p flits/cycle per network port.

6. A uniform spacial distribution is used for determining the destination of a packet when it is generated. With a uniform distribution, the probability that a packet is sent to processor i is $\frac{1}{N-1}$, where N is the number of processors and i can be any value between 1 and N except the identity of the processor issuing the packet.

7. If two or more customers arrive at the same destination queue simultaneously, the blocking order is random.

8. All packets have the same number of flits.

Section 4.2.1 describes the interconnection topologies for the particular networks considered here. Routing parameters are determined by aggregating the different customer classes corresponding to all combinations of source and destination. Since oblivious routing is assumed in each of the networks of interest, each class has a unique route determined by one of the routing algorithms in Section 4.2.1. Since destinations are selected uniformly, the arrival rate for each class is $\frac{p}{N-1}$. Input queues use the virtual channel queueing discipline. Channel queues use the FIFO queueing discipline. As mentioned previously, channel queues are put in the network only for determining the blocking delay they incur on input queues; the service time for the channel queues are therefore not included in the network delay calculation. A buffer queue with two or more virtual channels, each with k sub-channels, is modelled as a buffer queue with one virtual channel with k sub-channels.

Queueing models for symmetric networks (k-ary n-cubes and Delta networks) with a uniform workload can be simplified by considering the minimal representative portion of the network. In a k-ary n-cube, the arrival processes at each switch are stochastically identical from switch to switch, so the queueing model can be reduced to that of a single switch with direct feedback (Figure 4.19). In a Delta network, the arrival processes for switches in each horizontal slice are identical, so the model can be reduced to a single slice (Figure 4.20). Both of these simplifications introduce cyclic dependencies among buffers. To break the cycles it is assumed that that buffer queues never fill. Channel queues therefore never block, and only channel queues contribute blocking delay to buffer queues. This approximation is accurate for the networks of interest when four or more buffers are provided per buffer queue. Further details on the queueing models for specific networks are provided in Appendix B.

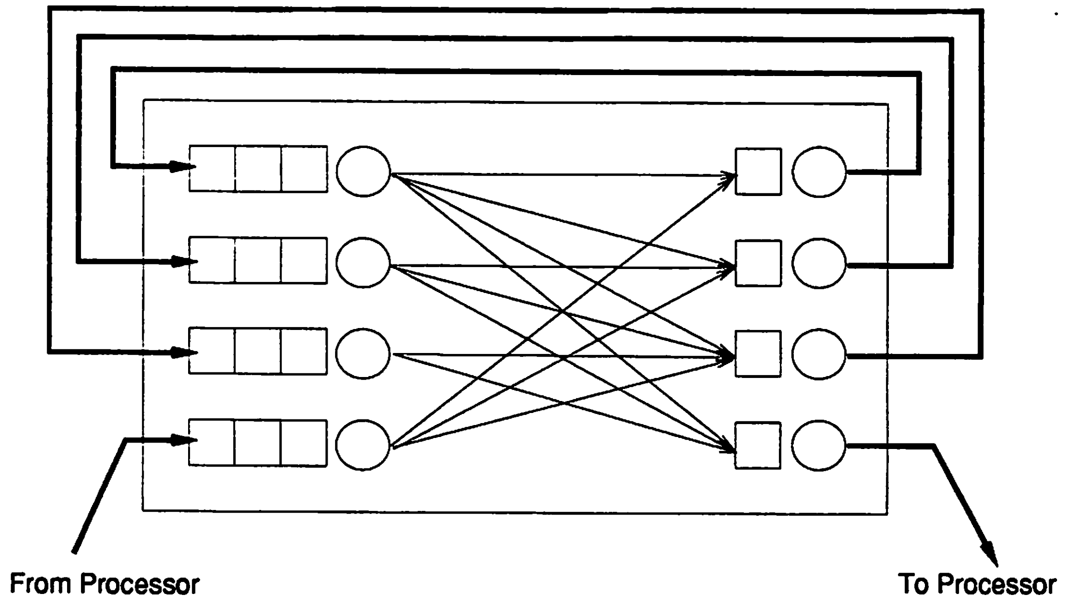


Figure 4.19: Simplified Model of Direct k-ary n-cube Under Uniform Load

4.4.2 Results

Figures 4.21 to 4.34 show plots of latency versus arrival rate, as predicted by the analytic model, for torus ($k = 3$), bidirectional torus ($k = 3$), mesh ($k = 3$), hypercube, and radix 4 and 8 Delta networks. Results are shown for 5 and 10 flit packet sizes. Simulation points are also shown

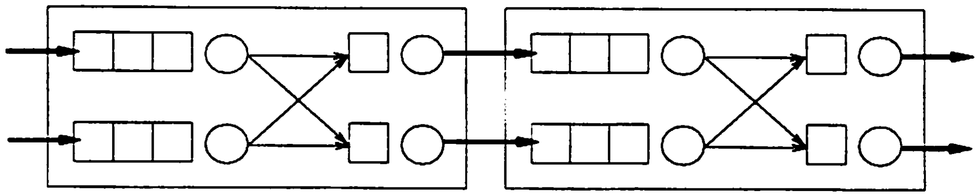


Figure 4.20: Simplified Model of a Delta Network Under Uniform Load

to indicate the accuracy of the model. Vertical lines indicate the saturation points predicted by the model; simulation points along the upper horizontal border indicate the saturation points predicted by the simulator. As the plots show, predicted latencies are usually within 5% of simulated values until heavy saturation (latency exceeds ten times the minimum) occurs. The predicted saturation points also corresponded well with simulation results.

As expected, the hypercube and Delta networks (Figures 4.27 to 4.34) exhibit the best performance: saturation bandwidth is independent of the number of processors. This results from the fact that the utilization of physical channels (channel queues in Figure 4.7) is independent of the number of processors [RF87]. The cost for this performance, however, is greater implementation cost: more complex interconnections, $\log N$ more switches for the multistage network, and larger switches for the direct network. For all networks, the saturation bandwidth (in flits/cycle) remains the same as the number of flits/packet changes, but latency rises faster because long packets occupy channels for longer periods of time. The torus network exhibits very low saturation bandwidth for large machines, even for small packet sizes. Adding two links between neighboring processors increases the saturation point by a factor of 2.5. For a given path width, a unidirectional toroidal mesh requires twice the number of interconnections through a node to embed the end-around connections [DS87]. The mesh results show that the extra interconnections are better used as back-links in a mesh, since this increases the saturation point by a factor of 2.

All of these results assume that a main memory and cache controller share a single network port. Saturation bandwidth could possibly be increased by adding more ports. This and other alternatives could be easily investigated using the same modeling technique.

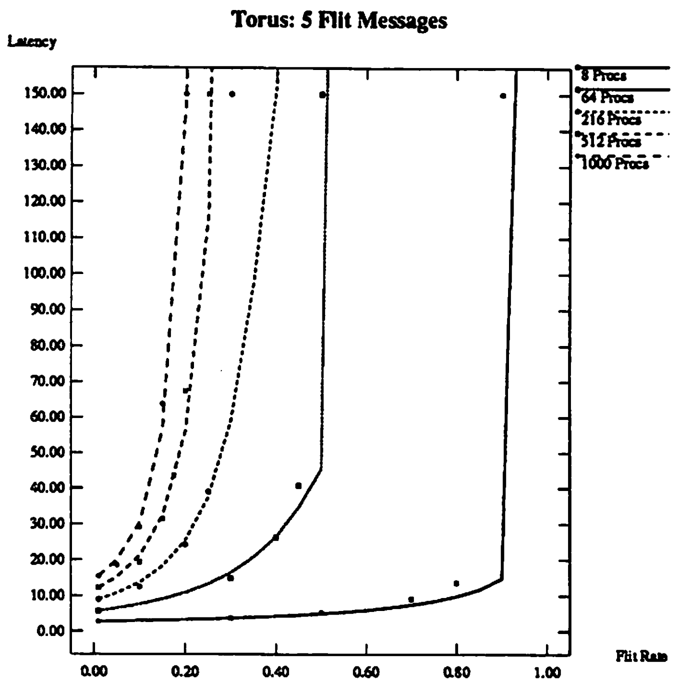


Figure 4.21: 3-D Torus Network Performance (5 Flits/Packet)

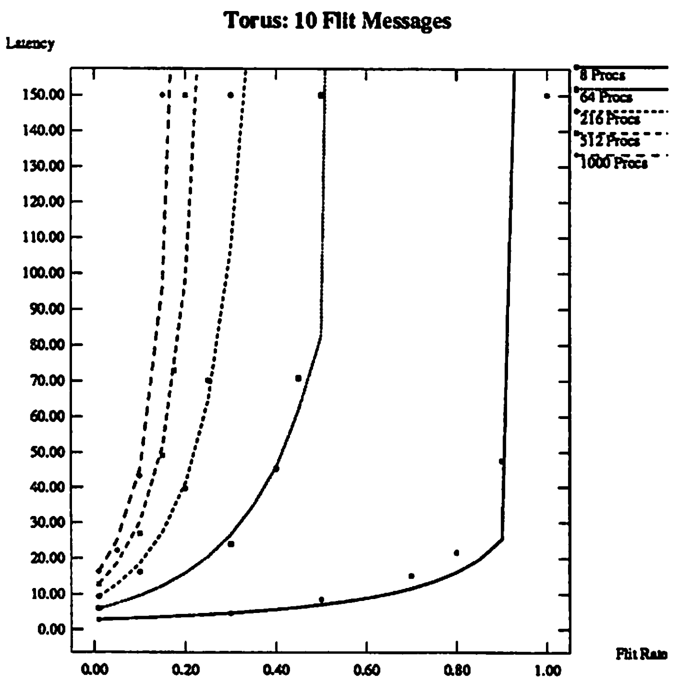


Figure 4.22: 3-D Torus Network Performance (10 Flits/Packet)

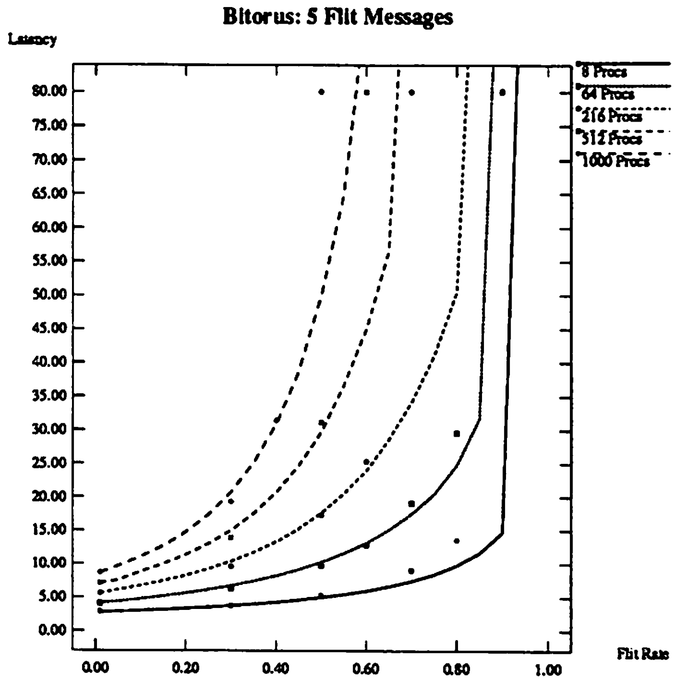


Figure 4.23: 3-D Bidirectional Torus Network Performance (5 Flits/Packet)

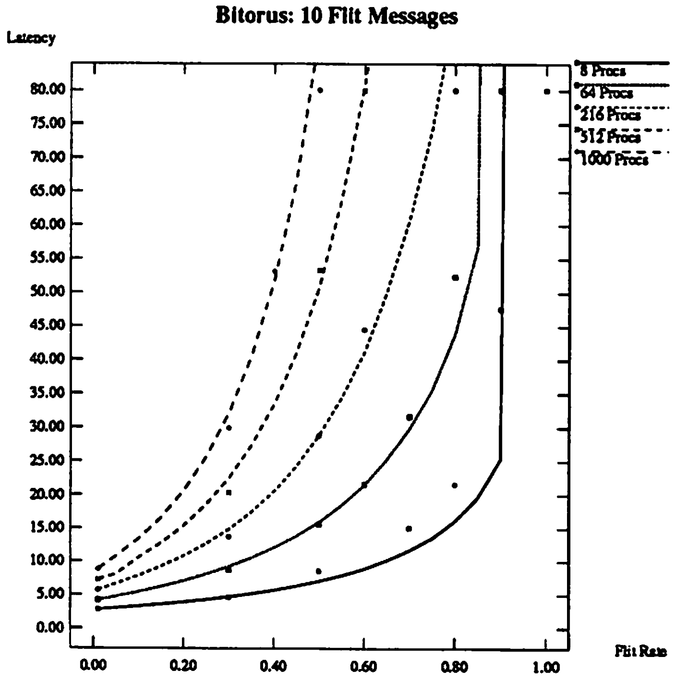


Figure 4.24: 3-D Bidirectional Torus Network Performance (10 Flits/Packet)

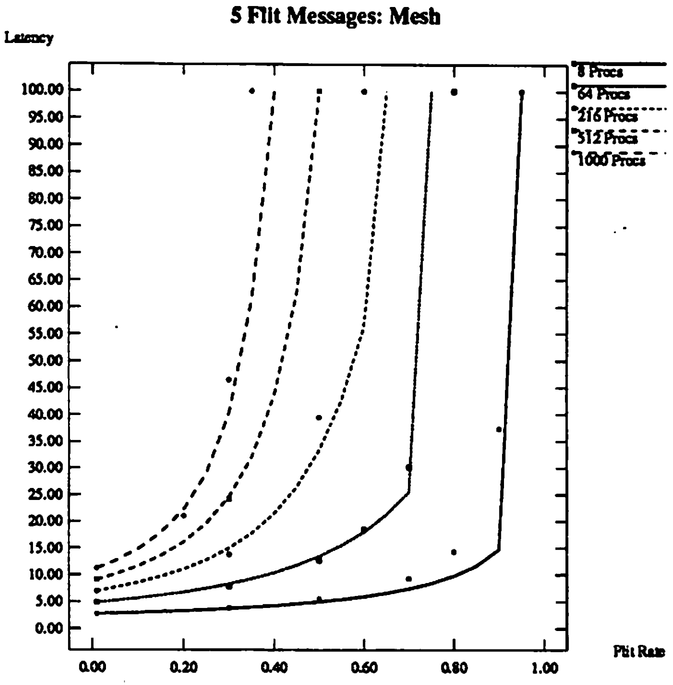


Figure 4.25: 3-D Mesh Network Performance (5 Flits/Packet)

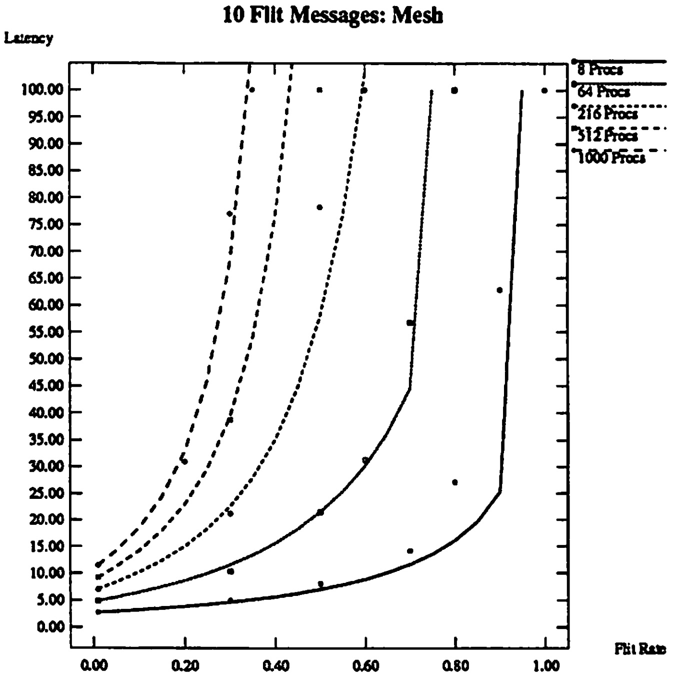


Figure 4.26: 3-D Mesh Network Performance (10 Flits/Packet)

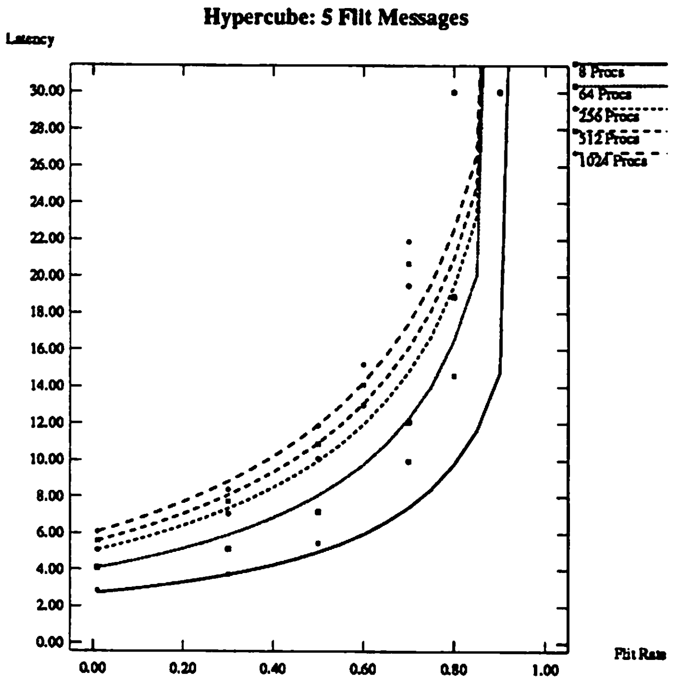


Figure 4.27: Hypercube Network Performance (5 Flits/Packet)

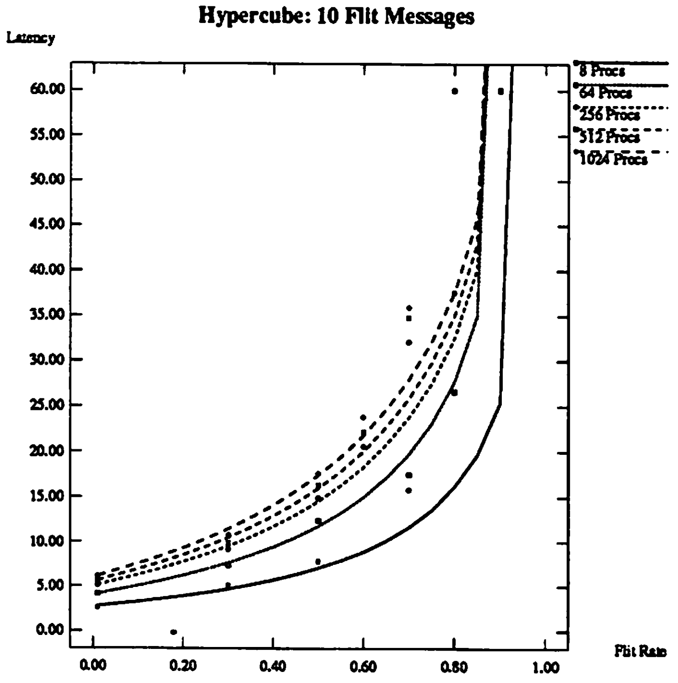


Figure 4.28: Hypercube Network Performance (10 Flits/Packet)

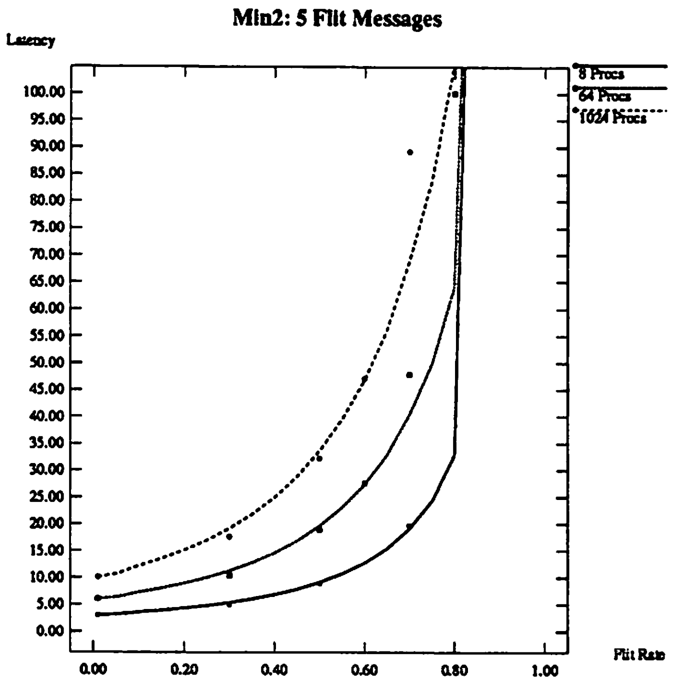


Figure 4.29: Radix-2 Delta Network Performance (5 Flits/Packet)

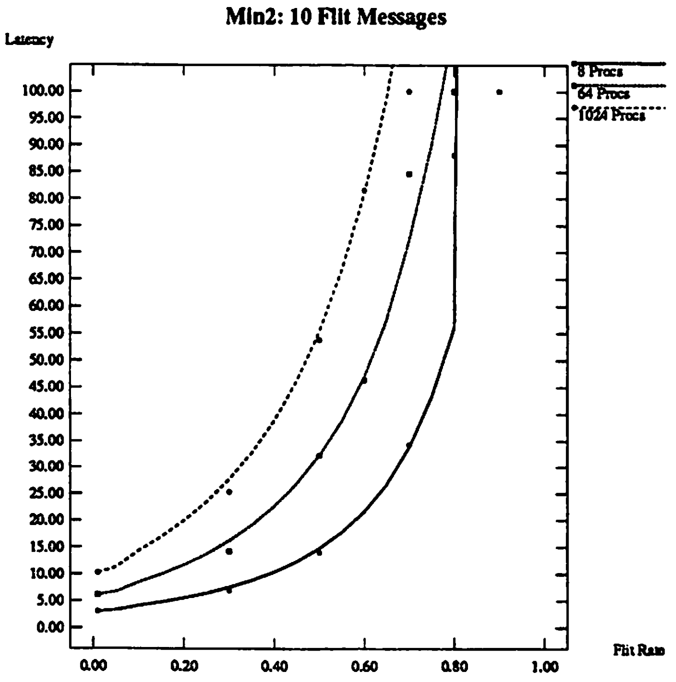


Figure 4.30: Radix-2 Delta Network Performance (10 Flits/Packet)

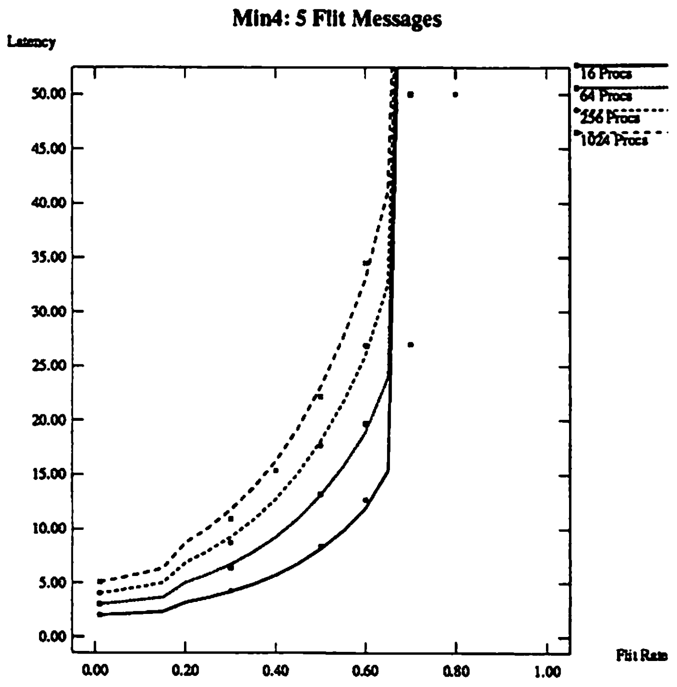


Figure 4.31: Radix-4 Delta Network Performance (5 Flits/Packet)

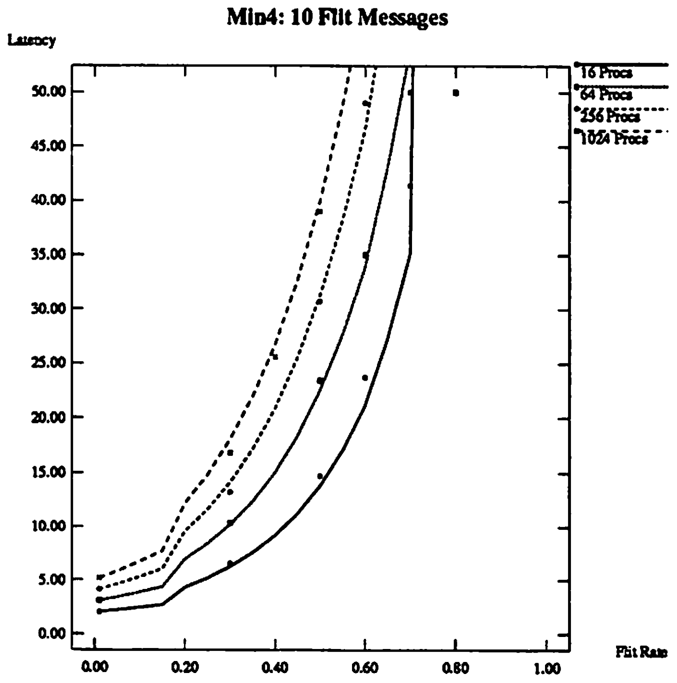


Figure 4.32: Radix-4 Delta Network Performance (10 Flits/Packet)

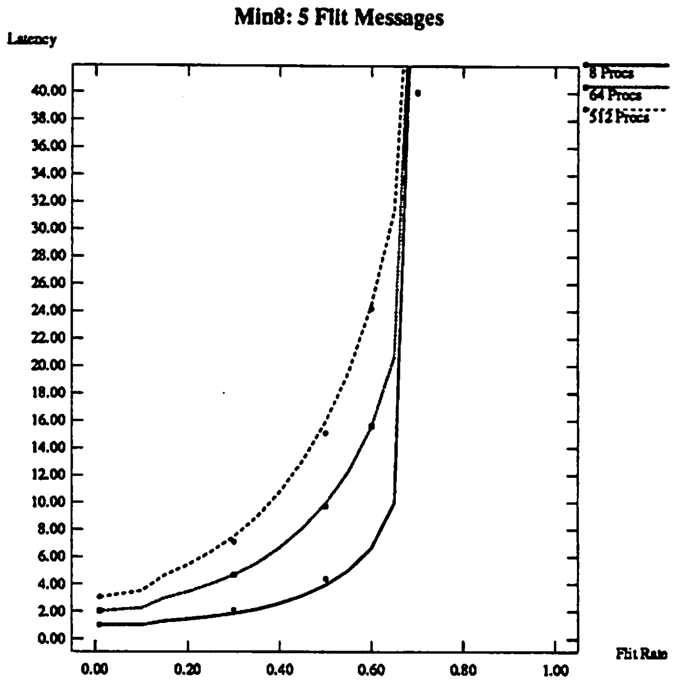


Figure 4.33: Radix-8 Delta Network Performance (5 Flits/Packet)

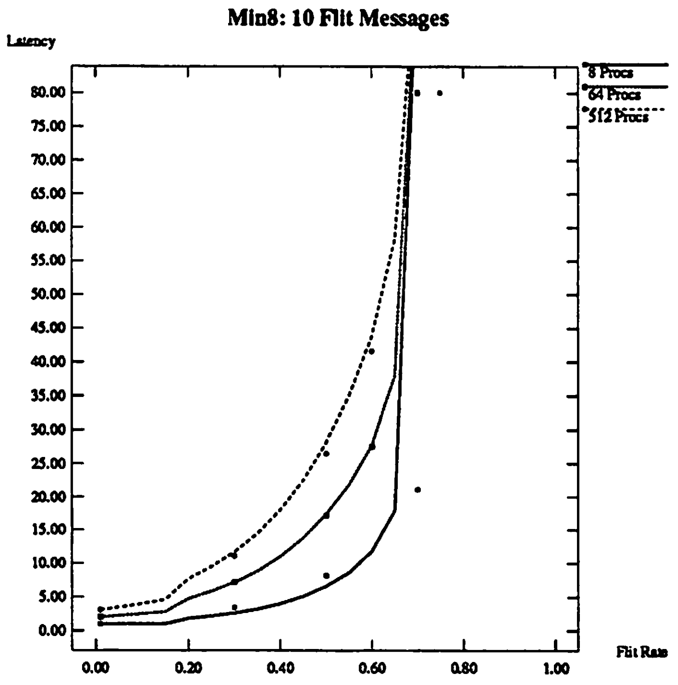


Figure 4.34: Radix-8 Delta Network Performance (10 Flits/Packet)

4.5 Processor Utilization of Caching Schemes

4.5.1 Methodology

The impact of network latency on processor utilization (or, equivalently, memory access time) can be taken into account using the modeling paradigm outlined in Section 1.3.3. In this paradigm, network-independent cache simulations are used to estimate arrival rates, and network models provide the latency with which a cache miss is processed. This, however, alters the arrival rate that was initially assumed, so a new arrival rate is estimated and the process is iterated until the assumed arrival rate corresponds to the arrival rate adjusted for miss latency. The miss latency can then be used to estimate the time during which a processor is stalled on a read miss, which corresponds to a decrease in utilization.

Figure 4.35 summarizes the algorithm. The input parameters are:

- n_{rm} is the average number of read misses, per processor, in the cache simulation;
- t_{busy} is the average number of cycles during the cache simulation in which a processor is busy (ie. not waiting for a lock or barrier);
- t_{sim} is the total number of cycles of the cache simulation;
- $f_{rm} = n_{rm}/t_{busy}$ is the average number of misses per busy cycle;
- n_{trans} is the average number of network transactions issued by a cache;
- U_{base} is the average processor utilization assuming zero network latency;
- $\lambda_{base} = n_{trans}/t_{busy}$ is the rate at which network transactions arrive at a network port, assuming zero network latency;
- n_{flits} is the average number of flits per transaction; It is determined using the cache simulation results and the transaction sizes in Tables 3.8 and 3.9.
- T_{dir} is the time to process a network transaction at its destination;

The following are calculated:

- U_{net} is the average processor utilization, accounting for network latency;
- λ_{net} is the rate at which network transactions arrive at a network port, accounting for network latency;

- $T_{net}(\lambda_{net}, n_{flits})$ is the average network latency (one way) for the specified arrival rate and packet size;

- T_{trans} is the total time to process a network transaction, including two network traversals and processing at its destination;

It is assumed that write misses are buffered, so a cache stalls only on read misses.

```
$i = 0$;
repeat {
    $T_{trans} = 2 * T_{net}(\lambda^i_{net}, n_{flits}) + T_{dir}$;
    $i = i + 1$;
    $\lambda^i_{net} = \frac{\lambda_{base}}{1 + f_{rm} T_{trans}}$;
} until ( $\frac{\lambda^i_{net} - \lambda^{i-1}_{net}}{\lambda^i_{net}} < 0.01$ )
$U_{net} = \frac{U_{base}}{1 + U_{base} f_{rm} T_{trans}}$;
```

Figure 4.35: Algorithm for Estimating Processor Utilization with Non-Zero Network Delay

The expression for λ^i_{net} is the simplified form of:

$$\lambda^i_{net} = \frac{n_{trans}}{t_{busy}(1 + f_{rm} T_{trans})} \tag{4.51}$$

The expression for U_{net} is derived as follows:

$$\begin{aligned} U_{net} &= \frac{t_{busy}}{t_{sim} + n_{rm} T_{trans}} && (4.52) \\ &= \frac{\frac{t_{busy}}{t_{sim}}}{1 + \frac{t_{busy}}{t_{sim}} f_{rm} T_{trans}} && (4.53) \\ &= \frac{U_{base}}{1 + U_{base} f_{rm} T_{trans}} && (4.54) \\ & && (4.55) \end{aligned}$$

4.5.2 Numerical Results

Table 4.3 shows utilization estimates for the most demanding benchmark, VERF, for several different network and protocol configurations. It is assumed that $T_{dir} = 20$ cycles, network paths are four bytes wide, and caches are infinite. The networks considered are a unidirectional 3-D torus, a bidirectional 3-D torus (bitorus), a hypercube, and a 3-D mesh. Since Delta networks have performance that is similar to a hypercube, they are omitted for brevity. Table 4.3 also shows the fraction of total network bandwidth used by the benchmark. The network saturation point is defined here as the load at which delay is ten times the minimum.

For a 16 byte block size all four networks provide reasonable utilizations (58 to 68 %). The UPDATE and COMP8 protocols improve utilization by 5 to 10 % at the expense of up to 50% more traffic. In all cases a large fraction (30% to 100%) of available network bandwidth is used. Traffic increases sharply for a 64 byte block size, and causes the torus network to saturate for all protocols. UPDATE and COMP8 improve utilizations by 10 to 15 %, but require up to 100 % more traffic. With a 64 byte block size, the UPDATE and COMP8 protocols cause the mesh network to saturate and use up most of the available bandwidth of the hypercube and bidirectional torus. These protocols are therefore only appropriate for smaller block sizes.

The table shows that coherent caches reduce the sensitivity of processor utilization to network latency. The UPDATE results, in particular, show that latency can be increased by a factor of 5 with a reduction in utilization of only 15%. This insensitivity explains why utilizations for different networks do not vary appreciably for networks operating below their saturation points. It also explains why utilizations are relatively insensitive to the number of flits per packet. Of the four networks, the mesh offers a good compromise between implementation cost and performance.

Network performance for multiprocessors with more than 64 processors can be estimated by scaling network traffic by the anticipated increase in coherence traffic. Results in Section 3.6.3 suggest that the frequency of cache invalidations (updates) grows linearly with the number of processors. The traffic for an N processor system relative to a 64 processor system can therefore be found by scaling the invalidation/update traffic for the 64 processor system by increase in machine size. Let:

- r_N be the ratio of traffic for the N processor system relative to the 64 processor system.
- f_{64} be the fraction of traffic in the 64 processor system due to invalidations or updates (from main memory to cache).

Table 4.3: Processor Utilizations for VERF, 4 Byte Path

Network	λ_{net} (flits/cycle)	Network Delay (cycles)	$\frac{\lambda_{net}}{\lambda_{sat}}$ [a]	Utilization (%)
IDEAL, 64B Block, 5 flits/packet				
Ideal	0.151	0.0	0.0	75
INVAL, 16B Block, 5 flits/packet				
Torus	0.224	55.9	0.45	58
Bitorus	0.249	36.0	0.29	63
Hyper	0.252	33.8	0.29	64
Mesh	0.244	39.6	0.34	62
INVAL, 64B Block, 10 flits/packet				
Torus	0.450	26.2	1.00	65
Bitorus	0.458	48.1	0.54	59
Hyper	0.473	42.3	0.54	60
Mesh	0.440	55.5	0.61	57
UPDATE, 16B Block, 5 flits/packet				
Torus	0.344	61.0	0.69	64
Bitorus	0.371	35.4	0.43	68
Hyper	0.374	33.2	0.43	69
Mesh	0.367	39.2	0.51	68
UPDATE, 64B Block, 5 flits/packet				
Torus	0.50	819	1.00	39
Bitorus	0.86	222	1.00	60
Hyper	0.87	222	1.00	60
Mesh	0.72	407	1.00	51
COMP8, 16B Block, 5 flits/packet				
Torus	0.279	50.3	0.56	64
Bitorus	0.296	33.3	0.34	68
Hyper	0.298	31.8	0.34	68
Mesh	0.293	36.3	0.41	67
COMP8, 64B Block, 5 flits/packet				
Torus	0.500	214	1.00	53
Bitorus	0.707	56.4	0.82	67
Hyper	0.724	46.3	0.83	68
Mesh	0.720	48.7	1.00	68

[a] λ_{sat} is the arrival rate at which network delay is ten times the minimum (light load) delay.

Table 4.4: Estimated Traffic Increases for Large Multiprocessors

N	Protocol	16B Block		64B Block	
		f_{64}	r_N	f_{64}	r_N
128	INVAL	0.0640	1.06	0.0445	1.05
128	UPDATE	0.482	1.48	0.675	1.68
128	COMP8	0.394	1.39	0.608	1.61
256	INVAL	0.0640	1.19	0.0445	1.13
256	UPDATE	0.482	2.45	0.675	3.03
256	COMP8	0.394	2.18	0.608	2.82
512	INVAL	0.0640	1.45	0.0445	1.31
512	UPDATE	0.482	4.37	0.675	5.73
512	COMP8	0.394	3.76	0.608	5.26
1024	INVAL	0.0640	1.96	0.0445	1.67
1024	UPDATE	0.482	8.23	0.675	11.13
1024	COMP8	0.394	6.91	0.608	10.12

The increase in traffic is then:

$$r_N = (1 - f_{64}) + f_{64}\frac{N}{64} \tag{4.56}$$

$$= 1 + f_{64}(\frac{N}{64} - 1) \tag{4.57}$$

Table 4.4 shows estimated traffic increases for 128, 256, 512 and 1024 processors. It shows that the UPDATE and COMP8 protocols are only appropriate for 128 processors or less. Traffic for the INVAL protocol, however, rises slowly enough that lower dimension networks with sufficiently wide paths should provide good support for up to 1024 processors.

4.6 Conclusion

In this chapter analytic techniques were presented for estimating the performance of a broad class of multiprocessor interconnection networks with realistic features: finite buffers, virtual channel queueing discipline, and virtual cut-through flow control. Numerical comparisons with simulation results showed that predicted values are usually within 5% of simulation values until heavy saturation is reached (latency exceeds ten times the minimum). The predicted saturation points also corresponded well with simulation results. The final section examined the impact of network performance on processor utilization in a cache coherent multiprocessor. The results suggest that, with an invalidation protocol and sufficiently wide paths, good utilization can be obtained for a number of different networks for machines with up to 1000 processors. The mesh network is particularly attractive because of its low implementation cost. The update and competitive protocols are only appropriate for small block sizes and machines with up to a couple hundred processors.

Chapter 5

Synchronization

5.1 Overview

Synchronization support is tied to the design of a cache coherence scheme, because poor implementations of synchronization primitives can negate much of the benefit of coherent caches. The availability of coherent caches can also permit more efficient implementations of synchronization primitives, such as locks and barriers [BD86, IEE90, $L^{+}90$]. This chapter begins with a review of published implementation techniques for the following hardware-supported synchronization primitives: fetch&op, locks, barriers, and multiprefix operations. A new technique is described for implementing fetch&op, barrier, and multiprefix operations in hardware or software. The hardware version applies to single or multistage networks constructed of $j \times k$ crossbar switches.

The goal of this chapter is to show that sophisticated synchronization support is possible in hardware or software using straightforward techniques of moderate complexity.

5.2 Previous Work

5.2.1 Overview

Some common hardware-supported synchronization primitives in shared memory multiprocessors, in order of increasing functionality, are [Sto87, RBJ88]

1. *read* and *write*

2. *fetch&op*

3. *locks*
4. *barriers*
5. *multiprefix operations*

The simplest primitives are the *read* and *write* instructions provided "for free" on all shared memory multiprocessors. Many of the first synchronization algorithms were based on read and write instructions because early high-level languages did not provide access to more powerful read-modify-write instructions [Sto87]. These algorithms, however, relied on the assumption that multiple reference streams are sequentially consistent: accesses by each process are performed in program order, as observed by *any* of the multiple processes. Although this assumption is reasonable for multiprogramming operating systems running on single processors, it severely impacts the performance of cache coherent multiprocessors; this was addressed in Chapter 1.

fetch&op instructions are the most common class of synchronization primitive, and include *test-and-set, compare-and-swap* and *fetch-and-add* [Sto87]. A *fetch&op* instruction takes two arguments: *addr*, the address of a shared variable, and *val*, a value to be used in the specified operation. *fetch&op* applies "op" to *val* and the contents of *addr*, stores the result in *addr*, and returns the contents of *addr* prior to performing the operation. This is all done atomically. test-and-set, compare-and-swap, and fetch&add are common examples of fetch&op primitives. In *test-and-set*, "op" is "set the contents of *addr* to one", and *val* is unused. In *compare-and-swap*, the contents of *addr* is compared with *val*; if equal, they are swapped, otherwise the contents of *addr* are unchanged. In *fetch&add*, *val* is added to the contents of *addr*. Machines that support fetch&op instructions include: IBM 370 (compare-and-swap, test-and-set) [Sto87], Sequent Symmetry (test&set) [LT88], IBM RP3 (fetch&add) [P+85], NYU Ultracomputer (fetch&add) [G+83c], and Cedar (various fetch&op) [G+83a].

locks provide a mechanism by which processes can gain exclusive access to shared data. Mutual exclusion is enforced by associating a lock with each collection of mutually exclusive data. Processes must *acquire* the lock before accessing the protected data, and must *release* it when they are finished. When a lock is acquired, a process has exclusive access to the critical section, and any other process attempting to acquire the lock stalls until the former process releases it.

barriers provide a mechanism by which multiple processes ensure that they have reached the same point in a program before proceeding. Barriers are typically used to ensure that results of one step of a computation, a parallel loop for example, are complete before proceeding with the next step.

multiprefix operations are much higher level operations that can be exploited in certain parallel algorithms [RBJ88]. Multiprefix operations are defined on a set of L value lists:

$$[l_{11}, l_{12}, \ldots, l_{1n_1}]$$
$$\vdots$$
$$[l_{L1}, l_{n_L 2}, \ldots, l_{Ln_L}] \tag{5.1}$$

It is assumed that:

1. There is at most one value stored per processor.
2. Processors are assigned distinct identifiers between 0 and $N - 1$, where N is the number of processors.
3. Let $proc(l_{ij})$ denote the processor holding value l_{ij} of list i. Values in a list i are ordered by processor identifiers such that $proc(l_{ij}) < proc(l_{i(j+1)})$.

Given an associative binary function F, a multiprefix operation computes a second set of L lists:

$$[r_{11} = l_{11}, r_{12} = F(l_{12}, r_{11}), \ldots, r_{1n_1} = F(l_{1n_1}, r_{1(n_1-1)})]$$
$$\vdots$$
$$[r_{L1} = l_{L1}, r_{n_{L2}} = F(l_{n_{L2}}, r_{L1}), \ldots, r_{Ln_L} = F(l_{Ln_L}, r_{L(n_L-1)})] \tag{5.2}$$

where r_{ij} is stored in processor $proc(l_{ij})$. This can be generalized by permitting a different binary function to be used for each list [Ran89]. Figure 5.1 shows a simple example of a multiprefix operation in which the minimum values in two lists are found.

Relatively few algorithms have been published that exploit the full power of multiprefix operations. Multiprefix operations, however, subsume many other less powerful primitives, including scans [Ble89], fetch&op [G$^+$83b], and Hillis' β operation [Hil85].

The following example illustrates the usefulness of this class of synchronization operation. This example uses the β operation and is taken from Hillis' dissertation on the Connection Machine [Hil85]. β operations can be thought of as a weaker form of the multiprefix operation in which the application of the binary function F is not constrained to processor order, and intermediate results are not returned. Consider the problem of finding the minimum path between two vertices in a graph:

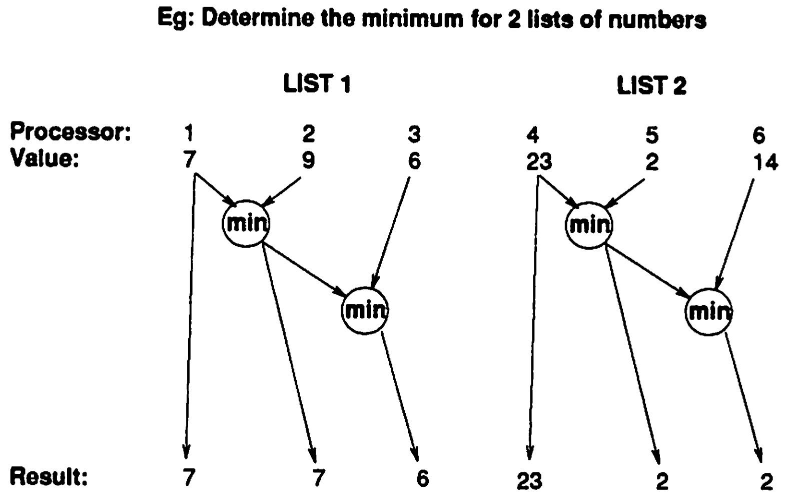


Figure 5.1: Example of a Multiprefix Operation

Given a set of vertices, V, and a set of edges $E = V \times V$, find the length k of the shortest path between two distinct vertices a and b.

Hillis describes the following parallel algorithm for solving the problem:

1. Label all vertices with $+\infty$.

2. Label vertex a with 0.

3. Label every vertex, except a, with 1 plus the minimum of the labels of all neighboring vertices.

4. Repeat the previous step until the label of b is finite. The label of b is the desired result.

The heart of the algorithm is Step 3. It can be directly implemented as a β operation with $F = \min(x, y)$ and a set of value lists, one per vertex, containing the labels of neighboring vertices. For many large graphs of interest, the average value of k for a randomly selected pair of vertices is a small integer. An efficient parallel implementation of the β operation can therefore provide considerable speedup over a serial solution.

The remainder of this section reviews published hardware and software implementations for these synchronization primitives.

5.2.2 Read, Write and Fetch&ops

Atomic reads and writes are provided "for free" on a sequentially consistent shared memory multiprocessor. This is not the case for weakly consistent multiprocessors, where the ordering of read and write operations are relaxed to improve performance. In a weakly consistent multiprocessor atomic reads and writes must be implemented as special synchronization instructions so that stricter ordering is enforced.

The most common way to implement atomic reads, writes or fetch&op instructions in a weakly coherent multiprocessor is to mark the corresponding synchronization variables as non-cacheable and build main memory controllers that support atomic read, write and fetch&op transactions. Since synchronization variables are uncached, only one copy of each exists in the system, and atomicity is easily enforced by the memory controller that contains the variable. This simple implementation serializes concurrent synchronization accesses. Higher performance can be achieved using a technique called *combining*. Since combining is also applicable to barriers and multiprefix operations, a discussion of it is deferred to Section 5.2.6.

5.2.3 Lock Implementations

Efficient locks can be implemented in software using lower level fetch&op instructions. The most effective technique is the MCS algorithm [MCS91], in which all spinning is done on local variables and minimal memory is required. The simplest version of the MCS algorithm is constructed using fetch&store and compare&swap primitives. The fundamental idea of the MCS algorithm is to maintain a distributed linked list of waiting processors, with the head of the list kept in a shared lock structure; this is analogous to the distributed linked list directory in Figure 2.3. All of the links other than the root are stored locally at the processors that are waiting. If a processor issues a lock request, it allocates a local link structure and appends itself to the list pointed to by the shared lock structure. When a processor issues an unlock request, it grants the lock to the next processor on the list and deallocate its local link. Details about how the list is updated atomically and about how spinning is performed locally are in [MCS91].

The MCS lock algorithm has the following properties:

1. It scales well.

2. It only requires space proportional to the number of processors that contend for a lock; this may be as high as O(N) but would typically be to O(1).

3. It requires the least amount of network traffic of all known software lock algorithms.
4. If a compare&swap primitive is available, it ensures that a lock is granted in FIFO order, guaranteeing fairness.
5. The time to access a free lock by a single processor is within a factor of two of the fastest algorithm.

Several techniques have been published on the implementation of locks completely in hardware [L+90, BD86, G+89, IEE90]. They are similar to the MCS algorithm in that a list is maintained of all processors waiting on a lock. The first class of hardware techniques maintains a distributed linked list using cache lines as link entries [IEE90]. The algorithm is essentially the same as the MCS algorithm, except that it is implemented entirely in hardware. This class of scheme is well suited to cache coherent multiprocessors employing distributed linked lists to store cache directory information (Section 2.3.3). Although the published schemes all use cache lines as links, they could be easily modified to use local (non-cache) memory.

The second class of hardware techniques maintains a centralized list at the main memory controller corresponding to the lock address [L+90]. These techniques typically exploit the hardware required by directory methods that maintain centralized directory information (Section 2.3.3). If directories are implemented as linked lists, the lock algorithm is essentially the same as the MCS algorithm except the links are all kept at one memory controller. If directories are implemented as bit vectors, process identifiers are kept as sets and FIFO information is lost.

Care must be taken with lock implementations that exploit cache coherence hardware to ensure that list information is not lost when cache lines are displaced or directory entries re-used. If cache lines holding lock links are "locked" in a set-associative cache, a deadlock situation can arise when all the entries of a set are assigned to lock link structures; if this situation arises and a cache miss occurs on a block that maps to the same set, special action must be taken to ensure that the processor does not deadlock and that list information is preserved. A similar situation can arise in a centralized linked-list directory scheme, in which there is usually a limited supply of link structures that is subject to exhaustion. Since this is a (hopefully) rare situation, lock operations that find no free links can be forced to retry until a link becomes available.

Empirical results in [MCS91] indicate that there is little performance to be gained with hardware implementations of locks, so they are desirable only if the extra hardware cost is very small. For multiprocessors with linked list cache coherence schemes, the added cost is a slightly more complex state machine in the memory or cache controllers, so the extra cost is minimal.

5.2.4 Barrier Implementations

Like locks, barriers can be efficiently implemented in software using fetch&op primitives. The tree barrier of Mellor-Crummey and Scott is representative of the most efficient algorithms [MCS91]. The tree barrier algorithm minimizes network contention by combining barrier requests in software. Each process is mapped to a node in a tree. When a process arrives at a barrier, it waits until all of its children arrive at the barrier and then indicates its arrival to its parent. When all processes have arrived, the root initiates wakeup by issuing "wake-up calls" to its children. When wakened, children wake up their children and the process repeats until all processes are resumed. The detailed algorithm features a number of low-level optimizations that are described in [MCS91].

The tree barrier algorithm has the following properties:

1. It has a critical path of length O(log N).

2. It only requires space proportional to the number of processors that meet at the barrier.

3. It requires O(N) network traffic, the minimum possible.

4. All spinning is performed on local variables.

Hardware barrier implementations have been published in [HRS88b, Hos89, GS89]. In [Hos89], all processors are connected to a large AND gate that asserts a barrier signal when all processers arrive at a barrier. When the barrier signal is asserted all processes are simultaneously wakened and execution continues. This scheme severely restricts the number of barriers that a multiprocessor can support at one time. The schemes in [HRS88b, GS89] implement barriers using a synchronization bus connected to all processors. By multiplexing the bus, a moderate number of separate barriers can be supported. Barrier algorithms based on fetch&op can also be considered hardware implementations because they rely on special *combining* hardware [AG89]. Combining hardware permits concurrent synchronization accesses to the same address to be satisfied simultaneously (or almost simultaneously). A detailed discussion of combining is in Section 5.2.6. Pseudo-code for a combining barrier is shown in Figure 5.2. It uses fetch&increment, fetch&decrement and fetch&no-op [1] operations. When a process arrives at a barrier it increments or decrements the arrival count based on its local sense flag (for efficiency, the sense flag obviates the need to reinitialize the arrival count after each barrer). If, based on the arrival count, the process is the last to arrive, it clears or sets the global arrival flag. If the process is not the last to arrive, it repeatedly

[1] fetch&no-op is just a read operation that can be combined.

reads the global arrival flag until it indicates that all processes have arrived. With a combining network, the minimum time to perform the barrier operation is only two round trip network crossings, independent of the number of processors. In spite of its low latency, a great deal of network traffic is generated as the processors spin on the wake-up flag. Section 5.3.3, however, shows how the polling traffic can be eliminated by making simple modifications to a particular implementation of a combining network.

The AND gate and bus barrier schemes permit barrier operations in tens of processor cycles, which is considerably faster than the best software algorithms. Barrier implementations that rely on a combining network can perform a barrier operation in a minimum of 2 round-trip network crossings. If a multistage network is used, this is $\log N$ fewer crossings than required by a good software scheme. If a single stage network is used, the performance improvement is less because the software combining tree can be mapped onto the network of processors to minimize the number of "hops" between tree nodes.

5.2.5 Implementation of Multiprefix Operations

Implementations of multiprefix operations are considerably more complex than barrier or lock algorithms because of the order in which the binary function F must be applied. Efficient hardware and software implementations have been published in [Ran87] and [Coh90], respectively.

The hardware scheme in [Ran87] employs randomization to construct an algorithm with time and space complexity $O(\log N)$ and $O(N)$, respectively, on a butterfly network with a modest amount of combining hardware. Although it is described in the context of special purpose combining hardware, it can also be completely implemented in software using the techniques of Section 5.3.2. The algorithm is easily generalized to networks other than the butterfly.

The software multiprefix algorithm in [Coh90] requires only basic communication facilities among processors. It assumes a binary hypercube interconnection network, but can be generalized to other networks. The algorithm is complicated, and a detailed description of it is beyond the scope of this summary. Time complexity is $O(\log N + T_{SORT}(S, S))$, where S is the length of the largest list of values, and $T_{SORT}(M, P)$ is the time to sort M items on P processors with a given multiprocessor network. Space complexity is $O(N)$.

Although the time complexity of both algorithms is the same, the software scheme is probably less efficient by a large constant factor. The hardware scheme scheme can also be pipelined, permitting multiprefix operations to be applied at a much higher rate.

```
/* globally shared barrier structure */
shared structure {
    /* nprocs initialized to # procs. arriving at barrier */
    int         nprocs;
    int         sense = 0;
    int         count = 0;
    boolean     all_arrived = FALSE;
} BARRIER;

/* barrier code */
void barrier(bar)
BARRIER   bar;
{
    local int count;

    if (bar.sense == 0) {
        count = fetch&inc(bar.count);
        if (count == bar.nprocs) {
            bar.all_arrived = TRUE;
        }
        else {
            while (fetch&nop(bar.all_arrived) == FALSE) {
                /* do nothing */
            }
            bar.sense = 1;
        }
    }
    else {
        count = fetch&dec(bar.count);
        if (count == 0) {
            bar.all_arrived = FALSE;
        }
        else {
            while (fetch&nop(bar.all_arrived) == TRUE) {
                /* do nothing */
            }
            bar.sense = 0;
        }
    }
}
```

Figure 5.2: Pseudo-code for Combining Barrier Algorithm

β operations are an important subset of multiprefix operations that can be implemented with fetch&op. This is because β operations do not impose an ordering constraint on the application of the binary function F. The implementation requires a distinct synchronization address a_i for each of the L value lists (Eqn. 5.1). The contents of a_i are initialized to l_{i1}. A β operation with function F is performed by having all processors $proc(l_{ij})$, $j \neq 1$, issue $fetch\&F(a_i, l_{ij})$. The result for list i is then the contents of a_i.

The use of fetch&op within a β operation provides a good example of where the high bandwidth of a hardware combining scheme can be exploited. This example considers how pipelined fetch&op's can be be used to emulate a very large data parallel computer (many thousands of processors) on a much smaller shared memory machine (several hundred processors). Emulation requires that the thousands of virtual processors be mapped onto several hundred physical processors. Given this mapping, the β operation can be performed by having each physical processor execute the following:

```
/* issue fetch&op's for all emulated processors */

for (all $l_{ij}$ on this processor) {
    deferred_fetch&F($a_i$, $l_{ij}$, address_of_result);
}

/* wait for fetch&op's to complete */
fence;
```

Each physical processor serially issues fetch&op accesses for each of the list elements mapped to it. With pipelined combining hardware, the fetch&op's can be issued at a very high rate. This can only be done, however, if processors can defer access to the results. The particular deferred access mechanisms used here are:

- A deferred access fetch&op primitive, which does not force the processor to wait for the result. The deferred fetch&op primitive requires an extra operand: a pointer to the memory location in which the result is placed.

- A *fence* instruction, which forces the processor to stall until all outstanding references are completed. This requires the processor to maintain a count of outstanding accesses. Fence instructions have been suggested elsewhere for similar synchronization functions [B+85, G+90a].

Assuming that most of the fetch&op operands and results reside in the local cache, most loop accesses (except the fetch&op's) will be satisfied locally. At the end of the loop the fence operation stalls the processor while the final fetch&op completes. With a sufficiently large number of virtual processors, the stall time should be small compared to loop time, and good utilization should be achieved. Good hardware support for fetch&op primitives provides a powerful way to emulate data parallel computation.

5.2.6 Combining

In some programming models, simultaneous accesses to the same location are performed concurrently, with the same result as if they were performed in some arbitrary serial order. This powerful abstraction is known as *combining*, and can be used to construct efficient parallel algorithms, including parallel queues, β operations, iteration assignment in parallel loops, and barriers [AG89]. Because of fanout restrictions, the combining abstraction can only be approximated in an implementation. In the simplest implementation, n concurrent fetch&op's are performed in serial order, requiring $O(n)$ time. If the "op" in fetch&op is associative, however, the time can be reduced to $O(\log n)$ by reordering the linear dependencies of a serial order into a tree. Implementations that use this technique are said to support *combining*. Two or more fetch&op accesses that generate an intermediate value are said to be *combined.*

Combining can be performed in hardware or software. One of the first combining implementations was developed for the multistage network in the NYU Ultracomputer [G+83c]. This implementation exploited the fact that simultaneous accesses to the same address pass through a tree of switching elements, with the root at the memory bank holding the addressed data. The tree of switching elements provides a natural way to reorder a serial application of the simultaneous accesses. Hardware within each switch detects two or more simultaneous accesses to the same location, and generates an intermediate result. To reduce network traffic, the switch forwards a single access with the intermediate operand, and retains a record of the accesses that are combined. When a result for the intermediate access is returned, it is split into return values for each of the combined accesses. Variations of this combining scheme have been published in [P+85, SB77, TR88, R+90]. The tree barrier described in Section 5.2.4 is an example of combining in software. Other examples are in [G+89, PCYL87].

There are two common combining situations with special properties that can be exploited in an implementation. The first is combining in the context of a *synchronous* reference model. In a synchronous reference model, processors issue shared accesses in lock-step. Several algorithms have been published that exploit this synchrony to provide provably efficient combining [Ran87]. These efficient combining schemes can also be used in an asynchronous reference model by emulating asynchronous accesses. Emulation, however, introduces overhead that can reduce performance. This was discussed in Section 4.2.3.

A second important combining situation arises in barrier and β algorithms based on fetch&op primitives. In barrier and β operations, a known set of processors issue fetch&op accesses to the same address in synchrony. We denote such fetch&op accesses as *static* fetch&op accesses. fetch&op accesses that do not statisfy these restrictions are *dynamic*. *static* fetch&op accesses can be exploited in efficient combining implementations [Ran87], which we call *static* combining schemes. Alternatively, implementations supporting *dynamic* fetch&op accesses are called *dynamic* combining schemes. The software and hardware barrier schemes in Section 5.2.4 are examples of static combining. Section 5.3.3 analogous ways to support multiprefix operations.

Most published hardware combining schemes support dynamic combining with asynchronous references. In its full generality, hardware implementations of asynchronous combining are very expensive, so most published implementations restrict the number and type of simultaneous combining operations that can take place. [SSG89] and [HRS88a] show how fetch&increment and fetch&decrement can be combined efficiently using a global synchronization bus. [R$^+$90] describes an efficient implementation of read combining. [TR88] and [LV88b] describe efficient combining schemes for networks with bit-wide datapaths. Published implementations of combining hardware for networks with wide paths and less restrictive operations have been very expensive; [PN85] estimated that a combining switch for a multistage network would require 6 to 24 times as much hardware as comparable non-combining network. Combining with synchronous references can be implemented efficiently, but synchronizing processor references has a potential performance penalty.

Hardware combining support is attractive for two reasons. The first is that latency of a combining operation can probably be reduced by a factor of 2 or more over a software implementation. The second is that a software implementation cannot be pipelined; the bandwidth of combining hardware is therefore higher because of reduced latency *and* pipelining. It is likely that combining hardware can be sufficiently pipelined to permit fetch&op requests to be accepted by a switch every 2 cpu cycles, with a processing latency of 20 or less (see [Joh90] and the combining implementation described in Section 5.3). A software implementation of either algorithm (using techniques described in Section 5.3.2) would probably require at least 40 cycles per fetch&op, and could not be pipelined. A hardware scheme would therefore increase bandwidth by an order of magnitude.

5.2.7 Summary

Some form of synchronous or asynchronous combining is necessary in software or hardware to construct many scalable parallel algorithms. In particular, static combining is useful for implementing efficient barriers and multiprefix operations, and dynamic combining is useful for implementing efficient parallel queues. Combining in hardware is attractive because of the much higher bandwidth it offers. A provably efficient algorithm is known for implementing dynamic combining in hardware for a synchronous reference model (or, using the techniques of Section 5.3.2, in software). Although the algorithm can be modified to emulate combining with asynchronous references, there is a potential performance penalty. On the other hand, dynamic combining hardware can be easily modified to support static combining. One approach is described in Section 5.3.3. Section 5.3 describes a dynamic combining implementation that should require far less hardware than conjectured in [PN85]. Furthermore, there seems to be no reason why the performance of the implementation should be significantly worse than previous dynamic combining schemes.

5.3 Efficient Implementation of Dynamic Combining

As previously mentioned, most published implementations of dynamic combining for networks with wide paths have been very expensive. This section presents a hardware technique that requires much less hardware than previously conjectured [PN85]. Because the design applies to networks built from jxk crossbar switches, it can be used in Delta networks, direct k-ary n-cubes and meshes. The same ideas can be used to construct an analogous software algorithm. The software equivalent is described in Section 5.3.1. This algorithm is simpler than that published by Goodman [G$^+$89], and requires no spinning. Section 5.3.3 shows how the hardware scheme can be easily modified to support barriers without the large number of network transactions required by a conventional combining barrier (Figure 5.2). The modifications result in what is essentially a hardware implementation of the software barrier algorithm described in Section 5.2.4. Similar modifications that support multiprefix operations are presented in Section 5.3.3.

5.3.1 Efficient Dynamic Combining in Hardware

The combining hardware described here assumes that combining is only required for fetch&op instructions, which are distinct from reads, writes and coherence transactions. It is also assumed that the multiprocessor network is constructed of jxk crossbar switches, with a routing algorithm that ensures that the response to a memory request returns on the same path as the request.

Figures 5.3 and 5.4 show the proposed combining hardware. Figure 5.3 is a typical kxk switch implementation (Figure 4.7) augmented with an extra input and output port on the crossbar switch, which are connected to a *combiner* composed of a combining table (Figure 5.4), an arithmetic/logic unit, and a controller. For simplicity, the combiners operate serially, so only one combining transaction can be handled at a time. All fetch&op transactions are routed through the combiner twice: once when they arrive at the switch and again after a combining entry has been established. The time interval while the transaction is in queue is the *combining window* during which other fetch&op transactions can be combined with the first. When it first arrives, a fetch&op transaction is compared against other buffered fetch&op transactions to see if the same operation is being made on the same location. If there is a match, the transaction is combined and the transaction need not be forwarded. If there is no match, a new table entry is inserted and the transaction is put back on the combiner queue; the new table entry is marked *active* to indicate that it is eligible for combination with future transactions. When a transaction makes a second pass through the combiner, the combiner determines if any combining has taken place since the first pass. If combining has occurred, the table entry must now be marked *inactive* and the fetch&op operand must be replaced with the value of the combined operand. If no combining has occurred, the entry is deleted since no decombining is required. There are clearly many ways in which the combiner queue can arbitrate among new combining transactions and transactions making a second pass.

The combining table provides associative access to combining entries. This can be implemented efficiently using a set-associative organization (Figure 5.4). In a set-associative organization [Smi82], the table is composed of n_l lines of n_s entries per line. To access entry x, a line is selected by mapping x to some value between 0 and $n_l - 1$; this mapping is typically done by making n_l a power of 2 and stripping off the $\log_2 n_l$ low order bits of x. The addresses of the n_s entries of the selected line are then compared, in parallel, to find the desired entry.

The identifier of a combining table entry has three components: the *address* of the synchronization variable, the fetch&op *operation*, and the *instance id*, which is used to differentiate two or more distinct combining entries for the same address. The need for instance id's is illus-

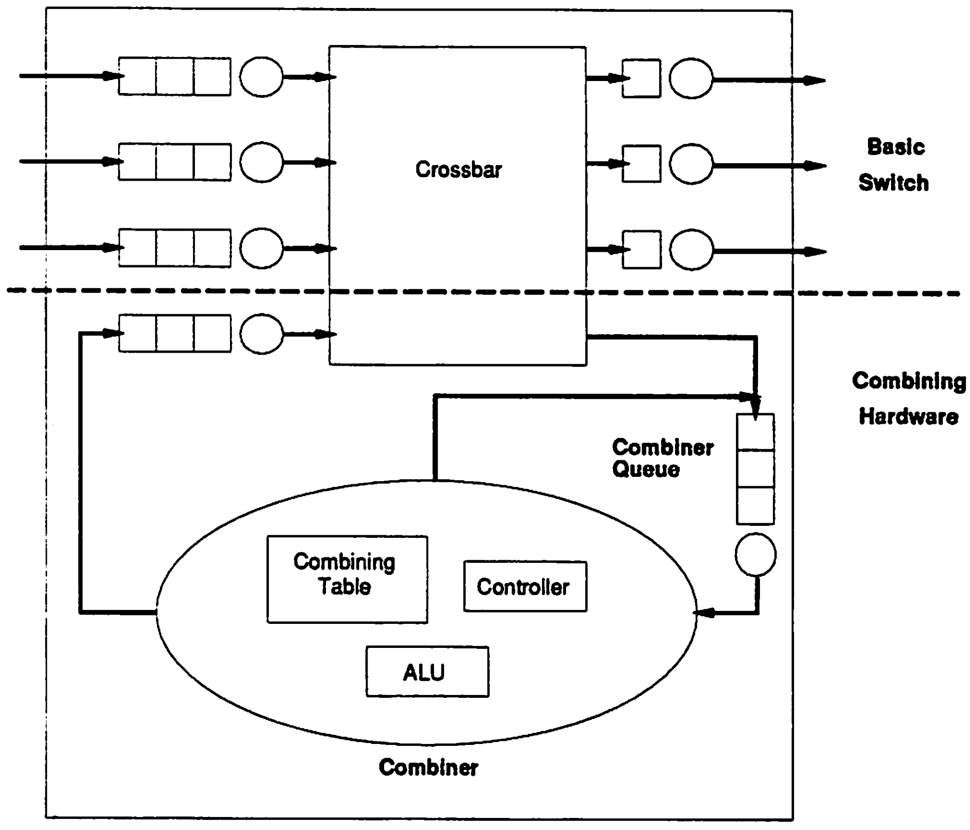


Figure 5.3: Combining Switch Architecture

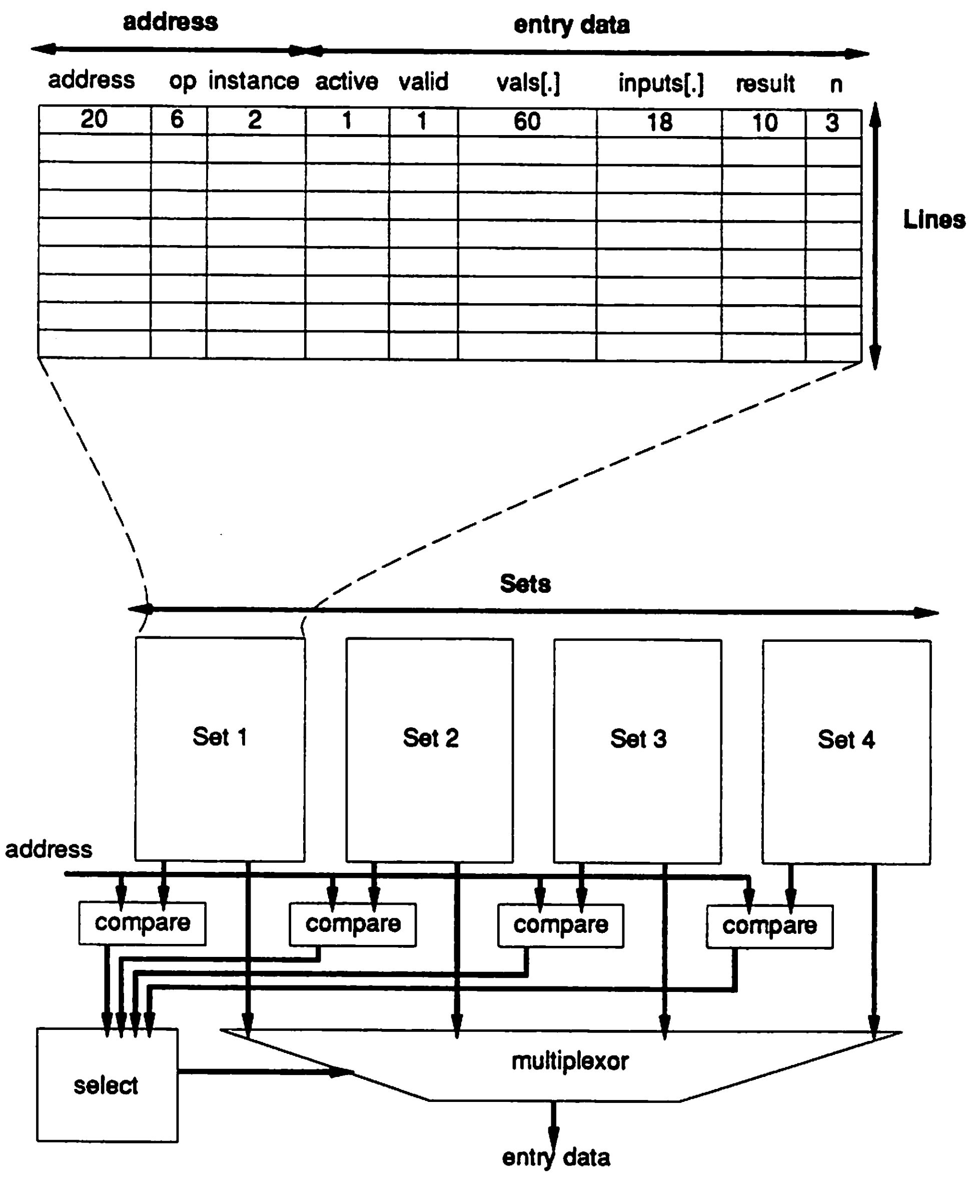


Figure 5.4: Combining Table Architecture

trated as follows. At time t_1 two fetch&add transactions arrive for address x and are combined. At time t_2, after the first combined fetch&add transaction has been forwarded, two other fetch&add transactions arrive for the same address. If these transactions are combined, they require a separate table entry and the entries must be differentiated by distinct instance id's.

A combining table lookup requires these steps:

1. Select the table line using the $\log_2 n_l$ bits of the fetch&op address.
2. In parallel, compare the address in each of the n_s sets against the fetch&op address and operation.
3. For the entries whose addresses and operations match, examine the valid bits, active bits and instance id's. If an active entry exist, combine the fetch&op. If one or more inactive entries exist, find the largest instance id id_{max}. Create a new entry and assign it an instance id of $id_{max} + 1$. If no free entries exist for this line, forward the fetch&op to the next switch.

The number of sets in the combining table limits the number of combining instances that can be stored for a particular address. Since more than one address can map to the same line in the table, the maximum number of combining instances may be less. We expect 4 to be a reasonable number of sets.

A combining entry is composed of 8 fields (Figure 5.4) (values in brackets indicate the number of bits devoted to this field assuming: switch fanin of 6, 1024 processors, 32 bit fetch&op address, 32 bit limit on fetch&op operands, 6 bit op-field, $n_s = 4$):

- fetch&op address (32)
- fetch&op operation (6)
- instance id (2)
- n_c: the number of combinings that have taken place (4)
- $vals_i[]$: an array of the fetch&op operands corresponding to the n_c combined transactions. This array has a maximum number of entries corresponding to the fanin of the switch. (6*32)
- $inputs_i[]$: an array of the switch input numbers corresponding to the n_c combined transactions This array has a maximum number of entries corresponding to the fanin of the switch. (6*3)

- an active flag, indicating whether or not combining can still occur for this entry (1)
- a valid bit, indicating whether or not the entry is free (1)
- the cummulative value of performing the fetch&op operation on all n_c $vals$ (32)

With the above assumptions, 36 bytes are required per entry, which is comparable to the size of a cache line. Several thousand entries could therefore be provided at reasonable cost.

The time to process a fetch&op transaction with this scheme will clearly be considerably longer than the time required to process a read or write; we estimate 5 to 10 times as long. Because of this, combining should be restricted to variables that are known to cause significant contention. The extra processing time, however, will certainly be no longer that the time required to do the equivalent combining operation in software.

The complexity of this combining scheme is minimized by handling only one combining or decombining transaction at a time. This is in contrast with other designs that attempt to combine two or more transactions simultaneously, requiring sophisticated multiport queues and tables [DKSS85, Lee87, DGK86, LV88b, SB77]. The impact on the performance of reads, writes and coherence operations should be small because the modifications to the basic switch are minimal: the buffer controllers have a few extra states, the buffer decoders have an extra case to consider, and the size of the crossbar is increased by one.

There are many ways to optimize the performance of the basic architecture of Figures 5.3 and 5.4, including:

1. Pipeline the combiner.
2. Provide a two ported combining table and separate *decombiner*.
3. Provide a mechanism to permit unblocked fetch&op transactions to bypass the combiner. The effectiveness of this optimization is probably workload dependent.

5.3.2 Efficient Dynamic Combining in Software

The hardware combining scheme of the previous section can be used as the basis of an analogous software scheme. The techniques described here could also be used in a software version of Ranade's combining algorithm [Ran87].

In the hardware scheme, a combining tree is dynamically mapped to a tree of switches in the network. In the software scheme, processors are dynamically mapped to nodes in a logical combining tree. Each node contains a table of combining data structures that is analogous to the hardware table in Figure 5.4. Distinct trees are used for distinct synchronization variables. For maximum performance, each mapping should attempt to minimize the communication required to traverse the tree. Figures 5.5 through and 5.7 provide a sketch of the software algorithm. They assume the availability of the following message passing primitives:

1. *send_and_interrupt(proc, msg)*: *proc* is interrupted and *msg* is passed to an interrupt handler; the sender does not block. In dynamic combining, the sequence of combining operations is not known in advance, so processors must be permitted to interrupt each other.

2. *send(proc, msg)*: *msg* is sent to *proc*; the sender does not block and *proc* can receive the message only with the *receive* primitive.

3. *receive(proc, msg)*: the receiver retrieves *msg* from *proc* if one has arrived, or blocks until one does.

These primitives could be built from lower level synchronization primitives and an interrupt facility. The following different *msg*'s are used:

1. *start_fetch&op(addr, val)*

2. *end_fetch&op(addr)*

3. *arrival_combined(addr)*

4. *arrival_not_combined(addr)*

5. *result(addr, val)*

As in the hardware scheme, two accesses to the combining table for each fetch&op (that is not itself combined) provide a window of time during which other fetch&op's can combine with the first. It is assumed that the interrupt handler enforces mutual exclusion for each table access. Processors send fetch&op transactions to their parent in the combining tree. When a *start_fetch&op(addr, val)* first arrives at a parent, a table lookup is performed for an entry corresponding to *addr* and *op*. If an active entry exists, the transaction is combined and *arrival_combined(addr)* returned to the child. The child then waits for a result. If an active entry does not exist, an active one is created with a distinct instance number, and *arrival_not_combined(addr)* returned to the child. The child must now wait for some length of time and issue *end_fetch&op(addr)* to the parent. The parent now

```
static_fetch&op(addr, val)
{
    send(parent, start_fetch&op(addr, val));
    receive(parent, msg);
    switch (msg) {
        case arrival_combined(addr):
            /* do nothing */

        case arrival_not_combined(addr):
            wait until combining window is over;
            send(parent, end_fetch&op(addr));

        case result(addr, val):
            return(val);
    }

    receive(parent, msg);

    switch (msg) {
        case result(addr, val):
            return(val);
    }
}
```

Figure 5.5: Software Combining Algorithm: fetch&op Routine for Leaves

```
interrupt_handler(msg, source)
{
    switch (msg) {
        case start_fetch&op(addr, val):
            if (table entry for addr) {
                send(source, arrival_combined(addr));
            }
            else {
                create a table entry;
                send(source, arrival_not_combined(addr));
            }
            return;

        case end_fetch&op(addr):
            get table entry for addr;
            send(parent, start_fetch&op(addr, val));
            return;

        case result(addr, val):
            get table entry for addr;
            for all children {
                send(child, result(addr, val));
            }
            delete table entry for addr;
            return;

        case arrival_combined(addr):
            /* do nothing */
            return;

        case arrival_not_combined(addr):
            wait until combining window is over;
            send(parent, end_fetch&op(addr));
            return;
    }
}
```

Figure 5.6: Software Combining Algorithm: Interrupt Handler for Interior Nodes

```
interrupt_handler(msg, source)
{
    switch (msg) {
        case start_fetch&op(addr, val):
            perform op;
            send(source, result(addr, val));
            return;
    }
}
```

Figure 5.7: Software Combining Algorithm: Interrupt Handler for Root

marks the corresponding table entry inactive and forwards a (possibly) combined *start_fetch&op* message to its parent. At the root of the combining tree, the sole copy of the combining variable is operated on and the values prior to each operation are returned via *result* messages to the children that issued *start_fetch&op*'s. The children calculate result values and return *result* messages to their children until the process completes.

The relative performance of the hardware and software combining schemes, like the barrier schemes, is dependent on whether a single or multiple stage interconnection network is used. With a multistage network, each arc in the combining tree requires the crossing of log N and 1 network links by the software and hardware schemes, respectively. In this case the hardware scheme is clearly superior.

With a single stage network, the combining tree can be statically mapped to the network topology in such a way that the link crossings would be similar for both schemes, and in this case a more detailed study is required. The software scheme has the significant advantage that it can support an arbitrary set of fetch&op operations. In particular, floating point operations could be easily provided. The software scheme, however, "steals" cycles from the processors to execute algorithm code and to do interrupt handling. For each fetch&op request from a single processor, log N processors must execute code from some portion of the algorithm, so the fraction of stolen cycles could become significant for finer grained parallel computations.

5.3.3 Modifications to Support Static Combining

The dynamic combining hardware in Figure 5.3 can be easily modified to support *synchronous* combining primitives such as barriers and multiprefix operations. Since each use of a static combining primitive involves a predetermined set of processors, a combining tree can be programmed into the combining hardware using special setup commands. When this is done, the combiner corresponding to a particular node in the combining tree can wait until all of its children have been combined before forwarding a fetch&op primitive to its parent. As before, an analogous scheme can be implemented in software.

Combining tree information can be stored in the combining table fields as follows. The number and identities of children are stored in the *result* and *values* fields, respectively. The count of children that have synchronized is kept in the combining count (n in Figure 5.4). To set up the combining table, a *register(addr, op)* transaction is sent at initialization time by all processors that participate in the combining tree. When a combiner receives a *register* request, it locates the corresponding table entry (allocating one if necessary), records the input number of the originating switch, and increments the child count. When a combiner receives a static combining transaction, it locates the corresponding table entry, combines the transaction with those that have already arrived, and increments the running child count. If all children have arrived, a single combined transaction is forwarded to the combiner's parent switch. At the root combiner, the synchronization variable is updated and the results are disseminated to the children recursively. The cost of these modifications are slightly more complex controllers for the combiner and main memory.

Since the combining table is of limited size, *register* requests may sometimes fail. If this occurs, *register_failure* transactions are forwarded to the root and to all participating processors, and some backup algorithm is used. The root must somehow remember that failure has occurred so that processors sending *register* requests in the future are notified of the failure; this could be done by storing a reserved value at the root.

It is easy to construct efficient barrier and multiprefix operations using this synchronous combining scheme. A barrier operation can be implemented using a static fetch&op with arbitrary "op" and discarding the result. A multiprefix operation can be implemented like the β operation in Section 5.2.5. Static fetch&op's must be used, and a register request is required to initialize combining table entries.

5.4 Conclusions

This chapter has summarized known techniques for implementing common shared memory synchronization primitives in hardware and software, indicating their advantages and disadvantages. A technique was described for implementing static and dynamic combining in hardware or software. It was also shown how this technique readily supports barriers and multiprefix operations. Such a technique is desirable because it supports the efficient emulation of many useful parallel programming paradigms, including those used in traditional shared memory programming environments, the Connection Machine [Hil85], the Fluent abstract machine [RBJ88], the Ultracomputer [Sch80], and the EPEX parallel Fortran model [D+88]. A duality exists among most of the described hardware and software synchronization techniques, so that each offers the same asymptotic performance. The hardware schemes should be faster by a constant factor, however, at the expense of extra circuitry. For combining operations the performance improvement should be considerable because the implementation can be pipelined. It is estimated that the proposed hardware combining scheme should require about 1.5 to 2 times the circuitry of a basic 7x7 network switch. Such a network switch is probably pin-limited, so the added cost appears minimal compared to the potential performance gain.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation implementation and evaluation techniques were presented for directory-based cache coherence schemes. Although the focus has been on scalable cache coherence, a thorough performance evaluation required detailed consideration of interconnection networks and synchronization schemes.

In Chapter 2, known methods were reviewed for efficiently implementing directories. The tag caching scheme was introduced. There are now a variety of efficient directory implementations that do not restrict the number of copies of a block.

In Chapter 3, the performance of full directory protocols were evaluated using execution driven analysis of three parallel programs. It built upon previous work in several ways. First, it applied efficient stack simulation techniques to the evaluation of directory protocols, extending the work of [Tho87] to support a competitive scheme. This is significant because it permits multiprocessors with several (uniform) cache sizes to be evaluated in in a single simulation run. Chapter 3 also introduced the concept of an *update-run* to a cached block. *update-runs* supplement the ping/cling and write-run locality models with a measure of invalidation/update traffic suitable for point-to-point networks.

The empirical data in Chapter 3 extends previously published data by evaluating update and competitive directory schemes, and by considering larger numbers of processors. Unlike Eggers' shared bus study [EK89b], the results here account for the extra point-to-point message traffic required for invalidates or updates. When considering point-to-point traffic, the update (competitive) protocols generated 0 to 300% (0 to 60%) more network traffic than the invalidation protocol.

Since the average number of updates per shared writes ranged from 4 to 16 (2 to 7) for the update (competitive) simulations, this was not surprising. Although considerable extra traffic was generated by the update and competitive protocols, the reduction in misses was substantial: read misses were reduced by 20 to 70% (15 to 60%). With buffered writes and blocking on read misses, the miss reductions corresponded to only modest increases in processor utilization (5 to 15%).

The growth of coherence traffic with multiprocessor size was also considered Chapter 3. The frequency of invalidations, updates and invalidation misses on a per-processor basis increased approximately linearly with the number of processors. Invalidations per shared write grew from 0.2 to 2.2 when the number of processors grew from 4 to 128. For the update (competitive) protocol, updates per shared write grew from 2 to 16 (0.5 to 6) over the same range of sizes. The simple update and competitive protocols considered here are therefore inadequate for more than 200 processors without some additional mechanism to further limit update traffic. The invalidation protocol appears suitable for more than 500 processors.

New network modeling techniques developed in Chapter 4 permitted a systematic comparison of a broad class of interconnection networks. The techniques built upon the parametric decomposition scheme used in the Bell Laboratories Queueing Network Analyzer and finite buffer algorithm of Altiok and Perros to support realistic network features: virtual cut-through flow control, and the use of virtual channels for congestion reduction. Furthermore, the models support arbitrary Markovian routing and two moment approximations of arrival processes. The models are also reasonably efficient, offering performance estimates in 5 to 100 times less cpu time that simulations. Results for hypercube, multistage, three dimensional toroidal mesh, and three dimensional mesh topologies showed the tradeoffs available among bandwidth and latency. The traffic estimates of Chapter 4 suggest that a three dimensional mesh topology with wide paths is a good compromise for several hundred processors.

The caching analysis in Chapter 3 assumed that the synchronization primitives used in the benchmarks (locks and barriers) had ideal implementations: This was done so that the comparison of coherence protocols would be unbiased by coherence activities generated by naive synchronization techniques. Chapter 5 justified this assumption by describing efficient implementations for locks and barriers, in hardware and software. Section 5.3 also described software and hardware implementations for combining with an asynchronous reference model. Section 5.3 showed how these implementations could be modified to support multiprefix operations, which offer direct support for several powerful programming paradigms. It is estimated that the proposed hardware combining scheme should require about 1.5 to 2 times the circuitry of a basic 7x7 network switch.

Such a switch is probably pin-limited, so the added cost appears minimal compared to the potential performance gain.

6.2 Future Work

This work can be extended in several ways. First, there is a need for better benchmark data, and evaluation techniques with much greater efficiency. The empirical data in Chapter 3 was collected from about 1 second of execution of three programs on a relatively small multiprocessor (64 cpu's). Furthermore, the three programs represent only a small class of programming paradigms. Innovative emulation techniques are desperately needed to permit large parallel programs to run to completion on realistically large data sets. Current trace-driven techniques are slow because they are too detailed. A better understanding of the significance of various modeling details would permit an intelligent tradeoff of accuracy for speed. A step toward this goal would be an investigation of trace sampling [LPI88].

There are several ways in which the network modeling techniques of Chapter 4 can be extended. It may be possible to model some of the recently proposed adaptive routing schemes [LH91]. Since adaptive routing schemes adjust the Markovian routing parameters based on network load, one approach is to place the algorithm of Figure 4.15 inside an additional iterative loop that adjusts the routing parameters based on the network load observed in the previous iteration. Modifications similar to those used to model virtual channels may permit the support of multiple message priorities. Priorities may be useful in improving the performance of synchronization and other critical transactions. Finally, more efficient modeling techniques for virtual channels and superposition could greatly reduce the currently large computation times for large networks.

Finally, the combining hardware described in Chapter 5 needs to be thoroughly evaluated and compared against competing schemes, such as Ranade's Fluent implementation [RBJ88].

Bibliography

[A+85] M. Ajmone Marsan et al. Generalized stochastic petri net models of multiprocessors with cache memories. In *International Conference on Supercomputing Systems*, pages 690–696, 1985.

[A+88a] A. Agarwal et al. An evaluation of directory schemes for cache coherence. In *Proceedings of the International Symposium on Computer Architecture*, pages 280–289, May 1988.

[A+88b] F. Allen et al. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[A+89] Y. Afek et al. A lazy cache algorithm. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 209–222, 1989.

[AB84] J. Archibald and J-L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the International Symposium on Computer Architecture*, pages 355–362, 1984.

[AB86] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov. 1986.

[ABC86] M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems*. MIT Press, 1986.

[ABC+89] M. Ajmone Marsan, G. Balbo, G. Chiola, A. Ciccardi, and G. Conte. Estimating the average delay in a delta interconnection network operating according to the cut-through packet switching technique. *Performance of Distributed and Parallel Systems*, pages 491–510, 1989.

[AG88] Anant Agarwal and Anoop Gupta. Memory-reference characteristics of multiprocessor applications under mach. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 215–225, 1988.

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 1989.

[AG90] Anant Agarwal and Anoop Gupta. Temporal, processor and spatial locality in multiprocessor memory references. *Frontiers of Computing Systems Research*, 1:271–295, 1990.

[AH90] Sarita Adve and Mark Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the International Conference on Parallel Processing*, pages 47–50, 1990.

[Aky88] Ian F. Akyildiz. Mean value analysis for blocking queueing networks. *IEEE Transactions on Software Engineering*, 14(4):418–428, April 1988.

[AN87] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *Proceedings of the PARLE Conference, Eindhoven, June 1987; in Lecture Notes in Computer Science*, pages 1–29. Springer-Verlag, 1987.

[AP87] T. Altiok and H. G. Perros. Approximate analysis of arbitrary configurations of open queueing networks with blocking. *Annals of Operations Research*, 9:481–509, 1987.

[AP89] Seth Abraham and Krishnan Padmanabhan. Performance of the direct binary n-cube network for multiprocessors. *IEEE Transactions on Computers*, 38(7):1000–1011, July 1989.

[AS83] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, March 1983.

[ASK85] W. Abu-Sufah and A. Y. Kwok. Performance prediction tools for cedar: A multiprocessor supercomputer. In *Proceedings of the International Symposium on Computer Architecture*, pages 406–413, 1985.

[B$^+$84] D. Bitton et al. A taxonomy of parallel sorting. *Computing Surveys*, 16(3):287–318, Sept. 1984.

[B$^+$85] W. C. Brantley et al. RP3 processor-memory element. In *Proceedings of the International Symposium on Computer Architecture*, pages 782–789, June 1985.

[BD81] F. A. Briggs and M. Dubois. Cache effectiveness in multiprocessor systems. In *Proceedings of the ACM Conference on the Measurement and Modeling of Computer Systems*, pages 306–313, 1981.

[BD86] P. Bitar and A. M. Despain. Multiprocessor cache synchronization: Issues, innovations, evolution. In *Proceedings of the International Symposium on Computer Architecture*, pages 424–433, June 1986.

[Bel82] Peter C. Bell. The use of decomposition techniques for the analysis of open restricted queueing networks. *Operations Research Letters*, 1(6):230–235, December 1982.

[BH89] Eugene D. Brooks and Joseph E. Hoag. A scalable coherent cache system with fuzzy directory state. Technical report, University of California, Lawrence Livermore National Laboratory, 1989.

[BK88] Richard G. Born and James R. Kenevan. Analytic derivation of processor potential utilization in straight line, ring, square mesh, and hypercube networks. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 94–103, 1988.

[Ble89] G. E. Blelloch. Scans as primitive operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *Computing Surveys*, 21(3):261–322, September 1989.

[C+89] D. R. Cheriton et al. Multi-level shared caching techniques for scalability in VMP-MC. In *Proceedings of the International Symposium on Computer Architecture*, pages 16–24, 1989.

[C+90] D. Chaiken et al. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–59, June 1990.

[C+91] David E. Culler et al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, Dec. 1978.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *Computing Surveys*, 21(3):323–358, September 1989.

[CKA91] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 1991.

[CKM88] R. Cytron, S. Karlovsky, and K. P. McAuliffe. Automatic management of programmable caches. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 229–238, 1988.

[Coh90] Evan Reid Cohn. Implementing the multiprefix operation efficiently. *Journal of Parallel and Distributed Computing*, 10:29–34, 1990.

[CS78] K. M. Chandy and C. H. Sauer. Approximate methods for analyzing queueing network models of computing systems. *Computing Surveys*, 10(3):281–317, Sept. 1978.

[CV88] H. Cheong and A. V. Veidenbaum. Stale data detection and coherence enforcement using flow analysis. In *Proceedings of the International Conference on Parallel Processing*, pages 138–145, 1988.

[CV90] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[D+86] M. Dubois et al. Memory access buffering in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, June 1986.

[D+88] F. Darema et al. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[Dal90a] Yves Dallery. Approximate analysis of general open queueing networks with restricted capacity. *Performance Evaluation*, 11:209–222, 1990.

[Dal90b] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[Dal90c] William J. Dally. Virtual-channel flow control. In *Proceedings of the International Symposium on Computer Architecture*, pages 60–68, 1990.

[DB82] M. Dubois and F. A. Briggs. Effects of cache coherency in multiprocessors. *IEEE Transactions on Computers*, C-31(11):1083–1099, Nov. 1982.

[DGK86] Susan Dickey, Allan Gottlieb, and Richard Kenner. Using VLSI to reduce serialization and memory traffic in shared memory parallel computers. In Charles E. Leiserson, editor, *Advanced Research in VLSI*, pages 299–316, 1986.

[DJ81a] D. M. Dias and J. R. Jump. Packet switching interconnection networks for modular systems. *IEEE Computer*, pages 43–53, Dec. 1981.

[DJ81b] Daniel M. Dias and J. Robert Jump. Analysis and simulation of buffered delta networks. *IEEE Transactions on Computers*, 30(4):273–282, April 1981.

[DKSS85] S. Dickey, R. Kenner, M. Snir, and J. Solworth. A VLSI combining network for the NYU ultracomputer. In *Proceedings of the International Conference on Computer Design*, pages 110–113, 1985.

[DR$^+$87] F. Darema-Rogers et al. Memory access patterns of parallel scientific programs. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 46–58, 1987.

[DS87] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[Dub85] M. Dubois. A cache-based multiprocessor with high efficiency. *IEEE Transactions on Computers*, C-34(10):968–972, Oct. 1985.

[Egg91] Susan J. Eggers. Simplicity versus accuracy in a model of cache coherence overhead. *IEEE Transactions on Computers*, 40(8):893–906, August 1991.

[EK88] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the International Symposium on Computer Architecture*, pages 373–383, June 1988.

[EK89a] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, 1989.

[EK89b] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–15, 1989.

[Fen81] T. Feng. A survey of interconnection networks. *IEEE Computer*, pages 12–27, Dec. 1981.

[G+83a] D. Gajski et al. Cedar–a large scale multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 514–529, August 1983.

[G+83b] A. Gottlieb et al. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, Apr. 1983.

[G+83c] A. Gottlieb et al. The NYU Ultracomputer–designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, Feb. 1983.

[G+89] J. R. Goodman et al. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–73, 1989.

[G+90a] Kourosh Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 15–26, 1990.

[G+90b] A. Gupta et al. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing*, pages 312–321, August 1990.

[Gec74] J. Gecsei. Determining hit ratios for multilevel hierarchies. *IBM Journal of Research and Development*, 18(4):316–327, July 1974.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–259, 1991.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[GL73] L. R. Goke and G. J. Lipovski. Banyan networks for partitioning multiprocessor systems. In *Proceedings of the International Symposium on Computer Architecture*, pages 21–28, 1973.

[GM88] Levent Gun and Armand M. Makowski. Matrix-geometric solution for finite capacity queues with phase-type distributions. *Performance '87*, pages 269–282, 1988.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, Scalable Coherent Interface Committee, March 1989.

[GS89] P. E. Green, Jr. and H. S. Stone. The implementation of a barrier for multiprocessors by means of an optical bus. Technical Disclosure YO888-0018, January 9, 1989, IBM Research, 1989.

[GW88] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, pages 422–433, May 1988.

[H+91] P. J. Hatcher et al. Data-parallel programming on mimd computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[Hil85] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hos89] T. Hoshino. *PAX Computer: High-Speed Parallel Processing and Scientific Computing*. Addison-Wesley, 1989.

[HRS88a] P. Heidelberger, B. D. Rathi, and H. S. Stone. A device for performing efficient task-distribution with a bus connection. Technical Report Technical Disclosure YO889-0053, January 20, 1989, IBM Research, 1988.

[HRS88b] P. Heidelberger, B. D. Rathi, and H. S. Stone. A low-cost device for contention-free barrier synchronization. Technical Report Technical Disclosure YO888-0218, March 16, 1988, IBM Research, 1988.

[HS89] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3):359–411, September 1989.

[HW88] P. Hudak and P. Wadler. Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR656, Department of Computer Science, Yale University, 1988.

[IEE90] IEEE. SCI (scalable coherent interface). Technical Report Standard P1596 (Draft), IEEE, 1990.

[Jen83] Y.-C. Jenq. Performance analysis of a packet switch based on single-buffered banyan network. *IEEE Journal on Selected Areas in Communications*, SAC-1(6):1014–1021, December 1983.

[Joh90] S. Lennart Johnsson. Communication in network architectures. In Robert Suaya and Graham Birtwistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, 1990.

[K+86] A. Karlin et al. Competitive snoopy caching. In *Proc. 27th Ann. Symp. Foundations of Computer Science*, pages 244–254, 1986.

[KLB76] W. Kraemer and M. Langenbach-Belz. Approximate formulae for the delay in the queueing system GI/G/1. In *Congressbook, Eight International Teletraffic Congress*, page 235, 1976.

[KRS86] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 218–228, 1986.

[KS83] C. P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, Dec. 1983.

[KSW88] C. P. Kruskal, M. Snir, and A. Weiss. The distribution of waiting times in clocked multistage interconnection networks. *IEEE Transactions on Computers*, 37(11):1337–1352, November 1988.

[Kue79] Paul J. Kuehn. Approximate analysis of general queueing networks by decomposition. *IEEE Transactions on Communications*, 27(1):113–126, January 1979.

[KX89] Demetres D. Kouvatsos and Nikos P. Xenios. Maximum entropy analysis of general queueing networks with blocking. *Queueing Networks with Blocking: Proceedings of First International Workshop*, pages 281–309, 1989.

[L+87] R. L. Lee et al. Multiprocessor cache design considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 253–263, June 1987.

[L+90] Dan Lenoski et al. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, pages 148–159, 1990.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[Law75] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, Dec. 1975.

[LDC89] Jesus Labarta, Jordi Domingo, and Olga Casals. Performance evaluation of packet switched omega networks with finite buffers. *Queueing Networks with Blocking: Proceedings of First International Workshop*, pages 249–255, 1989.

[Lee87] G. Lee. Another combining scheme to reduce hot spot contention in large scale shared memory parallel computers. In *Proceedings of the International Supercomputing Conference*, pages 68–79, 1987.

[LH91] D. H. Linder and J. C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[LPI88] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[LT88] T. Lovett and S. S. Thakkar. The symmetry multiprocessor system. In *Proceedings of the International Conference on Parallel Processing*, pages 303–310, 1988.

[LV88a] Scott T. Leutenegger and Mary K. Vernon. A mean-value performance analysis of a new multiprocessor architecture. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 167–176, 1988.

[LV88b] G. J. Lipovski and P. Vaughan. A fetch-and-op implementation for parallel computers. In *Proceedings of the International Symposium on Computer Architecture*, pages 384–392, 1988.

[LY90] D. J. Lilja and P. Yew. A compiler-assisted directory-based cache coherence scheme. Technical Report CSRD 990, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1990.

[MA89] Douglas E. Marquardt and Hasan S. AlKhatib. C2MP: A cache-coherent, distributed memory multiprocessor-system. In *Proceedings of Supercomputing '89*, pages 466–475, November 1989.

[Mar68] K. T. Marshall. Some inequalities in queueing. *Operations Research*, 16(3):651–665, May-June 1968.

[May90] Ernst W. Mayr. Theoretical aspects of parallel computation. In Robert Suaya and Graham Birtwistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, 1990.

[MB90] S. L. Min and J. L. Baer. A performance comparison of directory-based and timestamp-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing*, pages 305–311, August 1990.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[Neu81] M. F. Neuts. *Matrix-Geometric Solutions in Stochastic Models*. Johns Hopkins University Press, Baltimore, MD, 1981.

[NP85] A. Norton and G. F. Pfister. A methodology for predicting multiprocessor performance. In *Proceedings of the International Symposium on Computer Architecture*, June 1985.

[OA89] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, 1989.

[ON90] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proceedings of the International Symposium on Computer Architecture*, pages 138–147, May 1990.

[P+85] G. F. Pfister et al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 1985.

[Pat81] J. H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE Transactions on Computers*, C-30(10):771–780, October 1981.

[Pat82] J. H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computers*, C-31(4):296–304, April 1982.

[PCYL87] N. F. Tzeng P. C. Yew and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.

[Pea77] M. C. Pease. The indirect binary n-cube microprocessor array. *IEEE Transactions on Computers*, C-26(5):458–473, May 1977.

[Per89] H. G. Perros. A bibliography of papers on queueing networks with finite capacity queues. *Performance Evaluation*, 10:255–260, 1989.

[Per90] Harry G. Perros. Approximation algorithms for open queueing networks with blocking. *Stochastic Analysis of Computer and Communication Systems*, pages 451–498, 1990.

[PH88] N. M. Patel and P. G. Harrison. On hot-spot contention in interconnection networks. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 114–123, 1988.

[PN85] G. F. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, Oct. 1985.

[PP84] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the International Symposium on Computer Architecture*, pages 348–355, Jan. 1984.

[PS89] H. G. Perros and P. M. Snyder. A computationally efficient approximation algorithm for feed-forward open queueing networks with blocking. *Performance Evaluation*, 9:217–224, 1989.

[QD84] M. J. Quinn and N. Deo. Parallel graph algorithms. *Computing Surveys*, 16(3):319–348, Sept. 1984.

[R+90] R. Rettberg et al. The monarch parallel processor hardware design. *IEEE Computer*, 23(4):18–31, April 1990.

[Ran87] A. G. Ranade. How to emulate shared memory. In *Fundamentals of Computer Science*, pages 185–194, 1987.

[Ran89] Abhiram Gorakhanath Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, 1989.

[RBJ88] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The Fluent Abstract Machine. In *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 71–94, March 1988. Also available as Yale Univ. Comp. Sc. TR-573.

[RF87] Daniel A. Reed and Richard M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing*. MIT Press, 1987.

[SA88] R. L. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the International Symposium on Computer Architecture*, pages 186–195, 1988.

[SB77] H. Sullivan and T. R. Brashkow. A large scale homogeneous machine. In *Proceedings of the International Symposium on Computer Architecture*, pages 105–124, 1977.

[SB91] J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, April 1991.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, Oct. 1980.

[SD87] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 234–243, June 1987.

[SH91a] Richard Simoni and Mark Horowitz. Dynamic pointer allocation for scalable cache coherence directories. Technical report, Computer Systems Laboratory, Stanford University, 1991.

[SH91b] Richard Simoni and Mark Horowitz. Modeling the performance of limited pointers directories for cache coherence. Technical report, Computer Systems Laboratory, Stanford University, 1991.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *Computing Surveys*, 21(3):412–510, September 1989.

[Sie85] H. J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing*. Lexington Books, 1985.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[Smi85] A. J. Smith. CPU cache consistency with software support and using one time identifiers. In *Proceedings of the Pacific Computer Communications Conference*, pages 153–161, 1985.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the International Symposium on Computer Architecture*, pages 414–423, 1986.

[SSG89] G. S. Sohi, J. E. Smith, and J. R. Goodman. Restricted fetch&ϕ operations for parallel processing. In *Proceedings of the International Supercomputing Conference*, pages 410–416, June 1989.

[ST72] D. R. Slutz and I. L. Traiger. Evaluation techniques for cache memory hierarchies. Technical Report RJ 1045 (#17547), IBM, May 1972.

[Ste89] P. Stenstrom. A cache consistency protocol for multiprocessors with multistage networks. In *Proceedings of the International Symposium on Computer Architecture*, pages 407–415, 1989.

[Sto87] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, 1987.

[SV81] Alberto L. Sangiovanni-Vincentelli. *Circuit Simulation*. Sijthoff and Noordhoff, 1981.

[SW89] Moshe Segal and Ward Whitt. A queueing network analyzer for manufacturing. In *Teletraffic Science for New Cost-Effective Systems, Networks and Services*, pages 1146–1152, 1989.

[SWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical report, Computer Systems Laboratory, Stanford University, 1991.

[T$^+$90] Shreekant Thakkar et al. New directions in scalable shared-memory multiprocessor architectures. *IEEE Computer*, 23(6):71–83, June 1990.

[Tak89] Yukio Takahashi. Aggregate approximation for acyclic queueing networks with communication blocking. *Queueing Networks with Blocking*, pages 33–46, 1989.

[TF88] Y. Tamir and G. L. Frazier. High-performance multi-queue buffers for VLSI communication switches. In *Proceedings of the International Symposium on Computer Architecture*, pages 343–355, 1988.

[Tho87] J. G. Thompson. *Efficient Analysis of Caching Systems*. PhD thesis, University of California, Berkeley, 1987.

[TMH80] Yutaka Takahashi, Hideo Miyahara, and Toshiharu Hasegawa. An approximation method for open restricted queueing networks. *Operations Research*, 28(3):594–602, May 1980.

[TR88] Lewis W. Tucker and George G. Robertson. Architecture and applications of the connection machine. *IEEE Computer*, 21(8):26–39, August 1988.

[TRH89] Thomas H. Theimer, Erwin P. Rathgeb, and Manfred N. Huber. Performance analysis of buffered banyan networks. *Performance of Distributed and Parallel Systems*, pages 57–72, 1989.

[TS71] I. L. Traiger and D. R. Slutz. One-pass techniques for the evaluation of memory hierarchies. Technical Report Tech. Rep. RJ 892 (#15563), IBM, July 1971.

[TS89] James G. Thompson and Alan Jay Smith. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems*, 7(1):78–117, February 1989.

[V+88] M. K. Vernon et al. An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols. In *Proceedings of the International Symposium on Computer Architecture*, pages 308–317, May 1988.

[VH86] M. K. Vernon and M. A. Holiday. Performance analysis of multiprocessor cache consistency protocols using generalized timed petri nets. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 9–17, 1986.

[VJS89] Mary K. Vernon, Rajeev Jog, and Gurindar S. Sohi. Performance analysis of hierarchical cache-consistent multiprocessors. *Performance of Distributed and Parallel Systems*, pages 111–126, 1989.

[WG89a] W. D. Weber and A. Gupta. Analysis of cache invalidation patterns in microprocessors. In *Proc. ASPLOS III*, pages 243–256, 1989.

[WG89b] W-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, 1989.

[Whi82] W. Whitt. Approximating a point process by a renewal process, I: Two basic methods. *Operations Research*, 30(1):125–147, January 1982.

[Whi83] W. Whitt. The queueing network analyzer. *Bell System Technical Journal*, 62(9):2779–2815, November 1983.

[Wil87] A. W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 244–252, June 1987.

[Wol89] Ronald W. Wolff. *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, 1989.

[YBL89] W. Yang, L. N. Bhuyan, and B.-C. Liu. Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor. *IEEE Transactions on Computers*, 38(8):1143–1153, August 1989.

[YF82] W. C. Yen and K. S. Fu. Coherence problem in a multicache system. In *Proceedings of the International Conference on Parallel Processing*, pages 332–339, 1982.

[YLL90] Hyunsoo Yoon, Kyungsook Y. Lee, and Ming T. Liu. Performance analysis of multibuffered packet-switching networks in multiprocessor systems. *IEEE Transactions on Computers*, 39(3):319–327, March 1990.

Appendix A

Geometric Queue Models

A.1 GEO+1^k/GEO+1/1/N Queue Model

The state space of the discrete time Markov chain for a GEO+1^k/GEO+1/1/N queue is the number of customers in queue, n.

Let:

- $p_i = 1 - q_i$ be the probability that a customer arrives on input stream i,
- μ be the probability that a customer departs when the queue is non-empty.
- $N_{max} = N + k$ be the maximum number in queue. This assumes that the basic queue limit N is supplemented by k extra buffers to hold customers from upstream queues that block.
- $n_s = N_{max} + 1$ denote the number of states in the discrete time Markov chain.
- p_{s_1,s_2} be the transition probability between states s_1 and s_2.
- $p^h_{s_1,s_2}$ be the transition rate between s_1 and s_2 for events in which an arrival occurs on stream h, and there is no departure.
- $d^h_{s_1,s_2}$ be the transition rate between s_1 and s_2 for events in which an arrival occurs on stream h, and there is a departure.
- π_s be the steady state transition rate out of state s.
- P denote the matrix of transition probabilities $p_{s_1 s_2}$.
- π denote the vector $[\pi_{s_1}, \ldots, \pi_{s_{n_s}}]$.
- $f_a(h, i)$ denote the fraction of arrivals on stream h that find i customers in queue.

- $f_d(i)$ denote the fraction of departures that, just before departure, find i customers in queue.

State transitions result from $m \in \{0, \ldots, k\}$ simultaneous arrivals, and a possible simultaneous departure. With k arrival streams, m simultaneous arrivals can occur in $C(k,m) = \binom{k}{m}$ ways. Denote each distinct set of m arrival streams (selected from k possible streams) as $S_j^m = \{i_1^j, \ldots, i_m^j\}$, where $j = 1, \ldots, C(k,m)$. The probability that m arrivals occur on the streams in S_j^m is:

$$Pr(\text{arrivals on} S_j^m) = \left[\prod_{h=1}^{k}(1-p_h)\right]\left[\prod_{h=1}^{m}\frac{p_{i_h^j}}{(1-p_{i_h^j})}\right] \qquad \text{(A.1)}$$

The transition probabilities are:

$$p_{0,m} = \sum_{j=1}^{C(k,m)} Pr(\text{arrivals on } S_j^m) \quad \text{for } m = 0, \ldots, k \qquad \text{(A.2)}$$

$$p_{n_1,n_1-1} = 1 - \sum_{j=0}^{N_{max}-n_1} p_{n_1,n_1+j} \quad \text{for } n_1 = 1, \ldots, N_{max} \qquad \text{(A.3)}$$

$$\begin{aligned} p_{n_1,\min(n_1+m,N_{max})} &= \mu \sum_{j=1}^{C(k,m+1)} Pr(\text{arrivals on } S_j^{m+1}) \\ &+ (1-\mu)\sum_{j=1}^{C(k,m)} Pr(\text{arrivals on } S_j^m) \end{aligned} \qquad \text{(A.4)}$$

$$\begin{aligned} \text{for } n_1 &= 1, \ldots, N_{max} \\ \text{and } m &= 0, \ldots, k-1 \end{aligned}$$

$$p_{n_1,\min(n_1+k,N_{max})} = (1-\mu)Pr(\text{arrivals on } S_1^k) \qquad \text{(A.5)}$$

$$\text{for } n_1 = 1, \ldots, N_{max}$$

Equation (A.2) gives transition probabilities for an empty queue. In this case no departures can take place. Equation (A.3) is the probability that the number in queue decreases: a departure occurs with zero arrivals. Equations (A.4) and (A.5) give transition probabilities for for state changes in a non-empty queue in which the number in queue increases or remains the same.

The calculation of $f_a(h,i)$ requires $d^h_{s_1,s_2}$ and $p^h_{s_1,s_2}$: the fraction of transitions between two states in which an arrival occurs on stream h, with and without a departure. These are calculated with straightforward variations of Equations (A.2) to (A.5):

$$p^h_{0,m} = \sum_{\{j|h\in S^m_j\}} Pr(\text{arrivals on } S^m_j) \quad \text{for } m = 0,\ldots,k \tag{A.6}$$

$$p^h_{n_1,n_1-1} = 0 \quad \text{for } n_1 = 1,\ldots,N_{max} \tag{A.7}$$

$$d^h_{0,m} = 0 \quad \text{for } m = 0,\ldots,k \tag{A.8}$$

$$d^h_{n_1,n_1-1} = 0 \quad \text{for } n_1 = 1,\ldots,N_{max} \tag{A.9}$$

$$p^h_{n_1,\min(n_1+m,N_{max})} = (1-\mu) \sum_{\{j|h\in S^m_j\}} Pr(\text{arrivals on } S^m_j) \tag{A.10}$$

$$\begin{aligned} \text{for } n_1 &= 1,\ldots,N_{max} \\ \text{and } m &= 0,\ldots,k \end{aligned}$$

$$d^h_{n_1,\min(n_1+m,N_{max})} = \mu \sum_{\{j|h\in S^{m+1}_j\}} Pr(\text{arrivals on } S^{m+1}_j) \tag{A.11}$$

$$\begin{aligned} \text{for } n_1 &= 1,\ldots,N_{max} \\ \text{and } m &= 0,\ldots,k-1 \end{aligned}$$

These differ from the equations for p_{s_1,s_2} by summing over only those arrival sets containing h.

The steady state transition rates are the solution to the matrix equation [Wol89]:

$$\pi = \pi P \tag{A.12}$$

with the added constraint that

$$\sum_{j=0}^{N_{max}} \pi_j = 1 \tag{A.13}$$

These can be solved numerically with a sparse matrix package. Since the Markov chain is irreducible and has a finite number of states, Equations (A.12) and (A.13) have a unique solution [Wol89].

The probability that a customer arriving from stream h finds i customers in queue is determined using:

$$f_a(h,i) = \sum_{m=1}^{k} \frac{1}{m} \left[\sum_{j=\max(i-m,0)}^{i} \pi_j p^h_{j,\min(j+m,N_{max})} + \sum_{j=\max(i-m+1,0)}^{i+1} \pi_j d^h_{j,\min(j+m-1,N_{max})} \right] \tag{A.14}$$

$$\begin{aligned} \text{for } h &= 1,\ldots,k \\ \text{and } i &= 0,\ldots,N_{max} \end{aligned}$$

Equation (A.14) sums over all transitions such that one arrival (of m simultaneous arrivals, including an arrival on stream h) finds i in queue. The sums therefore involve all transitions between states n_1 and n_2 such that $n_1 \leq i \leq n_2$. Since m simultaneous arrivals are queued in random order, $\frac{1}{m}$ of the arrivals on a particular stream are queued at any particular position.

The probability that a departing customer leaves when there are i customers in queue is:

$$\begin{aligned} f_d(i) &= \frac{\mu \pi_i}{\sum_{j=1}^{N_{max}} \mu \pi_j} \\ &= \frac{\pi_i}{\sum_{j=1}^{N_{max}} \pi_j} \qquad \text{for } i = 1,\ldots,N_{max} \end{aligned} \tag{A.15}$$

The average number in queue is:

$$L = \sum_{i=0}^{N_{max}} \pi_i \min(i,N) \tag{A.16}$$

Applying Little's Law, the average delay in queue w (including service time) is:

$$\begin{aligned} w &= \frac{L}{\lambda} \\ &= \frac{L}{\sum_{j=1}^{k} p_j} \end{aligned} \tag{A.17}$$

For states in which the number in queue exceeds the queue limit N (ie. customers at one or more upstream queues are blocked), the arrival rates must be reduced to account for the fact that once an upstream queue blocks, it no longer sends customers until it becomes unblocked [1]. Modeling this in detail is expensive because it requires maintaining the blocking state of each upstream queue; this would increase the size of the state space by a factor of 2^k. A simpler, approximate method is

[1] Simulations of the superposition process showed that accuracy improves considerably if this effect is modeled.

to reduce the arrival rate of each stream h for states greater than N:

$$\lambda_h^i = \begin{cases} \lambda_h & i = 0, \ldots, N \\ \lambda_h \left(1 - \frac{\lambda_h}{\sum_{j=1}^{k} \lambda_j}\right)^{N-i} & i = N+1, \ldots, N+k-1 \\ 0 & i = N+k \end{cases} \tag{A.18}$$

λ_h^i is the adjusted arrival rate on stream h when i customers are in queue. This is based on the following approximations:

- If an arrival blocks, the probability that the arrival is from stream h is the fraction of all arrivals due to stream h.
- Each blocking event is independent of all others.

The probability that i blocked customers are from streams other than h is thus the probability that an arrival is from another stream $(1 - \frac{\lambda_h}{\sum_{j=1}^{k} \lambda_j})$ raised to the power i.

A.2 GEO2+1^k/GEO+1/1/N Queue Model

In this model interarrival times are distributed as a mixture of two geometric distributions, plus 1. Assume the parameters of the two distributions are p^0 and p^1, and that the mixing parameter is s. With probability s, an interarrival time is selected using a geometric distribution with parameter p^0. With probability $1 - s$, the interarrival time is selected using a geometric distribution with parameter p^1. The state space of the discrete time Markov chain for the GEO2+1^k/GEO+1/1/N queue therefore requires k extra variables to record the state of the k mixtures: $a_1, \ldots, a_k$, where $a_i \in \{0, 1\}$. $a_i = j$ indicates that the next interarrival on stream i will be distributed as 1 plus a geometric random variable with parameter p_i^j. These variables are denoted by the vector $A = [a_1, \ldots, a_k]$. The state space for the queue model is then (n, A), where n is the number of customers in queue. It is assumed that all GEO2+1 distributions have a common mixing parameter s.

This discussion uses the notation introduced in the previous section, plus the following:

- The number of states is now $n_s = 2^k(N_{max} + 1)$.
- Diff(A_1, A_2) denotes the set of indices of A_1 and A_2 for which $a_i^1 \neq a_i^2$. For example, Diff($[1,2,1],[1,1,2]$) $= \{2,3\}$. |Diff(A_1, A_2)| denotes the number of indices that differ.
- $s_0 = s$ and $s_1 = 1 - s$.

- $\overline{a}_i = 1 - a_i$. Hence $\overline{a}_i = 1$ if $a_i = 0$ and $\overline{a}_i = 0$ if $a_i = 1$.
- α_i denotes the fraction of time i customers are in queue.
- $\mathcal{A}^k$ denotes the set of all valid mixture vectors $[a_1, \ldots, a_k]$ of length k.

In this queue model, transition probabilities depend not only on the occurrence of arrivals or a departure, but also on the set of streams that have a change in mixture state. As before, the m arrivals can occur on $C(k,m)$ different sets of inputs, each denoted S_j^m for $j = 1, \ldots, C(k,m)$. The probability that m arrivals occur on streams S_j^m is similar to Equation (A.1), except that the per-stream arrival probability is indexed by mixture state:

$$Pr(\text{arrivals on } S_j^m) = \left[\prod_{h=1}^{k}(1 - p_h^{a_h})\right]\left[\prod_{h=1}^{m}\frac{p_{i_h}^{a_{i_h}}}{(1 - p_{i_h}^{a_{i_h}})}\right] \tag{A.19}$$

Given mixture states A_1 and A_2 and arrivals on S_j^m, the probability that only the mixture variables in Diff(A_1, A_2) change state is:

$$Pr(\text{changes on Diff}(A_1, A_2) \text{ in } S_j^m) = \left[\prod_{\{i|i\in S_j^m, i\notin \text{Diff}(A_1,A_2)\}} s_{a_i}\right]\left[\prod_{\{i|i\in \text{Diff}(A_1,A_2)\}} s_{\overline{a}_i}\right] \tag{A.20}$$

The transition probabilities are thus:

$$p_{(0,A_1),(m,A_2)} = \sum_{j=1}^{C(k,m)} Pr(\text{arrivals on } S_j^m)Pr(\text{changes on Diff}(A_1, A_2) \text{ in } S_j^m) \tag{A.21}$$

for $m = 0, \ldots, k$

and all A_1, A_2 such that Diff$(A_1, A_2) \leq m$

$$p_{(n_1,A_1),(n_1-1,A_1)} = 1 - \sum_{j=0}^{N_{max}-n_1} \sum_{\{A_2|A_2\in\mathcal{A}^k\}} p_{(n_1,A_1),(n_1+j,A_2)} \tag{A.22}$$

for $n_1 \;=\; 1, \ldots, N_{max}$

$$\begin{aligned} p_{(n_1,A_1),(\min(n_1+m,N_{max}),A_2)} &= \\ \mu \sum_{j=1}^{C(k,m+1)} & Pr(\text{arrivals on } S_j^{m+1})Pr(\text{changes on Diff}(A_1, A_2) \text{ in } S_j^{m+1}) \\ + (1-\mu) \sum_{j=1}^{C(k,m)} & Pr(\text{arrivals on } S_j^m)Pr(\text{changes on Diff}(A_1, A_2) \text{ in } S_j^m) \end{aligned} \tag{A.23}$$

$$\text{for } n_1 = 1, \ldots, N_{max}$$
$$\text{and } m = 0, \ldots, k-1$$
$$\text{and all } A_1, A_2 \text{ such that } \text{Diff}(A_1, A_2) \leq m$$

$$p_{(s_1,A_1),(\min(n_1+k,N_{max}),A_2)} = (1-\mu)Pr(\text{arrivals on } S_1^k)Pr(\text{changes on Diff}(A_1,A_2) \text{ in } S_1^k) \tag{A.24}$$

$$\text{for } s_1 = 1, \ldots, N_{max}$$
$$\text{and all } A_1, A_2 \text{ such that } \text{Diff}(A_1, A_2) \leq k$$

Equation (A.21) gives transition probabilities for an empty queue. In this case no departures can take place. Equation (A.22) is the probability that the number of customers in a non-empty queue decreases: a departure occurs with zero arrivals. Equation (A.23) and (A.24) apply to transitions in which the number of customers in a non-empty queue increases or remains the same.

As for the GEO^k model, the expressions for $d^h_{s_1,s_2}$ and $p^h_{s_1,s_2}$ are straightforward (but messy) variations of Equations (A.21) to (A.24):

$$p^h_{(0,A_1),(m,A_2)} = \sum_{\{j|h\in S_j^m\}} Pr(\text{arrivals on } S_j^m)Pr(\text{changes on Diff}(A_1,A_2) \text{ in } S_j^m) \tag{A.25}$$

$$\text{for } m = 0, \ldots, k$$
$$\text{and all } A_1, A_2 \text{ such that } \text{Diff}(A_1, A_2) \leq m$$

$$d^h_{(0,A_1),(m,A_2)} = 0 \tag{A.26}$$

$$\text{for } m = 0, \ldots, k$$
$$\text{and all } A_1, A_2 \text{ such that } \text{Diff}(A_1, A_2) \leq m$$

$$p^h_{(n_1,A_1),(n_1-1,A_1)} = 0 \qquad \text{for } n_1 = 1, \ldots, N_{max} \tag{A.27}$$

$$d^h_{(n_1,A_1),(n_1-1,A_1)} = 0 \qquad \text{for } n_1 = 1, \ldots, N_{max} \tag{A.28}$$

$$p^h_{(n_1,A_1),(\min(n_1+m,N_{max}),A_2)} = (1-\mu) \sum_{\{j|h\in S_j^m\}} Pr(\text{arrivals on } S_j^m)Pr(\text{changes on Diff}(A_1,A_2) \text{ in } S_j^m) \tag{A.29}$$

$$\text{for } n_1 = 1, \ldots, N_{max}$$
$$\text{and } m = 0, \ldots, k$$
$$\text{and all } A_1, A_2 \text{ such that Diff}(A_1, A_2) \leq m$$

$$d^h_{(n_1,A_1),(\min(n_1+m,N_{max}),A_2)} =$$
$$\mu \sum_{\{j|h \in S_j^{m+1}\}} Pr(\text{arrivals on } S_j^{m+1}) Pr(\text{changes on Diff}(A_1, A_2) \text{ in } S_j^{m+1}) \quad \text{(A.30)}$$

$$\text{for } n_1 = 1, \ldots, N_{max}$$
$$\text{and } m = 0, \ldots, k-1$$
$$\text{and all } A_1, A_2 \text{ such that Diff}(A_1, A_2) \leq m$$

The steady state transition rates are the solution to Equations (A.12) and (A.13) for the new state space.

The probability that a customer from stream h finds i customers in queue is analogous to Equation (A.14):

$$f_a(h,i) =$$
$$\sum_{m=1}^{k} \frac{1}{m} \left[\sum_{j=\max(i-m,0)}^{i} \sum_{\{A_1,A_2|(|Diff(A_1,A_2)|)\leq m\}} \pi_{(j,A_1)} p^h_{(j,A_1),(\min(j+m,N_{max}),A_2)} \right.$$
$$\left. + \sum_{j=\max(i-m+1,0)}^{i+1} \sum_{\{A_1,A_2|(|Diff(A_1,A_2)|)\leq m\}} \pi_{(j,A_1)} d^h_{(j,A_1),(\min(j+m-1,N_{max}),A_2)} \right] \quad \text{(A.31)}$$

$$\begin{aligned} \text{for } h &= 1, \ldots, k \\ \text{and } i &= 0, \ldots, N_{max} \end{aligned}$$

The probability that a departing customer leaves when there are i customers in queue is analogous to Equation (A.15):

$$\begin{aligned} f_d(i) &= \frac{\mu \alpha_i}{\sum_{j=1}^{N_{max}} \mu \alpha_j} \\ &= \frac{\alpha_i}{\sum_{j=1}^{N_{max}} \alpha_j} \text{ for } i = 1, \ldots, N_{max} \end{aligned} \quad \text{(A.32)}$$

Here α_i is the fraction of time that there are i customers in queue, for any mixture state. It is found using:

$$\alpha_i = \sum_{\{A|A \in \mathcal{A}^k\}} \pi_{(i,A)} \quad \text{(A.33)}$$

The average number in queue L is:

$$L = \sum_{i=0}^{N_{max}} \alpha_i \min(i, N) \tag{A.34}$$

Applying Little's Law, the average delay in queue w (including service time) is:

$$\begin{aligned} w &= \frac{L}{\lambda} \\ &= \frac{L}{\sum_{j=1}^{k}(sp_j^0 + (1-s)p_j^1)} \end{aligned} \tag{A.35}$$

As before, λ_h^i is reduced using Equation (A.18) when one or more customers are blocked ($i > N$). For this model λ_h^i is a function of two geometric probabilities:

$$\lambda_h^i = sp_h^{i0} + (1-s)p_h^{i1} \tag{A.36}$$

We derate p_h^{i0} and p_h^{i1} equally.

A.3 Distribution of the Minimum of Two GEO2+1 Random Variables

In this section the first two moments are derived for the minimum of two GEO2+1 random variables. First consider the minimum of two GEO2 random variables B_1 and B_2:

$$B = \min\{B_1, B_2\} \tag{A.37}$$

Let $\{p_{11}, p_{12}\}$ and $\{p_{21}, p_{22}\}$ be the parameters for the geometric distributions in the mixtures for B_1 and B_2, respectively. Let $q_{ij} = 1 - p_{ij}$. It is assumed that B_1 and B_2 have a common mixing parameter s.

The probability that $B_i = b, i \in \{1,2\}$, is:

$$Pr(B_i = b) = sq_{i1}^b p_{i1} + (1-s)q_{i2}^b p_{i2} \tag{A.38}$$

The probability that $B_i > b$ is:

$$\begin{aligned} Pr(B_i > b) &= \sum_{i=b+1}^{\infty} Pr(B_i = b) && \text{(A.39)} \\ &= \sum_{i=b+1}^{\infty} (sq_{i1}^i p_{i1} + (1-s)q_{i2}^i p_{i2}) && \text{(A.40)} \\ &= sq_{i1}^{b+1} \sum_{i=0}^{\infty} q_{i1}^i p_{i1} + (1-s)q_{i2}^{b+1} \sum_{i=0}^{\infty} q_{i2}^i p_{i2} && \text{(A.41)} \\ &= sq_{i1}^{b+1} \frac{p_{i1}}{1-q_{i1}} + (1-s)q_{i2}^{b+1} \frac{p_{i2}}{1-q_{i2}} && \text{(A.42)} \\ &= sq_{i1}^{b+1} + (1-s)q_{i2}^{b+1} && \text{(A.43)} \end{aligned}$$

The probability that $B = b$ is then:

$$\begin{aligned}
&Pr(B = b) = \\
&\quad Pr(B_1 = b)Pr(B_2 > b) + Pr(B_1 > b)Pr(B_2 = b) \\
&\quad +Pr(B_1 = b)Pr(B_2 = b) \qquad \text{(A.44)}
\end{aligned}$$

The first two moments of B are found using z-transforms [Wol89]. Let $G(z)$ be the z-transform of B:

$$G(z) = \sum_{i=0}^{\infty} Pr(B = i)z^i \qquad \text{(A.45)}$$

After some tedious algebra:

$$\begin{aligned}
G(z) &= \frac{s^2(1 - q_{11}q_{21})}{1 - q_{11}q_{21}z} + \frac{s(1-s)(1 - q_{11}q_{22})}{1 - q_{11}q_{22}z} \\
&\quad + \frac{s(1-s)(1 - q_{12}q_{21})}{1 - q_{12}q_{21}z} + \frac{(1-s)^2(1 - q_{12}q_{22})}{1 - q_{12}q_{22}z} \qquad \text{(A.46)}
\end{aligned}$$

The two moments are thus:

$$\begin{aligned}
E(B) &= \lim_{z \to 1} G'(z) \qquad \text{(A.47)} \\
&= \frac{s^2 q_{11}q_{21}}{1 - q_{11}q_{21}} + \frac{s(1-s)q_{11}q_{22}}{1 - q_{11}q_{22}} \\
&\quad + \frac{s(1-s)q_{12}q_{21}}{1 - q_{12}q_{21}} + \frac{(1-s)^2 q_{12}q_{22}}{1 - q_{12}q_{22}} \qquad \text{(A.48)}
\end{aligned}$$

$$\begin{aligned}
E(B^2) &= (\lim_{z \to 1} G''(z)) + E(B) \qquad \text{(A.49)} \\
&= \frac{s^2 q_{11}q_{21}(1 + q_{11}q_{21})}{(1 - q_{11}q_{21})^2} + \frac{s(1-s)q_{11}q_{22}(1 + q_{11}q_{22})}{(1 - q_{11}q_{22})^2} \\
&\quad + \frac{s(1-s)q_{12}q_{21}(1 + q_{12}q_{21})}{(1 - q_{12}q_{21})^2} + \frac{(1-s)^2 q_{12}q_{22}(1 + q_{12}q_{22})}{(1 - q_{12}q_{22})^2} \qquad \text{(A.50)}
\end{aligned}$$

The moments for the minimum of two GEO2+1 random variables are the moments of $A = B + 1$:

$$E(A) = E(B) + 1 \qquad \text{(A.51)}$$

$$\begin{aligned}
E(A^2) &= Var(A) + E^2(A) \qquad \text{(A.52)} \\
&= Var(B) + E^2(A) \qquad \text{(A.53)} \\
&= E(B^2) - E^2(B) + E^2(A) \qquad \text{(A.54)}
\end{aligned}$$

Appendix B

Queueing Models for Specific Interconnection Networks

B.1 Notes for All Models

1. Buffer queues without external arrivals have a queue limit of 4 messages. Buffer queues with external arrivals have no queue limit. All buffer queues use a virtual channel queueing discipline (with 4 channels). The service distribution is deterministic with a delay of one discrete time unit.
2. All channel queues have a queue limit of 1 message, and use a FIFO queueing discipline. The service distribution is deterministic with a delay of one discrete time unit.
3. Virtual cut-through flow control is assumed.
4. All arrival streams are geometric with common parameter p. *Messages* therefore arrive at a rate of p messages per cycle. *Flits* arrive at a rate of bp, where b is the number of flits per message.
5. As discussed in Section 4.4.1, the service time at channel queues is deducted from overall network delay.
6. As discussed in Section 4.4.1, only blocking due to channel queues is modeled. This allows a queueing network to have cycles as long as each cycle includes at least one buffer queue.

B.2 Unidirectional Three Dimensional Torus

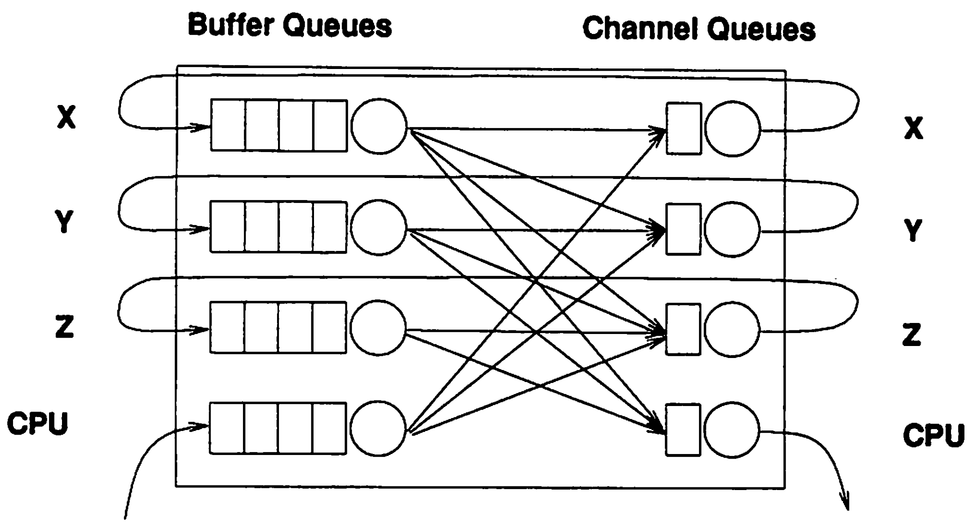


Figure B.1: Queueing Model of a Unidirectional 3-D Torus Network

Table B.1: Routing Parameters for 3-D Torus

q_{ij}	Number of Processors				
	8	64	216	512	1000
$q_{X,X}$	0.0	0.5	0.6667	0.75	0.8
$q_{X,Y}$	0.5	0.375	0.2778	0.2188	0.18
$q_{X,Z}$	0.25	0.09375	0.0463	0.0273	0.018
$q_{X,CPU}$	0.25	0.03125	0.0093	0.0039	0.002
$q_{Y,Y}$	0.0	0.5	0.6667	0.75	0.8
$q_{Y,Z}$	0.5	0.275	0.2778	0.2188	0.18
$q_{Y,CPU}$	0.5	0.125	0.0556	0.0313	0.02
$q_{Z,Z}$	0.0	0.5	0.6667	0.75	0.8
$q_{Z,CPU}$	1.0	0.5	0.3333	0.25	0.2
$q_{CPU,X}$	0.5714	0.7619	0.8372	0.8767	0.9009
$q_{CPU,Y}$	0.2857	0.1905	0.1395	0.1096	0.0901
$q_{CPU,Z}$	0.1429	0.0476	0.0233	0.0137	0.0090

B.3 Bidirectional Three Dimensional Torus

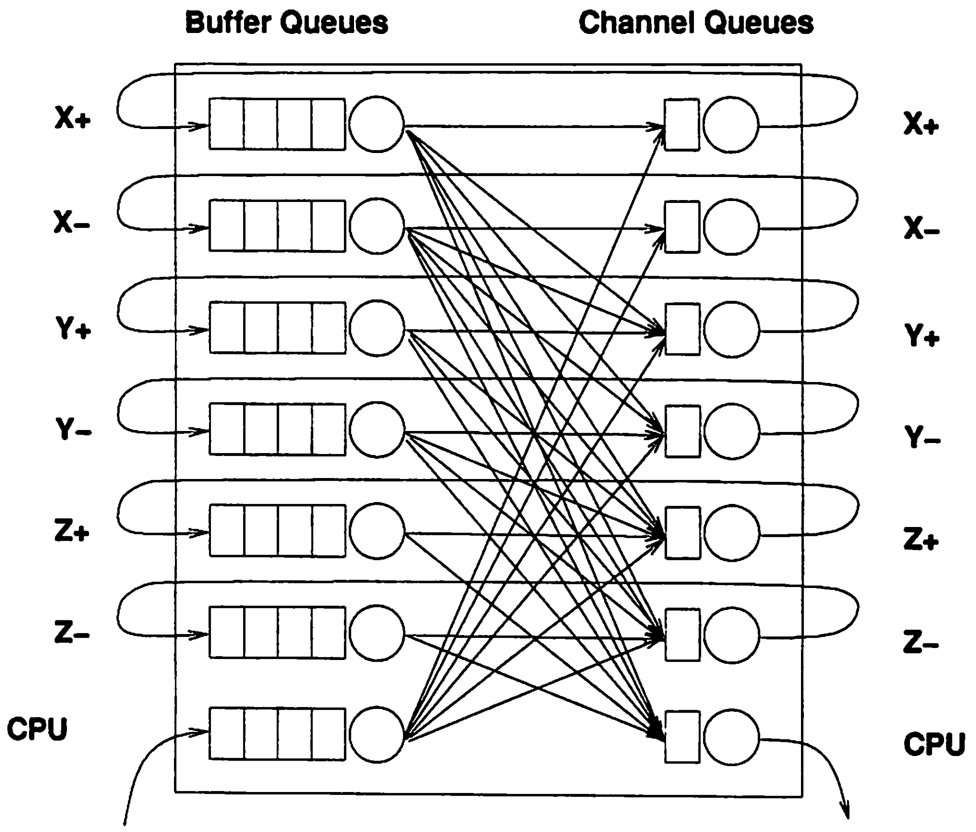


Figure B.2: Queueing Model of a Bidirectional 3-D Torus Network

B.4 Three Dimensional Mesh

A full queueing network of k^3 switches (of the form of Figure B.2) was used; it is too cumbersome to give the routing parameters for such large networks. The routing parameters were found as described in Section 4.4.1: by aggregating the $N(N-1)$ customer classes corresponding to all possible source/destination pairs.

B.5 Hypercube

B.6 Radix-2 Delta

All routing probabilities are $\frac{1}{2}$.

Table B.2: Routing Parameters for Bidirectional 3-D Torus

q_{ij}	Number of Processors				
	8	64	216	512	1000
$q_{X+,X+}$	0.0	0.3333	0.5	0.6	0.6667
$q_{X+,Y+}$	0.5	0.3333	0.25	0.2	0.1667
$q_{X+,Y-}$	0.0	0.1667	0.1667	0.15	0.1333
$q_{X+,Z+}$	0.25	0.0833	0.0417	0.025	0.0167
$q_{X+,Z-}$	0.0	0.0417	0.0278	0.0188	0.0133
$q_{X+,CPU}$	0.25	0.0417	0.0139	0.0063	0.0033
$q_{X-,X-}$	0.0	0.0	0.3333	0.5	0.6
$q_{X-,Y+}$	0.0	0.5	0.3333	0.25	0.2
$q_{X-,Y-}$	0.0	0.25	0.2222	0.1875	0.16
$q_{X-,Z+}$	0.0	0.125	0.0556	0.0313	0.02
$q_{X-,Z-}$	0.0	0.0625	0.0370	0.0234	0.016
$q_{X-,CPU}$	0.0	0.0625	0.0185	0.0078	0.004
$q_{Y+,Y+}$	0.0	0.3333	0.5	0.6	0.6667
$q_{Y+,Z+}$	0.5	0.3333	0.25	0.2	0.1667
$q_{Y+,Z-}$	0.0	0.1667	0.1667	0.15	0.1333
$q_{Y+,CPU}$	0.5	0.1667	0.0833	0.05	0.0333
$q_{Y-,Y-}$	0.0	0.0	0.3333	0.5	0.6
$q_{Y-,Z+}$	0.0	0.5	0.3333	0.25	0.2
$q_{Y-,Z-}$	0.0	0.25	0.2222	0.1875	0.16
$q_{Y-,CPU}$	0.0	0.25	0.1111	0.0625	0.04
$q_{Z+,Z+}$	0.0	0.3333	0.5	0.6	0.6667
$q_{Z+,CPU}$	1.0	0.6667	0.5	0.4	0.3333
$q_{Z-,Z-}$	0.0	0.0	0.3333	0.5	0.6
$q_{Z-,CPU}$	0.0	1.0	0.6667	0.5	0.4
$q_{CPU,X+}$	0.5714	0.5079	0.5023	0.5010	0.5005
$q_{CPU,X-}$	0.0	0.2540	0.3349	0.3757	0.4004
$q_{CPU,Y+}$	0.2857	0.1270	0.0837	0.0626	0.05005
$q_{CPU,Y-}$	0.0	0.0635	0.0558	0.0470	0.04004
$q_{CPU,Z+}$	0.1429	0.0317	0.0140	0.0078	0.00501
$q_{CPU,Z-}$	0.0	0.0159	0.0093	0.0059	0.00400

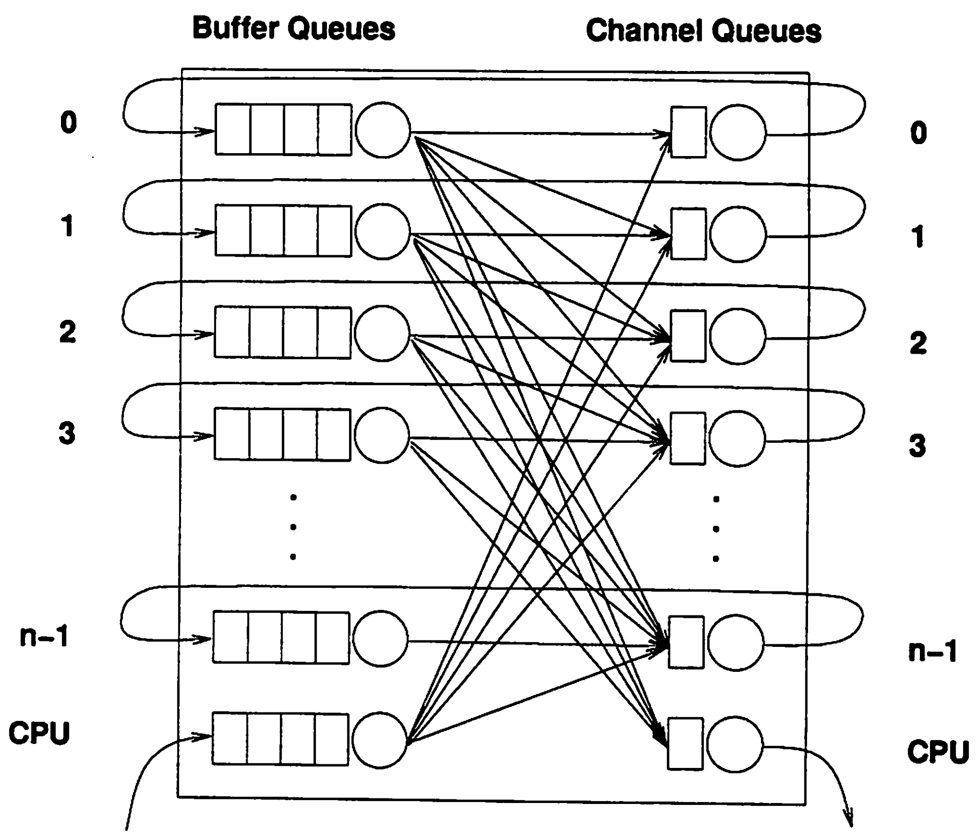


Figure B.3: Queueing Model of a Hypercube Network

Table B.3: Routing Parameters for a Hypercube

q_{ij}	Number of Processors				
	8	64	256	512	1024
$q_{1,2}$	0.5	0.5	0.5	0.5	0.5
$q_{1,3}$	0.25	0.25	0.25	0.25	0.25
$q_{1,4}$	0.0	0.125	0.125	0.125	0.125
$q_{1,5}$	0.0	0.0625	0.0625	0.0625	0.0625
$q_{1,6}$	0.0	0.03125	0.03125	0.03125	0.03125
$q_{1,7}$	0.0	0.0	0.015625	0.015625	0.015625
$q_{1,8}$	0.0	0.0	0.0078125	0.0078125	0.0078125
$q_{1,9}$	0.0	0.0	0.0	0.00390625	0.00390625
$q_{1,10}$	0.0	0.0	0.0	0.0	0.00195312
$q_{1,CPU}$	0.25	0.03125	0.0078125	0.00390625	0.00195312
$q_{2,3}$	0.5	0.5	0.5	0.5	0.5
$q_{2,4}$	0.0	0.25	0.25	0.25	0.25
$q_{2,5}$	0.0	0.125	0.125	0.125	0.125
$q_{2,6}$	0.0	0.0625	0.0625	0.0625	0.0625
$q_{2,7}$	0.0	0.0	0.03125	0.03125	0.03125
$q_{2,8}$	0.0	0.0	0.015625	0.015625	0.015625
$q_{2,9}$	0.0	0.0	0.0	0.078125.	0.0078125
$q_{2,10}$	0.0	0.0	0.0	0.0	0.00390625
$q_{2,CPU}$	0.5	0.0625	0.015625	0.0078125	0.00390625
$q_{3,4}$	0.0	0.5	0.5	0.5	0.5
$q_{3,5}$	0.0	0.25	0.25	0.25	0.25
$q_{3,6}$	0.0	0.125	0.125	0.125	0.125
$q_{3,7}$	0.0	0.0	0.0625	0.0625	0.0625
$q_{3,8}$	0.0	0.0	0.03125	0.03125	0.03125
$q_{3,9}$	0.0	0.0	0.0	0.015625	0.015625
$q_{3,10}$	0.0	0.0	0.0	0.0	0.0078125
$q_{3,CPU}$	1.0	0.125	0.03125	0.015625	0.0078125
$q_{4,5}$	0.0	0.5	0.5	0.5	0.5
$q_{4,6}$	0.0	0.25	0.25	0.25	0.25
$q_{4,7}$	0.0	0.0	0.125	0.125	0.125
$q_{4,8}$	0.0	0.0	0.0625	0.0625	0.0625
$q_{4,9}$	0.0	0.0	0.0	0.03125	0.03125
$q_{4,10}$	0.0	0.0	0.0	0.0	0.015625
$q_{4,CPU}$	0.0	0.25	0.0625	0.03125	0.015625
$q_{5,6}$	0.0	0.5	0.5	0.5	0.5
$q_{5,7}$	0.0	0.0	0.25	0.25	0.25
$q_{5,8}$	0.0	0.0	0.125	0.125	0.125
$q_{5,9}$	0.0	0.0	0.0	0.0625	0.0625
$q_{5,10}$	0.0	0.0	0.0	0.0	0.03125
$q_{5,CPU}$	0.0	0.5	0.125	0.0625	0.03125

Table B.4: Routing Parameters for a Hypercube (continued)

q_{ij}	Number of Processors				
	8	64	256	512	1024
$q_{6,7}$	0.0	0.0	0.5	0.5	0.5
$q_{6,8}$	0.0	0.0	0.25	0.25	0.25
$q_{6,9}$	0.0	0.0	0.0	0.125	0.125
$q_{6,10}$	0.0	0.0	0.0	0.0	0.0625
$q_{6,CPU}$	0.0	1.0	0.25	0.125	0.0625
$q_{7,8}$	0.0	0.0	0.5	0.5	0.5
$q_{7,9}$	0.0	0.0	0.0	0.25	0.25
$q_{7,10}$	0.0	0.0	0.0	0.0	0.125
$q_{7,CPU}$	0.0	0.0	0.5	0.25	0.125
$q_{8,9}$	0.0	0.0	0.0	0.5	0.5
$q_{8,10}$	0.0	0.0	0.0	0.0	0.25
$q_{8,CPU}$	0.0	0.0	1.0	0.5	0.25
$q_{9,10}$	0.0	0.0	0.0	0.0	0.5
$q_{9,CPU}$	0.0	0.0	0.0	1.0	0.5
$q_{10,CPU}$	0.0	0.0	0.0	0.0	1.0
$q_{CPU,1}$	0.5714	0.5079	0.5020	0.5010	0.5005
$q_{CPU,2}$	0.2857	0.2540	0.2510	0.2505	0.2502
$q_{CPU,3}$	0.1429	0.1270	0.1255	0.1252	0.1251
$q_{CPU,4}$	0.0	0.0635	0.0627	0.0626	0.0626
$q_{CPU,5}$	0.0	0.0317	0.0314	0.0313	0.0313
$q_{CPU,6}$	0.0	0.0159	0.0157	0.0157	0.0156
$q_{CPU,7}$	0.0	0.0	0.00784	0.00783	0.00782
$q_{CPU,8}$	0.0	0.0	0.00392	0.00391	0.00391
$q_{CPU,9}$	0.0	0.0	0.0	0.00196	0.00196
$q_{CPU,10}$	0.0	0.0	0.0	0.0	0.000978

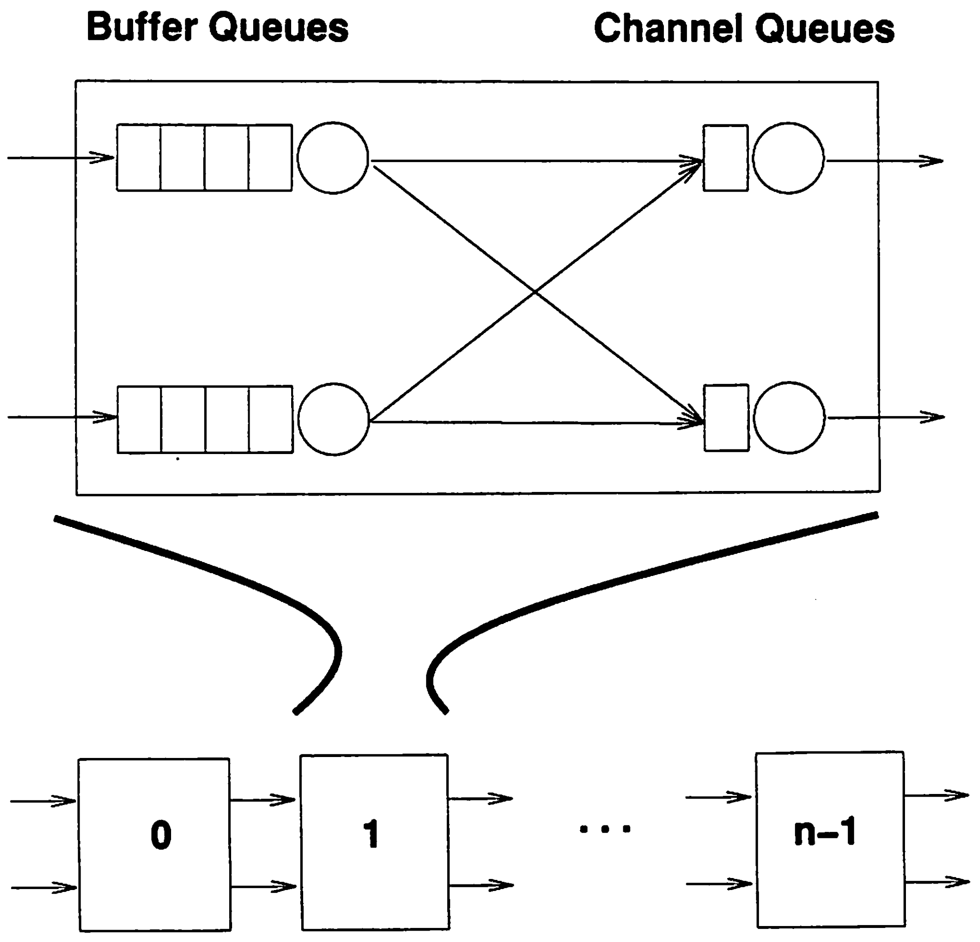


Figure B.4: Queueing Model of Radix-2 Delta Network

B.7 Radix-4 Delta

All routing probabilities are $\frac{1}{4}$.

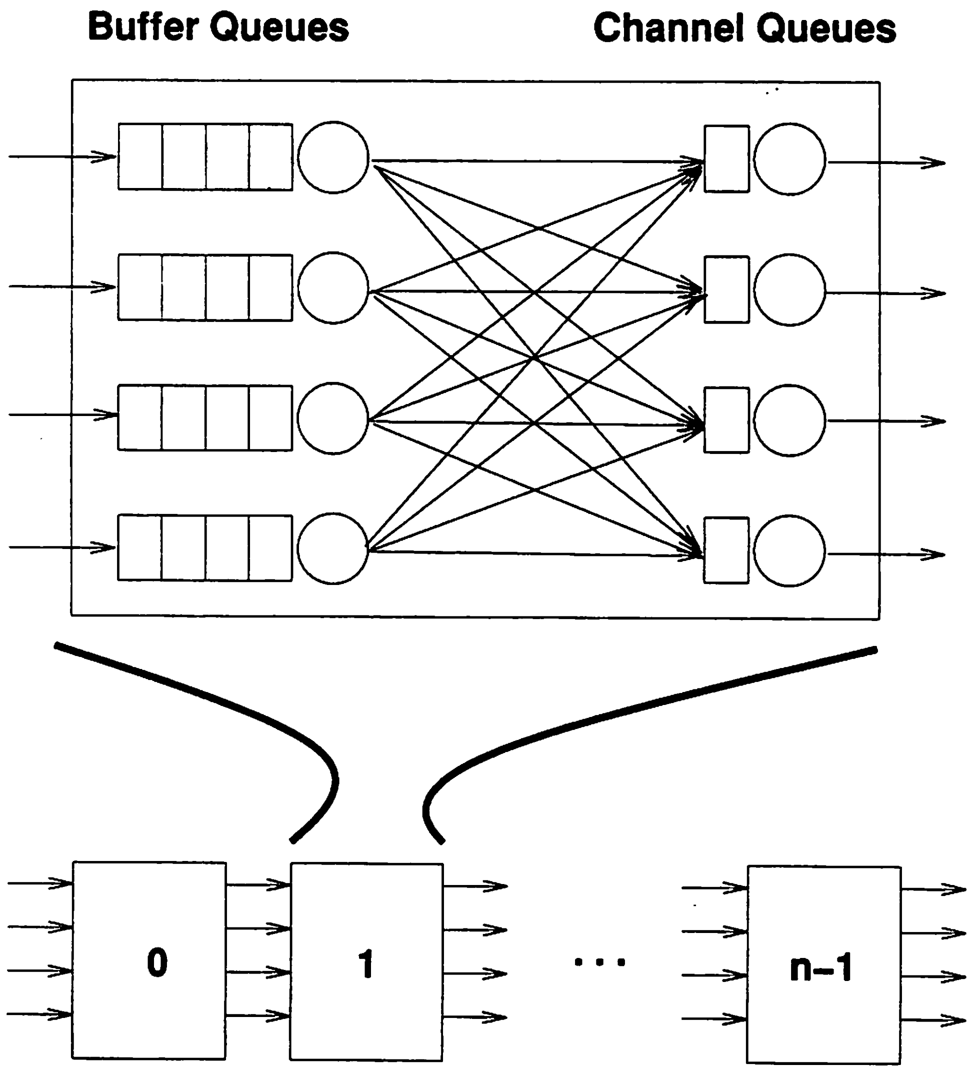


Figure B.5: Queueing Model of Radix-4 Delta Network

B.8 Radix-8 Delta

All routing probabilities are $\frac{1}{8}$.

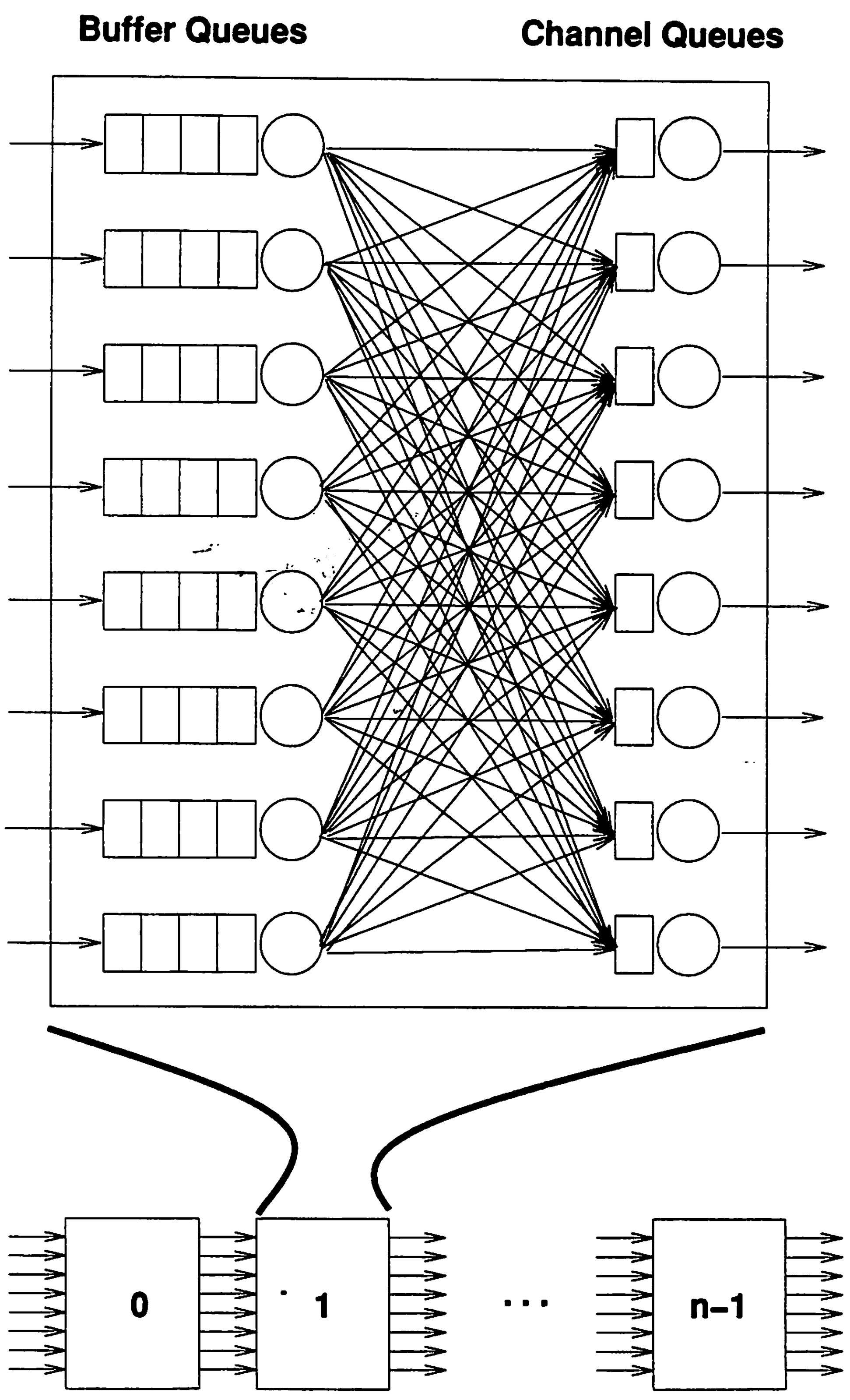


Figure B.6: Queueing Model of Radix-8 Delta Network