The Meerkat Multicomputer: Tradeoffs in Multicomputer Architecture

by

Robert C. Bedichek

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1994

Approved by____
(Co-Chairperson of Supervisory Committee)

____
(Co-Chairperson of Supervisory Committee)

____

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

The Meerkat Multicomputer: Tradeoffs in Multicomputer Architecture

by Robert C. Bedichek

Co-Chairpersons of Supervisory Committee: Professor Henry M. Levy
Professor Edward D. Lazowska
Department of Computer Science
and Engineering

A central problem preventing the wide application of distributed memory multicomputers has been their high price, especially for small installations. High prices are due to long design times, support for scaling to thousands of nodes, and high production costs. This thesis demonstrates a new approach that combines some carefully chosen restrictions on scaling with a software-intensive methodology.

We used a concrete design, Meerkat, as a vehicle for exploring the multicomputer design space. Meerkat is a distributed memory multicomputer architecture effective in the range from several to hundreds of processors. It uses a two-dimensional, passive backplane to connect nodes composed of processors, memory, and I/O devices. The interconnect is conceptually simple, inexpensive to design and build, has low latency, and provides high bandwidth on long messages. However, it does not scale to thousands of processors, provide high bandwidth on short messages, or implement cache coherent shared memory.

This thesis describes Meerkat's architecture, the niche that Meerkat fills, and the rationale for our design choices. We present performance results obtained from our hardware prototype and a calibrated simulator to demonstrate that parallel numerical workloads work well on Meerkat.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

Very few graduate students have the good fortune to work for such excellent professors as Hank Levy and Ed Lazowska. I thank them for their patience, wisdom, and encouragement. Robert Henry, the third member of my reading committee and my first advisor here in the department, has helped me in many ways, most recently by his critique of this thesis.

I would like to extend my appreciation to a number of people who helped me with Meerkat. George Forman built the first SUN-Meerkat interface. Joao Setubal, the first user other than myself, cheerfully tolerated many new-system bugs. Tektronix Research, Aaron Eastlund, and Motorola each donated one or more 88000 Hypermodules, the core of each Meerkat node. John Barr at Motorola is responsible for his company's donation. Horizon Research donated 128 MB of DRAM modules and shared their port of Mach 3.0 and OSF/1. Reuven Koblick and Takashi Kan, at Horizon Research and Mitsubishi Electric Company, were instrumental in these generous donations. Lattice Semiconductor donated all of the high speed PALs. Xilinx donated all of the FPGAs and the FPGA design software. Data I/O donated FGPA logic specification software.

Robert Short of Microsoft and Burton Smith of Tera gave valuable advice on design. Gary Hornbuckle of Tektronix generously shared his 88000-based workstation design on which my initial prototype circuit design was based. Jim Canon of Tektronix performed the legal maneuvers that allowed me to use Gary's design. Jeff Beachy of Tektronix and John Theus of The Theus Group provided expert guidance in the electrical engineering aspects of Meerkat. Arun Somani refined my understanding of fault tolerance issues. Rik Littlefield convinced me to use DRAM instead of VRAM for Meerkat node memory, sponsored my Delta account, and generally gave great advice. Craig Anderson wrote prototype run-time system code and reviewed chapters of this dissertation. Curtis Brown wrote and debugged the FFT benchmark and was my coauthor on a paper about Meerkat.

I would also like to thank the following people for their helpful comments and suggestions: Jean-Loup Baer, Kevin Bolding, Carl Ebeling, Susan Eggers, Ed Felten, Melanie

Fulgham, Dennis Gentry, David Keppel, Alex Klaiber, Peter Magnusson, Cathy McCann, Neil McKenzie, Dylan McNamee, Paul Messner, Larry Snyder, and Chandu Thekkath.

David Keppel, Alex Klaiber, Dylan McNamee, and Jeff Chase encouraged me to persist. David, Alex, and Dylan also spent many hours discussing Meerkat and commenting on drafts of this dissertation and on earlier papers about Meerkat.

Sandy Kaplan edited this dissertation, correcting my English, requesting further explanation in places, suggesting organizational changes, and for her support and encouragement. Remaining errors are mine.

The Computer Science Laboratory (CSL) staff of the department of Computer Science and Engineering often went beyond the call of duty to maintain the systems on which most of this work was performed. Warren Jessup, in particular, cheerfully took desperate calls on sunny Sunday afternoons, permitted device driver hacking on his systems that sometimes crashed them, and let me do other things that system support staff in other institutions rarely allow.

Another CSL staff member deserves special mention: Stephen Lee built most of the node boards while performing his official duty of supporting the undergraduate digital laboratory.

I extend my thanks to other CSL'ers: Erik Lundberg, Voradesh Yenbut, Nancy Johnson-Burr, Mark Murray, and Paul Barton-Davis, and to Center for Integrated Systems staff member Larry McMurchie. Researchers anywhere are lucky if they have support staff half as good as those in our department. In this respect, I am spoiled for life.

The office staff in the department have been very competent and helpful at every turn. I especially need to thank Frankye Jones, whose seems to be able to make the University bureaucracy do anything that she decides needs to be done on behalf of deserving students.

This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by The Pacific Northwest Laboratory of the United States Department of Energy, to whom I am grateful.

I dedicate this thesis to my late brother, John Bedichek, who infused me with a love for simple design and ambitious projects.

Chapter 1

INTRODUCTION

Parallel computers are systems of multiple processors capable of coordinated computation. They vary widely in their scalability, size, cost, application, architecture, construction, and complexity of design. This thesis focuses on parallel computers that scale to hundreds of nodes, are inexpensive to design and build, and are applicable to a range of applications. Our goal is to demonstrate that by limiting the scaling range, pushing function into software, and paying careful attention to hardware technology, we can design a multicomputer that has better performance for the price than existing systems.

1.1 The High Cost of Parallel Computers

Parallel computers come in many forms. Networks of workstations connected by Ethernet [65] are sometimes used as parallel computers. Ethernet allows workstations to communicate, albeit with low bandwidth and high latency. System software such as PVM [89], Munin [21], Emerald [47], and Amber [22] provides application support on such parallel computers. Some applications require neither high bandwidth nor low latency, and these are well served by networks of workstations.

On a global scale, the Internet can be considered a parallel computer. It provides an even lower bandwidth and higher latency interconnect than Ethernet. Yet, it can provide computing power many orders of magnitude higher than that available in any national laboratory. Cryptology researchers recently cracked a 129-digit RSA cypher using the Internet and thousands of cooperating hosts. This application required very little communication and a vast number of calculations.

If networks of workstations represent one end of the design spectrum, the older, tightly coupled Cray parallel computers, such as the Y-MP, represent the other. These systems have high-bandwidth, low-latency interconnects that allow a small number of fast processors to communicate through a shared memory. Memory is divided into banks, which operate in parallel. Each processor has a port, or an access path, that connects it to the memory system. Data are typically spread among the banks, allowing high-bandwidth access. Processors communicate by transmitting data to the memory system, and receiving data from the memory system, stored by another processor.

The older Crays are expensive because their high design costs are amortized over few production units, and they use large quantities of high-speed, bipolar circuitry. This circuitry is fast, but low density, and requires expensive cooling. The memory system requires a significant amount of this circuitry to provide multiple high-bandwidth and low-latency access paths for the processors. Dividing the memory into banks allows concurrent access, but poses an additional cost in hardware. Well-funded laboratories are willing to pay these high costs for fast systems that can solve numerical problems requiring either frequent interprocessor communication and/or the fastest single-stream performance.

Between these extremes of both cost and communication performance are systems such as Intel's Paragon [51], Thinking Machine's CM-5 [92], and Cray's T3D. These systems use conventional microprocessors, which lowers their design cost and reduces their physical size (compared to systems, such as the older Crays, whose processors are composed of many chips). However, these multicomputers are still expensive compared to networks of workstations for two primary reasons. First, they are complex systems sold in small numbers. Like the older Crays, they suffer from high design costs that are amortized over few production units. Second, they use communication structures that allow scaling to thousands of processors, which increases cost for all sizes of systems.

1.2 The Thesis

Our thesis is that a new approach to multicomputer design will yield faster and less expensive systems, with a price/performance ratio closer to that of workstation networks. This approach moves function from hardware to software, reducing design time and production cost, and increasing flexibility. Our approach also limits scaling range to allow the use of simple, high-performance hardware technology. We use a concrete design, Meerkat [10], as an example of our approach. We demonstrate that Meerkat can scale to hundreds of processors, has a short design time, is simple to construct, and is applicable to common parallel workloads.

Several design assumptions are key to this thesis. First, we assume that scaling range can be limited to a few hundred nodes while maintaining wide applicability. Second, we assume that by carefully choosing what function goes on each side of the hardware/software boundary, we can achieve high performance with simple hardware. Our third assumption is that available hardware technology greatly favors some architectural choices over others. For example, planar interconnects are cheaper and faster than non-planar ones.

As a secondary goal, this thesis demonstrates the value of an architectural evaluation methodology that synergistically uses a prototype and simulator. Building the prototype forced us to confront complex design issues, and operating it gave us a reference with which to calibrate the simulator. In turn, the simulator let us model variations that were impractical to construct.

1.3 Meerkat's Genesis

The Meerkat project began in the fall of 1990. At the time, the department was seeking a new system on which to do parallel systems research, but could spend only a few hundred thousand dollars. We evaluated machines from Intel, Thinking Machines, and Meiko. Affordable models had only 10 to 30 processors.

We considered the reasons for their expense, and thought about approaches that would yield systems with better price/performance ratios. By carefully choosing the interconnect topology, pushing function into software, and limiting the scaling range, it seemed that it should be possible to achieve performance as good or better than that of these systems, and at a lower cost. Our goal was to explore the architectural design space, not to build a production system.

The technology curves for commercial microprocessors and memory, i.e., the cost/performance as a function of time, are exceptional. Every year the price per unit of performance and per bit drops significantly. Kendall Square Research made the mistake of not using commercial processors and is suffering as a result. Cray Research for years designed their own processors, but has now switched to the DEC Alpha [35] as the core of their multiprocessors. We concluded that a viable machine would use commercial microprocessors and memory.

We envisioned a multicomputer of nodes composed of commodity components, running an operating system that supports message passing, at a minimum. The interconnection of these nodes is a crucial issue; existing systems use a variety of topologies and signalling technology. Silicon Graphics' multiprocessor workstations use a single synchronous bus. The Intel iPSC/2 and iPSC/860 systems use Hypercubes with synchronous bit-serial links built with conventional logic. The Intel Paragon uses meshes of 2-D routing chips and bit-parallel asynchronous links. The Thinking Machines' CM-5 uses a locally synchronous, globally asynchronous, fat-tree network.

We wanted hundreds of processors, more than could be connected to a single medium without unacceptably low internode communication performance. If each node were connected to all media, nodes would be large and expensive and perhaps limit the media's performance. To avoid this problem, the system must scale in two dimensions. But nodes should be kept close together, to limit the intrinsic speed of light delay. This constraint suggested a circle or a square of nodes.

Wires of some kind had to connect the nodes. We searched for a structure that required a minimum of circuitry and that would use the internode wires efficiently.

There could be point-to-point links between nodes, resembling the Intel Paragon. If all the communication were close-range, to a neighbor, the Paragon's mesh would work well. It allows the communication capacity to scale with the number of nodes. But most parallel workloads have a significant amount of communication that is not close-range. The 2-D mesh would still work for any workload, but we questioned whether it was the best.

From an aesthetic point of view, the mesh seemed unnecessarily complex. It performs a routing decision at every hop, for every message. In practice, it does not require much circuitry to do this; mesh router chip cost is dominated by the size of the package, which must be large to accommodate the pins needed to communicate with its four (or, in the case of the Hewlett Packard May-fly, six) mesh neighbors and with the local node. The logic may not cost a lot in silicon area, but it limits performance. Universities such as Caltech have had a hard time making their chips run reliably above 20 MB/sec. The mesh has a per-hop latency that can dominate communication time in fine-grain machines, such as the MIT Alewife.

It seemed that the simplest structure was a set of horizontal and vertical buses. We first thought about using multiple NuBUS's [69]. But like most existing buses, NuBUS is a backplane bus for connecting a small number of processors to a small number of I/O and memory devices. Its peak bandwidth is low (40 MB/sec), and its protocol is more complex than we needed. We then considered a simple bus tailored for software control and bulk transfers.

To make the design fast, we wanted to take advantage of the fast static-column mode of DRAMs. This meant setting up the DRAM for quick access to the words within a

DRAM page (which in our prototype is 16KB). Early on, we considered an interface where the processor would stuff each word into an interface register; however, this approach is unable to sustain more than about one-third of the local memory bandwidth. Avoiding this approach also meant that we did not have to do flow control on each word. Instead, flow control could be performed by software.

The next question we posed was how to define the network interface and low-level software. These two components are tied together closely. To keep the hardware simple, we required the software to manipulate much of the network interface's state.

Meerkat is the result of our choice to use commodity microprocessors and DRAM, a set of horizontal and vertical buses, and a software-intensive approach. The core of the Meerkat node is similar to the core of a conventional workstation. The main difference is a high bandwidth local memory connection to internode buses. This connection is integrated with the memory system and requires little extra hardware.

1.4 Methodology

Key to defending our thesis is an approach involving iterative prototyping, simulation, measurement, and evaluation. This approach is described below.

When the project began, we wrote a simple Meerkat simulator. It totaled only about 400 lines of C and had very rudimentary node processor execution. However, the results it generated indicated that there was no fundamental problem with Meerkat's interconnect: cross-sectional bandwidth was a linear function of the number of internode buses and transfer rate.

We then designed the hardware prototype. This process helped refine our architecture. We saw the benefits of eliminating flow-control hardware and coupling the DRAM system directly to the interconnect. The prototype effort yielded several results even before the first program ran. By keeping the architecture simple, we could achieve a high internode transfer rate and a short design time.

Once the hardware worked, the next step was to experiment with the system software and to make performance measurements. Although the simple hardware forced the software to do a lot, the software was tractable and the performance excellent.

The prototype design does not scale beyond 16 nodes because we made compromises to make the prototype design and construction simpler and less expensive. The limitation of the prototype does not reflect a limitation of the architecture. We could have made the prototype so that it scaled to 256 nodes, but this would have added person-months to the implementation effort.

The actual prototype has only four nodes. We needed a way to demonstrate the performance of larger Meerkats and of Meerkats that made different hardware/software tradeoffs. To do this, we wrote a detailed Meerkat simulator, calibrated it with the prototype, and used it to evaluate the performance of large systems.

We used several low-level benchmarks to measure raw internode communication performance. The benchmarks use a subset of the message-passing functions of NX, the Intel operating system [72], so that we could run them both on Meerkat and on Intel's parallel computers. Measurements on the prototype and simulator showed a high sustained bandwidth on long messages, and a low round-trip latency between node pairs. Simulator measurements also showed high bisection throughput for long messages on 256-node systems. Meerkat's communication performance equaled or exceeded all Intel systems we measured, often by a large margin.

These low-level benchmark results were encouraging, but did not directly answer the question of how Meerkat would perform on numerical applications, and how it would compare to commercially built systems. To answer these questions, we ran several numerical parallel programs on Meerkat and Delta. Meerkat's speedup curves were consistently above Delta's.

Although Meerkat had better speedups and higher internode bandwidth than Delta, we saw from simulator instrumentation that the internode buses were underused when traffic was dominated by small messages. This led us to reexamine our extreme approach of removing everything possible from hardware at the expense of software management.

In developing Meerkat, we knew that some applications would suffer as a result of our extreme approach. By being less extreme and allowing a slightly more complex network interface, we thought that we might retain Meerkat's essential benefits, yet substantially improve performance on programs that use short messages. In some cases extra hardware would be worth the improved performance on small messages; in other cases it would not. The desire for better performance on small messages led us to develop Meerkat-2. Meerkat-2 makes more efficient use of the interconnect through hardware handling of circuit management (establishment through disconnection).

In the past, multicomputer designers focused their efforts on the interconnect and node processor. The network interface, seen as significant to the operating system only, was deemed incidental to performance. More recently, designers have recognized the benefits of streamlining network access [30] and allowing application code direct access to the network interface [45, 38]. Our Meerkat-2 study is in line with this research: we show that the choice of network interface can dominate system performance. For the systems we modelled, this choice becomes more important as message sizes decrease.

To bolster our claim that Meerkat can scale to hundreds of nodes, despite the limited scalability of the prototype, we conducted a paper engineering study. We dealt with the most difficult problems of physical internode bus length, packaging, signalling, and clocking. We found that Meerkat's interconnect has potential for very high performance and the ability to connect hundreds of nodes.

Our study thus concluded. We started with research and intuition about how parallel computers could be built more simply, proposed an architecture, modelled it roughly, built a prototype, measured it, wrote a simulator, calibrated it to the prototype, and used this simulator to extend our measurements. We then modified our earlier hardware-lean approach by proposing a new hardware mechanism, which we modelled and measured.

1.5 Research Contributions

The Meerkat project significantly contributes to the study of multicomputer architecture by:

- Demonstrating that RISC principles can be applied to multicomputer design to yield a system with a short design time *and* high performance. By "RISC principles," we mean that hardware is tuned for the most important functions, while software is used for less important or less performance-critical functions.
- Analyzing tradeoffs in network interface design for circuit-switched multicomputers. We provide performance results and discuss the operating system implications of several network interface designs.
- Exercising a methodology that combines a hardware prototype and an efficient simulator that is calibrated to the hardware. Hardware prototypes and system simulators are tools often used in architectural tradeoff studies. Our approach is unique in the way in which we employed both tools. We used the hardware to calibrate the simulator and to test programs. We used the calibrated simulator to extend our results to systems with hundreds of nodes and various network interfaces. The simulator also allowed us to measure many things that would be difficult or impossible to measure on a hardware prototype. For example, the simulator counts the number of cycles each processor is delayed waiting to access the interconnect.
- Developing a detailed and highly efficient simulator that is able to model the execution timing of hundreds of processors running significant programs.

- Presenting an interconnect with an intrinsically lower latency than that of any others. While not significant for multicomputers such as Meerkat, where software overhead dominates interconnect latency, this property could be of great benefit in multiprocessors for which remote reference latency is dominated by interconnect latency.

1.6 Thesis Organization

Chapter 2 describes the Meerkat architecture and the design issues that guided its development. The hardware prototype, described in Chapter 3, provides insight into the complexity of a Meerkat implementation. The Meerkat simulator is presented in Chapter 4. Chapter 5 relates performance results obtained from measurements made on the hardware and the simulator. Chapter 6 explores a new network interface, Meerkat-2, discussing issues in network interface design and contrasting performance results with a range of alternatives. To show that large Meerkats with fast buses can be built we propose in Chapter 7 solutions to some of the difficult problems of internode signalling. Chapter 8 offers conclusions and discusses avenues for future research.

Chapter 2

MEERKAT DESIGN ISSUES AND ARCHITECTURE

Parallel computers can deliver great computational power. However, realizing their potential requires solving significant problems involving computer architecture, programming language design, compiler techniques, the writing and debugging of parallel programs, performance measurement, hardware design, and fault tolerance.

Architects of parallel computers take many factors into consideration as they develop a design. They make assumptions about how the system will be programmed, what transformations compilers will perform, what range of performance the design must scale over, what tools programmers will use to debug their programs, how reliable the system must be, and what hardware technology will be available when the system is built. This last consideration - the available hardware technology - changes rapidly and has a profound effect on architecture. Constantly improving hardware technology favors approaches that take advantage of commodity components and have short design times.

This chapter discusses the design process for a parallel computer – a multi[1] In particular, we are interested in multis that can be designed quickly, to take advantage of the most modern components. Section 2.1 introduces key design issues and tradeoffs considered in developing Meerkat. Section 2.2 describes Meerkat's structure and operation. Section 2.4 discusses related system software issues. Meerkat's network latency is compared with that of a mesh in Section 2.3.6.

2.1 Introduction

Six key design issues were weighed in arriving at Meerkat's architecture: choice of node processor, network scalability, private vs. shared address space, system design time, uniprocessor vs. multiprocessor nodes, and interconnect planarity. For each issue, we examine the choices considered in creating Meerkat and contrast Meerkat with other multi architectures.

[1] We use the term *multi* to refer both to multicomputers and multiprocessors. Our definitions for these terms are in section 2.1.3.

2.1.1 Node Processors

Node performance as a function of production cost is non-linear. This function grows quickly until it reaches the level of high-volume microprocessors. It then increases slowly, reaching the level of supercomputers at very high cost (see Figure 2.1). The best performance per unit of cost is at the knee of the curve, in the region of state-of-the-art microprocessors. While this curve moves up with time, its shape has remained the same for at least twenty years.

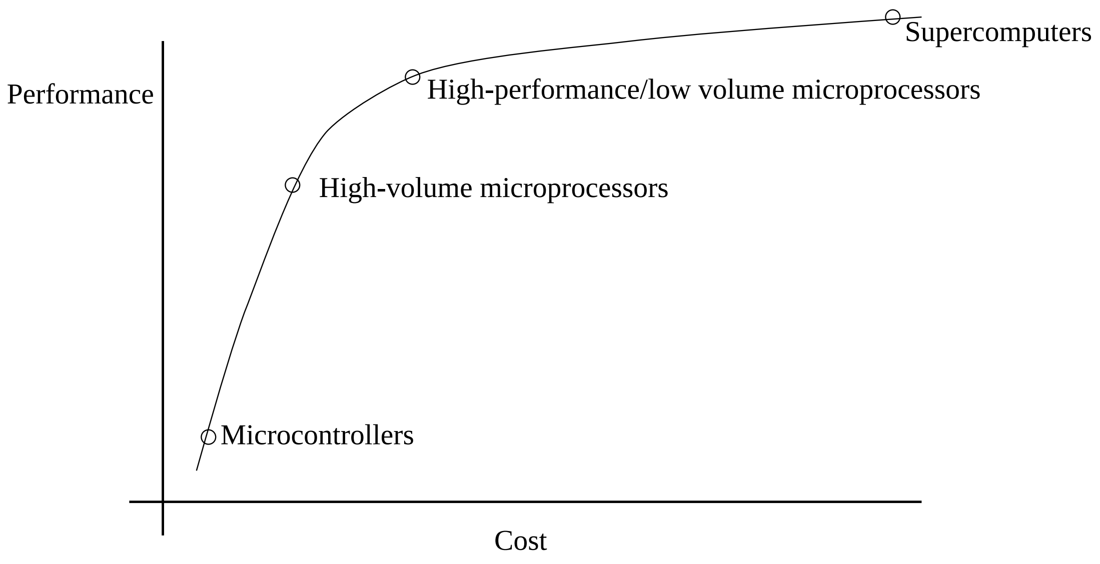


Figure 2.1: Cost-Performance Curve for Uniprocessors

Processors slower than commodity microprocessors serve applications that do not require high performance. While their cost is low, their relative performance is even lower. Aggregating many processors with poor price/performance ratios will not yield a multi with a good price/performance ratio. Furthermore, processors require the support of memory and network interfaces. Although these devices can be matched in speed with the processors they support, and thus be less expensive for slower processors, they will also be more numerous for a given level of system performance. Their lower expense will not fully compensate for their greater number. In addition, more parallelism must be available in a system with more processors. This means that the utilization may be lower, further lowering system price/performance.

Processors much faster than commodity microprocessors serve applications without efficient parallel solutions. Their extra performance comes at a significant cost. Aggregating a few fast processors with poor price/performance ratios will result in a multi with good price/performance *only* if the increased cost of the processors is offset by an increase in processor utilization or a decrease in the cost of supporting devices such as memory and network interfaces. Expensive processors generally require expensive support components: while their numbers will be lower, their cost will increase.

Some experimental parallel systems, such as Mosaic [80], the J-Machine [30], and the Connection Machine [46], use large numbers of small custom processors. Maspar and Thinking Machines Corporation offered the MP-1 [16] and the CM-2, respectively, both of which used many small custom processors.

These systems can yield high performance on regular problems with high parallelism. However, their performance suffers on irregular workloads and those without sufficient parallelism. More significantly, by opting for custom processors, they do not take advantage of the huge investment made in commodity microprocessors. This may be why Thinking Machines' more recent system, the CM-5, uses the popular Sparc microprocessor rather than the tiny-custom-processor approach.

Many experimental and commercial multis use commodity microprocessors. For example, the Stanford DASH [58] uses MIPS R3000s, the Cray T3D uses DEC Alphas, and the Intel Paragon uses Intel i860s. Some experimental systems use modified commercial parts. For example, the MIT Alewife uses modified Sparc microprocessors, while the MIT *T planned to use modified Motorola 88110s [37].

We conclude that commodity microprocessors are the processors of choice in multis where the price/performance ratio is important. High-end commodity microprocessors are probably the best choice because a small sacrifice in the price/performance of the node processor will be compensated for by a modest decrease in the amount of parallelism and support components required.

2.1.2 Limiting Scalability

Multicomputer interconnects scale over some range, allowing system configurations with a variable number of processing nodes. The range of nodes over which an architecture is effective is called its *scalability*, or its scaling range. The Thinking Machines CM-5, for example, is scalable from 32 to 16,384 nodes, a wide range.

Most multi installations range from a few tens to a few hundred processors, despite the ability of manufacturers to make similar models with thousands of processors. There is no lack of demand for the greater computational power of larger systems. Rather, size is limited by economics. Nodes of commercial multis, such as the Thinking Machine's CM-5 [92] and the Intel Paragon [51], use powerful microprocessors and large memories, and therefore cost thousands of dollars. A 256-node system, for example, would cost millions of dollars. Although larger systems are important, they are also rare: most prospective consumers of parallel systems can afford systems with a few tens to a few hundreds of nodes, but not thousands of them.

There is a wide range of scalability in multis. The Intel iPSC/860 is limited to 128 nodes, while the new Paragon can be built with several thousand nodes. The KSR-1 can in theory have 1088 processors; it is not clear how effective the system would be at this size. The Convex MPP scales to 128 nodes.

There are significant advantages to having a few hundred nodes as an upper limit. First, larger system designs must solve complex fault tolerance issues. Second, larger systems require correspondingly larger bisection bandwidths. The CM-5 keeps the bisection bandwidth constant over its scaling range by using a fat-tree network, which is quite expensive. The Intel Paragon does not increase bisection bandwidth in larger systems. Rather, all Paragons have a high-performance interconnect. In small Paragons, this interconnect is more powerful than it needs to be.

We conclude that limiting scalability to a few hundred nodes helps avoid difficult fault tolerance and bisection bandwidth issues, while still allowing all but the most massive and expensive systems to be built.

2.1.3 Shared vs. Private Address Spaces

Parallel computers can be divided into two broad categories: those with shared address space and those with private address space. The former are called *multiprocessors*, while the latter are called *multicomputers*.

Processors in multiprocessors communicate by reading and writing memory locations in a shared address space, using conventional load and store instructions. Communication between nodes is implicit in the execution of memory access instructions. The most common programming model used on multiprocessors in called *shared memory*, because parallel programs written for these machines use a shared address space to communicate and synchronize.

Processors in multicomputers, on the other hand, communicate by explicitly sending each other blocks of data through the interconnect. Load and store instructions access local memory only. The most common programming model used on these systems is called *message passing*, because parallel programs written for these machines explicitly send and receive messages.

Multiprocessors are generally more difficult to design and build than multicomputers. However, many researchers believe that multiprocessors are easier to program [85, 93, 52]. This has led a number of researchers to study how multiprocessors with hundreds of processors should be designed [58, 2, 12, 8]. Multiprocessors require hardware either to keep track of the location and status of cache lines as they move through the system, or to effect extremely low latency internode communication, as in the Cray T3D.

Software can make multicomputers perform as multiprocessors [59]. However, performance on many workloads will not be as good as it would be with hardware support. Some researchers propose intermediate systems that support a shared address space through a combination of hardware and software [15].

Multiprocessor design is interesting and important, however our investigation concerned simple, quick-to-design parallel computers. The effectiveness of multicomputers is well established, and they are simpler than multiprocessors to build, so we chose the former.

2.1.4 Design Time

The amount of time it takes to design a multi profoundly affects its performance. Hardware technology improves quickly, but not predictably. Thus, architects must select components before the first machine of a new architecture is built. The *technology gap* is the time between component selection and system production. The longer the gap, the older the components will be, the slower they will be, and the worse the resulting system may perform.

While hardware technology development makes simple architectures with short design times attractive, other factors argue for more complex designs. In general, the more performance designers try to extract from a given hardware technology, the longer it will take to design the system. A quickly designed multi may take advantage of modern hardware, but use that hardware less efficiently than a multi with a longer design time. Thus, there is a tension between using old hardware efficiently and using new hardware less efficiently.

The KSR-1 is a complex system that had a correspondingly long design time of seven years. Because of the technology gap, its processors were about one fifth the performance of commercial microprocessors at the time of the system's availability in 1992. Whether the KSR-1's complex memory system makes parallel programming easier, and whether this ease compensates for its slow processors, is an open question. But it is clear that the KSR architects traded node performance for other characteristics (such as making the system easier to program).

We were interested in exploring whether a quickly designed multi could be effective. Thus, in Meerkat, we opted for a simple design.

2.1.5 Uniprocessor versus Multiple-Processor Nodes

Multicomputers can have a single, general-purpose processor or multiple processors per node. Multiple processors allow exploitation of fine-grain parallelism within a node. They also spread the per-node cost of memory, the network interface, and support logic over several processors.

However, there are several practical and theoretical problems with having multiple, general-purpose processors per node:

- The design time is significantly longer. The designers of the Stanford FLASH multiprocessor chose to use one processor per node [55], citing design complexity of multiple processor nodes as the principal reason.
- The operating system is more difficult to write and debug, because it must support multiple nodes and multiple processors per node. Even the simple run-time system that we built for our prototype (see Chapter 3) was complicated and made slower by the need to deal with multiple processors.
- The programming model is a more complex mix of small-scale, fine-grain parallelism and larger-scale, coarse-grain parallelism. While some research systems have supported this model [22], it has not gained wide acceptance.
- User-level access to the network interface (see Chapter 6) is more complicated, because much of the network interface state must be duplicated for each processor.
- Physical node size will increase to accommodate the additional processors, which affects node performance. If the larger nodes make intranode wires longer, a fast node cycle time will be harder to achieve. If the larger nodes increase internode wires, internode latency will increase. One or both of these effects will occur in a given implementation.

While our prototype uses multiple-processor nodes, we envision future Meerkat implementations with single-processor nodes.

2.1.6 Interconnect Planarity

Multi interconnect performance is limited by the density of the wires that connect nodes [31]. Meerkat's interconnect is *planar*, which means that the wires connecting nodes can be routed in a limited number of wiring planes. These limits allow an inexpensive implementation with a conventional, controlled-impedance backplane driven by CMOS integrated circuits.

Wiring densities of the printed circuit boards (from which backplanes are made) are about 80 wires per inch per plane. Non-planar interconnects must use cables to connect the node circuit boards. The wiring density of cables is approximately 10 wires per inch. In addition to their order-of-magnitude advantage in density, printed circuit wires are less expensive and more reliable than cables.

Nonplanar interconnect topologies, such as the hypercube [78], cannot be wired using low-cost, high-density printed circuit wires. Instead, nodes of these systems are connected by links that use cables, which have relatively few wires, often just one. To maintain the same data rates, these cables must use much higher clock rates. High clock rates, in turn, require expensive coaxial or fiber-optic cables and GaAs drivers [40]. Thus, planar interconnects using printed circuit wiring are intrinsically cheaper per unit bandwidth.

The performance of GaAs is not improving as rapidly as CMOS [17]. Therefore, we expect the relative advantage of CMOS-driven interconnects to increase. With recent advances in CMOS technology, it is possible today to fabricate a single, low-cost CMOS gate array that drives over 100 terminated bus wires at 100 MHz [42].

Fat-tree networks [57] are planar in theory, but in practice manufacturers such as Thinking Machines are not able to take advantage of this property. The CM-5's fat-tree network, for example, uses coaxial cables to connect nodes of the fat tree.

Interconnects that are planar both in theory and in practice have substantial advantages in density, performance, and cost. To gain these advantages, we chose to make Meerkat's interconnect planar.

2.2 Meerkat's Architecture

Meerkat connects nodes with a two-dimensional, passive backplane. A *node* is a processor tightly coupled with cache, memory, and I/O devices (optional). Two or more nodes are connected by a set of wires, called a *bus*, used to exchange information. Buses can be *horizontal* or *vertical*. Arranged on a grid, each node *taps* one horizontal and one vertical internode bus with circuits that can send and receive.

Any pair of nodes can communicate using either one or two buses. A pair of nodes that taps the same vertical or horizontal bus communicates with a *one-bus* connection. A node pair that cannot use a one-bus connection must use an intermediate node as a *cross-point* to communicate using a *two-bus* connection. Every node is a potential cross-point for two-bus connections between nodes that share its vertical and horizontal buses. The *sender* owns the bus(es) for the duration of the connection through which it transmits information to the *receiver*.

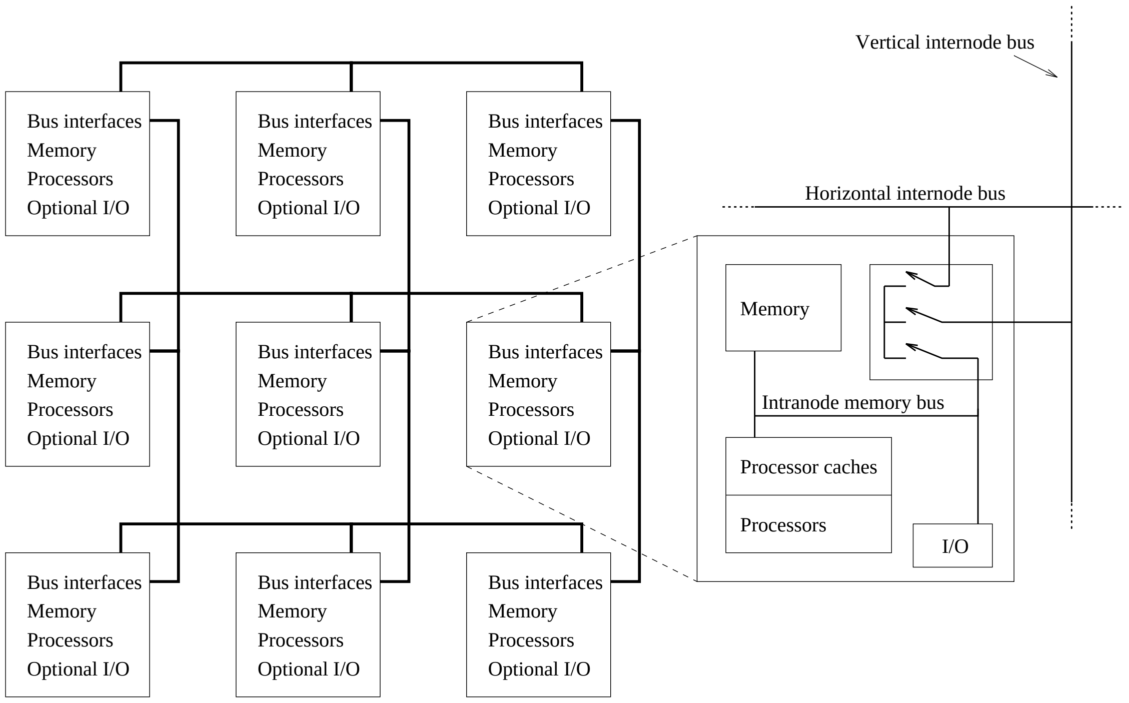


Figure 2.2: Structure of a 3x3 Meerkat

Meerkat's architecture (Figure 2.2) resembles those of the Wisconsin Multicube [41] and Aquarius Multi-multi [20]. However, these two machines put their buses to different uses than does Meerkat. The Multicube and Aquarius implement a coherent shared memory system in hardware. Therefore, transactions on their buses are initiated by memory reference instructions and cache coherence operations. Packets on these buses are small, usually carrying a cache line or less of data. In contrast, Meerkat's buses are manipulated by low-level, message-passing code that is invoked explicitly by user programs to send packets that often will be hundreds or thousands of bytes in length.

2.2.1 Communication in Meerkat

A node's processor accesses its two bus taps through a *bus interface* that occupies part of the processor's physical address space and is thus accessed via load and store instructions. The bus interface provides five primitive commands: **arbitrate**, **signal**, **data-send**, **data-receive**, and **release**. Each of these operations can be applied to either of the two taps of the bus interface.

The **arbitrate** command instructs the bus interface to acquire control of a vertical or horizontal bus. In addition, it can request that a second bus, an *orthogonal bus*, be acquired by a second node that shares the first bus with the owning node. If the arbitration for both buses is successful, the second node acts as cross-point for the owning node. The processor in a node acting as a cross-point is unaffected unless it attempts to use either of its buses. In this case, it will be unable to do so until the owning node relinquishes ownership. The tap connecting an owning node to a bus is said to be *active*.

After the owning node has established a one- or two-bus connection, and before it transfers data, it must alert the receiver by using the **signal** command. This command sets bits in the status registers of the sender and receiver, and it requests a processor interrupt in the receiver node. Either the set status bit or the interrupt tells the receiver to accept data from the interconnect.

To enter a receptive state, the receiver executes the **data-receive** command. This resets the status bits and interrupt request. The sender can then execute the **data-send** command to copy data from the owning node's memory through its active tap, onto a bus, possibly through a cross-point and onto a second bus, through a tap in the receiver, and into the receiver's memory. The sender includes the address of data to be copied from its memory as part of the **data-send** command. The receiver similarly includes the address of the buffer where the data are to be written as part of the **data-receive** command.

A *packet* consists of data sent in a single **data-send** operation. Packets can have length limits and restrictions on their alignment in memory. Multiple packets can be sent over a connection by repeating the **signal**/**data-receive**/**data-send** sequence. The receiver can mask internode interrupts and instead poll the status bit in its bus interface to determine when a sender is waiting for it to execute a **data-receive**. This is useful when a receiver knows from one packet that another is due to arrive shortly. The receiver can poll and thus avoid the interrupt latency for each packet received.

When the owning node has sent all relevant data, it **releases** the bus(es). Other nodes can then arbitrate and become owners of the bus(es).

2.3 Meerkat System Scalability

The design of Meerkats that can scale up to 256 nodes will have to solve problems of internode signalling, power distribution, and cooling. Sections 2.3.1, 2.3.2, and 2.3.3 address the signalling issue, while Section 2.3.4 discusses the relationship between a machine's dimensionality and its cooling and power requirements. We compare Meerkat's underlying network latency with that of a 2-D mesh in Section 2.3.6. Section 2.3.7 concludes this section with our view of how buses will be used in the future.

If buses are short, or if the signalling rate is low, bus design is easy. However, we are interested in high-performance buses that can span the width and height of a Meerkat array. To make the buses short, nodes should be small. This allows them to be packed closely together, thereby reducing bus length.

Chapter 7 proposes a Meerkat design that scales to 16 nodes per bus (256 nodes total), uses nodes that are 11cm along their shortest edge, and limits bus length to 75 cm. We believe that it is practical to build 11 cm-wide nodes with conventional printed circuit boards, high performance microprocessors, and copious memories. However, the combination of a bus length of 75 cm and the desire for high signalling rates still presents design challenges.

2.3.1 The Steady-State Design Rule

Bus designers choose protocols to make their job tractable. One such rule is to constrain the bus cycle time to be long enough for the bus signals to reach a steady state throughout the bus at a predictable point in the bus cycle. With this restriction, the maximum bus wire length is determined by the amount of time it takes for signals to propagate the length of the bus. A typical bound for the bus clock period is:

$$T_{cycle} > 2T_{prop} + T_{setup} + T_{skew}$$

The $2T_{prop}$ term represents the worst case time for a wired-OR glitch to dissipate [43]. T_{setup} is the time that the receiver's latch requires the data to be stable before it is latched. T_{skew} is the maximum uncertainty in the clock signal that tells the receiver's latch when to sample the value on the bus wire.

This rule allows the bus to be treated as simply another logic element. It also limits bus bandwidth to be a function of bus length. While convenient for the logic designer, this rule is not necessary. Some researchers have taken this rule to be one that is intrinsic to buses: Barroso and Dubois motivate small rings for cache-coherent multiprocessors by listing this as a fundamental problem with buses [8].

2.3.2 Beyond Steady State: High Performance Bus Signalling

If the steady-state design rule is used, bus clock speed decreases as bus length increases. However, if the system designer circumvents this rule, the bus can be clocked at a high rate that is mostly independent of bus length. There are several techniques that can be used to achieve this higher rate:

- Phase-tolerant signalling, such as that being used in the next generation of 2-D mesh routers from Intel's Scientific Systems Division [74]. This kind of signalling helps compensate for clock skew, thus allowing the receivers to know with great precision when they should sample the bus wires.

- Carrier Sense (CS), Multiple Access with Collision Detection (MACD), and Collision Enforcement (CE). This is the technique that Ethernet employs so that senders can transmit without knowing that they have exclusive access to the bus [65]. This optimistic protocol allows data transmission and arbitration to proceed concurrently. This technique can also be used in high performance parallel buses.

- Gunning Transceiver Logic (GTL) [42]. This technique requires a fraction of the signalling energy of older methods, such as ECL, BTL and CMOS. GTL transceivers can be fabricated with conventional CMOS transistors. GTL is used in the new XDBus from SUN Microsystems and Xerox Corp [84]. It allows many parallel bits to be sent by a small number of packages. This, in turn, makes it easier to keep stub lengths short, clock skew short, and settling times due to SSO (Simultaneous Switching Output) short.

- An idle-bus-undriven protocol. With this technique, when the bus is idle, no devices are sinking current, i.e., no devices are sending any kind of a signal. A bus that goes from such an idle state into a driven state will not experience a wired-OR glitch and thus can avoid having to wait for a bus round-trip propagation to let such a glitch settle [43]. When a bus goes from being owned by one node to being owned by another, it will have to wait for the wired-OR glitch to settle. But this wait has to be paid only once per tenure and only when tenure is passed directly from one bus master to another.

All of these techniques will allow buses to approach the clock speed, and thus the bandwidth, of short point-to-point connects.

2.3.3 Bus Loading

Buses have multiple taps, these taps affect per-unit-length inductance and capacitance, which, it turn, determine the propagation speed. This section discusses these relationships.

Characteristic impedance is:

$$Z_0 = \sqrt{\frac{L_0}{(C_0+C_d)}}$$

where C_d is the distributed tap capacitance per unit length, C_0 is the bus wire capacitance per unit length, and L_0 is the bus wire inductance per unit length [94]. A bus with no taps corresponds to $C_d = 0$. Propagation speed is:

$$T'_{pd} = \sqrt{L_0(C_0 + C_d)}$$

and is independent of impedance. Note that both propagation speed and impedance are functions of the per-unit inductance and capacitance. The distributed gate capacitance lowers propagation speed. If an untapped wire (i.e., $C_d = 0$) has propagation speed of T_{pd}, the tapped (i.e., capacitatively loaded) wire will have speed:

$$T'_{pd} = T_{pd}\sqrt{1 + \frac{C_d}{C_0}}$$

For example, a microstrip line with $Z_0 = 50$, $C_0 = 35pf/ft$, $P_d = 1.77ns/ft$ loaded with four $5pf$ taps per foot (i.e., $C_d = 20pf/ft$) would have a speed of:

$$1.77\sqrt{1 + \frac{20}{35}} = 2.21ns/ft$$

In this example, the capacitive loading slowed the propagation speed by 25%.

The alternative to buses are point-to-point networks, such as rings and meshes. In these networks, signals travel a short distance at a higher propagation rate and then are absorbed by logic that will reintroduce the signal on the next link. Section 2.3.6 analyzes the relationship between propagation rate and latency in buses and point-to-point networks.

2.3.4 Interconnect Dimensionality vs. Cooling and Power Requirements

Meerkat uses a two dimensional interconnect, and not a three dimensional interconnect, for two reasons. First, by limiting the interconnect to two dimensions, the bus wires can be kept short by letting the nodes be long in the third dimension. Let us call the two dimensions of the Meerkat buses X and Y. The nodes extend in the third dimension, Z.

To keep bus wires short, we limit the X and Y dimensions, at the expense of a larger Z (i.e., node boards extend farther in the Z dimension that they would if their X and Y dimensions could be greater). A Z larger than X and Y is not a problem as long as nodes do not become so long and narrow that the intranode wire lengths limit intranode performance (e.g., by limiting node cycle time).

If the interconnect were three dimensional, we would not be able to sacrifice Z to keep internode bus wires short. This is because there would be bus wires in all three dimensions, including Z. The bandwidth advantage of using the third dimension for internode connections would be mitigated by the reduction in performance due to longer bus wires.

The second reason to keep the interconnect two dimensional is that it makes power distribution and cooling easy. The need for power and cooling grows as the system's physical volume; the ability to provide cooling and power grows as the system's physical surface area. These grow together in a two dimensional system: $O(\frac{N^2}{N^2})$; volume outpaces surface area in a three dimensional system: $O(\frac{N^3}{N^2})$. (N is the length of one edge of the systems, $N = \sqrt{P}$ for Meerkat, P is the number of nodes.)

With single-rail, low-swing signalling, the bus termination will dissipate little power, and that only at the bus ends. The bus ends are easy to cool because they are on the system's surface.

In a two dimensional system such as Meerkat, it is no harder to cool the first node added than the last. This is not the case in a system with a three dimensional interconnect, e.g., the Tera or the Cray T3D. The average node added to a 3-D machine requires cooling/power cables which cost $O(N)$. The average node added to a 2-D machine requires cooling/power cables with constant cost.

2.3.5 Network Bandwidth: Circuit-Switched versus Packet-Switched Networks

Network bandwidth changes as a function of system size and communication pattern. The bisection bandwidth of 2-D meshes, which are packet-switched, and Meerkat, which is circuit-switched, are the product of: (1) the number of wires that cross an imaginary line dividing the nodes into two equal groups, and (2) the data rate sustainable across each wire. The number of wires that cross a bisection line increases as the square root of the number of nodes. If the bandwidth of the links and buses is kept constant, the bisection bandwidth will grow as the square root of the number of nodes in both kinds of networks. To keep bisection bandwidth constant, the bandwidth of the links or buses must double when the number of nodes doubles.

The bisection bandwidth is not the only measure of a network's ability to carry data. This metric is useful when the expected communication pattern shows no locality, i.e., when the sender and receiver are separated by a distance that is on the order of the length of the edges of the network. However, some workloads exhibit good locality; nodes exchange data with nearby nodes. For these workloads, the bandwidth of a mesh grows with increases in the number of nodes. The Meerkat interconnect cannot take significant advantage of locality. Therefore, its bandwidth is not significantly affected by whether the communication pattern is local.

2.3.6 Network Latency in Meerkat vs. 2-D Meshes

The latency of communication in a multi is the sum of the network latency, network interface latency, and the software overhead that lies on the critical path. This section compares the network latency of Meerkat with 2-D mesh networks. We show that the network latency in the case of light load is much lower than that of 2-D meshes.

Consider the latency of a datum in each of the two networks shown in Figure 2.3. Let:

- D be the distance that the datum has to travel, measured in node-distances, e.g., if the datum has to travel two node distances in the horizontal direction and one in the vertical, $D = 3$.
- T_s be the time for light to travel the internode spacing.
- $V_{meerkat}$ is the velocity factor of the Meerkat bus. This will be about $.5$ for a bus built with copper wires, conventional circuit board material (e.g., PF4), and with the moderate capacitive loading of bus taps.
- V_{mesh} is the velocity factor of the point-to-point links in the 2-D mesh. This will be about $.6$ for a link built with copper wires, conventional circuit board material (e.g., PF4), and capacitive loads at the end of the wires.
- T_f be the fall-through time for both the Meerkat cross-point switch and the 2-D mesh router chips. The T_f for the last generation of router chips is about 50 nsec; it should be around 20 nsec for cutting-edge chips available in the next few years.
- $T_{Meerkat}$ be the time for the datum to travel from source to destination in Meerkat, a distance of D nodes. We assume that the datum must travel through a cross-point

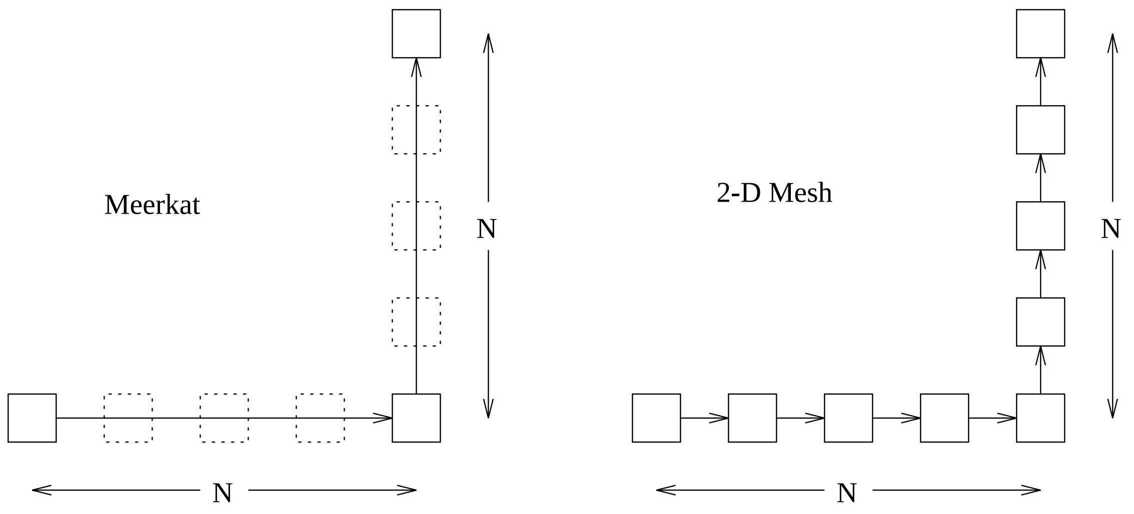


Figure 2.3: Non-Local Communication in Meerkat and a 2-D Mesh

switch in Meerkat, i.e., that the sender and receiver do not share a horizontal or vertical bus.

- T_{mesh} be the time for the datum to travel from source to destination in the 2-D mesh, a distance of D nodes.

The time for our datum to go from source to destination then, is:

$$T_{meerkat} = D\frac{T_s}{V_{meerkat}} + T_f$$

The internode bus arbitration in Meerkat can be overlapped with data transfer and therefore does not affect the latency. Such overlapping is feasible with an optimistic approach that handles collisions gracefully.

In a system with two-rail drivers, where each bit driver switches to either power or ground, a conflict between nodes must be avoided completely as it causes large current spikes and power dissipation. There are signalling schemes, however, that can prevent collisions from causing electrical problems. One such method involves using single-rail bus drivers, drivers that switch bus wires to only one power rail. In such a system, two nodes driving the same bus will cause the bus to have the wired-OR of the two values being driven, but no electrical problems result. The bus arbitration protocol can handle the wired-OR condition. The XDBus [24], used in high-end SUN SPARC systems, is an example of a bus that uses single-rail drivers.

In a mesh, the fall-through time is paid at every node, making the time:

$$T_{mesh} = D(\frac{T_s}{V_{mesh}} + T_f)$$

Notice that the T_f term in T_{mesh} is multiplied by $N-1$, whereas in $T_{meerkat}$ it is not.

Let us consider some concrete values for the variables in these equations. Take V_{mesh} to be $.6$, $V_{meerkat}$ to be $.5$, T_f to be 20 nsec, T_s to be $.16$ nsec (this corresponds to an internode spacing of 5 cm). Figure 2.4 shows a plot of the latency for Meerkat and a mesh. As the communication distance increases, Meerkat's relative advantage increases. In Meerkat, an increase in latency is dominated by the propagation speed of electrical signals in a transmission line. In a mesh, the latency is dominated by the time for the data to be absorbed and retransmitted at every node.

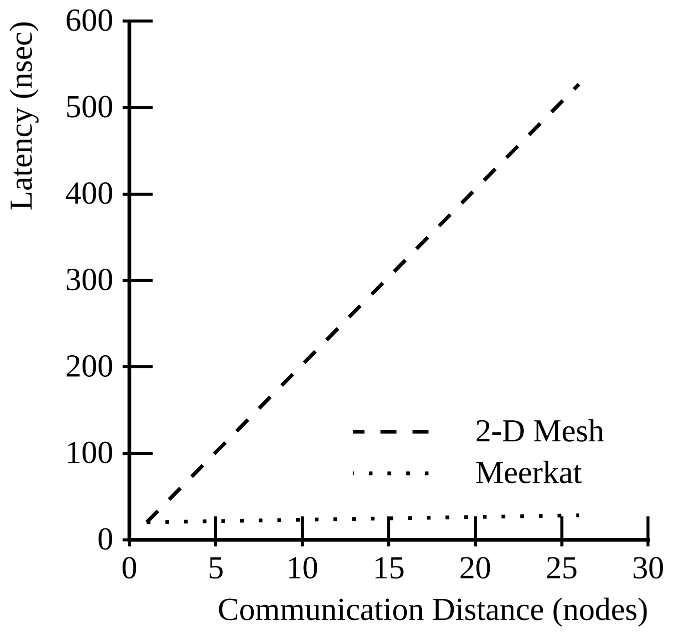


Figure 2.4: Ratio of Latency of Meerkat to 2-D Mesh

This discussion makes several assumptions. First, it assumes that the network is lightly loaded. If the network is heavily loaded, the network latency will be dominated by the queuing delay. This delay is determined by the ability of the network to handle heavy load. (Refer to section 2.3.5). Another assumption is that the arbitration can be overlapped with data transfer. If it cannot be completely overlapped, an additional delay on the order of the round-trip signal propagation time must be paid. Meerkat still will have a significantly lower latency for $D > 2$ (assuming that the bus length does not exceed about two meters).

The communication latency of both Meerkat and a 2-D mesh multi with a software-intensive network interface will be dominated by the software latency, not the network latency analyzed in this section. However, the Meerkat interconnect could be used in hardware-intensive design where there was little or no software overhead. In this case, the low network latency shown here would be a significant advantage.

2.3.7 Buses and Technology Trends

Buses have scaled with technology and have a lot of room to improve. System designers at SUN Microsystems and Silicon Graphics can put more processors on a bus than ever before (20 in the case of SUN, 36 in the case of SGI) despite the fact that the processors being connected to buses are faster than ever before. Designers are able to do this without even leaving the comfortable world of synchronous buses that reach steady state every clock cycle.

We believe that buses should be the fundamental building block of multiprocessors that cannot hide their interconnect latency. When the bandwidth of the bus is inadequate, the solution is not to use rings or meshes, but to add buses. This is what SUN Microsystems does in their Sparccenter 2000. SUN uses two buses with memory interleaved between them in 256 byte intervals. Meerkat takes this approach one step further by using a two-dimensional array of buses [2].

Tradeoffs put pressure on buses in two opposite directions: to be wide and narrow. The wider a bus, the higher its peak bandwidth. However, doubling the width does not double bandwidth. First, a wider bus will have a slightly greater clock skew, and thus will have a longer cycle. Second, if a wide bus is used to send data that takes fewer bits than the width of the bus, bandwidth is wasted. Third, a wide bus takes fewer cycles to transmit blocks of data than a narrow bus. This means that the cost of switching ownership of the bus is amortized over fewer cycles and the effective bandwidth is lower.

We believe that in systems where interconnect latency is critical, such as in a hardware cache-coherent multiprocessor, the bus should be narrow enough that its bandwidth is not wasted with short transactions, but wide enough that the receiver can proceed with what it receives in the first cycle. A bus that is too narrow, where the receiver must wait several cycles before it can continue, has the same latency problem as rings and meshes have.

[2] Meerkat does this to reduce hardware, not to reduce latency.

When interconnect latency is not critical, such as in a software-intensive multicomputer, the bus should be narrow enough to control the system cost and node size, and large enough to amortize the control circuitry and provide sufficient bandwidth.

Buses were used in the past because they required a minimum of logic to connect communicating agents. They will be used more in the future for a similar reason: they interpose a minimum of logic between communicating agents.

2.4 Meerkat's System Software

We developed Meerkat to support both message-passing and coarse-grain distributed shared memory (DSM). Chapter 5 discusses the structure and performance of a message-passing system we implemented. While we have not yet implemented DSM for Meerkat, we believe that Kai Li's success with DSM on systems with slower interconnects [59] will be repeated with even better results on systems such as Meerkat. This section discusses some of the software issues that are specific to Meerkat.

2.4.1 Software-Controlled Interconnect

Meerkat's interconnect is software-controlled. This software is responsible for choosing the route for node-to-node connections, arbitrating for the required buses, signalling, and transmitting messages. Software control allows the hardware to be simple, which, in turn, allows the hardware to be fast and quick to design. It also makes the system flexible, because software can be changed more easily than hardware.

This approach has several disadvantages, however. Software control is slower than hardware control. It imposes a higher fixed overhead for communication. Long messages can amortize this overhead, but performance on short messages suffers. In addition, the low level of control over buses precludes safe, user-mode access to the interconnect. Interacting with the network interface at such a low level makes it difficult to prevent broken or malicious software from causing system-wide problems.

Faced with these competing factors, we chose the hardware-minimalist approach. In Chapter 5 we show that Meerkat's performance on several workloads is excellent. Chapter 6 demonstrates how the performance improves when additional hardware is devoted to the network interface.

2.4.2 Buffering and Deadlock in the Interconnect

Meerkat has no buffering in the interconnect or in the node interfaces to the interconnect. Because of this, and because processors on the sending and receiving nodes rendezvous before data transmission, there is no need for hardware flow control. This allows nodes to inject data into the interconnect at high rates and is a principal source of the high internode bandwidth that Meerkat achieves.

The lack of buffering in the network also eliminates a source of potential deadlock that exists in other multi interconnects that buffer data. While these other interconnects avoid deadlocks as well, they usually pay some penalty in performance or complexity. There are no buffers in the Meerkat interconnect, making it impossible to have a circular dependency involving buffers.

Meerkat could deadlock during bus arbitration if two sending nodes each:

1. Acquire the first bus in a two-bus connection.
2. Require the same second bus.
3. Do not release the first bus.

This method would also lower interconnect utilization by preventing other nodes from using the first bus for a period of time. For both of these reasons, Meerkat's low-level arbitration software releases the first bus in a failed two-bus arbitration and tries again a short time later. The length of the back-off interval is random to prevent live-lock. This back-off method resembles that used by Ethernet [65], except that it is done in software and the Meerkat bus arbitration circuit grants ownership of a particular bus to one node at a time. In Ethernet, collisions occur when two nodes try to use the wire at the same time.

The lack of buffering in Meerkat also simplifies the fault model, lowers the cost of the interconnect implementation, lowers the latency (by removing levels of logic between the sender and the receiver), provides in-order delivery of packets, and allows the interconnect to be managed by software. The Meerkat interconnect does not require the design complexity of high-speed routing circuitry present in 2-D meshes.

2.5 Summary

This chapter presented Meerkat, a moderately scalable multi architecture using an interconnect composed of multiple passive buses. Based on research and our intuition that existing architectures were too complex to design quickly, we conceived of an architecture for which higher performance could be obtained with simpler structures.

Meerkat is at the extreme end of the design spectrum: it uses simple hardware and requires efficient, low-level software. We reinforced Meerkat's minimalist approach with a discussion of specific design choices. The most important choice was to limit scalability. Some multi customers require thousands of processors and can afford the high costs associated with them. Those for whom a few hundred nodes will suffice, however, also require the optimized price/performance achievable from systems designed for their needs.

Another tradeoff was to limit to two the dimension of the interconnect. This allowed the use of inexpensive, yet high performance, wiring technology. A third tradeoff was to make the interconnect circuit switched rather than packet switched. This allowed a simple implementation with high transfer rate. Chapter 5 shows that Meerkat performs well on several workloads.

The next chapter describes the Meerkat prototype. We used the prototype to make performance measurements, to calibrate the Meerkat simulator (described in Chapter 4), and to gauge Meerkat's design and implementation costs.

Chapter 3

THE MEERKAT HARDWARE PROTOTYPE

Meerkat's hardware prototype is notable for several reasons. First, we used the hardware design as a vehicle for testing the simplicity of various architectural ideas. By implementing Meerkat, we confronted design details that are often glossed-over in a paper design.

Second, we used the hardware prototype not only to gather performance results, but also to carefully calibrate the Meerkat simulator (described in the next chapter). Without this calibration, drawing conclusions from simulation results would have been much more difficult.

Third, we often used the prototype to debug programs that took a long time to execute on the simulator. This resulted in faster program development. Once a program ran properly on the prototype, we could then run it on the simulator to make additional measurements.

The prototype's extreme simplicity is demonstrated by the ease with which it was built. The prototype uses conventional, off-the-shelf integrated circuits. The detailed design, simulation, board layout, parts procurement, construction, debugging, host debugger development, and target monitor coding took only two person-years of effort. The prototype's simplicity reflects that of the Meerkat architecture.

The prototype does not scale beyond 16 nodes, because we made compromises to simplify prototype design and construction. However, the changes required to make a design that scales to 256 nodes are modest. In Chapter 7, we present a paper design of a system that scales to hundreds of nodes.

The next section identifies major prototype components and their interrelationships. Section 3.2 describes the physical arrangement of Meerkat's components, while Section 3.3 estimates the cost of Meerkat's interconnect. The core of each Meerkat node, its memory system, is described in Section 3.4. Internode connection arbitration and data transfer follow in Sections 3.5 and 3.6. Sections 3.7 and 3.8 describe the Meerkat node's local bus and system initialization strategy. The hardware that allows communication between Meerkat and its host is described in Section 3.9. We conclude the chapter with practical tips for others who design, build, and debug prototypes.

3.1 Introduction

The Meerkat prototype consists of four nodes connected in a two-by-two square. This square is connected to a Sparcstation II host through a host-Meerkat interface. Figure 3.1 shows the relationship between the nodes, the host-Meerkat interface, and the host system. The host runs a cross-debugger, `mg88`, that allows the user to control the prototype. A special device driver running on the host mediates between `mg88` and the parallel interface that connects to the prototype. The host-Meerkat interface accepts commands from the parallel interface and controls the square of nodes. It also generates the clock signals that are distributed to the nodes.

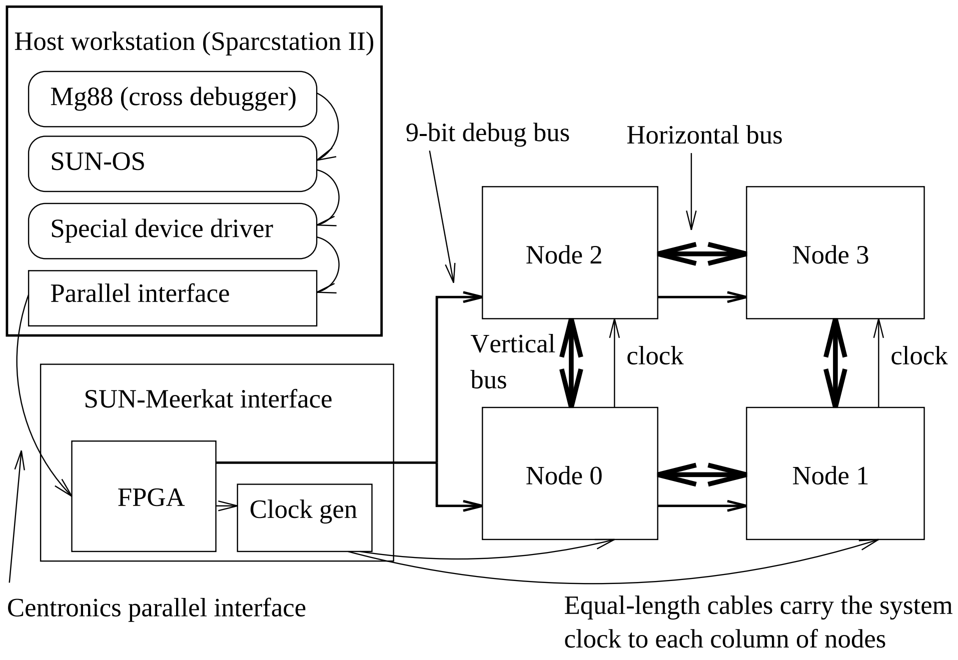


Figure 3.1: Meerkat Component Interrelationships

Figure 3.2 shows the main data and address paths of a Meerkat node [1]. The lower two rows of boxes represent the memory system and the internode data path. Two of the wide boxes in the middle are Field Programmable Gate Arrays (FPGAs) [95]. The FPGAs:

- Control internode arbitration and signalling.

[1] Our node design started as the modified core of a Tektronix workstation [90]. However, the design changed so much that nothing remains from the Tektronix core.

- Allow two-way communication between the nine-bit debug bus and the node processors.
- Control interrupts and allow processors to examine interrupt causes.
- Provide a 32-bit, 200-nanosecond cycle counter that programs use to measure execution time.
- Provide the S-Bus with a time-out mechanism, as is required by the S-Bus specification.
- Play a key role during system initialization: they load a bootstrap program into DRAM.
- Implement part of the memory refresh logic.

The wide boxes labeled "S-Bus slot 0" and "S-Bus slot 1" are connectors that allow connection of I/O adapters. The top box in Figure 3.2 shows the four Motorola MC88100 processors [63] and eight MC88200 Cache and Memory Management Units (CMMUs) that support them. Each CMMU contains 16 KB of cache. Four of the CMMUs are for code, and four are for data. Thus, each processor has one code CMMU and one data CMMU.

We designed the system to run at 25 MHz and verified that a single node would run at this speed. However, we were not able to obtain enough 25 MHz processor modules, so we clock the prototype at 20 MHz.

We designed the system to support up to four nodes per internode bus (a maximum of sixteen nodes per system). However, to fit all of the required logic in the FPGA that controls internode buses, we stripped down the logic to the minimum necessary to support our four-node system.

3.2 Physical Layout of a Meerkat Node

Figure 3.3 shows the physical arrangement of the node board. The node board measures 6.8 by 16 inches and contains four signal and two power planes. The processors and cache are on a Motorola Hypermodule daughter board [67]. The Hypermodule connects to the node board via three, 100-pin surface mounted connectors. The 32 MB of DRAM is in the form of eight, 4Mx9 SIMMs that plug into connectors at one end of the node board. At the other end of the node board are two S-Bus connectors, which allow I/O devices to be connected to each node.

Nodes plug together side-by-side to form horizontal buses and on top of each other to form vertical buses. Horizontal bus wires traverse the width of each node board to connect

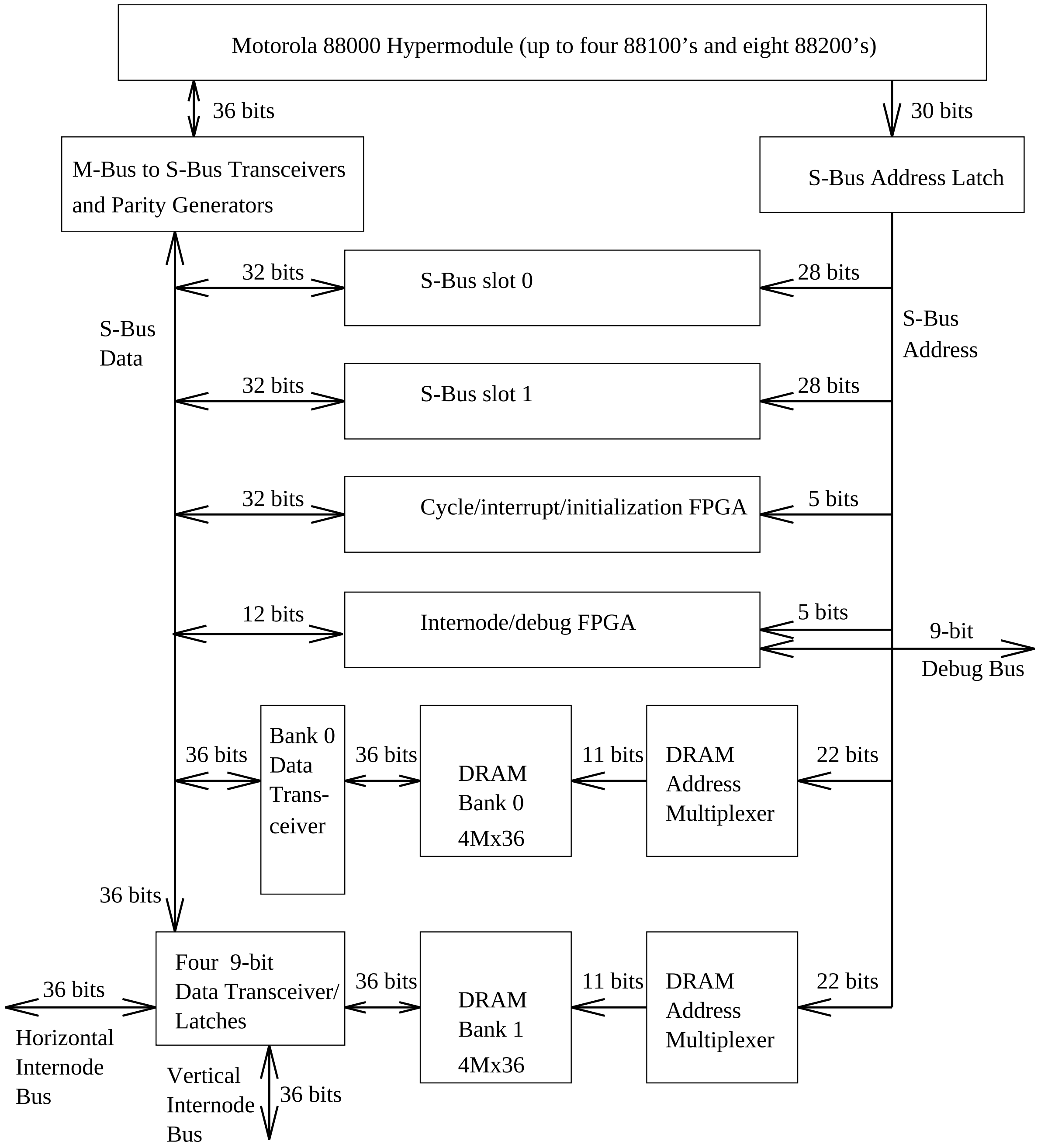
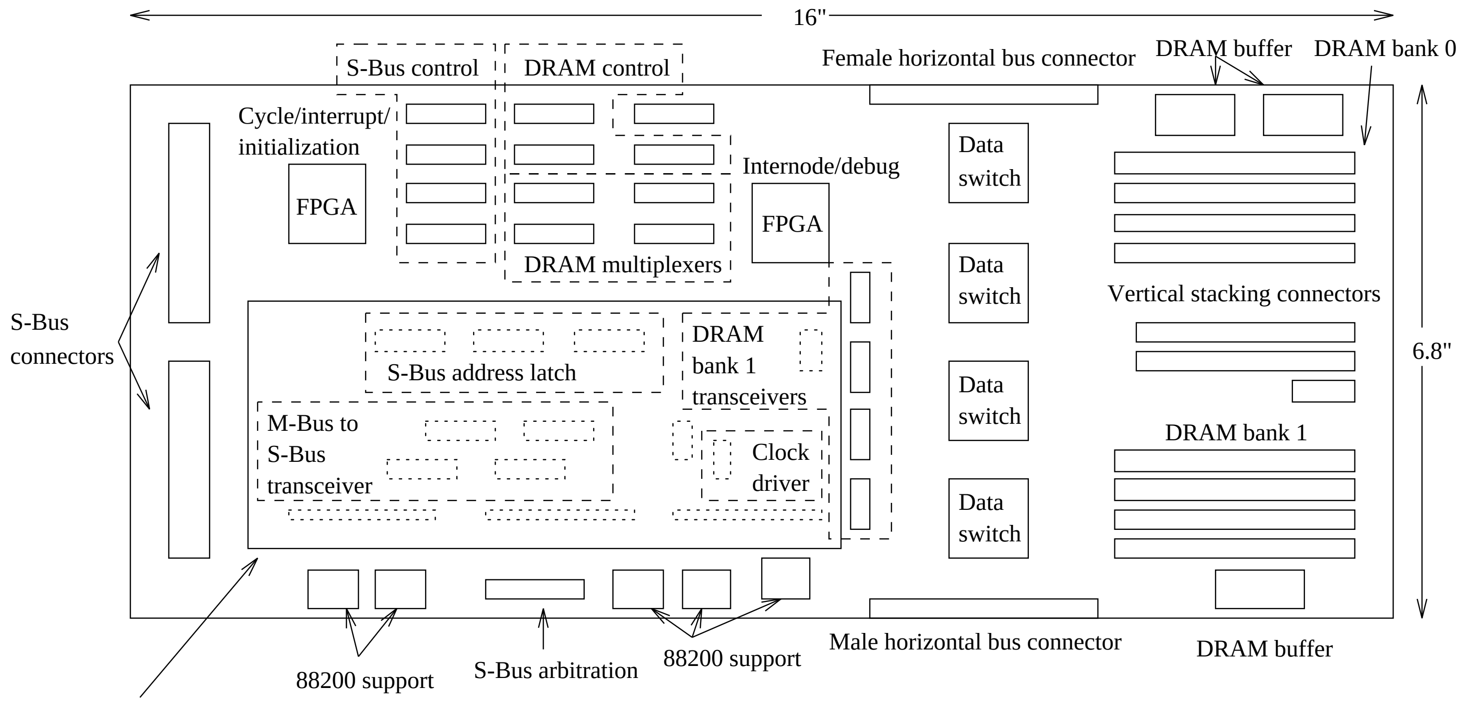


Figure 3.2: Main Data and Address Paths of a Meerkat Node



Motorola 88000 Hypermodule mounted above node board. The three, 100-pin connectors that connect the Hypermodule to the node board are shown as dotted rectangles. Chips mounted on the node board underneath the Hypermodule are also dotted.

Figure 3.3: Board Layout of a Meerkat Node

the male and female horizontal bus connectors. These wires are tapped by the node's internode bus data switch. The vertical bus is carried by stacking connectors that span the three cm spacing between vertically adjacent nodes.

Meerkat uses Motorola Hypermodules instead of separate processor and cache chips for several reasons. First, we avoided the considerable wiring problem of connecting the processors to the cache. Second, the Hypermodule's advanced packaging is denser than a standard Pin Grid Array (PGA). Third, we obtained yet higher density by putting chips underneath the Hypermodule. Fourth, the Hypermodule comes in a variety of configurations that trade off processors with cache. This gave us flexibility in experimenting with various configurations. Since we received only four-processor Hypermodules, that is the hardware configuration we used. Finally, insertion and removal is much easier due to the low pin forces on the Hypermodule connectors than it is with the high pin count and forces of PGA packages.

3.3 Meerkat Prototype's Interconnect Cost

It is common to compare gate counts, DRAM bits, SRAM bits, and silicon area in order to evaluate competing designs. Given the simplicity of our approach, the cost of connectors and circuit board area are significant. The current cost for these items and the requisite integrated circuits is about $90 per node. Compared to the total cost of a node, which is $7,500, this figure is insignificant. The overhead of the interconnect and distributed shared memory hardware in DASH [58] is about 20 percent of the system. While that amount is reasonable for a machine of its class, it represents a substantially higher system cost than Meerkat's two percent overhead.

Each node has components common to any system, such as processors, cache, main memory, and components used for internode communication. The main contributors to the cost of Meerkat's internode mechanism are: (1) the four, nine-bit-wide bus transceivers that form the internode bus data switch, (2) the four connectors per node that mate with the node's neighbors, and (3) the board area that these parts require. The internode control logic requires a few hundred gates. In our implementation the logic is in an FPGA, but any commercial implementation would fold these gates into an existing gate array. Hence, the incremental cost of the Meerkat interconnect would be low.

3.4 Meerkat's Memory System

Meerkat's memory system is the most complex part of the prototype's design. The MC88100 processors are each connected to a pair of MC88200 CMMUS [64]) that cache data and translate addresses. Each MC88200 within a node is also connected to a single MBus. The MBus is the CMMU's connection to memory and to other CMMUs for snooping. The MBus is translated into the S-Bus, which, in turn is connected to memory.

The memory system is 36-bits wide (32 for data, four for parity) to match CMMU requirements and is two-way interleaved for high bandwidth. Each bank consists of four, 4Mx9 Single Inline Memory Modules (SIMMs), for a total of 16 MB per bank and 32 MB per node. The banks are interleaved, so that every other word is stored in a given bank. One word is transferred to or from memory per clock tick for sequentially accessed memory locations. Each bank of DRAM has its own control, address, and data paths [66].

3.4.1 DRAM Control and Interleaving

DRAMs require special control signals and are often connected to specialized chips that generate these signals [83]. However, we were unable to find a controller chip that could give us the performance we wanted. Therefore, we designed our own controller using Programmable Array Logic (PALs) [1]. These devices sense S-Bus signals and generate the Row Address Strobe (RAS), Column Address Strobe (CAS), and write signals for the two banks of DRAM.

CMMUs read or write one or four words per bus transaction. Reads require the CMMUs to stall at least two cycles, while writes usually proceed with no stalling. Read stalls delay the CMMUs until the address reaches the DRAMs, the DRAMs return data, and the data are passed back to the CMMUs. Writes do not usually stall the CMMUs, because the address and data can flow out to the DRAM at the same time: the CMMUs do not wait for the data to flow through the memory system before they continue processing. However, if a CMMU gains mastery of the MBus and S-Bus immediately after a write, it will stall waiting for the DRAM system to finish the write cycle. Given the MC88200 and DRAM timing, our zero-cycle write stall and two-cycle read stall produce the fastest possible memory system running at 25 MHz (our design point).

Once the first word of a DRAM access is transferred, one subsequent word is moved between memory and the bus master per clock tick. At a system clock rate of 20 MHz, this yields a local memory system bandwidth of 80 MB/sec.

3.4.2 DRAM Addressing

Addresses are fed to DRAMs in a two-stage process. In the first stage, the top half of the address is sent to the DRAM address inputs, and RAS is activated. The DRAMs latch this part of the address on the falling (active) edge of RAS. In the second stage, one clock cycle later, the bottom half of the address is sent to the DRAM address inputs, and CAS is activated [2].

Address multiplexers take a full memory address and output either the top or bottom half. We considered using a commercial address multiplexer chip. However, we found that it was more efficient to use PALs. We did not sacrifice performance by using programmable logic, and we were able to fold into the PALs other circuitry that manipulates addresses before they are fed to the DRAMs. In addition, our PAL-based address multiplexer can handle both one- and four-megabit DRAMs.

The memory system can access up to 1024 sequential words of DRAM. Cache-line access requires accessing four sequential words, while internode transfers require accessing from four to 1024 words. An address counter generates the sequential address. In some systems, the address counter is part of a DMA controller; in others, it is part of the DRAM controller. In our prototype, this counter is in the same PALs as the address multiplexer. Thus, we economized on chips and signal routing because the address multiplexers already manipulated addresses on their way to the DRAMs.

Because the address counters are part of the address multiplexers, the CMMUs cannot use the DRAM system during internode transfers. The first DRAM request a CMMU makes on behalf of a processor will cause the CMMU and the processor to stall until the transfer is finished. Internode transfers can last up to 50 microseconds and can repeat with an inter-packet delay of 10 microseconds. While the processors will not be well used during this time, the memory system will operate at much higher efficiency than it does when it serves the CMMUs. Also, only a small fraction of an application's run time will be spent waiting for the internode transfers to complete.

---

[2] DRAMs use this multiplexing scheme to reduce the DRAM package pin count. This scheme does not reduce performance, because the two parts of the address are needed at different times within the DRAM.

3.4.3 Parity Checking and Generation

The processor checks and generates parity bits on every memory and I/O access. Memory stores parity bits. For memory accesses, parity bits are moved along with the data. Stores to I/O locations ignore the generated parity bits. Loads from I/O locations cause the parity to be generated in the bus transceivers that connect the processors to the S-Bus.

3.4.4 DRAM Refresh

Each DRAM chip has 2048 rows, with 2048 bits per row. Each row must be refreshed every 16 milliseconds. This means that, on average, rows must be refreshed at a rate of one per eight microseconds. To amortize the time-cost of refresh and to simplify the circuitry, the memory system refreshes eight rows every 64 microseconds.

A counter in the host-Meerkat interface generates a 200-nanosecond pulse every 64 microseconds. The pulse is transmitted to the internode/debug FPGA on each node. When triggered by the host-Meerkat interface, this FPGA requests eight refresh cycles from the DRAM controller. The DRAM controller gives preference to refresh requests over reads and writes. The longest the DRAM system can take to complete one request is 62 microseconds (50 microseconds for a long internode transfer plus 12 microseconds waiting for a sender to rendezvous). Because refresh has highest priority, and the longest DRAM operation is less than the refresh period, refresh overrun cannot occur.

3.5 Internode Bus Arbitration and Signalling

This section describes the hardware that implements the internode arbitration and signalling mechanisms. (Refer to Section 2.2.1 for a description of these mechanisms.)

The first step in internode communication is the sending node's arbitration for a vertical or horizontal bus. A processor on the sending node writes the *internode control register* with bits requesting that: (1) the cross-bar circuitry be disabled, (2) the horizontal or vertical bus be arbitrated for, and (3) the internode command be sent once the internode bus is acquired.

If the cross-bar circuit of a node is in use, it will stay enabled despite a disable request from the node's processor. The cross-bar circuit will become disabled once the bus that it is repeating becomes inactive. The sending node's process checks that it has disabled the node's cross-bar function and secured the required internode bus.

The prototype's arbitration scheme is simple and limited to two bus masters in order to fit internode control logic in the Xilinx 3064 FPGA. Along with data, parity, and arbitration signals, each internode bus also has two function bits. The sending node drives these bits to control the receiver or a cross point. The four functions that the sender can request are:

1. *Local interrupt*: signal another node on this bus that we wish to send it data.
2. *Cross-bar request*: ask the other node on this bus to act as a cross-point switch for us, to arbitrate for its other bus on our behalf, and to repeat our control and data on its other bus.
3. *Remote interrupt*: signal another node through a cross-point connection set up by a previous cross-bar request.
4. *Return status*: the other node should drive the status signal with one of two different status bits.

The status signals on the internode buses are available for the processor's inspection in the *internode status register*. Initially, the other node will drive this signal with its rendezvous bit. After a cross-bar request, it will use its "I-acquired-the-other-bus-for-you" bit to signal whether the cross-bar request was successful. After a remote-interrupt request, the status signal is driven with a copy of the other bus' status signal, which will indicate the value of the receiver's rendezvous bit.

3.6 Internode Data Transfer

Processors initiate transmission of internode data by writing a *transfer-length register* with the number of words to move; they then load a special region of physical address space. This region is called "magic DRAM" because: (1) the semantics of load and store instructions that access this region are not intuitive, and (2) access to this region affects the contents of DRAM (see Figure 3.4). A load instruction from a given magic DRAM location initiates the transmission of multiple words starting from the corresponding DRAM location. The transfer count register determines the number of words moved. The value loaded is itself meaningless but will have correct parity so as not to signal an error.

A receiving node goes into a receptive state by issuing a store instruction, with a magic DRAM address corresponding to the DRAM address where it would like the incoming data to go. Once the DRAM is ready to receive, a ready-status indication is propagated back to the sender. The sender polls its network interface to see this status and then loads from its magic DRAM space, as described above. The transmitted data are accompanied

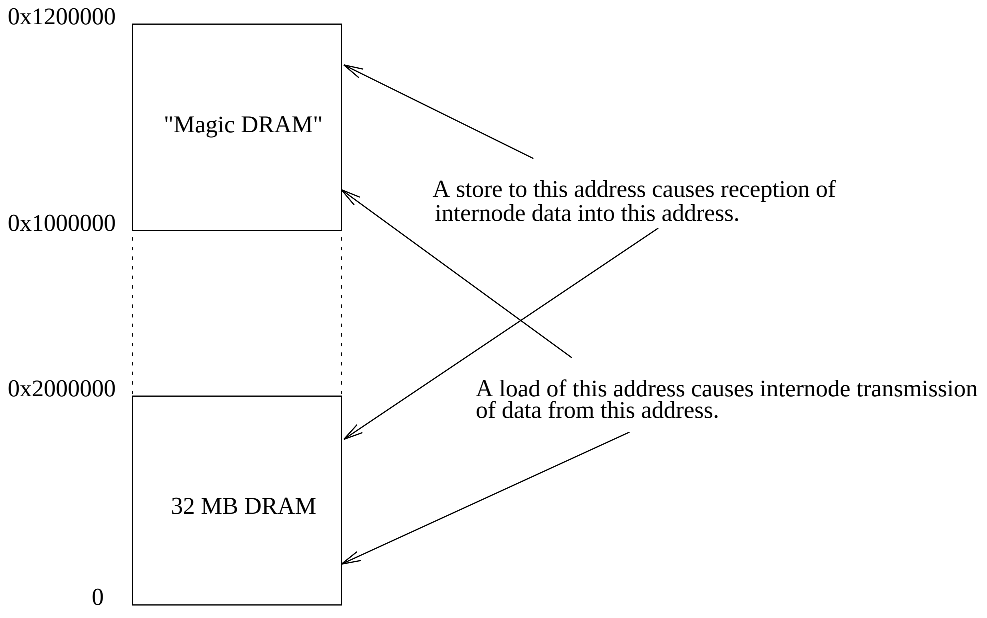

0x1200000
"Magic DRAM"
0x1000000
A store to this address causes reception of internode data into this address.
0x2000000
A load of this address causes internode transmission of data from this address.
32 MB DRAM
0

Figure 3.4: Magic DRAM Space

by a control signal that marks their beginning and end. The receiver uses this signal to synchronize its reception.

Using the processors to generate the memory addresses for internode data avoids the expense and complexity of a full-featured DMA controller. During the transfer, both the receiving and sending processors stall as soon as they try to use data from a register loaded by a load instruction that misses in the cache. Processors may stall sooner if they fill the three-stage data memory pipeline of the MC88100 with store instructions that miss the cache. This means that the processors will stall for most of the internode transfer time. Since transfers are short and there often is no useful work for the processors until the data are moved, stalling is not a drawback.

3.7 Initialization

There is no ROM in the prototype. Instead, we rely on the programmability of the FPGAs to bootstrap the system. When powered on, the DRAM contains random bits, and the FPGA pins are in high-impedance mode. By virtue of the FPGA's state, the processors are kept in a reset state. The host system, through the host-Meerkat interface, loads the two FPGAs on each node.

One of these FPGAs, the initialization-interrupt-cycle FPGA, is responsible for a number of functions. During initialization, it is first loaded with special logic that serves a key bootstrap function. This logic is encoded with a 13-word MC88100 bootstrap program along with enough extra logic to get control of the S-Bus and write this program into the first 13 words of DRAM. Once DRAM has been initialized, the FPGA is reloaded with the interrupt-cycle logic to serve the running Meerkat system as a cycle counter and interrupt controller.

After the initialization-interrupt-cycle FPGA is reloaded, the host signals the internode-debug FPGA to release the processors from the reset state. The processors then execute the bootstrap program. This small program loads DRAM with several kilobytes sent to it by the host through the debug channel. These kilobytes are the debug monitor, which serves as the cross-debugger's agent on each node.

Avoiding the use of ROM has two primary advantages. First, we avoid the expense, design complexity, and board area ROM requires. Second, there are no parts to replace every time a debug monitor bug is fixed or upgraded.

The disadvantages of no ROM include the time and effort spent to engineer the initialization mechanism and download the debug monitor each time the system is initialized. However, the initialization mechanism required less engineering effort than ROMs would have, and the debug monitor takes a fraction of a second to download.

3.8 S-Bus Interface

Each node provides two S-Bus connectors to allow connection of I/O devices. S-Bus also connects the processors, memory, and I/O devices within each node. We used S-Bus because of its simplicity, the availability of S-Bus devices, and its excellent performance.

The Meerkat S-Bus controller is implemented in four PALs:

1. One PAL arbitrates among the eleven possible bus masters: the eight CMMUs, the two S-Bus slots, and the initialization-interrupt-cycle FPGA.
2. A second PAL generates the three, S-Bus size signals and the bottom two address bits from the CMMU control lines.
3. A third PAL generates the S-Bus read and address strobe signals in addition to controlling various bus transceivers.
4. A fourth PAL generates bus grant lines for the two S-Bus slots, the cycle-interrupt-initialization FPGA, and various control signals.

3.9 Host-Meerkat Communication

The Sparcstation host is connected to the host-Meerkat interface via a Centronics parallel port in the Sparc with an attached parallel cable [71]. The host-Meerkat interface uses a Xilinx 3090 FPGA to interpret the Centronics signals and to control the Meerkat array. A special device driver in the Sparcstation controls the parallel port in response to commands from `mg88`, the Meerkat cross-debugger. We redefined the semantics of several of the Centronics signals for our own purposes. Therefore, it is only a Centronics interface at the lowest electrical level. At the protocol level, it is specific to Meerkat.

A nine-bit debug bus is controlled by the host-Meerkat interface FPGA. Each node taps this bus and can access it through the use of a pair of eight-bit registers. One register is written by the host and read by the node processor; the other is written by the node processor and read by the host. The host-Meerkat interface drivers the ninth bit on the debug to indicate whether a select byte or a data byte is present on the bus. A pair of full/empty status bits, visible to both the node processor and the host, support handshaking on both ends of the host-Meerkat communication.

The host-Meerkat interface is passive. It can only respond to requests; it cannot interrupt the host. The host sees the interface as a small number of registers that it can read and write. The primitives that access these registers include:

- Reading any node's debug transmit data register
- Writing any or all node's debug receive data register
- Turning the debug bus clock on or off or sending a single pulse on the debug bus clock line
- Sending a single pulse on the done/reprogram or reset lines of the node FPGA's
- Turning on or off the power-stable signal that feeds the node processor
- Turning on and off the node power
- Resetting the host-Meerkat error bit

A system status register in the interface allows the host to check the following:

- Whether the node power is on or off
- Whether the debug clock is running
- The state of the power-stable line that is generated by the host-Meerkat interface and which controls logic on the nodes

- The done/reprogram signal from each of the two FPGAs on each node
- The host-Meerkat interface error bit

To send a sequence of bytes to the nodes, the Meerkat device driver on the Sparcstation host writes a select register in the host-Meerkat interface once and then writes the data register once per byte transmitted. Overrun is possible if the node does not empty its receive data register quickly enough. To avoid this, the device driver can check for the receive-data-register empty condition before sending each byte. However, this would cut the transmission rate considerably. The compromise we made was to have the driver check for receive-empty status on only the first few bytes in a sequence. This assumes that, if the receiving node is ready for the first few bytes, it will continue to receive all the bytes in a sequence. This is a valid assumption because: (1) the program performing the reception is always the Meerkat debug monitor, and (2) interrupts are off during its execution. We achieve a host-to-Meerkat transfer rate of about 80,000 bytes per second.

The host has the option of sending data to one node or all nodes at once. The broadcast option is used to download the debug monitor and program text, data, and bss to all the nodes in parallel. This ability is frequently used and worth the modest additional implementation effort.

To receive a sequence of bytes from a node, the Meerkat device driver polls the node's transmit-buffer-full status bit until it indicates the presence of a byte. The driver then reads the byte and returns to the polling loop. We cannot make an assumption about the host's availability to read bytes from the target as we did in the host-to- node case. Therefore we must check status on both ends of the communication for each byte sent. We achieve a Meerkat-to-host transfer rate of about 17,000 bytes per second.

We found the host-target communication performance to be barely adequate. It takes about three seconds to completely reset the nodes and download the target debug monitor to each of them. It takes about six seconds to download the Mach microkernel to all of the nodes.

To alert the host of some event or request a service, a node sends a packet through the debug channel. Because the nodes are passive to the host-Meerkat interface, and the interface is passive to the host, the host must poll for events or requests from the nodes. Despite the fact that we poll each node 20 times a second, the latency is still a problem in some situations. Also, having to use the SUN-OS process timer to generate 20 SIGALRM signals per second exposed bugs in SUN-OS and unfortunate interactions with other system services. In retrospect, a more sophisticated interface that allowed the nodes to signal the cross debugger through the device driver would have required less effort and performed better.

3.10 Lessons Learned from the Meerkat Prototype

We entered schematics and ran the simulation on a Apple Macintosh FX running Capilano's DesignWorks. Bugs in this tool severely hampered our productivity and prevented us from simulating significant parts of the design. For example, our behavioral model of the data switch chips (refer to Figure 3.2) caused the simulator to crash. These parts are surface-mounted and practically impossible to remove once they are soldered onto the circuit board. There were two minor errors in the polarity of the control signals to these chips. The straightforward way to fix the problems was to change traces underneath the chips, but these could not be moved. Therefore, the solution was to engineer clock signals that took account of the connections that could not be changed.

We designed the circuit board using Racal-Redac's low-end CADSTAR layout program running on an IBM PS/2. CADSTAR's autorouter lacked the ability to solve our layout problem: it could only use two layers at a time and used simple horizontal and vertical wires. We manually routed all 1300 signals on four layers. The tool had no bugs that we encountered, but the user interface and lack of requisite features made progress slow.

Debugging was done with a four-channel, 500 MHz, digital Tektronix oscilloscope, the TD 540, and a low-end Tektronix logic analyzer, the 1230. The 1230's tedious user interface and limited sampling ability made logic analysis slow.

The following list distills lessons learned on Meerkat into practical tips for designing, building, and debugging future prototypes:

- Simulate *all* of the logic in a design. If something cannot be simulated, seriously consider changing the design. We were able to simulate Meerkat's PALs, and, as a result, the complex DRAM system worked with only a few patch wires.
- Use programmable logic when possible: it allows bugs to be fixed and the architecture to be adjusted without physically altering the prototype.
- An ECB autorouter can save a lot of time and effort. Try to find one that can route your whole design.
- Pay close attention to parts limitations (e.g., delay lines that require the duty cycle to be at most 25%, or FPGAs with large delays). These limitations are easily glossed over when the design is on paper only and, if ignored, will lead to major problems in making the prototype work.

- Perform careful analysis to determine which wires will behave as transmission lines. Properly terminate these wires to prevent ringing.
- A high-speed storage oscilloscope and logic analyzer are essential to any similar project. These tools allow diagnosis of system failures due to hardware design errors, software bugs, chip failures, and construction problems.
- Surface mount technology (SMT) offers two key advantages: (1) shorter device lead frames have lower inductance, and (2) SMT consumes less board area, making intra-node wires shorter with fewer behaving as transmission lines. Do not be discouraged from SMT by warnings about soldering difficulties. However, the lead pitch of our SMT devices was 50 mils, and finer pitch devices may prove difficult to solder.

3.11 Summary

The hardware prototype performed several important functions in evaluating the Meerkat architecture:

- Building the hardware brought us in close contact with design issues that are easy to miss on paper.
- It acted as a timing and semantic reference model that calibrated the Meerkat simulator.
- It gave us a fast execution vehicle for debugging long-running programs. This shortened the edit-execute-debug-fix program development cycle.
- It gave us a measure of the Meerkat architecture's implementation complexity.
- It helps convince others that our approach is realistic.

After processors and cache, the most expensive component of most multicomputers is its memory system. In Meerkat the memory system's performance affects the internode transfer rate and processor performance. We made Meerkat's memory system as fast as possible given the use of commodity DRAMs.

In keeping with the Meerkat architecture's lean approach, we economized in the prototype in several ways:

- We achieved economy of mechanism by building part of the internode transfer sequencing logic into the same programmable logic devices that serve as the DRAM address path.
- Meerkat uses S-Bus to connect processor and memory within each node and between nodes and external I/O devices. S-Bus is a low-cost, high-performance method for connecting computing components.
- The initialization strategy avoids ROMs. Instead, the cross-debugger controls an intricate initialization process that uses the FPGAs to initialize the DRAM.

- The connection between the Sparcstation host and Meerkat uses a standard, low-cost parallel interface. The connection allows the cross-debugger to rapidly download the target and control its operation.
- The cross-debugger is based on `gdb`, a powerful symbolic debugger. Using `gdb`, instead of developing a new debugger saved a significant amount of effort.
- We used PALs for circuits that had to be faster than those buildable with FPGAs. Similarly, we used FPGAs only for circuits that had to be faster than what can be done in software.

As a result of the prototype, we achieved a measurable system, one capable of demonstrating the value of Meerkat's design approach.

Chapter 4

THE MEERKAT SIMULATOR

This chapter describes the Meerkat system architecture simulator, the main tool for producing performance results and understanding architectural tradeoffs. Understanding the simulator's structure makes the performance data more comprehensible and credible. However, the simulator is also interesting in its own right.

Unlike other simulators used for architectural evaluation, the Meerkat simulator models the fine detail of hundreds of nodes running significant programs. It is unusually fast for a simulator that models fine detail: it simulates about 800,000 processor cycles per second on a Sparc 10 host. It achieves this speed without resorting to direct execution, and thus without constraining the target architecture to match the host's. In addition, it models the semantics of virtual address translation and processor supervisor and user modes. No other simulator that we are aware of has this combination of features.

We achieved simulation efficiency by starting with a fast, threaded-code simulator and adding only those timing models needed to achieve accuracy. This approach resembles that used to gain semantic accuracy in a previous simulator [9]. To measure timing accuracy, we ran a suite of benchmarks on the simulator and the Meerkat prototype. Test results guided our grafting of timing models onto the threaded-code simulator base.

We introduce the roles of simulation in Section 4.1. Benefits of system architecture simulation are discussed in Section 4.2, while Section 4.3 describes some of the varied approaches to simulation. Section 4.4 relates the structure of Meerkat's simulator, the performance of which is analyzed in Section 4.5. The development of timing models and simulator timing accuracy are discussed in Section 4.6. Section 4.7 concludes the chapter with a description of the simulator's execution profiling feature.

4.1 Introduction

Computer architecture simulators vary widely in their application. They are used by processor architects to evaluate uniprocessor design tradeoffs [26], operating system authors to debug their code [9] and to evaluate operating system performance [23, 82], parallel system architects to assess the performance of large systems [18, 75], and by end users to execute programs written for one system on a different host system [62, 86, 87]. We used the Meerkat simulator to debug and tune operating system code and to evaluate Meerkat's performance.

Simulators also vary in their performance and the level of detail they can model. A common metric is the *slow-down*, or the average number of simulator host instructions executed per simulated instruction. In general, the more detail that the simulator captures, the greater its slow-down. Slow but accurate simulators have the advantage of capturing subtleties of the target system. However, their slow speed limits the size of the system they can model and the number of simulated instructions they can execute. The simulator designer must choose a level of simulation detail that is fine enough to capture important performance artifacts, yet fast enough to model large systems and long-running applications in an acceptable timeframe.

4.2 Benefits of Architecture Simulation

There are several benefits of using a simulator for evaluating multicomputer architectures:

- Simulators can be augmented relatively easily with new measurement and debugging features.
- Large simulators do not cost millions of dollars, unlike large hardware implementations.
- New network interfaces can be added in a few days. It is often impractical to retrofit hardware with new interfaces. We can also model networks that are impossible to build, e.g., an infinitely fast network.
- Execution is deterministic on a simulator. This makes program bugs repeatable, which is not always the case for hardware implementations.
- Simulators are not subject to prototype failures.
- Simulators can be given to other researchers. While hardware is difficult to transport, a simulator can be sent through the Internet.
- As many simulators can be running as there are hosts and host memory. This means that multiple experiments can be run concurrently on a simulator. Hardware prototypes are usually few in number; often, just one exists.

There are other benefits of simulators, such as the ability to: (1) non-intrusively generate address traces of user and system code, and (2) stress-test operating system software by causing the most serious and complex interrupt and exception conditions.

4.3 Simulation Strategies

The best simulation method depends on the application of the simulation results. This section outlines several simulation strategies and their applications.

4.3.1 Microarchitecture Simulation

Microarchitecture simulators are built by logic designers to express and test their designs. They can also execute short sequences of code, enabling designers to evaluate architectural features and debug microcode. Microarchitecture simulators typically have a slow-down of 20,000 to one [73]. While useful, they are too slow for debugging all but the shortest sequences of code.

4.3.2 Macroarchitecture Simulation

Macroarchitecture simulators (also called *macro simulators* or *instruction-set-architecture simulators*) can execute longer-running programs. They are used for studying cache performance and debugging operating system code in advance of a chip's availability. Unlike micro simulators, macro simulators often model the chip's timing closely, but not perfectly.

Because they do not model systems as closely, they are much smaller and faster than micro simulators. Conventional macro simulators have slow-downs on the order of 100 to 1000. They dispatch instructions by fetching from a simulated memory, isolating the operation code fields, and branching based on the values of these fields. Once dispatched, the instruction's semantics are simulated by reading and manipulating simulation variables that represent the target system's state.

Several techniques can improve the performance of macro simulators. Instead of decoding the operation fields each time an instruction is executed, we can translate the instruction once into a form that is faster to execute. This idea has been used in a variety of simulators for a number of applications [26, 32, 61, 62, 86]. It is also used in some processors to translate an instruction set that programmers see into a more RISC-like form that is more efficient to execute [25, 34].

4.3.3 Direct Execution

A popular approach to efficient architecture simulation is to execute the target program directly on the host [19]. The target program is encased in an environment that makes it execute as though it were on the simulated system. This requires that either the host system have the same instruction set as the target or that the program be recompiled. Instructions that cannot execute directly on the host are replaced with procedure calls to simulator code. If every instruction were rewritten, the simulator would resemble Shade [26], which translates every target instruction to a sequence of host instructions.

Direct execution simulators are usually capable of executing only user-space code. To evaluate tradeoffs in multicomputer architecture, we needed to simulate both user and kernel code. It is possible to use direct execution techniques to simulate kernel code. However, every memory access instruction would require a call to a simulator routine that modelled virtual address translation and the data cache. The Meerkat simulator uses this routine, which consumes half of the host's execution time when running our simulator. Thus, the upper bound on the performance improvement anticipated from using direct execution would be a factor of two.

4.3.4 Blurred Lines Between Simulation Techniques

Several tools can be considered fast macro simulators that dynamically translate code, or direct execution simulators, or profiling tools. The UNIX utility `prof` is a profiling tool that is not usually called a simulator. But it could be considered a direct execution simulator that wraps the target program in an environment enabling execution measurement. Shade is thought of as a fast macro simulator that uses dynamic compilation. While it is more flexible than `prof` and uses dynamic instead of static compilation, it is also a tracing tool. The line between different tracing and simulation techniques is often blurred despite efforts to neatly categorize them.

4.3.5 The Meerkat Approach: Threaded-Code

Measuring Meerkat's design required a simulator efficient enough to run significant programs on hundreds of simulated processors. In addition, it had to model timing accurately. We could not afford to spend years constructing a complex simulator or waiting for results from a slow one.

For these reasons, we wrote a simulator that translates instructions to threaded code [11, 56], which is then executed. The threaded code is cached, so that the price of translation for most instructions is paid just once, the first time they are encountered in the code stream. The result is a simulator that has a slow-down of about 100 per simulated processor. Its timing is close enough to the prototype's that we can use it to run large programs and make meaningful measurements.

4.4 Structure of the Meerkat Simulator

Figure 4.1 shows the structure of the Meerkat simulator. Users interact with the simulator through a symbolic debugger, called gdb [88]. The simulator consists of an instruction translator, a threaded-code interpreter [11], cache models, a TLB model, a physical memory system model, and I/O models.

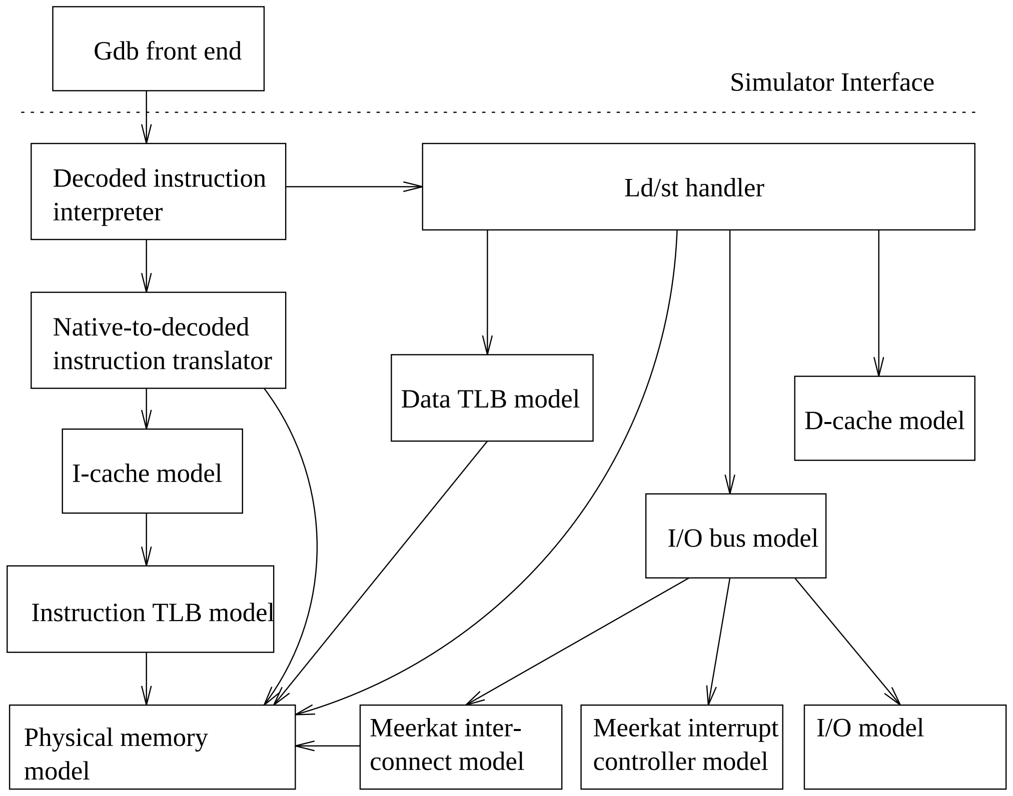


Figure 4.1: Meerkat Simulator Structure

Meerkat programs are compiled, assembled, and linked with a set of GNU cross-development tools on a Sparcstation host. The simulator, called `mg88` (a contraction of "Meerkat", "gdb", and "88000"), is run on the resulting executable image. Mg88 loads the code, data, and symbol table. It then calls through the simulator interface to: (1) place the code and data in simulated memory, (2) set up profiling structures, and (3) initialize registers.

When the user issues the `run` command, the front end calls the threaded code interpreter. Execution returns to the front end upon an exceptional condition, such as a breakpoint or a special trap instruction indicating that the simulated program has finished.

4.4.1 Translation to Threaded Code

The simulator does not interpret MC88100 instructions directly. Instead, instructions are first translated to *decoded instructions*, which are cached in structures called *decoded instruction pages*. Only instructions encountered during execution are translated and cached, so unlike Mimic [62], there is little startup overhead.

Decoded instructions contain up to six fields. The first is always a pointer to the instruction's *handler*, the code that interprets the instruction. This makes it easy to dispatch decoded instructions. On most simulator hosts, this dispatch consists of two instructions: a load followed by an indirect jump. For triadic instructions, three of the decoded instruction fields are pointers to host memory that models the MC88100 registers. The last two fields hold the type and length of memory access instructions.

Figure 4.2 shows an unsigned add instruction followed by a load instruction. The add instruction sums the contents of r5 and r6 and stores the result in r4. The load calculates the effective address as the sum of r4 and 1000. It loads a word from this address and puts it in r2.

4.4.2 Instruction Pointers and Branching Instructions

There are two instruction pointers: the decoded instruction pointer (DECIP) and the modelled instruction pointer (IP). The IP is the value of the MC88100 virtual address of the next instruction to execute. After non-branching instructions, the IP and the DECIP are incremented.

Branch instructions whose target is within the decoded instruction page are translated to a decoded form that includes a pointer to the target of the branch (i.e., a pointer to another decoded instruction). Off-page branches must be resolved during execution, because the instruction translator makes no assumptions about the contents or allocation of other decoded instruction pages. In addition, the decoded instruction pages correspond to physical pages; DECIP points to instructions that correspond to physical addresses. The IP points to virtual addresses. The correspondence between virtual MC88100 instruction addresses and decoded instruction slots is dynamic and can change after threaded code is generated.

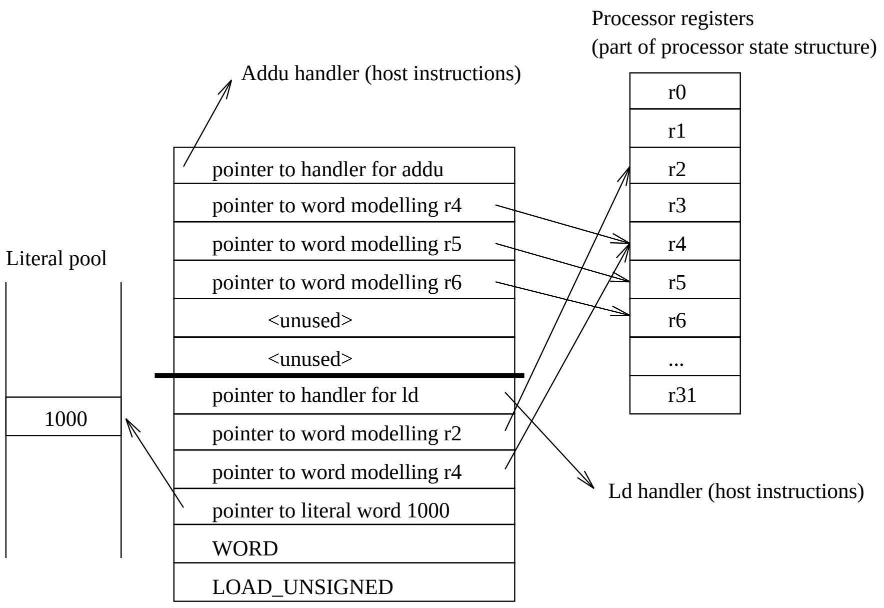

Processor registers
(part of processor state structure)
Addu handler (host instructions)
r0
r1
r2
r3
r4
r5
r6
...
r31
pointer to handler for addu
pointer to word modelling r4
pointer to word modelling r5
pointer to word modelling r6
<unused>
<unused>
pointer to handler for ld
pointer to word modelling r2
pointer to word modelling r4
pointer to literal word 1000
WORD
LOAD_UNSIGNED
Literal pool
1000
Ld handler (host instructions)

Figure 4.2: Decoded Instructions for `addu r4,r5,r6` and `ld r2,r4,1000`

Delayed branches increment both the DECIP and the IP, just as non-branching instructions do. In addition, they set a flag if the delayed branch is taken (i.e., the branch condition is true). Non-branching instruction handlers check this flag, which, when true, indicates that the previous instruction was a taken delayed branch. In this case control is passed to the target of the delayed branch after the non-branching handler finishes its semantic actions. Branching instructions need not verify that they are in a branch delay slot, because the MC88100 specification prohibits this from occurring.

Jump instructions transfer control to locations whose targets come from a register and are thus not known until execution. For this reason, a jump is treated like an off-page branch. Its execution requires translating a virtual MC88100 code address into a DECIP. A small cache of translations from target virtual instruction addresses to DECIPs speeds the interpretation of these transfer control instructions.

4.4.3 Decoded Instruction Pages

Decoded instruction pages contain slots that hold decoded instructions. They correspond to a physical page of a particular node's memory. Decoded instruction pages are allocated when a running program attempts to execute code on a physical page that does not yet have a corresponding decoded page. Because both physical page structures and decoded instruction page structures are allocated lazily, it is possible for neither to exist for a particular physical page in the modelled system. It is also possible that a physical page structure, but not a decoded instruction page, is allocated for a particular physical page.

When a decoded instruction page is first allocated, and when it is flushed, it is filled with the *decode-me* pseudo instructions. When a decode-me pseudo instruction executes, an MC88100 instruction is translated to a decoded instruction. The MC88100 instruction is fetched from the address on the physical page that corresponds to the position of the decode-me pseudo instruction in the decoded instruction page. For example, if the decode-me pseudo instruction in the tenth slot is executed, the tenth word in the corresponding physical page is translated. The new decoded instruction replaces the pseudo instruction, and this new instruction is executed.

There are 1025 decoded instruction slots in each decoded instruction page. The first 1024 of these hold decoded instructions that can be in a physical page (a physical page is four kilobytes, and each MC88100 instruction takes four bytes). The 1025th slot holds a sentinel called the *requalify-decoded-ip* pseudo instruction. When a non-branching decoded instruction in the 1024th slot is finished executing, the DECIP is incremented and points to the requalify-decoded-ip pseudo instruction. Because the flow of control has moved off of the page, the DECIP must be recomputed to point to the first decoded instruction of the new page. The requalify-decoded-ip speeds instruction handlers, because they never need to check for the end-of-page condition.

Dividing physical and decoded instruction space into pages allows incremental, demand-driven allocation of memory. This conserves host virtual memory (a point discussed in the next section).

`Mg88` does no garbage collection. Avoiding garbage collection kept the simulator simpler at the expense of higher virtual memory consumption. Since the units of allocation are large (4k for a physical page structure, 20k for a decoded instruction page), garbage collection would probably be redundant with the host's virtual memory system. That is, garbage collection would reclaim memory that would be identified as pages that are not recently used, and thus candidates for paging out to secondary storage. Thus, garbage collection would conserve host virtual memory, but not physical memory, at the expense of higher simulator complexity.

4.4.4 Processor State

Figure 4.3 shows the relationship among processor state, simulated physical memory, and the decoded instruction pages. Upon simulation start-up, the user specifies the number of nodes, number of processors per node, and size of node memory. The simulator allocates an array of processor state structures to match the user's request. The processor state holds pointers to the physical memory map and the decoded instruction map for the node. All of a given node's processor state structures point to the same maps: a node's memory is shared by all of its processors. The state structure also contains all of the processor's registers, in addition to various counters to generate execution statistics, such as the cache hit rate.

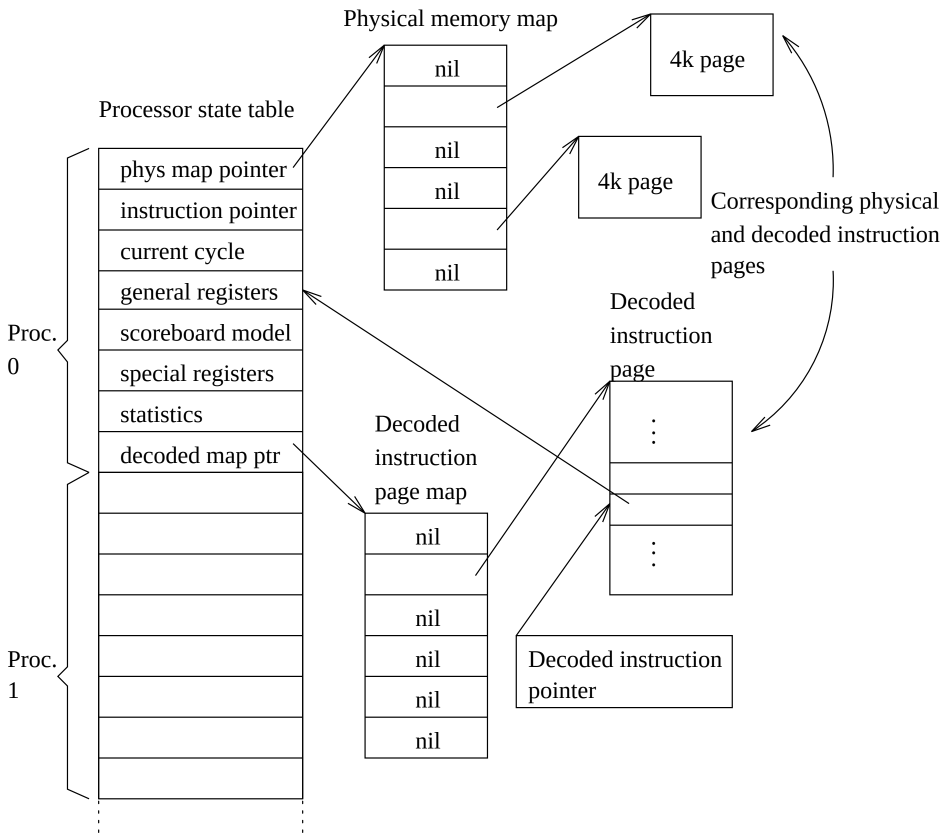


Figure 4.3: Simulator Data Structures

In Figure 4.3, note that the second page of physical and decoded instruction memory is allocated. This indicates that either the front end or a running program touched this page, and that an attempt has been made to execute an instruction on this page.

As simulation proceeds, the memory maps will gradually fill in as the running program executes memory access instructions and branches to new pages. Typically, however, large pieces of both maps are vacant. This is especially true of the decoded instruction map, where vacancy represents host virtual address space that we have conserved. Because these maps are replicated for each node, large simulations would require far more virtual memory than is available on our largest hosts if we did not allocate memory lazily. The startup time to initialize this memory would also be enormous. For example, an FFT (described in Section 5.9) of 32,768 points on 256 nodes, each with 1 MB of simulated physical memory, requires about 50 MB of decoded instruction memory out of a total process requirement of 260 MB. If we did not allocate on demand, the FFT would consume 1342 MB for decoded instruction memory alone.

4.4.5 Modelling Basic Instruction Execution Time

To model the number of cycles an instruction takes, each processor has an associated current cycle count. This variable is incremented by every handler to reflect the number of cycles it takes to issue the instruction.

The MC88100 has a register scoreboard. When multicycle instructions issue, they set the scoreboard bit corresponding to the instruction's destination registers. When multicycle instructions finish, their destination register scoreboard bits are reset. Issued instructions stall the processor until the operand register's scoreboard bits are clear. This mechanism ensures correct operation of instructions that use values produced by multicycle instructions.

To model the scoreboard, we use an array of *time-available* values that correspond to general registers (see Figure 4.4). Every multicycle instruction handler sets the time-available slot corresponding to its destination register to the cycle count value when the register in the hardware would be available. The handler then produces the result and stores it in the destination registers. Instructions that read registers advance the processor cycle count to the maximum of the time-available slots corresponding to each operand register and the current cycle count.

For example, if an `addu` instruction reads registers three and four and takes one cycle to issue, and if registers three and four have time-available counts of 105 and 107 respectively, and if the cycle count before the `addu` issue is 100, then the processor cycle count will be set to 107 by the `addu` handler. In this case, the one-cycle issue time and the time to wait for register three are hidden by the time to wait for register four, as in the real processor.

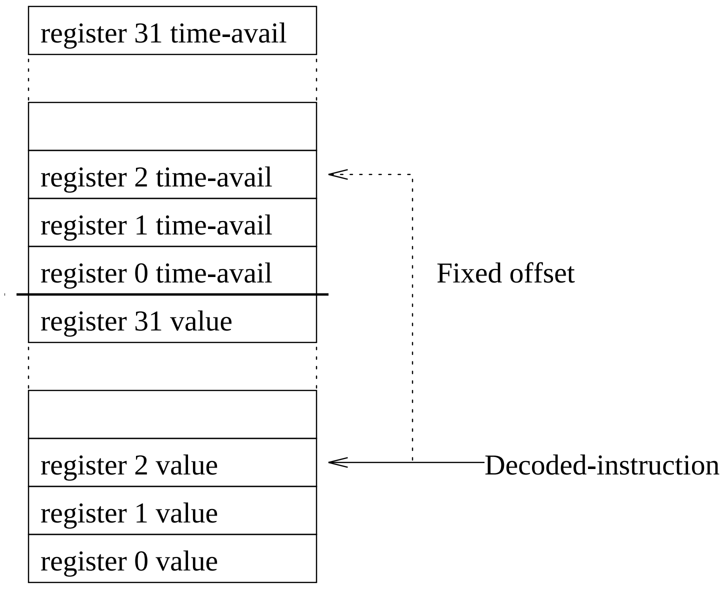


Figure 4.4: Relationship between General Registers and Time-Available Array

Our scoreboard model depends on being able to calculate the time-available for all multicycle instructions at the time the instructions issue. In our case this was possible, and we believe that it will be so for most similar systems.

Instruction handlers access the time-available slots using the same pointers they use to access the register values. Handlers perform this access by adding a constant to the register pointer. This mechanism depends on the time-available slots being the same width (32 bits) as the general registers. A subtle implication of this method is that the literal pools must have dummy time-available slots that are in the same relationship to the literal values as the real time-available slots are to the register values: instruction handlers dereference operand pointers the same way for both register and immediate operands.

The timing of the data cache is modelled by keeping track of what the tag state of a real MC88200 cache [64] would be. We model the MC88200's Least Recently Used (LRU) behavior by keeping the cycle count of the most recent access to each cache line. While this takes more storage than the LRU bit scheme that the hardware uses, it is simpler to understand and faster to execute. The data cache model calculates the time-available value for load instructions and puts this value in the destination register's corresponding time-available slot. It can do this because the cache state is kept current by memory access instructions that touch cachable memory. These instructions cause the cache state to be updated to reflect which cache line was touched, whether the line is clean or dirty, etc.

Our MC88200 model does not model bus snooping. As a result, simulations of programs that use multiple processors per node will be inaccurate. However, all of the benchmarks reported in this dissertation use only one processor per node.

The MC88100 processor has a three-slot pipeline in the Data Memory Unit (DMU) through which all memory access instructions must flow after they are issued (see Figure 4.5). Each load or store of a word or less uses one DMU slot; double-word loads and stores use two slots. When the pipeline is full, a memory instruction attempting to issue will stall until the slots it requires become available. When the request in slot one is satisfied, the pipeline shifts the contents of slot one to slot zero and slot two to slot one. After this shift, slot two is free to accept a new request.

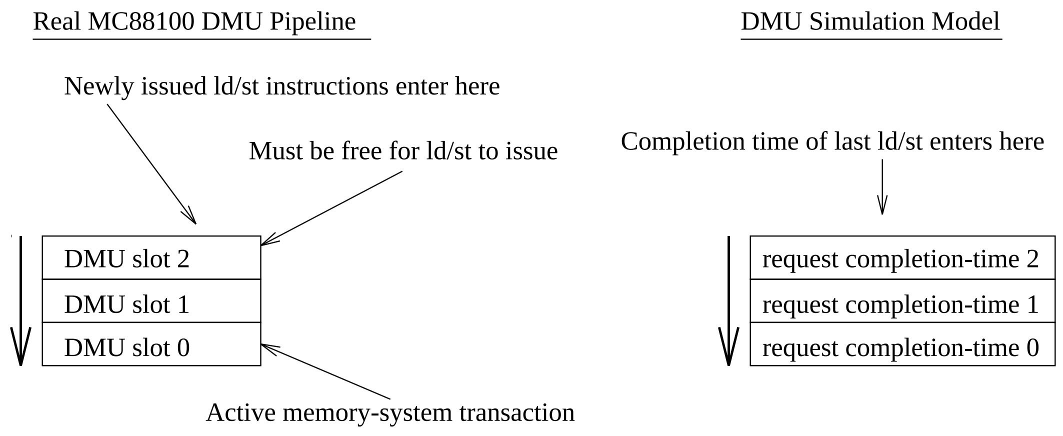

Real MC88100 DMU Pipeline
DMU Simulation Model
Newly issued ld/st instructions enter here
Must be free for ld/st to issue
Completion time of last ld/st enters here
DMU slot 2
DMU slot 1
DMU slot 0
request completion-time 2
request completion-time 1
request completion-time 0
Active memory-system transaction

Figure 4.5: Real MC88100 DMU and Simulation Model

We keep a three-element array of time-available values to model when the reference in each slot of a real MC88100 DMU pipeline would be available. Each DMU slot used advances the current processor cycle to the maximum of the current cycle and the time-available value in the oldest slot. The array is shifted down to eliminate the oldest slot and to make available the slot at the other end of the array. This slot is then filled with the time at which the reference in a real MC88100 would vacate the DMU.

4.4.6 Processor Switching

We simulate a multiple-processor system on a single-processor host by executing a few cycles of each simulated processor before switching to the next processor. The default number of cycles per switch is 10. The user can change this number. Each instruction handler checks to see if the processor cycle quantum has expired. If it has, the handler branches to the processor switch code. This code tries quickly to find another processor to execute and then dispatches the next instruction for the new processor. Because each instruction handler runs to completion, processor switching is done between instructions only. A processor can thus take more time than the quantum provides.

The processor switch code picks a new processor so as to minimize skew between any pair of processors. It does this by examining a circular list of running processors in a round-robin order and picking the first one it finds whose cycle count is below the current system cycle-count threshold. The switch code increases the threshold only when all processors have executed beyond it.

It is possible for individual instructions to take up to 10,000 cycles [1]. The processor will jump far ahead in simulated time and will not execute its next instruction until the other processors have caught up. Skew induced by long-running instructions has caused no problems or timing anomalies.

To keep the cost of switching low, there are only four key processor-dependent interpreter variables in registers (DECIP, IP, a pointer to the processor state structure, and the processor current cycle). An earlier version of the simulator kept just the DECIP in a host register and calculated IP when its value was needed [9]. The earlier version was written for a host with a small number of registers, and it therefore made sense to calculate the IP this way. Doing so in the current version would complicate the threaded-code interpreter, and our simulator host has enough registers for both DECIP and IP variables.

4.4.7 Modelling Data Memory Access

While most instruction handlers are in a single large function, memory access instruction handlers are complex and so call a separate function for most of their semantic effect. This function first translates the data's virtual address by calling the TLB model. This model simulates the Motorola MC88200's 56-entry TLB. TLB misses delay the memory access, as they do in the hardware.

The physical address returned by the TLB model is checked to see if it is an I/O address or a memory address. If it is the former, the I/O module is called. Otherwise, the physical memory map is accessed to find the host memory that models the addressed simulated

[1] This happens on a store of a **cache-copyback** command to the control register of a data cache that is full of dirty data.

memory. Host memory is allocated if it is not yet in the map. The memory operation is performed, and the data cache state is updated using the physical address returned by the TLB model.

Unlike the real cache, our data cache model does not contain the data itself. It contains tags, LRU information, and state bits. This difference means that memory coherence errors can be latent on the simulator that are manifest on the hardware, and vice versa. In other words, an artifact of our data cache modelling is that memory is always coherent. If the operating system does not flush the data cache when it should, the error will not be seen on the simulator, though it may be seen on the hardware. Or, a bug may occur running on the simulator that is the result of cached memory being overwritten. This bug is latent on the hardware, because the hardware processor reads the correct value from cache and does not see the erroneous value in memory.

The difference in memory coherency is a small price to pay for a simpler cache model and smaller cache state. In addition, it causes no timing inaccuracies.

4.4.8 Modelling Instruction Memory Access

The simulator models instruction cache cold misses, but not capacity misses. It models cold misses by distinguishing between translated and untranslated instructions. When a decode-me pseudo-instruction is encountered:

1. The processor's cycle count is advanced to reflect the time to fetch an instruction cache line.
2. All four MC88100 instructions in the cache-line that corresponds to the executed decode-me pseudo-instruction are translated to decoded instructions.

This is a small extension of the decoded instruction cache. It requires no extra work in the case of a decoded instruction cache hit and very little extra work in the case of a miss. However, it simulates only cold misses: while the real instruction cache is finite (4096 instructions), the decoded instruction cache is unlimited. A straightforward extension would model capacity misses by invalidating a cache-line of decoded instructions when a real instruction cache would replace a valid cache line. We considered doing this, but felt it was unnecessary because our benchmarks fit in the instruction cache and experienced few, if any, capacity misses.

4.4.9 Modelling I/O

When the load/store function encounters an address outside the range of physical memory, it calls the I/O module with this address and a pointer to the decoded load/store instruction. We pass a pointer to the decoded load/store for convenience: it has the size and type values that the I/O models need.

The I/O module searches a table that relates address ranges to particular I/O model functions. The table lookup corresponds to the address decoders found in hardware between the processor address bus and the I/O device select signals. An older version of our simulator used a hashing scheme to speed the lookup [9]. However, the Meerkat simulator has a small table of devices, and the lookup time is not significant.

The I/O models include Meerkat's 32-bit cycle counter, interrupt controller status and control registers, internode status and control registers, magic DRAM space (described in Section 3.6), CMMU control pages, and some pseudo-devices used for controlling execution profiling.

4.5 Meerkat Simulator's Performance

The simulator's performance is a function of workload and the processor switch time. All of our tests were made with a processor switch time of 10 cycles. We found the difference in simulator accuracy between switching every cycle and switching every 10 cycles to be insignificant. At this setting, the performance ranged from 500,000 to 750,000 simulated processor cycles per second on a 36 MHz SUN Sparc-10/30 host [2]. Thus, on average, we simulate one Meerkat processor cycle in 54 to 72 Sparc-10 cycles. While some workloads could cause much lower or higher performance, the tests we used were all in this range.

The Sparc-10 host can do more per cycle than the processor we model. Therefore, our figure of 54 to 72 Sparc-10 cycles per Meerkat processor cycle must be adjusted to make a fair estimate of the simulator's slow-down. We estimate that the Sparc-10 has half the Clocks Per Instruction (CPI) of the Meerkat processors. This means that there is roughly a slow-down of 100 to 150 per simulated processor.

The system slow-down is the ratio of the number of host cycles it takes to simulate one cycle of a whole Meerkat. Simulating one cycle of a multi-node Meerkat requires simulating one cycle of each processor and the interconnect. To calculate this figure, we

[2] This workstation has a SPECint rating of 45.2.

multiply the per-processor slow-down by the number of simulated processors. Thus, a simulated 256-node Meerkat has a slow-down of 27,000 - 37,000 to one. Because our host is about four times faster than the Meerkat processor, however, the ratio between wall-clock time and simulated time is not this large. For example, a 32k FFT simulation runs for 850,000 cycles, or 42 milliseconds. This takes six minutes of Sparc-10 time, which means that the ratio of Sparc-10 time to Meerkat time is 8,600 to 1.

4.6 Timing Models and Simulator Accuracy

We used a suite of tests both to guide development of timing models and to evaluate the simulator's overall accuracy. This section addresses these two issues.

4.6.1 Timing Model Development

Our method of timing model development was to iteratively:

1. Measure the difference between execution times of a moderate or complex benchmark program on the simulator and the prototype.
2. Identify which aspect of the prototype's timing was most responsible for the difference.
3. Write a low-level test that is more sensitive than the benchmark to the aspect identified in Step 2.
4. Verify that the low-level test shows a significant performance difference between the simulator and the prototype.
5. Add a timing model to the simulator that captures behavior identified in Step 2. The new timing model often has parameters that the user can adjust. Pick default values for these parameters.
6. Rerun the low-level test to verify that the new timing model makes the simulator accurate on the low-level test. This may require adjusting the model's parameters. If not, examine the test and the simulator on a cycle-by-cycle level. Fix the model.
7. Rerun the higher-level benchmark to see if the aspect identified in Step 2 was correct. If not, repeat. If so, check the accuracy of the simulator on another benchmark test.
8. Stop when the simulator is accurate for all the tests.

Figure 4.6 shows output from `mg88`'s `timing` command. This command also lets users change each of the listed parameters. Lines marked with asterisks are user-settable parameters that affect some aspect of the simulation unrelated to timing. All other parameters are in units of machine cycles, which correspond to 50 nanoseconds on the prototype.

```
  < 0>      DRAM refresh penalty = 36
  < 1>       DRAM refresh period = 1280
  < 2>DRAM latency to read 1 wrd = 8
  < 3>DRAM latency to write 1 wd = 7
  < 4>         cache hit latency = 2
  < 5>        cache miss latency = 2
  < 6>  cache miss line copyback = 7
  < 7>flush segment base latency = 1032
  < 8> read cache miss line fill = 11
  < 9>write cache miss line fill = 18
  <10>   flush line base latency = 10
  <11>       flush line copyback = 9
  <12>   flush page base latency = 264
  <13>invalidate all base latenc = 260
  <14> copyback all base latency = 1024
  <15> lca I/O read word latency = 8
  <16>lca I/O write word latency = 10
  <17>    TLB miss update cost   = 30
  <18>internode local int latency= 12
  <19>internode remote int laten = 16
  <20> CMMU ctl read word latency= 5
  <21>CMMU ctl write word latency= 5
  <22> DRAM read offset          = -1
  <23> DRAM write offset         = 0
  <24> internode status latency  = 16
  <25> cycles per cpu slice      = 10
* <26> bytes per stack per cpu   = 16384
  <27> no-scoreboard read offset = 3
  <28> no-scoreboard write offset= 3
  <29> internode cycles per word = 1
  <30> internode receive buffers = 0
  <31> overhead cycles per mesg  = 3
* <32> switches per display updat= 10000
* <33> histogram interval in cycl= 500
```

Figure 4.6: User-Settable Timing Model Parameters

In some cases we changed our run-time system to make the simulated and prototype execution times closer. For example, we initially used processor 1 on every node to process internode interrupts, while processor 0 did everything else. On a message-exchange test, the simulator reported half the execution time of the hardware: after processing interrupts, processor 1 had dirty cache state that had to be flushed to memory and then reloaded by processor 0. The simulator did not model the MC88200's snooping and associated cache operations, because we were not interested in the performance of programs that used multiple processors per node. We changed the run-time system to use processor 0 for everything. This improved the performance of the prototype on small messages and brought the simulator and prototype execution times in line.

Some programs have different execution results on the simulator and the prototype: there are aspects of the prototype's timing that we could not - or did not want to - model. We did not model the timing of operations that we believed would not affect the outcome of our measurements and were difficult to model. For example, the prototype runs a debugging monitor that mediates between the cross debugger running on the Sparcstation host and the running Meerkat program. This monitor: (1) lets the cross debugger control the running Meerkat program, and (2) fields requests from the Meerkat program for operating system services performed on the host. The semantic effect of the monitor is modelled in the simulator, but not its timing or its effect on instruction and data caches. The time it takes the prototype to perform operating system services is a function of the load on the Sparcstation host, the relationship between the time of the request and the host's process interval timer, and other factors.

To enable accurate measurements, we were careful to measure only the execution time of sections of code that do no I/O through the monitor and whose initial cache state is not dependent on previous calls to the monitor. In practice, we found this easy to do, and it usually meant avoiding **printf**'s until after a measurement was taken.

4.6.2 Timing Accuracy Tests

Table 4.1 shows the results of low-level tests of individual floating point and memory access instructions. The times reported are in microseconds and are an average of the time to execute 1000 iterations of the loop containing the measured instruction. The largest error is in the floating point add instruction test. The simulator overstates the cost of this instruction by one cycle, or 50 nanoseconds: the simulator does not model contention for the single write-port of the MC88100's register file. The MC88100 is capable of writing one word per cycle to its register file. The simulator makes a pessimistic guess as to whether contention will occur. In the case of the floating point add below, this guess is incorrect. We considered adding a timing model to correct this, but decided that the increased accuracy would not compensate for the effort, simulator performance degradation, and added simulator complexity.

Uncached read and write tests exercise the DRAM system. The simulator models interference with DRAM refresh, which consumes about two percent of the memory system's bandwidth. DRAM refresh is rarely modelled in system simulations, because it is considered such a small factor. Note, however, that if we did not model this aspect, our errors

Table 4.1: Low-Level Simulator Accuracy Test Results

Test name	Hardware (μ-seconds)	Simulator (μ-seconds)	Difference	
			μ-seconds	Percent
loop with no mem activ	0.301	0.301	0.000	0.0
FP add	0.704	0.754	0.050	6.6
FP multiply	0.857	0.855	-0.002	-0.2
FP divide	3.404	3.404	0.000	0.0
FP mem add/multiply	1.711	1.710	-0.001	-0.1
cache read hit	0.501	0.501	0.000	0.0
double cache read hit	0.551	0.551	0.000	0.0
cache write hit	0.350	0.351	0.001	0.3
double cache write hit	0.401	0.401	0.000	0.0
uncached read	0.867	0.862	-0.005	-0.6
double uncached read	1.336	1.323	-0.013	-1.0
uncached write	0.360	0.357	-0.003	-0.8
double uncached write	0.718	0.714	-0.004	-0.6
I/O read	0.851	0.851	0.000	0.0
I/O write	0.452	0.449	-0.003	-0.7
CMMU control page read	0.701	0.701	0.000	0.0
CMMU control page write	0.301	0.301	0.000	0.0

would be several times larger. Many of the errors are below one percent. These errors may be due to slight timing differences between the hardware and the simulator. These differences can cause the simulator to model one more, or one less, refresh cycle than occurs on the hardware. In fact, the hardware measurements show variation from run to run of about one percent.

Table 4.2 show the results of more complex operations. These include various cache operations, synchronization instructions, cache flushing, and sequences that load the local memory bus. The largest error is shown in the "copyback full line" test, where the simulator understates the time for a copyback of a cache line by a little over one clock cycle.

Table 4.3 shows the results of a message exchange test. In this test the data cache is cold before each message exchange. Table 4.4 shows the results of the same test run with a warm cache. To preload the cache, the test is run twice for each message size, and the results are taken from the second iteration.

To verify the simulator's accuracy for the tests used to compare Meerkat and Delta, we included several parallel applications in our accuracy suite. Table 4.5 shows the correspondence of simulator and hardware results for a global combine, SOR, and FFT (see Sections 5.9 and 5.10). The largest error is 7.8% on a 32-byte global combine.

Table 4.2: Medium-Level Simulator Accuracy Test Results

Test name	**Hardware**	**Simulator**	**Difference**	
	(μ-seconds)	**(μ-seconds)**	**μ-seconds**	**Percent**
cache read misses	1.074	1.079	0.005	0.5
dbl cache read misses	1.132	1.125	-0.007	-0.6
cache write misses	0.926	0.927	0.001	0.1
dbl cache write misses	0.975	0.977	0.002	0.2
full write pipeline	0.923	0.931	0.008	0.9
instruction cache fill	0.782	0.781	-0.001	-0.1
copy uncached to cache	0.938	0.936	-0.002	-0.2
exchange memory instruction	0.971	0.974	0.003	0.3
bus contention 3 reads	1.381	1.391	0.010	0.7
write back on write miss	1.286	1.291	0.005	0.4
write back on read miss	1.427	1.430	0.003	0.2
invalidate empty line	0.502	0.505	0.003	0.6
copyback full line	0.925	0.860	-0.065	-7.6
invalidate empty page	15.868	15.784	-0.084	-0.5
copyback half full page	75.060	75.008	-0.052	-0.1
copyback full page	134.224	134.252	0.028	0.0
copyback empty data cache	54.800	54.640	-0.160	-0.3
copyback full data cache	528.800	530.360	1.560	0.3
copyback+invalid whole cache	529.800	530.720	0.920	0.2

Table 4.3: Message Exchange Test Results: Cold Cache

Test name	**Hardware**	**Simulator**	**Difference**	
	(μ-seconds)	**(μ-seconds)**	**μ-seconds**	**Percent**
4-byte message	74.800	70.200	-4.600	-6.6
8-byte message	75.800	70.400	-5.400	-7.7
16-byte message	96.800	91.200	-5.600	-6.1
32-byte message	84.800	82.400	-2.400	-2.9
64-byte message	97.400	92.800	-4.600	-5.0
128-byte message	115.400	109.600	-5.800	-5.3
256-byte message	153.400	145.400	-8.000	-5.5
512-byte message	144.200	140.400	-3.800	-2.7
1024-byte message	150.800	145.200	-5.600	-3.9
2048-byte message	176.200	170.800	-5.400	-3.2
4096-byte message	259.800	254.800	-5.000	-2.0
8192-byte message	422.000	419.200	-2.800	-0.7
16384-byte message	754.000	747.000	-7.000	-0.9
32768-byte message	1192.000	1190.800	-1.200	-0.1
65536-byte message	2134.400	2139.400	5.000	0.2

Table 4.4: Message Exchange Test Results: Warm Cache

Test name	**Hardware**	**Simulator**	**Difference**	
	(μ-seconds)	**(μ-seconds)**	**μ-seconds**	**Percent**
4-byte message	71.200	68.400	-2.800	-4.1
8-byte message	74.000	68.400	-5.600	-8.2
16-byte message	91.600	88.000	-3.600	-4.1
32-byte message	85.600	83.000	-2.600	-3.1
64-byte message	97.400	95.000	-2.400	-2.5
128-byte message	120.000	113.800	-6.200	-5.4
256-byte message	159.200	154.400	-4.800	-3.1
512-byte message	155.400	153.200	-2.200	-1.4
1024-byte message	176.000	175.400	-0.600	-0.3
2048-byte message	233.200	229.800	-3.400	-1.5
4096-byte message	258.800	252.800	-6.000	-2.4
8192-byte message	421.600	422.200	0.600	0.1
16384-byte message	750.600	750.400	-0.200	-0.0
32768-byte message	1655.000	1650.800	-4.200	-0.3
65536-byte message	2592.000	2582.600	-9.400	-0.4

Table 4.5: High-Level Simulator Accuracy Test Results

Test name	**Hardware**	**Simulator**	**Difference**	
	(μ-seconds)	**(μ-seconds)**	**μ-seconds**	**Percent**
Global Sync Average	94.558	90.142	-4.416	-4.9
Global Combine 8 bytes	130.200	127.400	-2.800	-2.2
Global Combine 16 bytes	94.400	95.800	1.400	1.5
Global Combine 32 bytes	97.400	105.600	8.200	7.8
Global Combine 64 bytes	128.200	124.000	-4.200	-3.4
Global Combine 128 bytes	160.000	150.200	-9.800	-6.5
Global Combine 256 bytes	226.200	224.200	-2.000	-0.9
Global Combine 512 bytes	381.600	368.800	-12.800	-3.5
Global Combine 1024 bytes	635.800	614.400	-21.400	-3.5
Global Combine 2048 bytes	887.000	885.400	-1.600	-0.2
Global Combine 4096 bytes	1697.800	1698.000	0.200	0.0
Global Combine 8192 bytes	3323.600	3498.000	174.400	5.0
Global Combine 16384 bytes	8292.200	8234.400	-57.800	-0.7
Global Combine 32768 bytes	15671.200	15223.200	-448.000	-2.9
R/B SOR 32x32	30252.000	29379.000	-873.000	-3.0
R/B SOR 32x32	59248.800	58650.600	-598.200	-1.0
FFT 16 points	2266.200	2300.200	34.000	1.5
FFT 512 points	5152.000	4883.800	-268.200	-5.5
FFT 1024 points	10136.000	9658.400	-477.600	-4.9
FFT 2048 points	21088.600	19987.600	-1101.000	-5.5
FFT 4096 points	46496.400	43895.000	-2601.400	-5.9

4.7 Profiling Features

Meerkat's simulator collects a variety of statistics as it executes. The most basic performance figures are execution times expressed in cycles. The current system cycle number is available on both the hardware and the simulator in a 32-bit register that is incremented every four processor cycles. At a 20 MHz system clock rate, the cycle counter has 200 nanosecond resolution. Our benchmark tests contain code to sample this counter at the start and end of the computation and to display the result upon completion. While overall execution time is useful and interesting, it does not explain where the time is spent.

The `mg88` user can select one processor to be monitored by a profiling feature similar to the UNIX `prof` utility. After execution, this feature will display a list of the functions called and how many cycles were spent in each. Also displayed is the number of cycles each function is delayed by an interrupt routine.

Figure 4.7 shows the profile of node 3 running the light-load bandwidth test (described

```
<CALLS ,TOT CYCLE,PERCALL,INTER>               CALLER:LINE/ADDR   CALLEE
<     1,     2705,   2705,    0>                 main:34       -> do_bwtest
<     1,     1584,   1584,    0>            do_bwtest:87       -> crecv
<     1,     1188,   1188,    0>                crecv:0x10700  -> msgwait
<     1,     1087,   1087,    0>            do_bwtest:88       -> csend
<     1,      434,    434,    0>       ibus_interrupt:0x102a0  -> message_handler
<     1,      373,    373,    0>                crecv:0x106fc  -> irecv
<     1,      331,    331,    0>        pending_match:0x10624  -> receive_buffer
<     1,      329,    329,    0>                csend:0x10cf8  -> flush_on_send
<     1,      316,    316,    0>        flush_on_send:0x10bf4  -> copyback_data_cache_
<     1,      144,    144,    0>                csend:0x10d3c  -> send_buffer
<     1,      120,    120,    0>                csend:0x10d80  -> send_buffer
<     1,       60,     60,    0>                csend:0x10d28  -> arbitrate_for_bus
<     1,       60,     60,    0>                 main:9        -> __main
<     1,       31,     31,    0>                csend:0x10d2c  -> send_buffer
<     1,       30,     30,    0>                csend:0x10cfc  -> packetize_data
<     1,        8,      8,    0>     crecv_wt_for_msg:0x1083c  -> get_buffer_header
<     1,        7,      7,  567>              msgwait:0x10958  -> free_buffer_header
<     1,        5,      5,    0>            do_bwtest:82       -> mynode
<     1,        4,      4,    0> copyback_data_cache_:485      -> mydatacmmuaddr
```

Figure 4.7: Sample Profiling Output

in Section 5.5). Each row of the display represents one call site. The first column shows the number of times the call was executed; the second shows the total number of cycles spent in the called function and all of its descendants; and the third shows the average number of cycles per call (i.e., the second column divided by the first). The fourth column gives the number of cycles a function call was penalized by an interrupt routine. The fifth column indicates the call site, which is followed by the name of the called function.

In the example, the function `csend` is called once by `do_bwtest` at line 88 and consumes 1087 cycles. To filter out time spent in initialization and other program segments that we do not wish to measure, there is a pseudo device that allows the running program to control profiling.

The simulator uses a myriad of counters to keep track of the performance of each processor's TLB, data cache, and local bus. It counts the number of cycles each processor is delayed waiting for internode buses. The number of successful and failed arbitration requests is kept for each internode bus, in addition to the longest and shortest bus tenures, the number of times a bus was acquired and then dropped before data could be sent, the number of words moved across each bus, and the total number of cycles each bus was active.

All of these counters were added to answer questions that arose. For example, speedups on our FFT benchmark were initially much lower than expected. Suspecting that internode bus contention was the cause, we added per-processor counters that tracked the number of cycles each processor was delayed due to internode bus contention. We found the delays were small and did not account for FFT's poor speedup. After we added the `prof`-like feature, we saw that the error was due to inefficient coding of a constant calculation in the benchmark.

The `prof`-like feature and event counters are useful, but they summarize a computation after its execution. Some aspects of program behavior are hard to understand from summaries. To give the `mg88` user a running image of internode bus activity, we included an option to animate internode communication. When this option is chosen, `mg88` opens two X windows, one showing each node and bus and one displaying a histogram of bus contention. The former window shows active buses as thick blue lines. Sending nodes have a thick blue border, while receiving nodes are green. Inactive nodes and buses are drawn with thin black lines. The histogram window shows the system cycle count on the horizontal axis, with bus contention on the vertical axis.

These animations of Meerkat internode communication helped us visualize the dynamic properties of our benchmark programs. They also helped explain the architecture to those unfamiliar with Meerkat.

4.8 Fast Conditional Breakpoints

This section describes a feature of `mg88` that improves the performance of conditional breakpoint evaluation.

Breakpoints are a fundamental facility in most debuggers. Many debuggers allow breakpoints to be set both in terms of machine addresses and program line numbers. It is also common to allow the user to supply conditional expressions that are evaluated when a breakpoint is set. If the expression is true, the breakpoint is recognized and execution stops. If it is false, program execution proceeds as though the breakpoint were not set. In traditional implementations, the time to evaluate the breakpoint condition is significant if it evaluates false many times.

We improved the performance of conditional breakpoint evaluation by evaluating the condition using target machine instructions. Instead of returning to gdb to evaluate conditional expressions, our system executes tens or hundreds of simulated cycles to execute out-of-line code to evaluate the breakpoint condition. We patch a branch to this code into the text where we would otherwise have inserted a breakpoint instruction.

Fast breakpoint condition evaluation has been implemented before [53]. What we report here is new in that we are doing it in a system debugger, we found it useful in this environment, and that it took only a few days to implement.

G88 generates Motorola 88100 code to evaluate the conditional expression on the target for most expressions. Some expressions are tricky to compile and occur rarely as breakpoint conditions. In these cases we evaluate the condition with the slow but general mechanism that is already present in gdb.

The evaluation code is generated on the host and downloaded to an area set aside by the run-time system for this purpose. This allocation of space is the only modification to the run-time system that was needed to implement fast conditional breakpoints.

Figure 4.8 shows an example of a conditional breakpoint and the code that gdb generates to evaluate it. Our code generator uses some otherwise idle registers (these registers were reserved by Motorola for the linker, but our linker does not use them). Architectures without such idle registers require some register spill/restore code surrounding the condition evaluation code. While our code generator is target dependent, it is small, around 900 lines of C, and should be easy to port to most architectures.

Fast conditional breakpoints have an even greater benefit when `mg88` is used to cross-debug programs running on the hardware prototype. The cost of communicating with the

```
[0] <kernel> (g88) break machdep.c:144 if scpus < 3

        or.u    r26,r0,hi16(0x85494)
        ld      r26,r26,lo16(0x85494)   | r26 := scpus
        or      r27,r0,0x3              | r27 := 3
        subu    r26,r26,r27             | BINOP_LESS
        bcnd.n  lt0,r26,L0
        or      r26,r0,1                | Eval true
        or      r26,r0,0                | Eval false
L0:
        bcnd    ne0,r26,L1              | Branch if the break condition is true
        or.u    r26,r0,hi16(0x667ac)    | Load the address of the instruction
        or      r26,r26,lo16(0x667ac)   | to return to
        jmp.n   r26                     | branch back to program
        .word   0x5c400008              | Displaced instruction (copied from
                                        | the breakpoint location)
L1:
tb0 0,r0,254 | breakpoint-trap
```

Figure 4.8: Conditional Breakpoint Followed by Code Generated To Evaluate It

debugee is much higher in the hardware environment than it is in the simulator. Operating system debuggers usually have a larger cost of communicating with the debugee than do conventional debuggers. As a result, the benefit gained from evaluating the breakpoint on the target is even higher than in the systems that Kessler considered [53].

4.9 Summary

Simulators vary widely in their application, structure, accuracy, and performance. We outlined several simulator applications and the simulators typically used. The Meerkat simulator is unique in its combination of speed, efficient use of memory, fine modelling detail, portability, user/supervisor modelling, and modelling of address translation. With this combination of features we were able to model large systems at a fine level of detail and make accurate predictions about the performance of large systems.

There are a number of benefits of using simulators over hardware, e.g., adaptability, deterministic execution, low cost. In addition to the benefits that most simulators provide, our simulator has three features that made debugging programs easier:

- `mg88` allows single-stepping and breakpointing of exception handlers. Large portions of the message-passing system execute as an exception handler, making it much easier to debug this code on the simulator.
- `mg88` has a precise memory breakpoint feature. When set, execution of all processors in the system stops the instant the location being watched by the breakpoint is

accessed. This aids certain debugging problems tremendously. Building as precise a breakpoint into the prototype would require redesigning the processor.

- `mg88` compiles conditional breakpoint expressions into target machine code. This speeds conditional breakpoint evaluation by orders of magnitude. The user can make more liberal use of conditional breakpoints without having to wait long periods for frequently-false conditions to be evaluated.

We described the Meerkat simulator after placing it in the context of a range of simulation strategies. The simulator can execute instructions quickly, yet models timing accurately, and can efficiently multiplex amongst many simulated processors.

Simulator construction began with a fast threaded-code interpreter and a translator to generate threaded-code from machine instructions. To make the simulator usable, we chose a powerful symbolic debugger for the front end. We achieved timing accuracy by carefully adding functional models to the behavioral simulator base. These functional models slowed the simulator down, but because we added only those functional models that substantially affected accuracy, the degradation was less than an order of magnitude.

Our simulator executes about 100-150 host instructions per simulated instruction. This compares favorably with other timing-accurate macro simulators. The high performance allows us to model multicomputers with 256 processors running substantial programs.

A number of performance monitors and animation features give the user a comprehensive view of the simulated system. These features were easy to add to the simulator, but would be difficult or impossible to add to the prototype. They are nonintrusive, i.e., their presence does not affect the execution behavior of the simulated system.

We showed the results of running a suite of tests on both the Meerkat prototype and the simulator. These tests show that the simulator is a faithful model of the prototype, usually differing from the prototype by only a few percent.

Chapter 5

A PERFORMANCE ANALYSIS OF MEERKAT

This chapter compares the performance of programs running on both the Meerkat prototype and Meerkat simulator[1] with those running on Intel's Touchstone Delta [49]. These comparisons demonstrate that Meerkat is an effective multicomputer architecture when compared to Delta, a heavily used, commercially built multicomputer. Our tests prove that: (1) Meerkat can drive its interconnect at nine times the rate of the Delta, (2) the bisection bandwidth of a 256-node Meerkat is three times higher than that of the same-sized Delta, and (3) Meerkat's speedups exceed Delta's up to the limit of Meerkat's scaling range.

Our simulated Meerkat, which can have up to 256 nodes, is calibrated to our prototype hardware, which has four nodes. The prototype cannot scale beyond its small number of nodes. The limited scalability of the prototype is not indicative of the architecture's potential. Rather, it was a decision made to facilitate timely results from a one-person effort. In contrast to the prototype, the architecture can scale to hundreds of nodes. That is, the architecture permits designs that have many nodes and that are only slightly more complicated that our prototype.

We introduce the differences between Meerkat and Delta in Section 5.1. Section 5.2 discusses the relationship between multicomputer interconnect performance and message size. Section 5.3 describes the message-passing primitives that we implemented for Meerkat. Section 5.4 gives insight into the implementation of these primitives. The interconnect's performance under light and heavy loads is analyzed in Sections 5.5 and 5.6. Meerkat's performance on two numerical applications is shown in Sections 5.9 and 5.10.

5.1 Introduction

Like Meerkat, Delta is a multicomputer composed of RISC processors, local memory, and an interconnect that is used explicitly by application software. The two systems differ in their interconnects: Delta employs a conventional mesh of 2-D routers, while Meerkat uses

[1] The Meerkat simulator is described in Chapter 4.

sets of vertical and horizontal internode buses.

There are other differences between Meerkat and Delta that complicate the comparison of the two interconnect architectures:

- The 40 MHz Intel i860 processors in Delta are about twice as fast as the 20 MHz Motorola 88100 processors in Meerkat.
- Meerkat's message-passing code is written in carefully crafted assembler, while Delta runs the NX/M operating system [48], written in C.
- The Meerkat internode interface copies to and from memory, whereas the Delta interface requires the processor to load and store each byte moved through the interface (i.e., programmed I/O).
- The Delta test program runs in an address space separate from NX/M and thus incurs context-switching costs, while Meerkat's test program runs in the same address space as the message-passing library.

Some of these differences, however, offset others. For example, the slower Meerkat processors executing our small, message-passing library offset the effect of faster Delta processors executing the larger NX/M operating system.

Despite these differences, there are conclusions we can draw from experiments comparing the systems. We know that the Delta is an effective multicomputer. If we can show that Meerkat's performance is better, we can conclude that Meerkat is also effective.

5.2 Message Granularity

Interconnect bandwidth and latency are functions of message size. Message size itself is a function of the: (1) algorithm, (2) data layout, (3) number of nodes applied to the problem, and (4) problem size. In general, if the number of nodes increases while other parameters remain constant, the size of messages will decrease. Likewise, an increase in problem size often increases the message size.

While a comprehensive discussion of message size exceeds the scope of this thesis, we provide the following examples as background for our subsequent performance discussion:

- The butterfly FFT algorithm using a cyclic layout generates messages that are $\frac{N}{P}$ points long, where N is the number of points in the FFT, and P is the number of nodes. The input to the FFT is a sequence of points. Each point is a complex number which, in our implementation, consumes 16 bytes of memory. Thus, a 32,768-point FFT on 256 nodes will send 2048-byte messages.

- We speculate that operating system traffic on Meerkat would be composed of both short control messages and page-sized messages to support file system and virtual memory traffic. Process migration would generate messages in excess of 64K bytes.
- Blocked iterative solution methods send messages that are proportional to the size of one edge of the block. A red/black successive over relaxation algorithm on a 4096 by 4096 grid running on 256 nodes will use a block that is 16 by 16. For double-precision numbers, each message will be 1024 bytes.

5.3 Message Passing

All benchmarks use a subset of Delta's message-passing library, which we implemented on Meerkat. The subset contains the following primitives:

- `Void` **csend**`(int type, const char *buf, int size, int dest)`
- `Int` **irecv**`(type, char *buf, int size)`
- `Void` **msgwait**`(int key)`

`Csend` sends a message of type `type` to node `dest` of length `size` whose data starts at address `buf`. Message delivery is reliable and in-order. Control does not leave `csend` until the message is delivered to the destination node, although it can be buffered by the receiving node until the application asks for it. If `dest` is -1, the message is broadcast to all nodes.

`Irecv` tells the message-passing system that: (1) the application is ready to receive a message of type `type` in a buffer of length `size` bytes or less, and (2) that the buffer starts at address `buf`. A type of -1 matches any received type. `Irecv` returns a key that can subsequently be passed to `msgwait`. `Msgwait` returns when the buffer associated with the passed key is filled with a received message. If `irecv` is called with a type field matching a message that has been received but not yet been delivered to the application, the message will be copied into the buffer supplied by `irecv`.

`Irecv` checks for a message with a matching type. If one has arrived, it copies the message to the user buffer and returns a value that will tell a subsequent call to `msgwait` that the message has arrived. If not, it allocates a pending-receive record and puts this record, which contains the user's buffer description and the expected message type, on a list of pending receive records. When the message arrives, it is delivered directly into the buffer supplied by `irecv`.

In Meerkat, the key returned by `irecv` is zero if the message is waiting. Otherwise, it is a pointer to the newly allocated pending-receive record. `Msgwait` returns immediately if it is passed a zero. If not, it spins, waiting for the passed pending-receive record to be marked "done" by the internode interrupt handler.

5.4 Chronology of `Csend` and `Irecv` Message Primitives

Figure 5.1 shows timelines with a simple `csend` and `irecv/msgwait`. A sending node is on the top timeline, and a receiving node on the bottom. The sender executes `csend`, while the receiver executes an `irecv` followed by a `msgwait`.

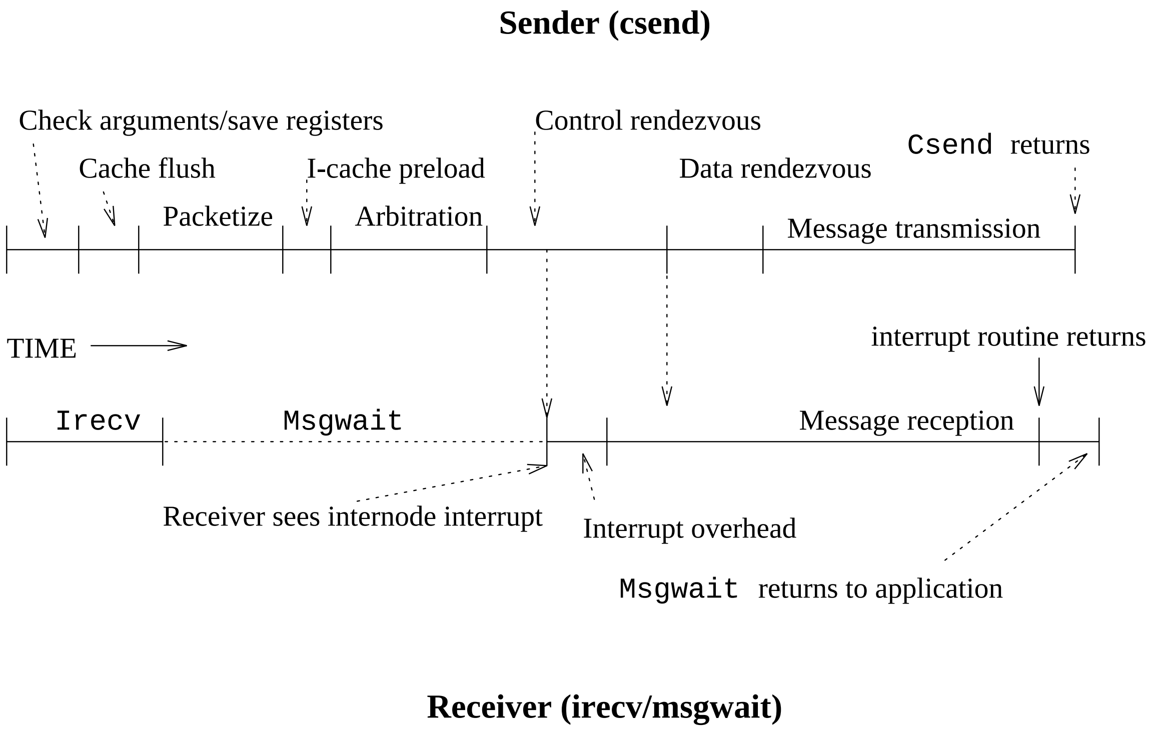


Figure 5.1: Simple `Csend` and `Irecv/Msgwait` Timeline

The steps of `csend` are:

1. *Argument checking*: check arguments to `csend`, test for broadcast, test for destination node being equal to the sending node, determine routing (i.e., 1-bus vs. 2-bus; if 1-bus, horizontal vs. vertical).
2. *Data cache flush*: instruct the data cache to write back to memory any dirty cache lines that hold data in the range of memory addresses to be sent.
3. *Message packetize*: divide the message into one or more data packets and, in so doing, construct a control packet.
4. *Internode bus arbitration*: acquire ownership of the required buses.

5. *Instruction cache preload*: preload the sender-receiver rendezvous code in the sender's instruction cache. This code can cause a bus timeout on the receiver if it does not execute certain groups of instructions quickly enough. Preloading the instruction cache makes the critical groups execute quickly. Preloading is done by executing critical code with parameter values that cause no internode activity.
6. *Control packet rendezvous*: signal the receiver node, wait for it to enter a receptive state, and begin to transmit the control packet. The receiver takes an interrupt when it is signalled.
7. *Data packet rendezvous*: signal the receiver node, wait for it to enter a receptive state, and begin to transmit a portion of user-specified data. The receiver polls for signals from the sender and thus avoids the interrupt overhead.
8. *Data transfer*: move data from the sending processor's memory to the receiver's memory. Each word moved requires one cycle.
9. *Miscellaneous processing*: call subroutines, save/restore registers, return to caller, etc.

Table 5.1 shows the execution cost of **csend** steps as a function of message size. Each pair relates the number of cycles for a given step followed by the percentage this step represents of the total time for `csend`.

Table 5.1: Number of Cycles and Percentage of Total Time Taken by `Csend` Step (Percentages Rounded to the Nearest Point)

`Csend` **Step**	**Message Size (in bytes)**							
	0		**128**		**1,024**		**32,768**	
1+9. Miscellaneous overhead	114 /	25%	125 /	16%	164 /	15%	531 /	5%
2. Data cache flush	46 /	10%	240 /	30%	322 /	29%	1,134 /	10%
3. Message packetize	50 /	11%	21 /	3%	28 /	3%	172 /	2%
4. Connection arbitration	52 /	11%	61 /	8%	53 /	5%	81 /	0%
5. I-cache preload	30 /	7%	30 /	4%	30 /	3%	30 /	0%
6. Control-packet rendezvous	144 /	32%	144 /	18%	144 /	12%	158 /	1%
7. Data-packet rendezvous	0 /	0%	148 /	18%	156 /	14%	769 /	7%
8. Data transfer	1 /	0%	32 /	4%	256 /	23%	8,192 /	74%
Total (cycles/percent)	456 /	100%	801 /	100%	1,113 /	100%	11,102 /	100%

The table shows that, as message size increases, the percentage of time spent sending data through the interconnect (Step 8) increases. Similarly, for all steps except Step 8, as message size increases, the percentage of time spent decreases. For example, zero-byte messages spend 11% of execution time in arbitration; this figure decreases to five percent for 1,024-byte messages, and to less than half a percent for 32,768-byte messages.

Miscellaneous overhead figures include time spent on argument checking and routing decisions.

Table 5.2 shows the execution cost to a receiving node given four message sizes. The `irecv` row shows the cost of posting a receive, i.e., of recording the user's request for a message in advance of the message's reception. The next row shows the cost of executing the interrupt subroutine that causes the rendezvous with the sending node, places data in the user's buffer, etc. The 'interrupt overhead' line contains time spent by the processor: context-switching to the interrupt exception handler, decoding the interrupt, saving registers, acquiring a semaphore that guards against conflicting use of the network interface, and receiving the control packet. The time spent receiving a message overlaps with the sending node's time for sending.

Table 5.2: Number of Cycles Taken by Message-Reception Step

Reception Step	**Message Size (in bytes)**			
	0	**128**	**1,024**	**32,768**
`irecv` (post a receive)	116	298	373	1,314
Message reception	105	210	434	9,345
Interrupt overhead	137	135	134	160
total (cycles)	358	643	941	10,819

Table 5.2 reveals several characteristics of the message-passing library. First, posting a receive requires a cache flush in order to maintain consistency. The larger the buffer, the more cache state must be adjusted, and the longer the time will be to post the receive. However, the increase is sublinear; beyond a certain message size, the whole cache is flushed. The first line of the table shows: (1) 182 cycles to flush the first 128 bytes, or about 1.4 cycles per byte, (2) 75 cycles to flush the next 896 bytes, or 0.083 cycles per byte, and (3) 941 cycles to flush the next 31744 bytes, or 0.029 cycles per byte.

Second, the table shows that, for large messages reception itself, (i.e., moving the data from the interconnect to the receiver's memory) dominates execution time. The reception time goes from 0.82 cycles per byte for small messages to 0.28 for long messages. If there were no software overhead, this figure would be 0.25 cycles per byte (one cycle per word). Thus, for long messages, Meerkat's message reception approaches the hardware transfer rate.

Lastly, the interrupt overhead is fairly constant, rising slightly for the largest messages.

This rise of 26 cycles (160-134) for 32,768-byte messages is due to cache effects: large messages flush the whole cache, including variables used by the message-passing library. The 26-cycle increase represents the time to reload these variables.

5.5 Bandwidth under Light Load

This section reports the performance of Meerkat and Delta on a simple test that measures the ability of nodes to exchange data of varying sizes. The core of the program for this test appears in Figure 5.2:

```
light_load_test(char *buf, int size)
{
  double start, end, elapsed_time, bytes_per_second;
  int key;

  switch (mynode()) {
    case 0:
      start = time_in_seconds();
      key = irecv(0, buf, size);
      csend(0, buf, size, 3, 0); /* Send to node 3 */
      msgwait(key);
      end = time_in_seconds();

      elapsed_time = end - start;
      bytes_per_second = 2 * (size / elapsed_time);
      printf("MB/sec for message of %d bytes=%7.3lf\n",
                      size, bytes_per_second / 1000000.0);
      break;

    case 3:
      crecv(0, buf, size);
      csend(0, buf, size, 0, 0); /* Send to node 0 */
  }
}
```

Figure 5.2: Message-Passing Program to Test Meerkat under Light Load

This test causes node zero to send a message of length `size` to node three. Node three waits for the message, which it sends to node zero. The total number of bytes sent is twice the message size. We measure the time interval from immediately prior to node zero's transmission until immediately after node zero receives a reply from node three. We chose the node numbers (zero and three) so that the communication requires a two-bus connection. We execute the sequence above for message sizes ranging from four to 256k bytes.

Figure 5.3 shows the bandwidth achieved by a pair of nodes for both systems as a function of message size. We include in these graphs the bandwidth measured on the Intel Hypercube iPSC/2 and iPSC/860 [5]. In this test the bandwidths reported by the Meerkat simulator and the hardware differed by about one percent. Therefore, the Meerkat curve can be viewed both as a measurement of a real system and as a simulation result.

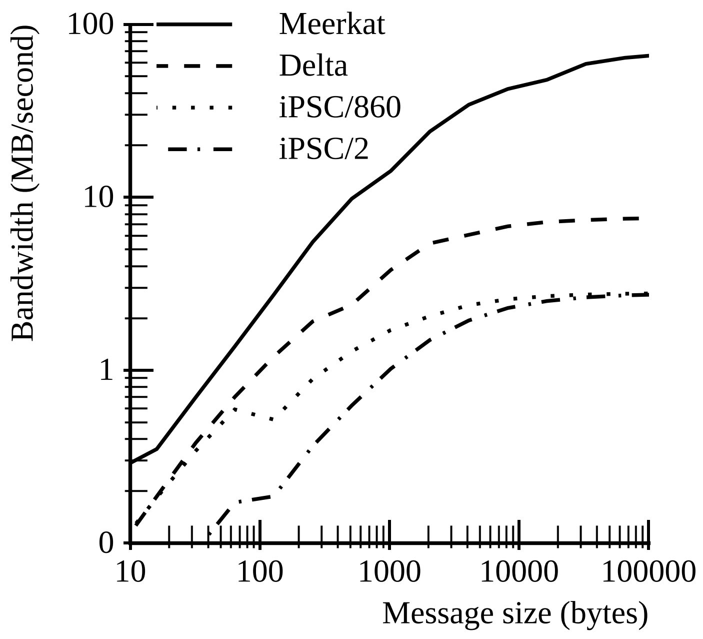


Figure 5.3: Bandwidth as a Function of Message Size under Light Load

The bandwidth of both Meerkat and Delta on small messages is limited by the ability of the nodes to inject messages into the network. The lower performance of Delta on small messages may be due to the extra work done in NX/M that is not done in the Meerkat message-passing library. Meerkat achieves 67 MB/sec for 100,000-byte messages, while Delta's bandwidth levels off at about 8 MB/sec for 2,000-byte messages.

Both Meerkat and Delta are limited on long messages by their different abilities to drive their interconnects. Meerkat's internode bandwidth reaches 83 percent of its peak rate of 80 MB/sec. Delta reaches 10 percent of its theoretical rate, which is also 80 MB/sec [81].

Delta's ability to drive its interconnect in this test is limited by its network interface and the speed of the node processor. The network interface requires that the processor manipulate each byte sent through the interconnect. In the next section, we will see that Delta's network can handle more traffic than a single node can generate.

While Meerkat's maximum interconnect performance is seen at a message size that is longer than most applications will generate, its performance on shorter messages is still high. It may make sense, however, to reduce the per-message overhead by moving logic from low-level software into hardware. This would push the solid curve in Figure 5.3 higher and to the left. Chapter 6 discusses changes to the network interface that can reduce per-message overhead.

5.6 Bandwidth under Heavy Load

In this experiment there are two groups of 128 nodes each, A and B, as shown in Figure 5.4. Each node in group A sends a message to its partner in group B and waits for a reply. Each node in group B waits for a message from its partner in group A; it then sends a message back to its partner. The nodes of each group are physically contiguous in an 8 by 16 block, and the two groups are adjacent to form a 16 by 16 block of nodes. The distance between each node and its partner is eight. The total number of bytes moved between the groups is the product of the message size and the number of messages sent, which is 256. We calculate the bandwidth by dividing the total number of bytes moved by the round trip time.

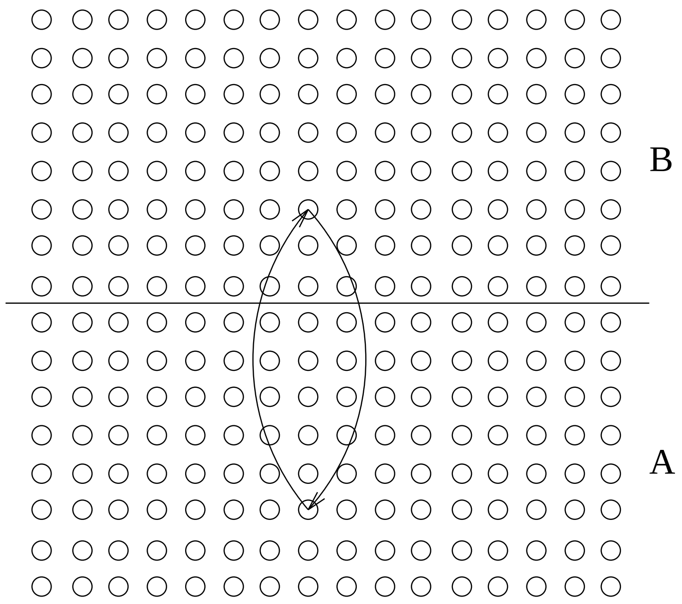


Figure 5.4: Pattern of Communication during Heavy Load Test

Figure 5.5 shows both Meerkat and Delta with a nearly linear increase in bandwidth with increasing message size for messages of less than 500 bytes. As with the light interconnect load, this increase results from the amortization of a fixed processor overhead per message over longer messages. The Delta bandwidth reaches a maximum of 280 MB/sec with a message size of 1000 bytes. Meerkat peaks at 750 MB/sec at a message size of 4000 bytes.

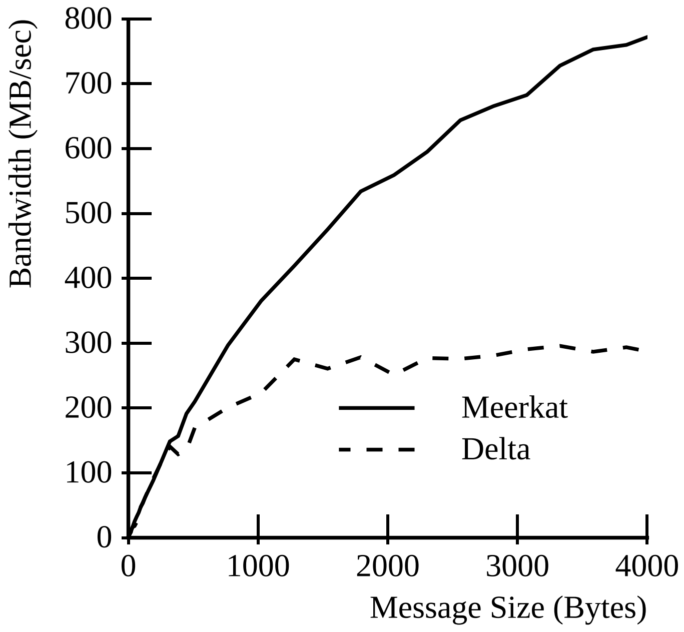


Figure 5.5: 256-Node System Bisection Bandwidth

Dividing each of these bandwidths by the number of channels through which the data move, in this case 16, we find that Meerkat's buses are driven at an average of 46 MB/sec. Delta's channels at the midpoint are driven at 17 MB/sec. Delta's channels are composed of pairs of unidirectional links. Through separate tests, we determined that the maximum undirectional link speed is 11 MB/sec. Thus, the 17 MB/sec figure shows that our heavy-load test overlaps the use of these unidirectional links.

The earlier plateau in Delta's bandwidth is due to Delta's higher ratio of processor to interconnect performance. That is, Meerkat's slower processors need longer messages to saturate its faster interconnect. However, the level of the plateaus depends on interconnect performance rather than processor speed.

5.7 Per-Node Bandwidth

To better understand what the results in the previous section mean for application performance, we consider the amount of internode bandwidth available to each node. We assume that the system is heavily loaded and that each node uses an equal amount of bandwidth.

Figure 5.6 shows system bisection bandwidth divided by the number of nodes. This graph is in units of double words (DW) (8 bytes), as this is often the unit in which numerical applications programmers consider problem and message sizes. The per-node bandwidth is shown for several sizes of Meerkat and Delta. Larger systems have lower per-node bandwidths because the bisection bandwidth increases with the square root of the number of nodes.

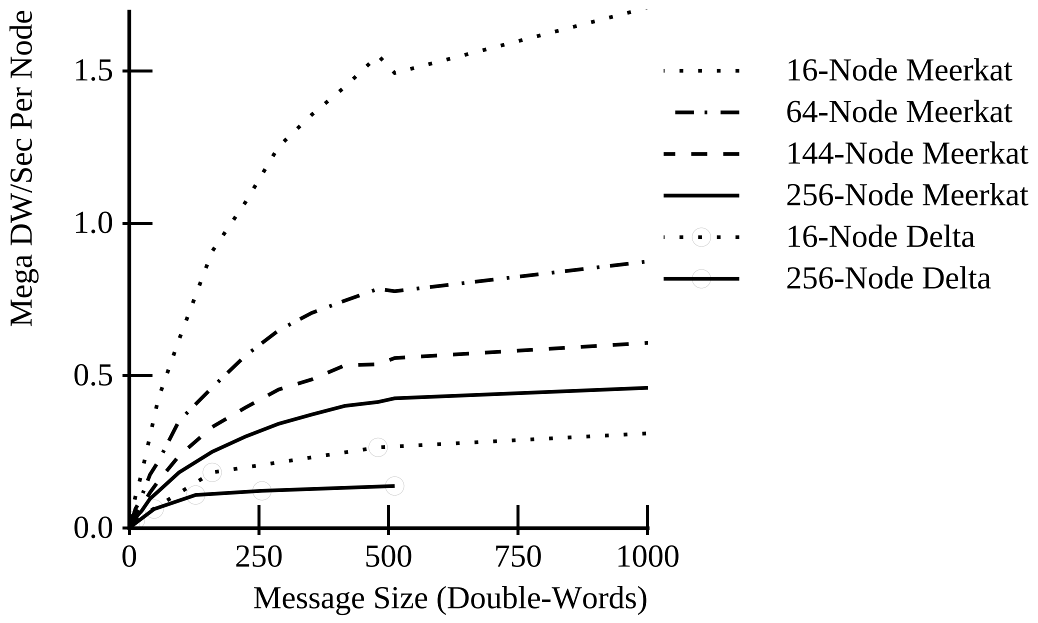


Figure 5.6: Per Node Bandwidth

Consider this example: if an application running on a 144-node Meerkat has every node send 500-DW messages and does no computation, every node will sustain a transfer rate of 550,000 DW/sec. All applications will spend some of their time computing, so these curves represent the upper bound on internode performance. These figures assume that nodes share the interconnect equally. The internode bandwidth seen by a single node can be much higher if other nodes make only light use of the interconnect.

These curves show that Delta has a lower per-node bandwidth than Meerkat for the same reasons given in the prior two sections for lower bandwidth. These differences are mostly a function of implementation, not of architecture. However, the ease with which we were able to design and build a fast implementation *is* a result of Meerkat’s simplicity.

5.8 Application Performance

It is important for prospective parallel computer users to be able to anticipate how well their applications will run. A given parallel application may run well on some computers and poorly on others. "Running well" is, of course, subjective. The definition we adopt in this section is that the processors execute application code for at least half of the overall execution time.

This section characterizes the applications that are likely to run well on Meerkat. We define these applications in terms of the application’s computation and communication demands. Our goal is to give application developers enough information to decide what combinations of parallel applications, input data sizes, and Meerkat configurations will run well.

The principal application characteristics that determine performance on Meerkat are: (1) the ratio of computation to communication, (2) message size, and (3) load balance.

The first characteristic, the *comp/comm ratio*, is the ratio of the instructions executed to the amount of data sent between nodes [77]. For example, an application that executes one million double-precision floating point instructions on each of 256 processors, and which passes two million double words between nodes, has a comp/comm ratio of 128. That is, each node will perform an average of 128 FP operations for each DW value it sends to another node.

The second application characteristic of importance is the *grain size*, the size of the message the application uses to package internode data. As the grain size decreases, the time spent to process messages increases. This computational effort comes at the expense of application running time.

Both of these characteristics are functions not only of the application, but also of the problem size, the number of nodes, and the details of how the application is implemented. Some algorithms, such as Cholesky factorization, use messages with sizes that are also a function of the input data. Other algorithms, e.g., modified Gram-Schmidt with partial pivoting [96], have a comp/comm ratio that is a function of the input data. Estimating the ratio for these applications may be difficult.

Applications that have very high comp/comm ratios, e.g., over a million to one, will perform well on almost any parallel computer. They spend almost all of their time computing. Even if internode communication is expensive, they spend little time communicating. These applications are perhaps best served by parallel computers with low-performance, inexpensive interconnects.

Applications that have very low comp/comm ratios, e.g., between zero and one, will perform poorly on any parallel computer. They will spend most the time communicating and little time computing, leaving processors idle most of the time. These applications are probably best run on uniprocessors, where no internode communication is necessary.

Between these extremes are applications that will run well on some systems but not on others. To gauge whether they will run well, i.e., with high processor utilization, we consider the comp/comm ratio of the parallel computer. This is the ratio of the rate at which each node can execute instructions to the rate at which each node can transmit data. For example, a system with processors that can sustain 10 MFLOPS and in which each processor can transmit data at 1 million floating point values per second (while the other processors are doing the same) has a comp/comm ratio of 10.

Comparing the system's ratio with the application's allows us to determine whether the application will run well. If the system's ratio is much lower than the application's, we expect that the application will spend most of its time computing, little time waiting for internode data, and will run well. On the other hand, if the system's ratio is much higher than the application's, the application will spend most of its time waiting for communication phases to complete, and it will run poorly.

Processors can also have low utilization if the computational load is not balanced. Thus, it is not good enough to have an application comp/comm ratio that is higher than the system's. Many regular problems, such as FFT and SOR (described in the next two sections), present the same computational load to each node, regardless of the input data. Other algorithms, however, require variable amounts of computation. In Cholesky factorization [44, 7], each processor is assigned a fixed portion of the matrix on which to operate. However, the number of operations performed on each portion is a function of the input data. Some processors may be idle, while others are busy. This load imbalance can be lessened by choosing a small block size. However, this decreases grain size and thus communication efficiency. There is a tension between making the block size small, to increase load balance, and making it large, to amortize the fixed message cost overhead.

Meerkat's comp/comm ratio is a function of message size and system size, as shown in Figure 5.7. These curves were generated by dividing the nominal node execution rate (10 MFLOPS) by the per-node internode bandwidth (shown in Figure 5.6).

Consider a sample use of these curves. Assume that we want to know how small we

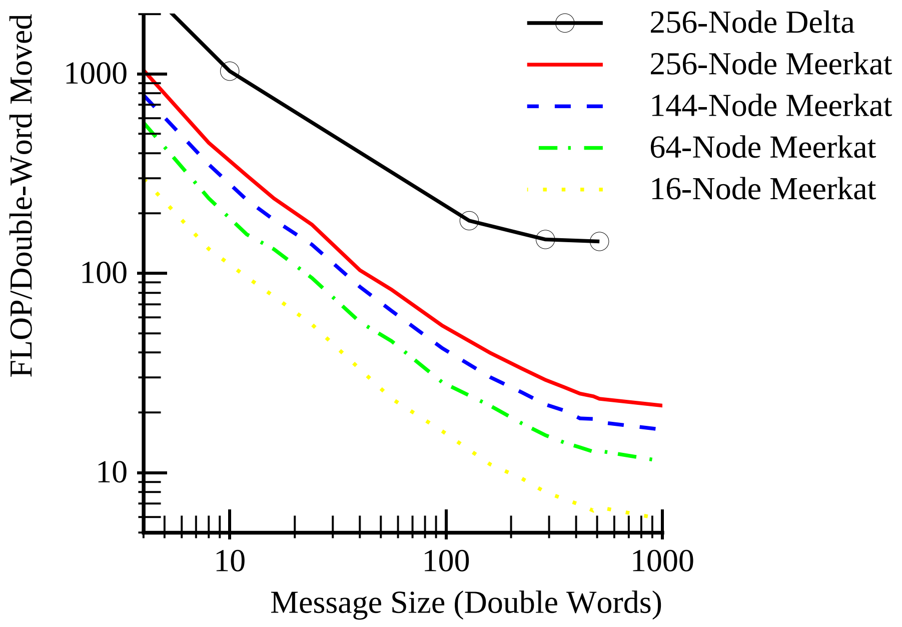


Figure 5.7: Meerkat and Delta Computation/Communication Ratio

can make the input to an FFT algorithm on a 64-node Meerkat without poor processor utilization. The input size is measured in points; each point requires two double words. Because FFT divides the input data evenly across all of the nodes, and communicates this data between pairs of nodes at each communication step, the input size will be 32 times the message size:

$$input\_size\_in\_points = message\_size\_in\_DW * \frac{64}{2}$$

From examination of the FFT algorithm, let us assume that we find that it has a comp/comm ratio of 100. By looking at Figure 5.7, we see that a 64-node Meerkat has a comp/comm ratio of 100 for messages of 25 words. For FFT to have messages of this size, the input data would have to be 32 times this, or about 1500 points. Larger inputs will yield longer messages and even better processor utilization.

The Delta curve never reaches down to a comp/comm of 100. This means that, for our FFT example, all input sizes will yield less than 50 percent processor utilization on Delta. The next section shows that measured speedup of FFT on Meerkat and Delta are consistent with this analysis and our assumption of a comp/comm ratio of 100.

J.P. Singh gives large-message ratios for several commercial multicomputers [77]. These are reproduced in Table 5.3, along with the values we measured on Delta and Meerkat. While we have listed our figures with Singh's, we note that his were derived from simple calculations, while ours started with measured bisection bandwidths. This makes the two sets of numbers not perfectly comparable, but we feel they are close enough to warrant rough comparison.

Table 5.3: Large-Message Computation/Communication Ratios For Meerkat and Several Commercial Systems (partly from J.P. Singh)

System	**Comp/Comm Ratio**
Meerkat (16 node)	5
Paragon	8
Meerkat (64 node)	11
Meerkat (144 node)	16
Meerkat (256 node)	20
CM-5 (Vector, nearest-neighbor)	50
CM-5 (Vector, random)	100
Delta (256 node)	144

Singh also gives the computation and communication functions for several representative numerical algorithms as a function of input and system sizes. These are shown in Table 5.4; N is the problem input size, P is the number of processors.

Table 5.4: Computation/Communication Ratios For Several Numerical Algorithms (From J.P. Singh)

Algorithm	**Computation**	**Communication**	**Comp/Comm Ratio**
LU Decomposition	N^3	$N^2\sqrt{P}$	$N/\sqrt{P}$
Conjugate Gradient	N^2	$N\sqrt{P}$	$N/\sqrt{P}$
FFT	$Nlog_2N$	$Nlog_2P$	$Nlog_2P$
Volume Rendering	N^3	N^3	constant

By comparing an algorithm's comp/comm ratio with the system's ratio, we can estimate how a particular problem and input size will run on a particular system.

5.9 Performance of 1-D FFT

FFT, a common computational problem for large parallel systems, often has poor performance, although speedup would be linear if communication cost was zero. Figure 5.8 shows Meerkat's and Delta's speedup achieved on a one-dimensional FFT [28] as a function of the number of nodes applied to the problem. Curves for two input sizes are shown: 4096 and 32768 points. The input points are evenly spread amongst the nodes; each point consists of a pair double precision floating point values.

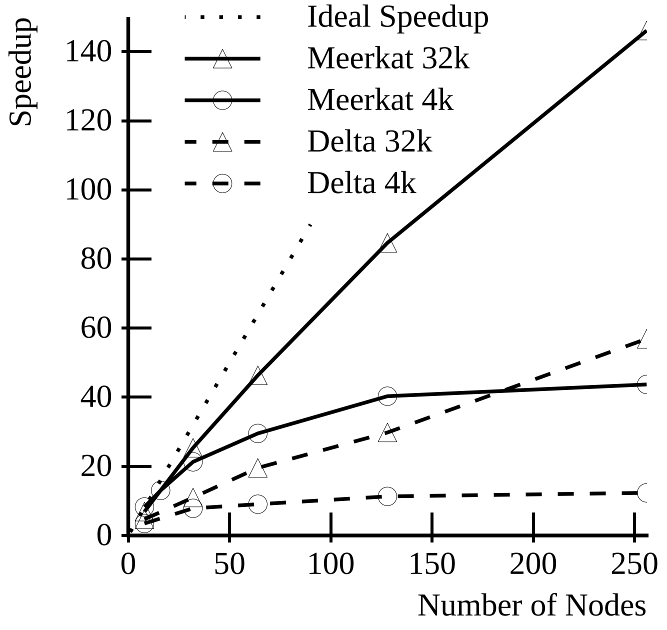


Figure 5.8: Speedup of 4k- and 32k-Point FFT

The total amount of computation is a function of the problem size, not the number of processors applied to the problem. With N points and P processors, there are $log_2 N$ steps. Each step takes time proportional to N, and the last $log_2 P$ steps require communication. Thus, as the number of processors increases, and the total running time decreases, the effect of communication may dominate. Also, while the number of messages each node sends per step is constant, the message size is proportional to $\frac{1}{P}$. This means that when the number of processors doubles, the data per message halves. Since the overhead of sending a message is fixed, the overhead per byte sent doubles when P doubles. In addition, another communication step must be performed when P doubles.

Figure 5.9 shows the internode bus contention of a 128-node Meerkat running a 4k-point FFT. The horizontal axis is the system cycle number of the simulated program, and the vertical axis shows the number of nodes that have failed in their request for an internode bus. The initial spike is due to the processors reaching the first communication step simultaneously. Contention falls off rapidly as requests are satisfied. The second of the seven ($log_2 128$) communication steps shows little contention, because contention during the first step skews the nodes: during the second step, nodes request internode buses at different times. During the last four steps, contention increases as the nodes resynchronize.

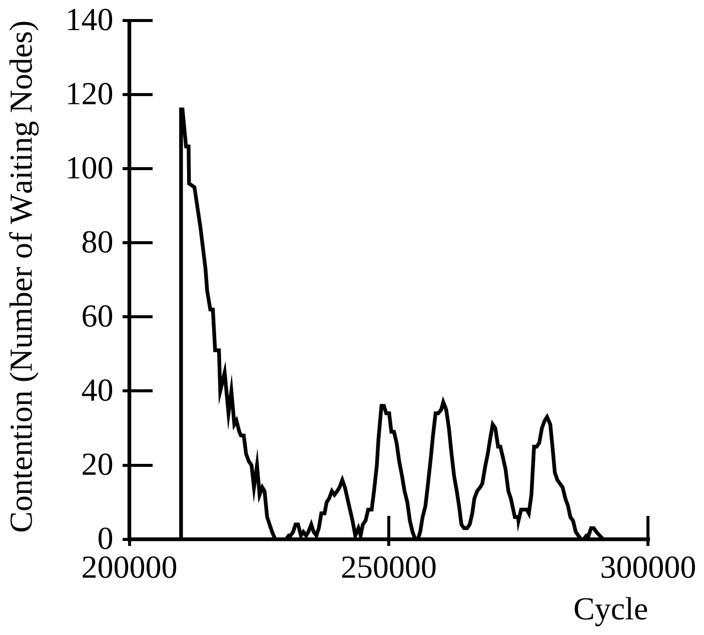


Figure 5.9: Internode Bus Contention during a 4k FFT on 128 Nodes

Meerkat achieves a higher speedup than Delta on both problem sizes because of its lower comp/comm ratio. The differences in these ratios derive from Delta's more powerful processors and Meerkat's faster interconnect.

5.10 Performance of SOR

Iterative parallel algorithms for solving systems of equations represent an important class of applications. We measured the speedup of one such algorithm: Red/Black Successive Over Relaxation [70]. The core of our test appears in Figure 5.10:

```
start = TIMERVAL();
for (iteration = 0 ; iteration < iterations ; iteration++) {
  register int i, j;

  send_black_border_values();
  receive_black_border_values();

  /* Compute new red values. */
  for (i = 1 ; i <= NPN ; i++)
    for (j = (i&1) ? 1 : 2 ; j <= NPN ; j += 2)
      X[i][j] = (1.0-omega)*X[i][j] +
        omega*0.25*(X[i-1][j] + X[i+1][j] + X[i][j+1] + X[i][j - 1]);

  send_red_border_values();
  receive_red_border_values();

  /* Compute new black values */
  for (i = 1 ; i <= NPN ; i++)
    for (j = (i&1) ? 2 : 1 ; j <= NPN ; j += 2)
      X[i][j] = (1.0-omega)*X[i][j] +
        omega*0.25*(X[i-1][j] + X[i+1][j] + X[i][j+1] + X[i][j - 1]);

  /* Test for convergence omitted */
}
end = TIMERVAL();
```

Figure 5.10: Core of Red/Black Successive Over Relaxation Program

We ran this algorithm on a 480 by 480 grid of double precision values with 1, 4, 16, 64, 144, and 256 nodes on both Meerkat and Delta. The algorithm divides the input matrix evenly, with each node getting an equal-sized block of contiguous values. Each node iteratively updates all the values in its block and then exchanges edge values with neighboring nodes. Since nodes communicate only with neighboring nodes, all communication is nearest-neighbor. This should run well on Delta, where the interconnect supports the simultaneous communication of each node with any of its four immediate neighbors without any contention. In contrast, Meerkat's sending nodes tie up buses while they communicate with their neighbors, locking out other nodes that wish to talk to their neighbors using the same bus.

To magnify Delta's advantage, we deliberately omitted the convergence test that would cause non-neighbor communication. Despite Delta's natural advantage on this workload, Figure 5.11 shows that Meerkat has a better speedup. Meerkat is faster for the same reason that it is faster on the parallel FFT: its comp/comm ratio is lower than Delta's. Delta's natural advantage on this communication pattern is not large enough to compensate for its less-efficient interconnect.

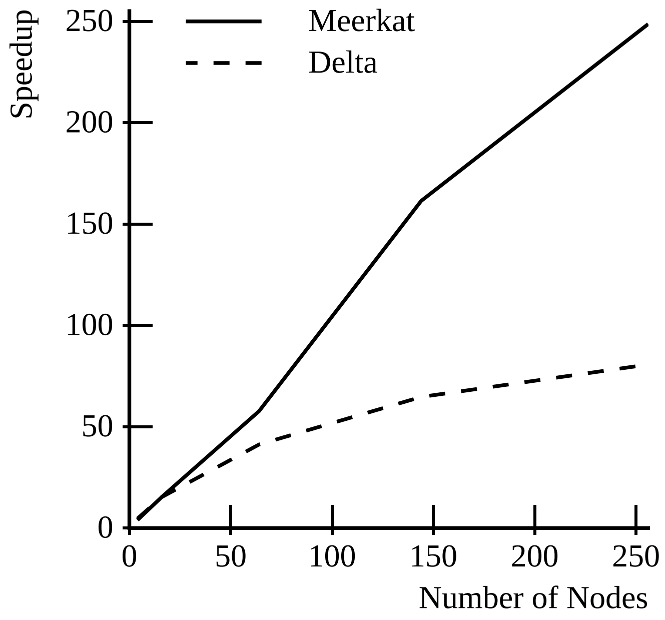


Figure 5.11: Red/Black Successive Over Relaxation Program Speedup

5.11 Performance of SIMPLE: A Fluid Dynamics Benchmark

Crowley et al. introduced SIMPLE in 1977 as a benchmark to evaluate new computers [29]. SIMPLE models the hydrodynamics of a pressurized fluid inside a spherical shell. It has been widely studied for a variety of purposes, including programming environment evaluation, computer performance evaluation, and portability studies [60].

The message-passing implementation we used modeled 4096 points spread evenly across the nodes. The computation is similar to that of SOR: communication is all nearest-neighbor and alternates between computation and communication phases. Unlike SOR, however, there is one phase during which only one row or column of processors can be computing at a time. Thus, during this phase, the parallelism drops to $\sqrt{P}$.

Like the speedup curves for FFT and SOR, Figure 5.12 shows Meerkat's speedup exceeding Delta's. The gap between the two speedup curves grows with increases in the number of processors applied to the problem. Delta's absolute performance is five times than of Meerkat for the single-processor case. We believe that this is due to SIMPLE's heavy use of floating point square root and divide, which are much faster on the Delta's i860 processors than on Meerkat's MC88100's.

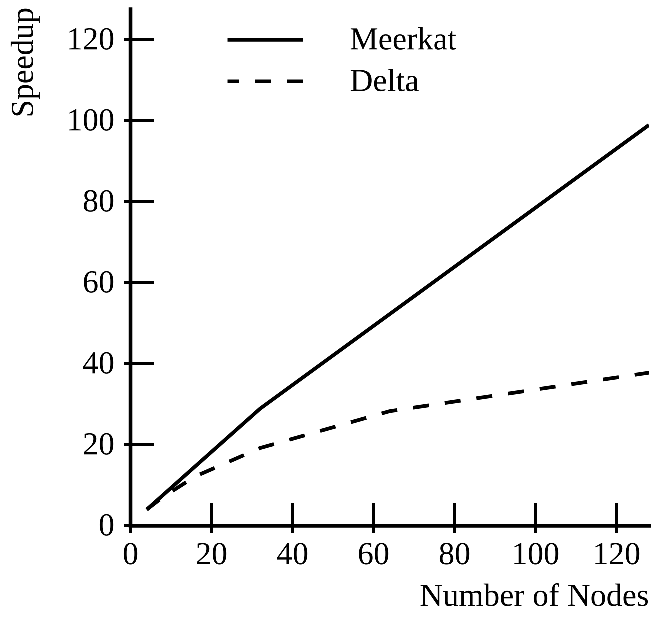


Figure 5.12: SIMPLE Speedup

5.12 Summary

This chapter demonstrated that Meerkat is an effective multicomputer by comparing it to Intel's Touchstone Delta, a multicomputer widely regarded as effective in solving large numerical problems. Differences between the Meerkat and Delta complicated our comparison. However, the differences did not preclude an evaluation of the overall running time of a series of benchmarks on the two systems.

Our benchmarks were message-passing programs. We described the semantics of the message-passing primitives these programs use, and their Meerkat implementation. For Meerkat, we displayed a breakdown of the time spent in the various steps of these primitives. It was not possible to make these detailed measurements for Delta.

The performance of both systems is a function of message size. Benchmark running times were therefore measured over a range of message sizes. Under a light load, Meerkat drove its interconnect nine times faster than Delta. In doing so, Meerkat's injection rate was close to its theoretical maximum, which is its peak local memory bandwidth. Delta's injection rate, one ninth that of Meerkat, was a small fraction of its theoretical maximum.

Under a heavy load, 256-node Meerkat's had three times the bisection bandwidth of the same-size Delta. Delta's communication links operated at a fifth of the rate for which they were designed. Meerkat's buses operated over three times faster and achieved 83 percent of their peak rate.

We related these performance figures to application performance by considering the ratio of computation to communication required by applications and the ratio multicomputers can deliver. We presented curves that show the ratio of several sizes of Meerkat and Delta as a function of message size and gave the ratios for several representative algorithms.

To understand what these advantages mean for application performance, we measured the speedup of several numerical applications on Meerkat and Delta: 4k-point FFT, 32k-point FFT, SOR, and SIMPLE. Meerkat's speedups on the first three of these problems were 3.5, 2.5, and 3 times higher than Delta's, at 256 nodes. At 128 nodes, SIMPLE's speedup on Meerkat was double that of its speedup on Delta. Despite Meerkat's slower processors, the absolute running time of a 256-node Meerkat on all tests, except SIMPLE, was smaller than the same sized Delta.

Chapter 6

THE MEERKAT 2 NETWORK INTERFACE

Meerkat is the result of a minimalist design philosophy, one that attempts to optimize cost/performance and keep design time short. It is at the extreme end of the multicomputer design space, one where all possible communication functions are moved into software, leaving just enough hardware to sustain high internode bandwidth.

Chapter 5 showed that despite this minimal hardware, Meerkat performs well. However, statistics gathered by the simulator show that internode bus utilization is low when short messages predominate. In developing Meerkat, we knew that some applications would suffer as a result of our extreme approach. By being less extreme and allowing a slightly more complex network interface, could we retain Meerkat's essential benefits, yet substantially improve performance on programs that use short messages? The answer proved to be "Yes."

This chapter introduces Meerkat-2, a new network interface tailored to circuit-switched interconnects. On all workloads, Meerkat-2 performs better than its predecessor, especially on workloads with short messages. Section 6.1 outlines the models and measurements detailed in this chapter. Section 6.2 describes the Meerkat-2 interface, operating system implications, use of the interface by user-space code, and how Meerkat-2 compares to other network interfaces. Section 6.3 relates performance results of running benchmarks described in Chapter 5 on the network interface alternatives.

6.1 Introduction

In the past, multicomputer designers focused their efforts on the interconnect and node processor. The network interface, seen as significant to the operating system only, was deemed incidental to performance. More recently, designers have recognized the benefits of streamlining network access [30] and allowing application code direct access to the network interface [45, 38].

To evaluate tradeoffs in network interface designs, we use the Meerkat simulator and the same suite of parallel applications described in Chapter 5. We augment the original Meerkat simulator with device models for three additional network interfaces. Each interface device model has corresponding run-time interface code that interprets the message-passing primitives. We run the applications on the simulator and measure the elapsed simulation time. From these measurements, we derive data bandwidth for some tests and speedups for others.

Our four network interface models represent different hardware and software tradeoffs:

1. The most hardware-lean and software-intensive model, called *Meerkat-1*, was described in Chapter 2.
2. *Meerkat-2*, a new interface model that makes more efficient use of the interconnect, requires more hardware than Meerkat-1, but has better performance on small messages.
3. To bound the improvement attainable by putting more function in hardware, we measure *Meerkat-msg*, a Meerkat with the entire message-passing system in hardware.
4. To bound the improvement attainable by making the interconnect faster, we model *Meerkat-infinite*, a variant of Meerkat-msg with an interconnect that is infinitely fast (i.e., no contention and zero time for transmittal of any size message).

Although all of our experiments use message-passing benchmarks, the Meerkat-1 and Meerkat-2 interfaces are not specific to this programming model. These interfaces could be programmed to support efficiently remote procedure call [14], distributed-shared-memory [59], or remote write [27]. Meerkat-msg and Meerkat-infinite, on the other hand, directly support message passing and thus would be less well suited to other programming styles.

6.2 Meerkat-2 Network Interface Architecture

This section identifies problems encountered with the Meerkat-1 interface that gave rise to Meerkat-2. We then describe the Meerkat-2 interface to the operating system and applications.

6.2.1 Rationale for a New Network Interface

Meerkat-1 requires little hardware and shows excellent performance compared to systems such as Intel's Delta. As expected, programs making heavy use of small messages spend a significant proportion of their execution time in the message run-time system.

In Meerkat-1, the sending node's processor: (1) arbitrates for the required circuit, (2) signals the receiving node, (3) waits to rendezvous with the receiving node, and (4) sends data. After arbitration and until data flow, the circuit is active but idle. For long messages, idle time is small compared to overall message time: we achieve 83 percent interconnect utilization for long messages. However, for workloads with many short messages, the delay while circuit-owning nodes rendezvous dominates communication time.

We examined how to improve performance of programs that use small messages. We realized that we could simplify the system software and improve performance on small messages by accepting a modest increase in network interface hardware. In some cases extra hardware would be worth the improved performance on small messages; in other cases it would not. The desire for better performance on small messages led us to design Meerkat-2.

Meerkat-2 makes more efficient use of the interconnect through hardware handling of circuit management (establishment through disconnection). The Meerkat-2 network interface contains a set of registers that define the entire transfer. Communication software on a sending node loads these registers; hardware completes the send operation. Efficiency is promoted because the hardware holds a circuit for the duration of the transfer only.

Like Meerkat-1, Meerkat-2 requires the sending and receiving nodes to rendezvous. However, rendezvous time overlaps with data transmission: Meerkat-2 contains a FIFO in the network interface to buffer incoming data until the receiving node decides where to place it. Figure 6.1 shows a Meerkat-2 node (which resembles Meerkat-1, except for the addition of the FIFO).

To summarize these differences, then, Meerkat-2 eliminates dead time on internode buses in two ways: (1) the sender has registers that describe a whole transfer, and (2) the receiver has a FIFO that overlaps the receipt of data with execution of the receiver's interrupt handler.

In addition to having better performance and requiring simpler low-level software, the Meerkat-2 interface allows safe user-space access and is compatible with virtual memory.

6.2.2 Programmer's View of Meerkat-2

Node programs see the Meerkat-2 network interface as a small set of registers, as shown in Figure 6.2. These registers are divided into the following groups: *sending user-space*, *receiving user-space*, and *system-space*. As their names imply, user registers are accessible

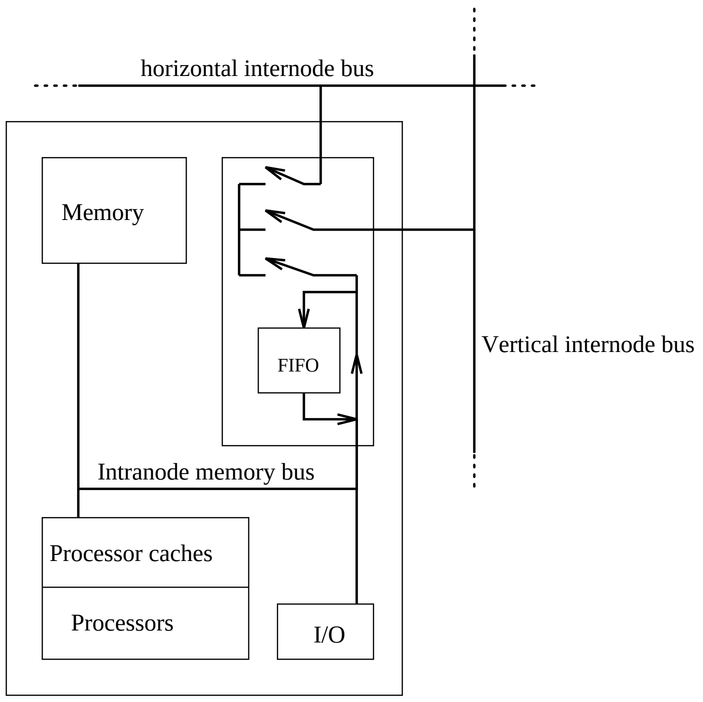


Figure 6.1: Data Paths and Major Components of a Meerkat-2 Node

in user space only. Application programs need just these registers to communicate. System registers regulate application access to the network and ensure that applications do not interfere with each other or the operating system.

To send data, a sending node writes zero or more of the first six send-user registers. It then writes the destination node register to initiate internode bus arbitration, which creates a connection between the sender and receiver. The sender usually places control information into the immediate data registers. If the data do not fit in these four registers, the sender also loads: (1) the virtual address register with the base address in memory of the data to be sent, and (2) the length register with the extent of the data. If the length register is zero, no memory access occurs. The network interface sets the length register to zero after every send operation; this register does not need to be loaded if no memory data are needed.

When the required connection is established and the receiver's FIFO empty, the receiver signals the sender to proceed. If the sender is told not to send, it drops the connection and tries again after a brief delay. When the sender finds the receiver ready, it copies its immediate data registers into the the receiver's corresponding registers and then sends the memory data. The receiving node can either poll for incoming data or take an interrupt upon

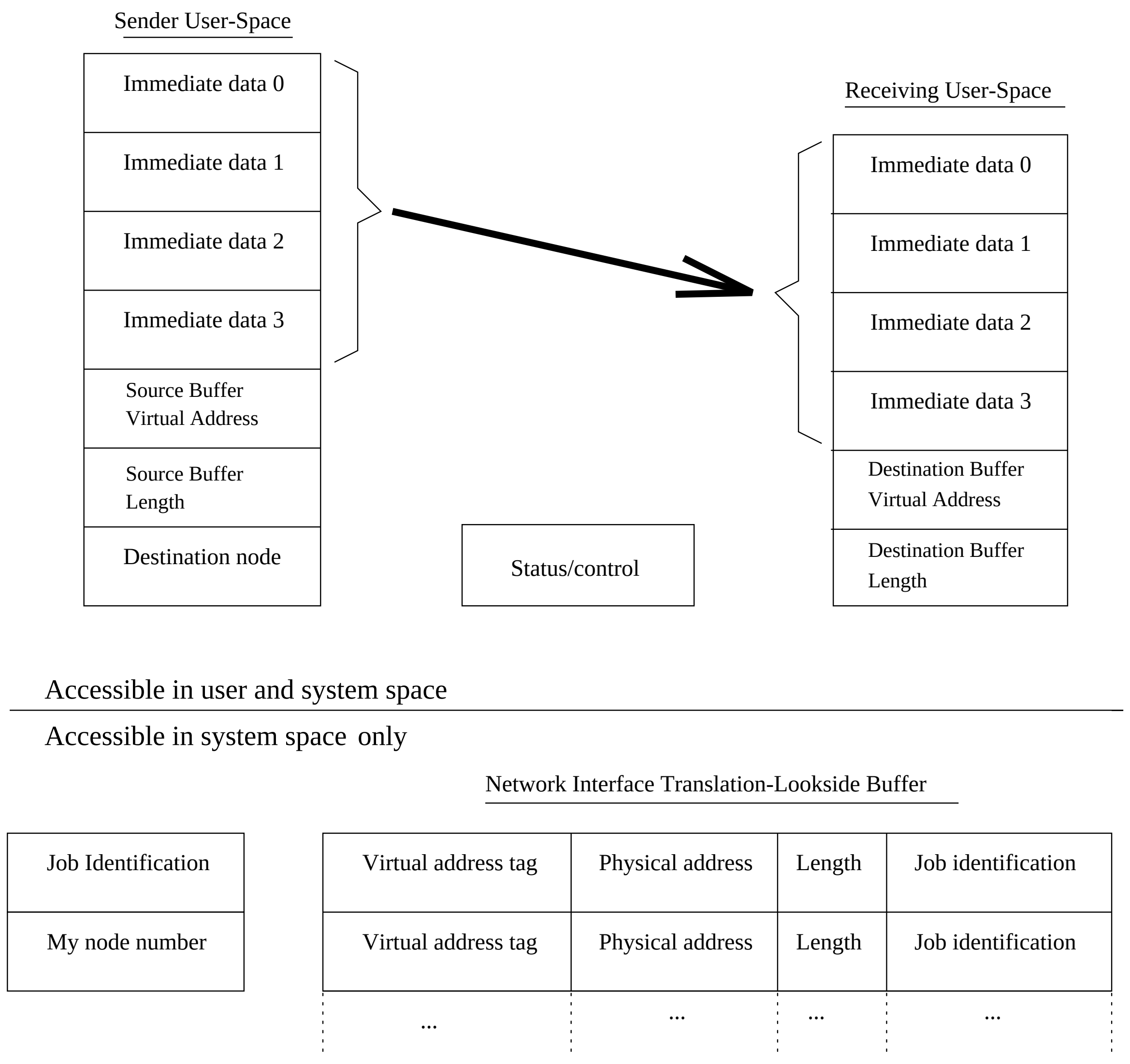


Figure 6.2: Meerkat-2 Network Interface Registers

receiving data. When the data arrive, the receiver's immediate data registers are loaded with the values sent through the interconnect. Its processor inspects control information in these registers to decide what to do with the rest of the information, if any. Additional data are buffered in the receiving node's network interface FIFO until its processor loads the receive virtual address and length registers. When the processor writes the length register, buffered data are written to memory.

The receiving node has a finite amount of time in which to determine where the data should be placed in memory. If this time is exceeded, the FIFO overflows and an error indication is passed back to the sending node. That node must check a status bit in the status/control register to ensure that the data were received successfully. If they were not, the sender repeats the send operation by rewriting the destination node register. The FIFO should be large enough so that performance does not suffer if data must be resent due to buffer overflows.

The sending node can determine when an internode transfer is complete in two ways: (1) it can poll a bit in the status register, or (2) it can take an interrupt. The interrupt option is useful if the transfer is large or if network queueing delays are long. During long data transfers, the processor may perform poorly if starved for access to memory. This will be true in systems where the interconnect injection rate is close to the peak local memory bandwidth.

To avoid the receive-interrupt cost, a user-space program can set the 'please-don't-interrupt' bit in the control register and then poll for incoming data. Data for the operating system, or for jobs other than the one currently running, will cause an interrupt regardless of the value of the 'please-don't-interrupt' bit. The network interface determines whether incoming data support the currently running application by comparing the job id register with the job id transmitted when the connection is established. If they match, the application is free to accept the received data. If not, an interrupt is generated, and the operating system must determine what to do. The received data may be operating system traffic that it will handle directly, or it may be data for another job, one that is sharing the system with the currently running job.

6.2.3 Operating System Implications of Meerkat-2

The Meerkat-2 interface requires operating system support to allocate send and receive buffers and to handle context switches. The next two sections discuss these problems and outline solutions.

We assume that a Meerkat-2 operating system will use gang scheduling of jobs running on multiple nodes. This means that when a job is running on one node, it is running on all nodes on which it is scheduled. Job switching requires that all nodes switch at the same time. Because nodes do not execute in lock-step it is possible that, for a brief time, a job may be running on one node, but not another. Meerkat-2 is robust in the face of this skew.

Send and Receive Data Buffers

The network interface contains a Translation Buffer (TB) that turns virtual addresses in the send-user and receive-user virtual address registers into physical addresses required by the hardware. The TB also allows the operating system to recognize which parts of memory a user-space program is using for buffers. Identification of send and receiver buffers is important for two reasons. First, the operating system must pin the buffer pages, i.e., freeze the mapping of virtual to physical address for buffer pages. Second, to simplify the hardware, we do not support scatter/gather. That is, the data sent from a sender's memory or loaded into a receiver's memory must be physically contiguous.

When a send or receive operation is attempted, a TLB lookup is performed. The first such operation on a given buffer causes a TLB fault. The fault handler looks at the virtual address and length of the buffer and:

1. Ensures that there are physical pages to back the virtual memory of the buffer.
2. Ensures that the physical pages are physically contiguous.
3. Flags the pages as pinned. This ensures that the pages will not be moved or paged out by the operating system.

The operating system swaps non-contiguous buffer pages with ones that are. Swapping overhead can be avoided in several ways. First, the operating system can use a careful allocation policy for all memory that may become a buffer. It may restrict the amount of such memory by requiring that the user notify the operating system of the location of its buffers when the user program is loaded. Such a policy attempts to allocate physically contiguous pages for virtually contiguous pages. For cases where this is was not possible, and a discontinuity occurs in a buffer, the operating system would swap page contents. Less than half of the pages would require swapping unless there were more than one discontinuity. Finally, swapping occurs only the first time the buffer is used.

A second way to avoid page-swapping is to allocate the memory for message buffers carefully. Such a policy would be implemented in a special allocation function for message buffers or in a general-purpose allocation function, such as `malloc()`. The location of physical discontinuities is passed from the operating system to the user-level allocation functions to avoid allocating any buffer that crosses a discontinuity.

Perhaps the best approach combines the operating system and user-level methods. The former reduces the number of physical discontinuities in the user space's virtual map. The latter is then able to avoid the few remaining discontinuities with little effort.

Druschel and Peterson describe an operating system facility to support a user-level allocation function for I/O and inter-domain buffers [36]. Their facility resembles ours in that its goals are high-performance I/O, and it involves a careful user-level buffer allocator working in concert with the operating system. It differs from our approach in that its buffers are passed between domains, whereas our allocator simply avoids physical discontinuities in user buffers.

Context Switches

A processor may take an interrupt and switch to another user-space program at any point during an internode send or receive. In this event, the system must ensure that communication is reliable. When network interface registers are active, we say that the application is in a *critical section*. The operating system must ensure that the application executes properly in the presence of interrupts that occur in critical sections. We first consider interrupts on the sending node, and then on the receiver node.

The operating system can handle an interrupted sender in two ways. First, it can save and restore the user-send registers on every context switch. Processor registers are usually handled in this way. However, this approach slows down every context switch, and it reduces overlap of communication and computation. A faster method is to invalidate partially constructed send operations with a single store of an abort code to the status/control register on every context switch. The operating system can do this even if a send is in progress.

Writing the abort code to the status/control register causes the first subsequent data-send operation to fail. To understand how the operating system and message library interact, consider an example. Suppose process A resumes after having loaded some of the user-send registers. Also suppose that those registers now have different values: while A was suspended, process B used the network interface, overwriting A's values. Process A, unaware of this, writes the rest of the user-send registers and begins the send by writing the length register. Because the operating system wrote the abort code to the status/control register just before it resumed A, the send immediately fails. The sender reloads all of the registers. The second time, the send will most likely succeed since the probability is low of any particular send failing. The sender already has to check the status register, as we described in Section 6.2.2.

In the common case, for which the send is not interrupted, the cost is only one extra store per context switch. If the send is interrupted, the sending processor will have to reload the user-send registers only once. Both cases require little overhead to support context switching. The send-register loading code resembles a restartable atomic sequence, in that it will be retried if a context switch occurs during its execution [13].

Another possible solution is called *roll-forward*. With this technique, the operating system ensures that it does not context-switch while the network interface registers are in use. It does this by testing for active interface registers when it is ready to context switch. If they are active, the operating system lets the application execute until it is finished its internode operation and the interface registers are no longer active. This technique is used to prevent context switches of user-level threads that are in critical sections [4].

On the receiving node, a context-switch interrupt may occur after data starts filling the FIFO and before the receiving node loads the receive address register. If the FIFO overflows because the context-switch interrupt takes too long, the sender is signaled with a negative acknowledgement. This will cause the sender to try again. If the FIFO does not overflow, the operating system must unload the FIFO into a system buffer, just as it would for any other received data intended for an inactive process.

Data can arrive for a receiving user process while the operating system or some other user process is running. In this case, the operating system buffers the data. When the recipient user process resumes, it is signalled that the operating system has data for it. It then calls the operating system to retrieve the data. This mismatch between arriving data and running job will be rare with gang scheduling.

6.2.4 Implementing Message Primitives on Meerkat-2

`Csend` writes: (1) the message type and length into send-immediate registers, (2) the buffer address and length into the send-virtual address and length registers, and (3) the destination node number into the destination node register. The `csend` code then polls the status register and either exits on success or repeats the send on failure.

`Irecv` checks for a message with a matching type. If one has arrived, it copies the message to the user buffer and returns a value that will let a subsequent call to `msgwait` know that a message has arrived. If there are no messages, `irecv` allocates a pending-receive record and puts this record, which contains the user's buffer description and the expected message type, on a list of similar records.

The receiver can wait for data by waiting for an interrupt or for a status register bit to change value. The former method is better when the receiver has other work it can do while waiting. The latter method is better when the receiver wants to avoid the interrupt latency and has no other useful work to do in the interim.

The interrupt or polling routine reads the incoming message type from a receiver-immediate register and seeks a match on the pending-receive list. When a match is found: (1) buffer lengths are checked to ensure the incoming message will fit in the user-supplied buffer, and (2) the buffer address and length are loaded into the receive virtual address and length registers. The pending record is marked 'done,' so that `msgwait` will know that a matching message has been received and placed in the user's buffer. If a pending receive is not found, the message is placed on a list of messages that have arrived but remain unclaimed by applications.

6.2.5 Comparison With Other Network Interfaces

Network interfaces for both multicomputers and local area networks vary in many ways. In this section we describe some of the variations and compare the Meerkat-1 and Meerkat-2 interfaces with existing ones.

One way to categorize network interfaces is by whether they are *user-safe*. A user-safe interface can be used directly by user-space software running in a multi-user environment. An interface that is not user-safe, on the other hand, cannot be exposed to untrusted software. Because of this, user-space software running on a system with a non-user-safe interface must let the operating system act as an intermediary. This level of indirection hurts performance in several ways.

First, transferring control to the operating system in order to send or receive data causes context switches. Anderson et al. argue that the relative cost of context switches is increasing with advances in processor technology [3]. Second, the operating system must check that the communication is valid. Third, the operating system is written for the general case: its general code is slower than the tailored code in user-space programs. However, Thekkath and Levy propose injecting tailored code into the operating system to mitigate the slowness [91]. This is not always easy to do, and it lessens only one of the costs of non-user-safe network interfaces.

Examples of systems with network interfaces that are not user-safe include workstations, Intel parallel systems, and Meerkat-1. The CM-5 and Meerkat-2 have user-safe interfaces. Henry and Joerg proposes extensions to their network interface to make it user-safe [45].

A second way to categorize network interfaces is by the way in which data are moved into them. We present a number of such categories:

- *Asynchronous-DMA*. The processor programs the controller to read or write a single block of system memory. The controller then performs the transfer while the processor executes independently. The controller interrupts the processor when the transfer is complete. The Intel iPSC/2 and iPSC/860 have this type of interface.
- *Synchronous-DMA*. The processor programs the controller to read or write a single block of system memory. The controller then performs the transfer while the processor stalls. No interrupt is needed, because the processor knows that the transfer is complete when it continues past the DMA-initiating point. The Meerkat-1 interface has this property.
- *Scatter/gather-DMA*. This resembles asynchronous-DMA, except that the controller can move data to and from multiple regions of memory. The DEC AN1 controller [79] is a recent example of such a controller.
- *Limited-DMA*. This resembles asynchronous-DMA, except that the processor can access only a small, dedicated region of memory located on the controller. It is typically hundreds of kilobytes in length. The DECstation PMADD-AA TurboChannel Ethernet interface [33] is an example of a limited-DMA interface.
- *Separate-DMA*. This resembles limited-DMA, except that the processor cannot access the small buffer memory that the controller can access directly. Data move between the separate buffer memory and system memory by *remote-DMA*. Separate-DMA network interfaces support a memory-to-memory copy operation to copy data between the separate buffer and system memory. Although the processor cannot access the separate buffer memory, it must manage allocation of this memory. The National Semiconductor NS8390 Ethernet controller [68] supports both DMA into an isolated memory buffer and DMA between this buffer and system memory.
- *FIFO*. The processor reads and writes a single location that maps to the ends of a pair of FIFOs. The FIFOs buffer data between the processor and the network. The Intel Delta [50] and Fore System's ATM interface [39] use this type of interface.
- *Fixed-register*. The processor can access a small array of memory in the interface. The processor writes data into this array and then touches a location to start the transfer. The entire contents of the array are transmitted. This differs from limited-DMA because the number of memory locations in a fixed-register interface is on the

order of tens of words, and the whole array is sent. In a limited-DMA interface, the buffer is on the order of tens of thousands of words, and the operating system must specify both the starting address and the data's length.

- *Protocol-processor*. A separate processor dedicated to handling network traffic mediates between the node's main processor and the network. This allows a wide variety of interface styles to be exposed to the processor. Examples of systems with a protocol processor that handles internode traffic include the FLASH multiprocessor [55], Intel's Delta, and Nectar [6].

DMA interfaces allow the full bandwidth of system memory to be coupled to the interconnect, but they also present complexities that FIFO interfaces do not. FIFO interfaces usually require less overhead to transmit and receive short messages. DMA interfaces perform better when the interconnect bandwidth exceeds the rate at which the processor can move data into a FIFO or register interface. However, DMA interfaces present complexities with cache coherency and page pinning that FIFO interfaces do not. If DMA transfers are not kept coherent by the hardware, as in Meerkat-1, the software must explicitly flush the cache. DMA buffers must be pinned in memory. These are not difficult problems, but they do not exist with FIFO interfaces.

Meerkat-2 is a hybrid between an asynchronous-DMA and a register interface. We included DMA to couple the full memory bandwidth to the interconnect. Meerkat-2's four immediate registers transmit a small amount of control information without the overhead of putting it in memory and pointing DMA registers at this memory. The inclusion of immediate registers also allows the building of an interface without scatter/gather. Scatter/gather is often used to send a few control words followed by a mass of data.

Meerkat-2 resembles the Alewife message interface [54], except that the latter is more general. Meerkat-2 has a fixed number of registers and cannot perform scatter/gather. The Alewife interface has: (1) an array of registers that can hold immediate data, and (2) address/length pairs that describe regions of memory. Thus, Alewife lets the user decide how many immediate parameters to send and whether the interface will transmit data from non-contiguous memory blocks.

We opted for a simpler scheme to shorten the hardware design time. The number of immediate registers is arbitrary. Our message-passing system requires three registers. We proposed that the network interface have four registers, because this number was sufficient for several communication schemes while keeping the hardware cost low. The registers hold control information that:

- Has a short lifetime, and thus does not need to be stored in memory.

- Is needed by the receiving processor to determine where incoming data should be placed. The receiving processor can make this determination faster if it can read the control information directly from the network interfaces.
- Is difficult to place in memory next to data buffers. Data buffers are often allocated by users; unrelated user data may precede the buffers. Two solutions we did not chose were to: (1) support scatter/gather, which we avoid to simplify the hardware, and (2) copy the user data into a system buffer, which was expensive. We avoided the complexity of scatter/gather, and the execution time of copying data, by providing immediate control registers that obviate the need to store the control information in memory.

6.3 Performance

This section compares the performance of several network interface models that represent distinct design points. First, we briefly recapitulate the differences among network interface models. We then assess each model's performance on four benchmarks.

6.3.1 Meerkat-1, Meerkat-2, Meerkat-msg, Meerkat-infinite

Meerkat-1 requires the processor to do tens of loads and stores of network control registers to arbitrate for the required buses, signal the receiving node, and transfer data. During heavy internode bus contention, nodes waiting for an internode bus must wait for the node that owns the bus to execute instructions to carry out its communication. When message payloads are large, this overhead is amortized over many bytes, and bus utilization reaches 83 percent. However, payloads of messages as large as 1000 bytes result in bus utilization of around 50 percent.

Meerkat-2 requires far fewer manipulations of interface control registers to transfer data. Moreover, once the registers are loaded, hardware can complete the communication without processor intervention. This both decreases processor overhead for communication and greatly improves bus utilization for small messages. We model the Meerkat-2 with an infinite FIFO. This assumes that, in a real system, the FIFO would be large enough, and the receive-interrupt-service routine fast enough, to make the number of FIFO overflows insignificant.

Meerkat-msg resembles Meerkat-2, except that the entire message-passing system is implemented in the network interface. Thus, message transmission and reception consume only a few processor cycles. We do not envision any implementor actually building Meerkat-msg, because it is tied to one model and would be complex to implement in hardware. It is interesting because it is at an extreme end of the network interface design space. It bounds how much the performance of Meerkat-2 could improve by putting more function in hardware.

Meerkat-infinite has the same network interface as Meerkat-msg and a network with zero time to communicate any message. Meerkat-infinite is impossible to realize, but is interesting because, like Meerkat-msg, it gives an upper bound on performance. By comparing the speedups of programs running on Meerkat-infinite with the same programs running on Meerkat-msg, we can see how much performance could improve if we improved the network itself. This last model helps us calibrate our benchmark programs.

Meerkat-2 and Meerkat-msg have the same network as Meerkat-1. The network moves four bytes per clock tick (50nsec) during data transfer, and thus has a transfer rate of 80 MB/sec.

6.3.2 Bandwidth under Light Load

Figure 6.3 shows the performance of the Meerkat models and the Delta running the light-load bandwidth test (described in Section 5.5). The Delta and Meerkat-1 curves are the same as those in shown in Figure 5.3.

Meerkat-1 peaks at 67 MB/sec, whereas Meerkat-2 and Meerkat-msg peak at close to 80 MB/sec. The reason for this difference is that Meerkat-1 can send at most 4,096 bytes in one packet and requires the communicating nodes to synchronize before each packet. Meerkat-2 and Meerkat-msg do not require their run-time systems to packetize the data, and so achieve bandwidths closer to the peak rate.

The Meerkat-infinite curve is shown for small messages. This curve is a straight line that has no limit, since it represents the bandwidth of a Meerkat with an infinitely fast interconnect. The curve is not at infinity, because there is a small cost to launching the send and posting the receive. Bandwidth is limited by the need to pay this small fixed cost.

Meerkat-2 has over twice the bandwidth and half the latency of Meerkat-1 on small messages. More importantly, as later graphs show, Meerkat-2 uses the interconnect more efficiently under heavy traffic.

6.3.3 Bandwidth under Heavy Load

Figure 6.4 shows the bisection bandwidth in megabytes per second of a 256-node Meerkat and a 256-node Delta. The heavy-load bandwidth test used in this measurement is described

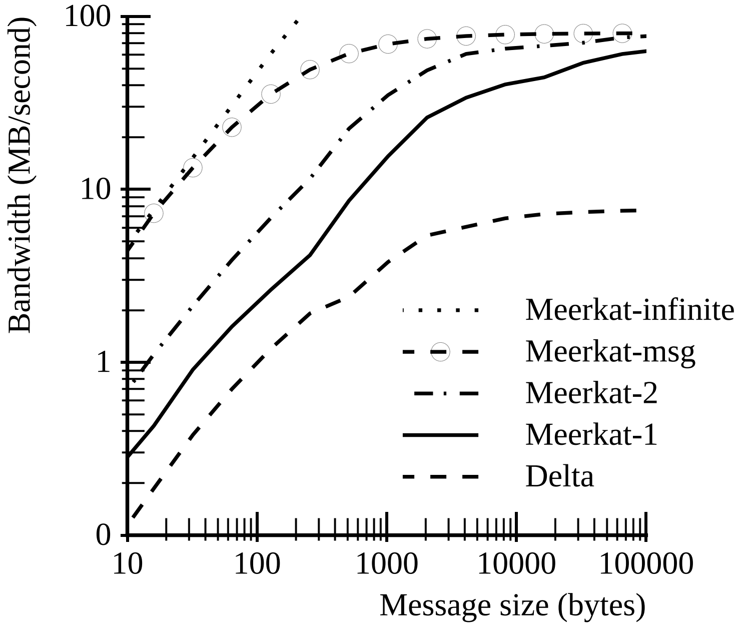


Figure 6.3: Two-Node Bandwidth (under Light Load)

in Section 5.6.

The difference between Meerkat-1 and Meerkat-2 is more pronounced in this test than in the light-load bandwidth test. For 32-byte messages in the light-load test, Meerkat-2 achieves twice the bandwidth of Meerkat-1. On the heavy-load test for the same message size, Meerkat-2 reaches nearly eight times the bandwidth of Meerkat-1.

The improvement in the light-load case is due to time spent by the processors in low-level network interface code. The improvement in the heavy-load case is due to the reduction by a factor of roughly eight in the time a node spends waiting to acquire internode buses. Our goal in designing the Meerkat-2 network interface was to improve the efficiency of network connection utilization, and the heavy-load test proves that we were successful.

6.3.4 Speed-up of FFT

The low-level performance results reported in the prior sections show the raw performance of various network interfaces. We now examine the application-level performance implications of our interface models.

In particular, we discuss the performance of a one-dimensional FFT on two different

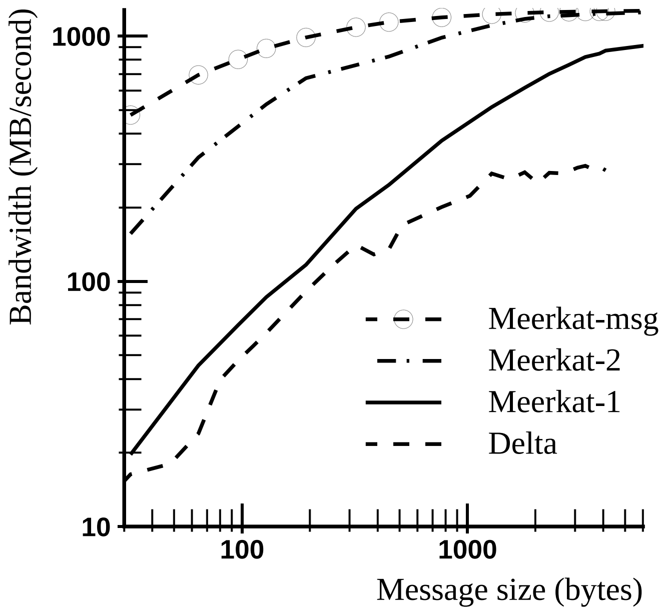


Figure 6.4: 256-Node System Bisection Bandwidth (under Heavy Load)

problem sizes: 4,096 and 32,768 points. (This is the same test program described in Section 5.9.) Our metric is speedup, the ratio of the time the algorithm takes on one node to the time it takes on multiple nodes. We plot the number of nodes on the horizontal axis, and the speedup ratio on the vertical axis.

Figure 6.5 shows that the Meerkat-2 interface has a much better speedup than Meerkat-1 and close to the speedup of Meerkat-msg. As the previous section showed, Meerkat-2 is able to use the interconnect more efficiently and thus cause shorter queueing delays for nodes contending for the interconnect.

Figure 6.6 shows that the benefits of Meerkat-2 over Meerkat-1 are less pronounced, but still noticeable, on a larger message size. In this test there is a total of 32,768 points, which results in a total of 524,288 bytes of data system-wide. At the largest system size we model, 256 nodes, each node has 2048 bytes of data. There are eight parallel steps, each of which consists of node pairs that differ in node address by one bit exchanging messages of 2048 bytes. At this message size, the higher fixed cost of Meerkat-1's communication software is amortized over these large messages. As a result, Meerkat-2's advantage is not as pronounced as it is in the case of the 4k FFT, where messages are 256 bytes.

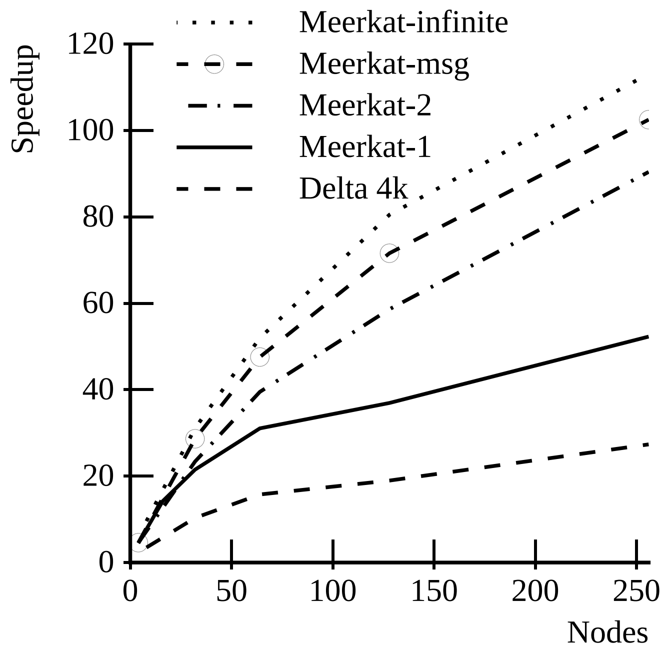


Figure 6.5: Speedup of 4k-Point FFT

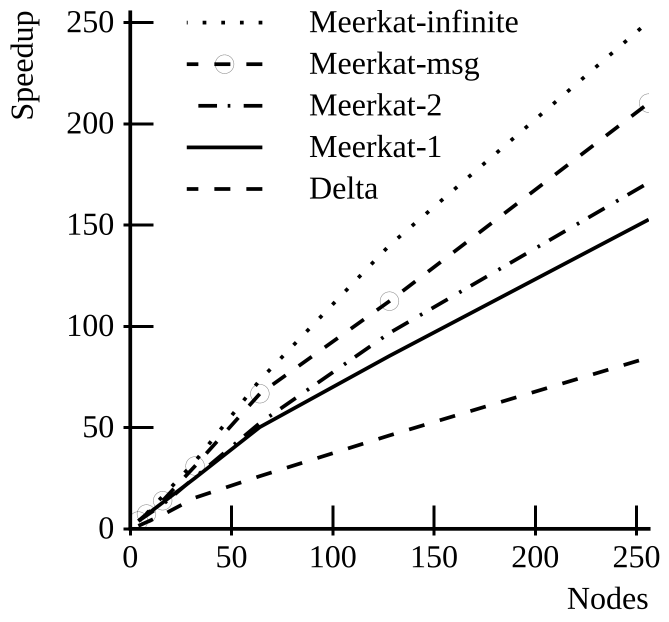


Figure 6.6: Speedup of 32k-Point FFT

6.3.5 Performance of SOR

Figure 6.7 shows the speedup of a SOR on up to 256 nodes [70]. The relationship between the curves is similar to those for the 4k FFT (Figure 6.5). Meerkat-2's advantage is significant only when the number of nodes is large.

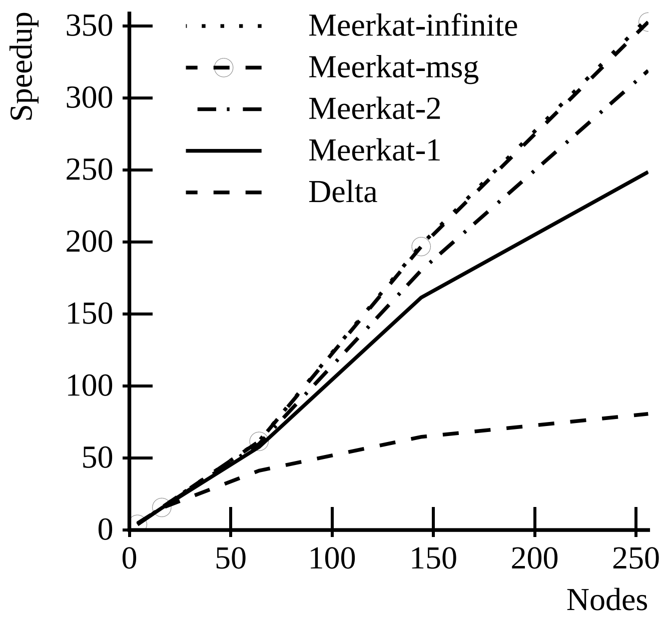


Figure 6.7: Speedup of R/B SOR

Note that the curves for Meerkat-msg and Meerkat-infinite are coincident. This indicates that the network itself is lightly loaded in this test. As a result, Meerkat-2's advantage over Meerkat-1 is due to the reduced processor overhead in running the message-passing code. The reduction is modest compared to the reduced bus waiting times for tests that cause heavy loading, such as that shown in Figure 6.4.

6.4 Summary

We compared the performance of four network interface models on several benchmarks. One of the models, Meerkat-1, corresponds to the system described in Chapter 2. As expected, Meerkat-1's performance suffers under a heavy load of small messages for two reasons: (1) Meerkat-1 requires significant processor overhead, and (2) during rendezvous in Meerkat-1, internode resources are active but carrying no data. When messages are small and traffic is heavy, nodes will queue waiting for a bus that is being underused. These problems led to the development of Meerkat-2.

Meerkat-2 modestly improves performance when the interconnect is loaded lightly and messages are long. Its relative performance improves for many small messages under a light load. When the load increases, however, Meerkat-2's performance is eight times better on small messages. In addition to better performance, Meerkat-2 offers safe, user-mode access and provides a translation mechanism to allow the network interface access to user virtual memory without imposing onerous requirements on the operating system.

The choice of network interface can dominate system performance. For the systems we modelled, this choice becomes more important as message sizes decrease.

Chapter 7

MEERKAT IMPLEMENTED WITH CURRENT TECHNOLOGY

This chapter outlines a possible implementation of the Meerkat architecture that uses 1994 vintage technology. Comparing our 1990 implementation described in Chapter 3 with the paper design in this chapter will demonstrate that Meerkat's architecture scales with technology.

A second purpose of this chapter's paper study is to show that a Meerkat with hundreds of nodes can reasonably be built. This is important because the Meerkat prototype is not scalable beyond 16 nodes [1]. Our use of the prototype did not require such scalability. However, we contend that this is a property of the prototype, and not of the architecture. It is important, therefore, to show how large Meerkats could be built.

Bus length and the nature and number of taps on a bus are critical in determining the rate at which data can be sent over the bus. These characteristics limit the size of a Meerkat. With current circuit technology, a system with the following characteristics could be engineered without much difficulty:

- 16 nodes per bus
- Internode bus lengths of 75 cm, stub lengths of 5 cm or less
- A maximum of sixteen taps (each node having a tap) per internode bus
- Internode buses clocked at 100 MHz with one 32-bit word transferred per clock tick
- Use of GTL [42] logic levels for internode communication

These characteristics yield a system that can have 256 nodes and a raw, per-bus internode transfer rate of 400 MB/sec. We estimate that under heavy load, such a system would provide 20 MB/sec per node, assuming that most communication requires two buses. If the maximum packet size is 1024 words, as in our prototype, the average latency under heavy load to send 1024 words from one node to another would be 100 microseconds. Under light load, we estimate that a pair of nodes could achieve a sustained bandwidth of 330 MB/sec with a latency of 20 microseconds.

[1] It does not scale because we made compromises to simplify prototype design and construction.

7.1 Node Size

Nodes need to be as small as possible in modern designs to keep wires within the node as short as possible. The pressure to reduce intranode wire lengths is increasing because of higher clock rates and the concomitant use of logic technology with fast edge rates. For example, the current cluster of the DASH prototype is spread on five boards, each about 1800 square cm. For the next generation of DASH, plans call for a cluster to be on a single 516 square cm board. The trend towards smaller nodes benefits Meerkat, because it allows more nodes to tap a bus in a given length of internode bus. Circuit loading and cooling requirements due to an increased number of nodes will also limit the internode clock rate, but to a lesser degree.

7.2 Clocking

While asynchronous interconnects are popular, we see advantages in a synchronous design, namely that the hardware is simpler and much easier to debug. We believe that the difficulty in reproducing and isolating design errors and device failures in asynchronous designs is a serious problem. The principal disadvantage with synchronous designs is that clock skew must be controlled to achieve good performance.

Most synchronous backplane systems are limited by maximum clock skew, worst case device performance, round-trip propagation delay, and jitter. In these systems bus tenure is often just a few cycles, and the bus standard was proposed before large and inexpensive standard cell CMOS gate arrays were available. We propose a clocking scheme that takes advantage of these inexpensive gate arrays and the relatively long bus tenures that we expect in Meerkat in order to compensate for clock skew, wire propagation delay, and device performance.

A system clock is distributed to each node in the system in a way that is convenient to implement and with a concomitant loose bound on skew of half a cycle. Each node is thus run at the same frequency, but nodes may be out of phase with each other by as much as half a clock cycle. The internode logic on each node contains a programmable delay chain for each bus tap that can shift the received system clock by as much as half a cycle. The bus protocol guarantees that on every cycle all nodes know which node is the current bus master. The internode logic uses this information to adjust its delay chains to compensate for clock skew, propagation delay, and device performance. It does this by looking up the delay value in a skew table that is indexed by node and tap number. In a system where B is 16, the table will be two by 16. Each element of the table is a delay value that will be four to six bits wide.

The skew table is loaded in a calibration procedure that is performed when the system is first started and could be updated periodically to adjust for temperature variations. When ownership shifts from one node to another, there will be a loss of at least one cycle as each receiver adjusts its delay chain. Since we expect bus tenure to last for tens or hundreds of cycles, this is a small price to pay for the high clock rate that we hope to achieve. By this dynamic adjustment, we compensate for many factors that limit performance in other systems.

We believe that clock skew will not be the limiting effect in a high-performance Meerkat design. Rather, clock jitter and bus reflections from the taps that each node places on the internode buses will the primary constraints. The latter effect can be helped by keeping stub lengths as short as possible, using a controlled-impedance backplane [76], and using GTL.

Any pair of horizontal and vertical buses can be connected through cross-point switches. This requires that all buses be clocked at the rate of the slowest bus. In an optimal design, all of the buses would reach their limit at the same clock frequency. To do this we chose a physical arrangement that makes the wires that constitute the horizontal and vertical buses the same length.

Figure 7.1 shows how a Meerkat backplane would be laid out in order to maximize the number of nodes in two dimensions while keeping the horizontal and vertical bus lengths equal and as short as possible. This figure shows just a portion of a Meerkat backplane, the lower left corner. Connectors on the front of the backplane are shown as solid, and connectors on the back are shown as dashed. Node boards have connectors along one edge that mate with the backplane connectors. Nodes boards are plugged into the backplane connectors and thus are at right angles with the backplane circuit board. Having node connectors, and thus nodes, on both sides of the backplane doubles the number of nodes that can be attached relative to the number that can be attached with a single-sided backplane. Node connectors, and thus the nodes themselves, are set at a 30 degree angle with respect to the bottom edge of the backplane circuit board. The backplane is a multiple layer circuit board with wires for the internode buses running straight, i.e, not having any bends.

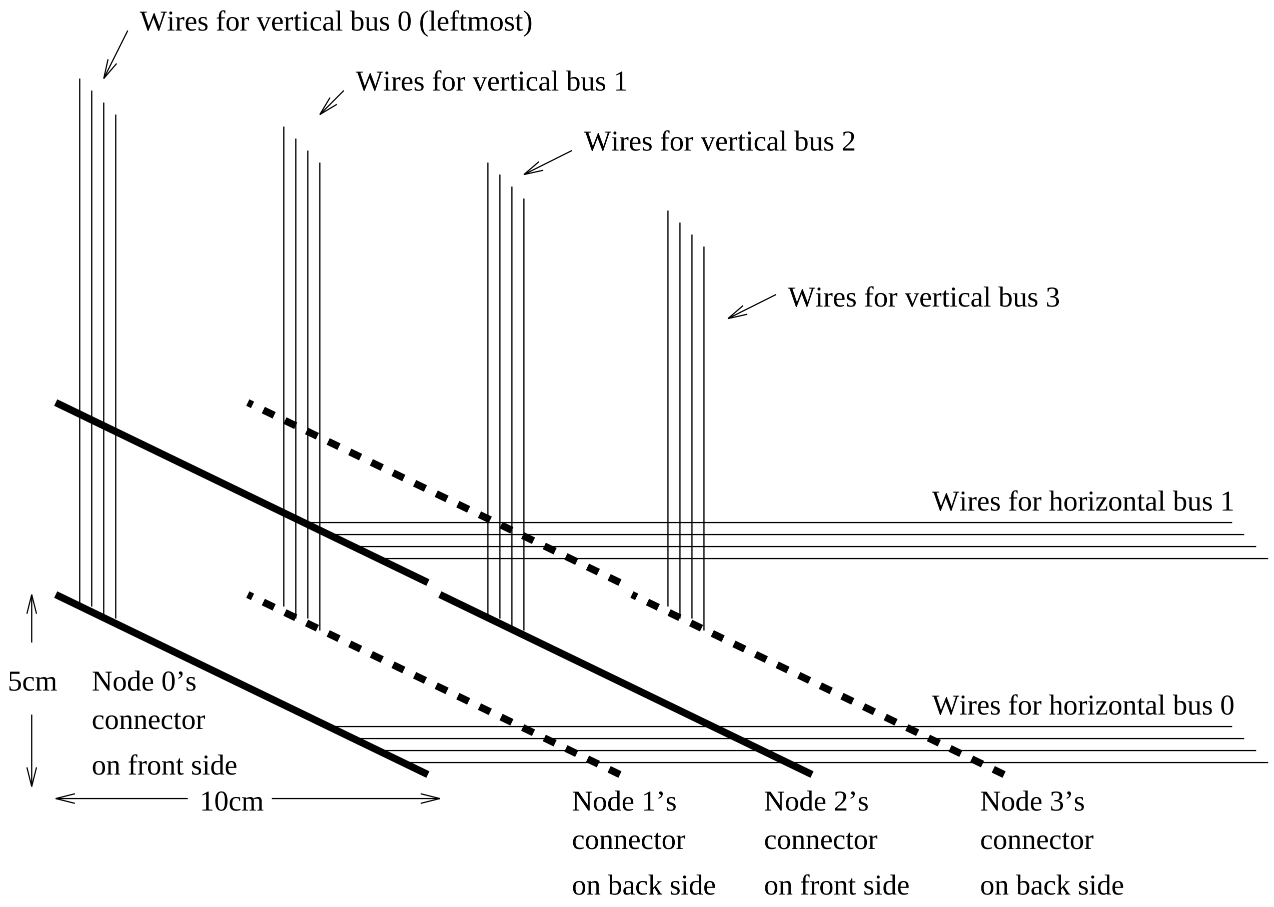

Wires for vertical bus 0 (leftmost)
Wires for vertical bus 1
Wires for vertical bus 2
Wires for vertical bus 3
Wires for horizontal bus 1
Wires for horizontal bus 0
5cm
Node 0's connector on front side
10cm
Node 1's connector on back side
Node 2's connector on front side
Node 3's connector on back side

Figure 7.1: Lower-Left Corner of a Meerkat Backplane

Figure 7.1 shows nodes spaced every 5 cm on horizontal and vertical buses. For a given number of nodes per bus, B, there are $(B - 1)$ 5 cm bus segments. Thus, a system that supported up to 16 nodes per internode bus at 5 cm intervals has buses whose wires are 75 cm long. A system with 16 nodes per bus, 16 vertical buses, and 16 horizontal buses has 256 nodes. A backplane that is 75 cm on a side would probably have to be fabricated in four 37.5 by 37.5 square cm segments. The connections between the backplane segments would be designed to match the backplane's target impedance as closely as possible.

7.3 Summary

The key problems for a Meerkat designer are the physical connection of many nodes to a two-dimensional backplane, limiting the length of backplane wires, the transmission of data, and clocking of this data.

This chapter outlined solutions to some of these design problems. These solutions show how a Meerkat might be built that scales to hundreds of nodes and has interconnect performance well above that of the Meerkat prototype.

Chapter 8

CONCLUSIONS

This thesis has shown the benefits of a hardware-minimalist approach to multicomputer design. By keeping the design simple, we were able to keep design time short and hardware expense low.

We used a concrete architecture, Meerkat, to represent our vision and to test our ideas. Meerkat is at the extreme end of the design spectrum: it uses simple hardware and requires efficient, low-level software. We reinforced Meerkat's minimalist approach with a discussion of specific design choices. The most important choice was to limit scalability. Some parallel computer customers require thousands of processors and can afford the high costs associated with them. Those for whom a few hundred nodes will suffice, however, also require the optimized price/performance achievable from systems designed for their needs.

Another tradeoff was to limit the interconnect to two dimensions. This allowed the use of inexpensive, yet high performance, wiring technology. A third tradeoff was to make the interconnect circuit switched rather than packet switched. This allowed a simple implementation with high transfer rates.

To test our ideas, we designed and built a hardware prototype. This brought us face-to-face with issues that are often glossed-over in paper designs. It also gave us a fast execution vehicle for Meerkat programs, one that allowed us to measure the performance of small systems. The prototype's high performance and simple design demonstrate the value of our approach.

To further evaluate Meerkat, we developed a versatile and efficient simulator. This allowed use to extend our prototype results to systems with hundreds of processors. The calibration of the simulator with the prototype gave us confidence in the simulation results.

The simulator is interesting in its own right, as well. It combines high accuracy, fast execution, host portability, a powerful user interface, and the ability to model multicomputers with hundreds of nodes. We achieved this combination of qualities by starting with a fast, threaded-code interpreter and a translator to generate threaded-code from machine instructions. To make the simulator usable, we chose an existing symbolic debugger, gdb, for the front end. We achieved timing accuracy by carefully adding functional models to the behavioral simulator base. These functional models slowed the simulator down; however, because we added only those functional models that substantially affected accuracy, the degradation was less than an order of magnitude.

We used a combination of low- and high-level benchmarks on both the prototype and simulator. Low-level tests showed that pairs of nodes can achieve a substantial fraction of the peak internode bandwidth. They also showed excellent bisection throughput of a 256-node system under heavy load. High-level tests showed excellent speedup of several numerical applications.

We ran the same tests on Intel's Delta. The Delta is a commercially built system that is heavily used by scientists running parallel numerical codes. All of our tests showed Meerkat outperforming Delta, often by a considerable margin.

The simulator results indicated that there was an opportunity to improve communication performance by a moderate enhancement of the network interface. This led us to design a network interface, Meerkat-2, that used the network more efficiently, especially on small messages, in trade for a moderate increase in design complexity. We reran our tests and found performance improvements, sometimes substantial, on all tests.

To show that the Meerkat architecture scales with technology, we made a paper design of a 256-node system implemented with 1994 technology. The result of this design exercise indicates that Meerkat's interconnect performance can keep pace with improvements in processors and memory.

8.1 Contributions

This dissertation presented several significant contributions to the field of multicomputer architecture. Foremost among these contributions, we demonstrated that RISC principles can be applied to multicomputer design to yield a system with a short design time *and* high performance.

We augmented our Meerkat-1 study with an analysis of tradeoffs in network interface design for circuit-switched multicomputers. We provided performance results and discussed the operating system implications of several network interface designs.

A methodological contribution is the combined use of a hardware prototype and an efficient simulator that is calibrated to the hardware. Hardware prototypes and system simulators are tools often used in architectural tradeoff studies. Our approach is unique in the way in which we employed both tools. We used the hardware to calibrate the simulator and to test programs. We used the calibrated simulator to extend our results to systems with hundreds of nodes and various network interfaces. The simulator also allowed us to measure many things that would be difficult or impossible to measure on a hardware prototype. For example, the simulator counts the number of cycles each processor is delayed waiting to access the interconnect.

The simulator is itself interesting: it a detailed and highly efficient simulator that is able to model the execution timing of hundreds of processors running significant programs. We described its structure, accuracy, performance. In addition, we detailed the methodology we used in building the simulator.

The Meerkat interconnect has an intrinsically lower latency than that of any others that can connect hundreds of nodes. While not significant for multicomputers such as Meerkat, where software overhead dominates interconnect latency, this property could be of great benefit in multiprocessors for which remote reference latency is dominated by interconnect latency.

8.2 Avenues for Future Research

This section describes four ideas for future research: the benefit of Meerkat's interconnect in a multiprocessor, fault tolerant Meerkats, the engineering of fast buses, and ways to improve the simulation technology we used.

8.2.1 Meerkat Interconnect in a Multiprocessor

Some of the lessons that we have learned could be applied to parallel systems very different from ours. Meerkat's interconnect latency matches that of mesh router systems when load is high and greatly exceeds it when load is low. This low network latency is of no advantage in a system like Meerkat-1, for which network latency is dwarfed by software overhead. However, it would be a significant advantage in systems with network interfaces that themselves have low latency and very low software overhead. For example, a cache-coherent multiprocessor could use Meerkat buses instead of mesh routers to connect nodes. This would reduce the latency, making it much closer to the speed-of-light delay between nodes and lessening the need for latency-hiding techniques.

8.2.2 Fault Tolerance

There are many multicomputer applications that demand some degree of fault tolerance. The choice of interconnect can affect the difficulty of achieving a given level of fault tolerance. Meerkat's passive backplanes have a simple fault model that should make tolerating faults easier than it is in many other interconnects.

There are several failure modes to consider. If a node is fail-stop, a stopped node will not prevent communication on that bus: node interfaces tap the buses only, they do not intercept them. The node pairs that would have otherwise used the stopped node as a cross-point switch can use an alternative node. In any rectangular array, there are always two choices of cross-point node for any pair of sender and receiver.

If a node fails without releasing internode buses, or fails and jams a bus, the neighbor nodes can turn off the offending node by using simple voting and power control circuitry. The bus will continue to function with a powered-off node, because the bus-tapping circuitry will present an open circuit when power is removed.

Node power-control circuitry is also useful for allowing nodes to be added and removed while the system is running. When a node is inserted in a live system, the power to its socket is off. Once inserted, its power is turned on, and other nodes initialize it so that it can join the system. Broken nodes, or nodes in need of upgrading, can be unloaded by the operating system, have their power turned off, and then be physically removed.

8.2.3 Electrical Engineering of Faster Buses

An interesting question for future exploration is how Meerkat's buses will scale up with technology in comparison to low-dimension, point-to-point networks. In either case, the electrical links will be transmission lines, and there is little difference between driving a terminated point-to-point link and driving a terminated bus. In the latter case, however, the data can go much farther without incurring logic delays.

Scaling up Meerkat's buses will use many of the same circuit techniques that point-to-point links require. Key techniques include: phase-shift-tolerant signalling, low energy signalling, parallel optimistic bus arbitration using collision detection, and adaptive active signal termination. To relieve signal reference problems, differential signalling may be needed.

8.2.4 High Performance and Accurate Simulator

The quantitative results in this thesis are largely based on results from the Meerkat simulator. There are a number of ways in which the simulator could be made faster while maintaining its accuracy. First, native host code could be generated, as in Shade [26]. Second, the simulator could be parallelized and run on a multicomputer [1].

The simulator could be made more accurate while sacrificing minimal performance by making the instruction cache model accurate and by modelling the single register write-back port. Currently, the simulator models instruction cache cold misses only. It does not model contention for the single write-back port of the 88100's register file.

8.3 Final Word

This thesis sought to convince the reader of several key points. First, a minimalist, performance-oriented approach to multicomputer design has great rewards. It lowers design time and improves performance. Moving function into software simplifies hardware, and software can be improved and tailored long after hardware is built. Second, detailed simulations of multicomputers with hundreds of nodes running significant programs are possible even on a modest host computer. In addition, the costs of constructing such a simulator are moderate and yield a tool that can be changed and instrumented much more easily than can hardware. Third, the network interface is one of the most important aspects of multicomputer architecture. Our measurements show that the choice of network interface is often more significant than the network itself.

[1] We structured the simulator so that it would be easy to run on a multicomputer. We did not implement it, because performance was satisfactory for our benchmarks with `mg88` hosted on a uniprocessor.

Bibliography

[1] Advanced Micro Devices, Inc., 901 Thompson Place, PO Box 3453, Sunnyvale, Ca. 94088-3453. *PAL Device Data Book and Design Guide*, 1993.

[2] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Laboratory for Computer Science, MIT, 1991.

[3] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. *Proceedings of the Fourth International Conference of Architectural Support for Programming Languages and Operating Systems*, April 1991.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[5] Ramune Arlauskas. iPSC/2 System: A second generation hypercube. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 38–42, January 1988.

[6] Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H.T. Kung, Robert D. Sansom, and Peter A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the Third International Conference of Architectural Support for Programming Languages and Operating Systems*, April 1989.

[7] C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman. A comparison of three distributed sparse factorization schemes. In *SIAM Symposium on Sparse Matrices*, May 1989.

[8] Luiz Andre’ Barroso and Michel Dubois. The performance of cache-coherent ring-based multiprocessors. Technical Report CENG-92-19, University of Southern California, November 1992.

[9] Robert Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the Winter 1990 USENIX Conference*, pages 53–63, January 1990.

[10] Robert Bedichek and Curtis Brown. The Meerkat multicomputer. In *Proceedings Fifth Annual International Symposium on Parallel and Distributed Systems*, December 1993.

[11] James R. Bell. Threaded code. *Communications of the ACM (CACM)*, 16(2):370–372, June 1973.

[12] John K. Bennett, Sandhya Dwarkadas, Jay Greenwood, and Evan Speight. Willow: A scalable shared memory multiprocessor. *Proceedings. Supercomputing ’92*, pages 336–345, November 1992.

[13] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[14] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[15] Roberto Bisiani and Mosur Ravishankar. Plus: A distributed shared-memory system. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990.

[16] T. Blank. The MasPar MP-1 architecture. In *Thirty-Fifth IEEE Computer Society Internation Conference*, pages 20–24, February 1990.

[17] Gaetano Borriello. Personal communication, April 1994.

[18] Eric A. Brewer, Chrysanthos N. Dellacrocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[19] Eric A. Brewer and William E. Weihl. Developing parallel applications using high-performance simulation. *ACM/ONR Workshop on Parallel and Dist. Debugging. ACM SIGPLAN Notices*, 28(12):158–168, December 1993.

[20] Michael Carlton and Alvin Despain. Aquarius project. *IEEE Computer*, pages 80–83, June 1990.

[21] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[22] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[23] J. Bradley Chen and Brian N. Bershad. The impact of operating system performance on memory system performance. *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 120–133, Dec 1993.

[24] Christopher Cheng and Leo Yuan. Electrical design of a 1 GByte/sec high performance backplane using low voltage swing CMOS (GTL). In *Proceedings of Research on Integrated Systems*, pages 291–299. MIT Press, 1993.

[25] D. W. Clark. Pipelining and performance in the VAX-8800 processor. *Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct 1987.

[26] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *1994 ACM SIGMETRICS Conference on Modeling and Measurement of Computer Systems*, May 1994.

[27] Douglas Comer and James Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135, June 1990.

[28] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Computation*, 19:297–301, 1965.

[29] W. Crowley, C.P. Hendrickson, and T.I. Luby. The SIMPLE code. *Technical Report UCID-17715*, 1978.

[30] William Dally. *The J-Machine System*, volume 1. MIT Press, Cambridge Mass., 1990.

[31] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, editor, *Proceedings Stanford Conference on Advanced Research in VLSI*, pages 391–415, Cambridge Mass., 1987. MIT Press.

[32] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the Smalltalk-80 system. *11th Annual Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[33] Digital Equipment Corporation, Workstation Systems Engineering. *PMADD-AA TurboChannel Ethernet Module Functional Specification, Rev 1.2.*, August 1990.

[34] David R. Ditzel, Hubert R. MeLellan, and Alan D. Berenbaum. The hardware architecture of the CRISP microprocessor. *Proceedings of the 14th Annual International Symposium on Computer Architecture; Computer Architecture News*, 15(2):309–319, June 1987.

[35] Dan Dobberpuhl, R. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R. Conrad, D. Dever, B. Gieseke, S. Hassoun, G. Hoeppner, J. Kowaleski, K. Kuchler, M. Ladd, M. Leary, L. Madden, E. McLellan, D. Meyer, J. Montanaro, D. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam. A 200MHz 64 bit dual issue CMOS microprocessor. In *International Solid-State Circuits Conference 1992*, February 1992.

[36] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, December 1993.

[37] G.M. Papadopoulos et al. *T: Integrated building blocks for parallel computing. In *Proceedings of Supercomputing ’93*, pages 624–635, 1993.

[38] Edward W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. Ph.D. thesis, Department of Computer Science and Enginerring, University of Washington, July 1993.

[39] FORE Systems, 1000 Gamma Drive, Pittsburgh PA 15238. *TCA-100 TURBOchannel ATM Computer Interface, User’s Manual*, 1992.

[40] GigaBit Logic, Inc., 1908 Oak Terrace Lane, Newbury Park, CA 91320. *GigaBit Logic GaAs IC Data Book and Designer’s Guide*, 1991.

[41] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings 17th Annual Symposium on Computer Architecture*, pages 422–431, May 1990.

[42] Bill Gunning, Leo Yuan, Trung Nguyen, and Tony Wong. A CMOS low-voltage-swing transmission-line transceiver. *IEEE International Solid-State Circuits Conference*, pages 58–59, 1992.

[43] D. Gustavson and J. Theus. Wire-Or logic on transmission lines. *IEEE Micro*, pages 51–55, June 1983.

[44] M.T. Heath, E. Ng, and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33, 1991.

[45] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[46] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge Mass., 1985.

[47] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. Ph.D. thesis, University of Washington, January 1987. Department of Computer Science technical report 87-01-01.

[48] Intel Corp, 2065 Bowers Avenue, Santa Clara, California 95051. *Touchstone Delta C System Calls Reference Manual*, 1991.

[49] Intel Corp, 2065 Bowers Avenue, Santa Clara, California 95051. *A Touchstone Delta System Description*, February 1991.

[50] Intel Corp, 2065 Bowers Avenue, Santa Clara, California 95051. *Touchstone Delta System User's Guide*, 1991.

[51] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[52] Alan H. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[53] Peter B. Kessler. Fast breakpoints: Design and implementation. In *Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, volume 25, pages 78–84, White Plains, NY, USA, June 1990.

[54] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *7th ACM International Conference on Supercomputing*, 1993.

[55] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Charachorloo, John Chapin, David Hakahira, Joel Bater, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *21th Annual International Symposium on Computer Architecture*, May 1994.

[56] T.G. Lang, J.T. O'Quin, and R.O. Simpson. Threaded code interpreter for object code. *IBM Technical Disclosure Bulletin*, pages 4238–4241, March 1986.

[57] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[58] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[59] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Ph.D. thesis, Yale University, September 1986. Technical Report YALEU/DCS/RR–492.

[60] Calvin Lin and Lawrence Snyder. A portable implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, 1991.

[61] Peter S. Magnusson. A design for efficient simulation of a multiprocessor. *MASCOTS '93 – Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, January 1993.

[62] Cathy May. Mimic: A Fast S/370 Simulator. In *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices*, volume 22, pages 1–13, St. Paul, MN, June 1987.

[63] *MC88100 RISC Microprocessor User's Manual*. 2900 South Diablo Way, Tempe, Arizona 85282.

[64] *MC88200 Cache/Memory Management User's Manual*. 2900 South Diablo Way, Tempe, Arizona 85282.

[65] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[66] Mitsubishi Electric, 1050 East Arques Avenue, Sunnyvale, CA 93086. *Target Spec: Fast Page Mode 4194304-it Dynamic RAM*, 1990.

[67] Motorola Corporation, 2900 South Diablo Way, Tempe, Arizona 85282. *HM88K HYPERmodule 32-bit RISC Processor Mezzanine Module User's Manual*, March 1990.

[68] National Semiconductor Corporation. *DP8390C/NS32490C Network Interface Controller*, 1986.

[69] *IEEE standard for a simple 32-bit backplane bus: NuBus*. New York, NY, USA, August 1988.

[70] J. M. Ortega and R. G. Voight. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(149), 1985.

[71] Paradise Systems, Inc., 99 South Hill Drive, Brisbane, CA 94005. *Paradis Printer Controller*, January 1988.

[72] Paul Pierce. The NX/2 operating system. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[73] Carl Ponder. Personal communication, February 1994.

[74] Justin Ratner. Talk given at the University of Washington, 1993.

[75] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. *Performance Evaluation Review*, 21(1):48–60, May 1993.

[76] Lee W. Ritchey. Controlled impedance PCB design. *Printed Circuit Design*, pages 23–28, June 1989.

[77] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[78] Youcef Saad and Martin H. Schultz. Topological properties of hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, vol. 1*, pages 867–872, January 1988.

[79] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[80] Charles L. Seitz, Nanett J. Boden, Jakov Seizovic, and Wen-King Su. The design of the Caltech Mosaic C multicomputer. *Research on Integrated Systems*, pages 1–22, 1993.

[81] Charles L. Seitz and Wen-King Su. A family of routing and communication chips based on the Mosaic. *Research on Integrated Systems*, pages 320–337, 1993.

[82] Margo Selzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324, January 1990.

[83] Signetics Corp, 811 E. Arques Avenue, Sunnyvale, CA 94088-3409. *Signetics FAST Logic Data Handbook*, 1989.

[84] Pradeep Sindhu, Jean-Marc Frailong, Jean Gastinel, Michel Cekleov, Leo Yuan, Bill Gunning, and Don Curry. XDBus: a high-performance, consistent, packet-switched VLSI bus. *IEEE*, pages 338–344, 1993.

[85] J. P. Singh. Talk given at the University of Washington, 1993.

[86] Richard L. Sites, Anton Chernoff, Mathew B. Kerk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[87] Insignia Solutions. *SoftPC Product Information*, 1991.

[88] Richard M. Stallman and Roland H. Pesch. *Using GDB: The GNU Source-Level Debugger*. Free Software Foundation, 545 Tech Square, Cambridge, Ma. 02139, March 1992.

[89] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[90] Tektronix, Inc., P.O. Box 1000, Wilsonville, Oregon 97070. *XD88/12 Engineering Requirements Specification*, September 1990.

[91] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2), May 1993.

[92] Thinking Machines Corp., 245 First St., Cambridge MA 02142. *CM-5 Technical Summary*.

[93] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: a hierachically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[94] Jr. William R. Blood. *MECL System Design Handbook*. 2900 South Diablo Way, Tempe, Arizona 85282, 1983.

[95] Xilinx. *The Programmable Gate Array Data Book*, 1993.

[96] E.L. Zapata, J.A. Lamas, F.F. Rivera, and G. Plata. Modified Gram-Schmidt QR factorization on hypercube SIMD computers. *Journal of Parallel and Distributed Computing*, 12:60–69, 1991.