Picasso: An Experiment in Hypercube Operating System Design

David K. Bradley Bobby A. A. Nazief Dirk C. Grunwald Daniel A. Reed

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

Any new computer system organization raises many questions about hardware, system software, algorithms, and programming; multicomputers are no exception. The commercial emergence of hypercubes has made it possible to move many research questions from theoretical to experimental venues. However, hardware availability does more than permit implementation of previously tested ideas. The feasibility of many application and operating system algorithms can only be determined experimentally. We believe it is difficult, if not impossible, to experimentally examine these issues singly — they interact in both obvious and subtle ways. This paper describes the Picasso hypercube operating system, a part of the Picasso project at the University of Illinois. Motivated by the need for a flexible operating system testbed, the design of Picasso has exposed several important, and necessary, features of multicomputer operating systems designed for research. We present the design and current performance of Picasso, discuss the lessons learned, and conclude with an overview of future research plans.

1 Introduction

The recent explosion of interest in multicomputers can be traced to the construction of the CalTech Cosmic Cube [15]. Following the success of the Cosmic Cube, Intel developed the iPSC [12], based on Intel 80286/80287 processor pairs; Ametek countered with the System/14 also based on the Intel 80286/80287 chip set. Both the Intel iPSC and Ametek System/14 designs used an existing microprocessor coupled with additional logic to manage communication. The Floating Point Systems T-Series [5], based on the Inmos Transputer [16], first integrated computation and communication in a single processor. Ncube followed with the Ncube/ten [6] using a custom microprocessor containing hardware instructions for message passing.

Despite the enormous attention generated by multicomputers, all commercial multicomputers are conceptually similar. Differences do exist in communication performance and computation speed, but these differences are quantitative rather than qualitative. These similarities extend to commercial multicomputer operating systems. All provide basic support for internode communication; some also provide the process and memory management associated with traditional operating systems.

The homogeneity of commercial multicomputer networks does not reflect an intellectual consensus on the hardware and software design paradigms. Instead, the immaturity of the field, both in research and design experience, coupled with marketing pressures, has encouraged conservative hardware and software design. Designs based on existing microprocessors provided effective experimental tools with minimal hardware costs. Similarly, the addition of explicit send and receive primitives to sequential programming languages such as C and Fortran is the simplest extension sufficient to support message-based parallel programming.

Any new computer system organization raises many questions about hardware, system software, algorithms, and programming; multicomputers are no exception. On the multicomputer operating systems front, there is much work still to be done, both in operating system organization and algorithms, and in

operating system user interfaces. Pragmatically, the commercial emergence of multicomputers has made it possible to move many operating systems research questions from theoretical to experimental venues. However, hardware availability does more than permit implementation of previously tested ideas. The feasibility of many operating system algorithms can only be determined experimentally. We believe it is difficult, if not impossible, to experimentally examine these issues singly — they interact in both obvious and subtle ways.

Unfortunately, commercial multicomputer operating systems, because they are minimal extensions of traditional operating systems, do not provide the flexibility necessary to investigate resource management. This paper describes the Picasso hypercube operating system, a part of the Picasso project at the University of Illinois. During the design of Picasso we discovered several important and, we believe, necessary, features of multicomputer operating systems. Thus, this paper should be viewed as a manifesto for multicomputer operating system design.

In §2, we present our view of the important multicomputer operating system research issues. The current state of one Picasso implementation on the Ametek System/14 is the subject of §3. In §4 we discuss the performance of the current Picasso implementation, followed in §5 by a discussion of related performance studies. We conclude in §6 with plans for future research.

2 Motivations

In our research over the last five years[13], we have developed performance models of multicomputer interconnection networks, permitting network comparisons under a variety of workloads, explored the feasibility of dynamic load balancing on multicomputers, and proposed adaptive routing algorithms for multicomputer networks. In all our investigations of multicomputer resource management, *performance* has been the guiding motivation; this is the *raison d'etre* for multicomputer networks.

Unfortunately, many multicomputer resource management algorithms interact in subtle ways (e.g., dynamic task scheduling and message routing). A well-crafted operating system is a gestalt, more than the sum of its constituent algorithms. Decomposition and separate evaluation of component policies may not capture the subtle, but often crucial, policy interactions. Analysis, by its nature, requires simplifying assumptions to permit tractability. Similarly, the cost of simulation restricts the scope and depth of testing. Thus, the feasibility of many resource management algorithms can only be determined by conducting parametric performance studies on operating system implementations that combine variants of selected resource management policies.

As noted at the outset, commercial multicomputer operating systems are conservative, minimal extensions of operating systems for sequential processors. However, just as operating systems for sequential processors have posed a plethora of interesting, and controversial, research problems (e.g., virtual memory, deadlock resolution, synchronization, process scheduling, and software organization), multicomputer operating systems provide fertile ground for research. In the remainder of this section, we sketch some, but by no means all, of the important issues we believe must be addressed by future multicomputer operating systems.

2.1 Communication Paradigms

The current generation of hypercube-based multicomputer networks relies on store-and-forward packet switching using fixed path, *e-cube* message routing.[1] Although the hypercube topology has a rich interconnection network, with $K!$ overlapping routes between any two nodes separated by K hops, the *e-cube* routing algorithm relies on only one of these paths, the one obtained by resolving differences in the binary source and destination addresses from least to most significant. Although *e-cube* routing is simple and free from deadlocks, the inability to investigate alternate routes limits its flexibility when faced with network hot spots and highly variable communication traffic. Ideally, a routing algorithm should use knowledge of the network state to select the path from source to destination node.

Adaptability is not without cost; the overhead for acquiring network knowledge depends on the source and frequency of that acquisition. Centralized routing techniques use global information to make routing decisions. Because these algorithms use information distributed from a central source, they do not adapt quickly to changes in the network state. In contrast, distributed routing techniques use only local information and can quickly adapt. Unfortunately, decisions based purely on local information are usually unstable, and packets can be trapped in transmission loops. A hybrid routing procedure combines the two strategies. In the best case, a hybrid technique achieves the

[1] The Ametek 2010 and Intel iPSC/2 are the first of the second generation multicomputers based on circuit-switched communication.

stability of the centralized techniques and the adaptability of the distributed techniques[8].

The performance overhead for adaptive routing algorithms varies greatly, and the efficacy of a particular choice depends on communication pattern variability in both time and space. If the communication pattern is highly regular, simple, non-adaptive algorithms like *e-cube* suffice. Similarly, if the traffic varies rapidly in time, only a distributed routing scheme can respond quickly enough to match the changing workload.

Adaptive routing algorithms are not universal; some are ill-suited to certain communication workloads. Just as virtual memory management algorithms generated considerable controversy and early debate, the costs, implementation complexity, and relative benefits of adaptive routing algorithms remain in doubt. Only by exploring the implementation overheads and realms of routing algorithm applicability can a consensus emerge. In the interim, application programmers are best served by providing a suite of routing algorithms, each matched to certain communication characteristics.

2.2 Task Scheduling

There are two primary approaches to parallel processing and task scheduling[2] on a multicomputer network. In the first, all parallel tasks in a computation are known *a priori*; tasks are statically mapped onto network nodes before the computation is initiated and remain there throughout the entire computation. This paradigm corresponds naturally to two different software design styles: a universal task and a network of communicating tasks.

As an example of the universal task design style, iterative partial differential equations solvers update a regular grid of points. The value of each grid point is successively updated using values from neighboring points; the algorithm at each point is the same. Thus, it is natural to partition the grid into blocks and assign each block to an instance of the algorithm, a task. Although the tessellated tasks do communicate, the pattern is, by definition, regular. A static scheduling algorithm need only map the (fixed) tessellation of the single task onto the nodes of the multicomputer network.

Alternatively, one can design a network of communicating tasks similar to that supported by CSP[7]. The topology of the network of communicating tasks can be arbitrary and irregular, and both the amount and frequency of intertask data transfer may vary during the computation. Nevertheless, the number of tasks and their potential communication patterns are known, permitting a static mapping of tasks onto network nodes.

In the second approach to parallel processing on a multicomputer network, a parallel computation is defined by a dynamically created task precedence graph. New tasks are initiated, existing tasks terminate as the computation unfolds, and the mapping of tasks onto network nodes is dynamic. Precedence constraints among tasks arise because the initiation of new tasks depends on the completion of prior ones. This dynamic view of computation differs in several significant ways from the static view. In particular, the workload varies over time, and multiple tasks may become eligible for execution given the results from a single task. Most importantly, tasks must be dynamically mapped onto network nodes using only partial knowledge of the global system state.

Returning to the example drawn from partial differential equations, the regular grid must be fine enough (i.e., contain enough grid points) to capture the behavior of the function at its greatest variation. Where the variation is small, considerably less computation is required. Grid refinement techniques adapt to variation by repeatedly *refining* the grid in appropriate regions. This refinement creates new tasks, each containing a new block of grid points, and disrupts the initially regular communication pattern. These new tasks must be dynamically mapped onto processors while simultaneously restricting the communication overhead to acceptable levels.

Currently, the universal task paradigm is most common, perhaps because commercial multicomputer operating systems support neither static nor dynamic task scheduling. Instead, programmers statically assign tasks to nodes based on their knowledge of the intertask communication pattern. Static task schedulers are ancillary to multicomputer operating systems; these schedulers need only measures of communication and communication performance to assign tasks to nodes. In contrast, dynamic task schedulers are intimately tied to the multicomputer operating system structure and communication subsystem. Acquisition of scheduling information for dynamic task assignment and task migration must complement the communication network. For example, both the adaptive message routing algorithm and the dynamic scheduling algorithm should reflect the common communication patterns of the application. The dearth of performance studies for both static[2] and dynamic[13] task scheduling suggests that additional experiments are imperative.

[2]Although the term task scheduling denotes assignment of tasks to nodes, not processor scheduling on a single node, the two are not independent.

2.3 Global Virtual Memory

The maturation of multicomputer design is reflected by the improved performance of second generation interconnection networks (e.g., the Intel iPSC/2 and the Ametek 2010). Early network implementations constrained the space of feasible algorithms, requiring strict communication locality to offset the high communication latency. Not only do second generation networks exhibit smaller average message latency, but the latency is nearly independent of the number of communication links traversed by a message. This improved performance greatly narrows the gap between transmission latency in multicomputers and global memory access times in hierarchical, shared memory systems[9], making it feasible to consider supporting global, virtual memory[3] on multicomputer networks.

Although hierarchical, shared memory multiprocessors provide a flat, global address space, the access times for global and local memory often differ by an order of magnitude. This differential encourages data migration from global to local memory. Movement of code pages or array sections is typical.

In contrast to shared memory systems, the access times for local and remote memories differ by two or three orders of magnitude, even for second generation multicomputers; this disparity makes data migration imperative. Because executable code segments are read-only, code sharing is a logical first step in realizing global, virtual memory. Most tasks of multicomputer programs execute similar code, often with identical spatial and temporal locality; the number of distinct pages in the union of all task localities is small. Preliminary studies[11] suggest that assigning code pages to "home" nodes and transferring these pages on demand can significantly reduce the total memory required.

In contrast to shared code, a writable, shared *data space* would dramatically change the multicomputer programming paradigm, obviating message passing primitives, but introducing synchronization primitives for shared memory access. A shared, writable data space, coupled with the implementation need for data migration, introduces thorny data coherence problems. If only one copy of shared objects can exist, multiple tasks may compete for the single copy. This *network thrashing* may offset any potential performance gains from data sharing. If multiple copies of shared objects are permitted, the programmer or the compiler must insure that modifications are correctly propagated to all copies. In the former case, the programmer must delimit each region of *read-only* and *read-write* access with synchronization primitives. In the latter, a sophisticated interprocedural data dependency analysis would identify those regions of *read-only* and *read-write* access.

Data dependency analysis to extract parallelism is well-established[17]. However, the performance and overhead of operating system heuristics to reduce memory access penalties by automatically migrating code and data are unknown.

[3]In a global, virtual memory, all tasks of a multicomputer program share a virtual memory that spans many nodes. Modification of any memory location is visible to all nodes.

2.4 Heterogeneous System Resources

At present, multicomputers are homogeneous ensembles of computation nodes. Although individual algorithms are often homogeneous, most applications consist of disparate algorithms with differing characteristics. As an example, computer vision encompasses image capture, low-level pixel processing, feature extraction, and high-level symbolic computation. The algorithm diversity in typical vision systems makes them highly suitable candidates for heterogeneous multicomputer networks containing pixel and symbolic processors. Ideally, a multicomputer operating system should support diverse nodes (e.g., vector and symbolic) and permit configuration of node ensembles for specific applications. Such configurations pose unsolved problems in task scheduling, adaptive routing, and resource management.

2.5 Peripheral Management

Second generation multicomputer networks, such as the Intel iPSC/2, support peripherals (e.g., secondary storage devices) as node adjuncts rather than as peripherals of the multicomputer host. This raises several important configuration questions, particularly for secondary storage devices. Which is preferable, a large number of slow, small devices that distribute communication traffic to secondary storage across the network, or a small number of large, fast devices that introduce communication bottlenecks? The answer clearly depends on the performance of the underlying communication network, its software support, and the demands of application algorithms. These issues can only be explored by operating system instrumentation and benchmarking.

If peripherals are distributed throughout the network in a non-uniform pattern, the communication traffic generated by remote access may adversely affect intertask communication. Moreover, the temporal and spatial distribution of peripheral traffic will likely differ from that generated by intertask commu-

nication, particularly if some devices require real-time response (e.g., video cameras). Message priorities, coupled with adaptive routing algorithms, may ameliorate the deleterious effects of a bicameral traffic distribution.

2.6 Observations

Like traditional, uniprocessor operating systems, multicomputer network operating systems define a *virtual machine* that not only extends the underlying hardware but also hides its idiosyncrasies. Typically, such virtual machines provide resource management, scheduling, and some measure of fault tolerance. In addition, a multicomputer network operating system should also provide support for efficient internode communication, parallel task scheduling, heterogeneous nodes and peripherals, and ideally, support for global virtual memory. Algorithms for these resource management problems are coupled by their mutual need for resource status information. As noted at the outset, we believe the coupling of resource management algorithms is such that separate study is not viable. Moreover, the dynamic nature of these interactions makes analysis impossible and simulation prohibitively expensive.

Existing multicomputer operating systems are too inflexible to permit rapid replacement of specific resource management policies. Important policies are often intertwined and scattered throughout the operating system. Picasso is a multicomputer operating system schema designed to permit systematic exploration of policy interactions by quickly replacing specific policies.

3 The Picasso Operating System

What are the characteristics of a research operating system? Most importantly, it should be *flexible*, providing a facility for testing novel ideas, while retaining efficiency, allowing realistic appraisal of those ideas. It must allow experimentation and the sharing of results by many researchers. We believe such a system will have the following properties.

Separation of Policy and Mechanism:
Long recognized as a requirement in operating system research [18], the use of distinct policies is predicated on a formal definition of operating system interfaces. Interface standards, the mechanism for policy interaction, permit simultaneous, disjoint policy development by multiple researchers.

Modularity:
Not only must policy and mechanism be separate, policy implementations must be independent. Although policies interact, and their performance is interrelated, a change in the implementation of one policy should only affect other policies through defined interfaces. For example, changes in message routing should not affect virtual memory support.

Portability:
Significant, quantitative performance differences permit implementation of qualitatively different policies (e.g., lower message latency permit efficient implementation of global, virtual memory). To identify those policies suitable for particular architectures, the operating system core should be implemented on a variety of multicomputer networks. To be sure, some policies, such as global virtual memory, may be prohibitively expensive on some multicomputer networks; however, quantifying performance differences provides insight into the costs and benefits of each policy.

Instrumentation:
Accurate performance measurements of commercial multicomputer operating systems *or* applications are difficult at best. Existing multicomputer operating systems, organized to minimize overhead, lack the flexibility and facilities for gathering performance data. A modular design with performance instrumentation interfaces permits capture of detailed performance data *when needed.* Coupled with hardware performance monitors, this software instrumentation provides support for both performance measurement and program debugging.

Performance:
Although it is naïve to expect the performance of a research operating system to match that of a crafted production system, poor performance makes research results suspect. Some performance degradation is acceptable, provided the performance degradation due to modularity and enforced interface standards penalizes all policies equally.

In toto these design properties permit construction of flexible, reconfigurable research operating systems; the Picasso multicomputer operating system is one such example. The remainder of this section describes an implementation of Picasso on the the Ametek System/14, one of the few first generation multicomputers with a separate communication co-processor.

The presence of a co-processor on the Ametek System/14 permits partitioning of operating system components based on functionality, while minimally affecting the performance of the computational processor.

3.1 Hardware Organization

Each node of the Ametek System/14 contains a computational processor, an 8 MHz Intel 80286 microprocessor with an 80287 floating point co-processor, and a communications processor, an Intel 80186 microprocessor. As Figure 1 shows, these processors share a one megabyte dual-ported memory, and each has a software monitor stored in private PROM. In addition, the 80186 has a small amount of fast, private memory.

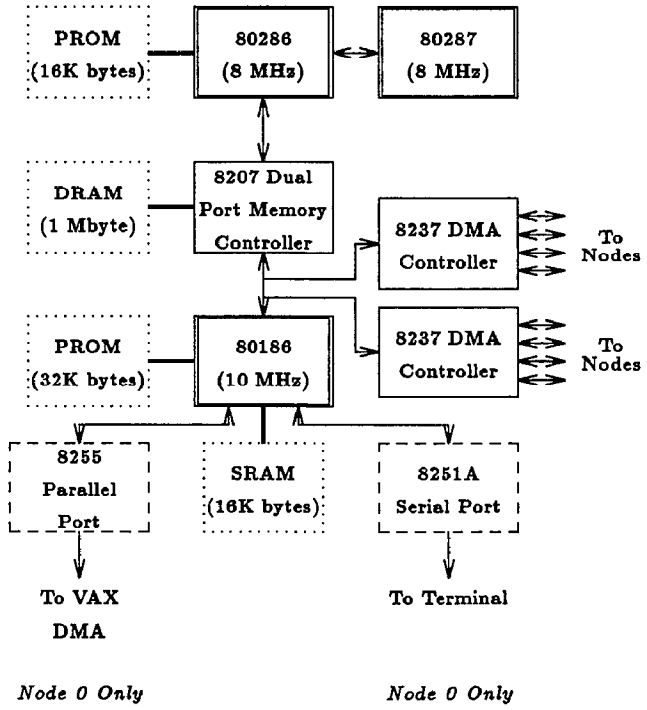

Figure 1: Block Diagram of System/14 Node

Messages are transmitted between adjacent nodes using serial communication lines that are connected to one of two direct memory access (DMA) controllers; however, only one DMA controller can be active at a time.

3.2 Software Organization

As in commercial multicomputer operating systems, every node maintains a private copy of the Picasso operating system; each processor within a node executes a separate operating system component. Although there is one operating system, different parts have been designed to execute on the 80286 and 80186 processors. Quite naturally, the 80186 communication processor manages operations related to communication (e.g., transmitting messages, managing message buffers, updating routing tables, etc.). The computation processor is responsible for task management and scheduling. These two processors communicate through a shared data structure in each node. A cross-interrupt mechanism permits fast intranode communication, and a hardware memory lock guarantees exclusive access to data structures shared by both processors.

Internode communication is initiated by application tasks on the computational processor, but it is largely implemented by communication processor. The communication processor polls the transmission lines for incoming messages and checks a message transmission list for outgoing messages. The transmission list is shared between the two processors; messages are enqueued by the computation processor and serviced by the communication processor. Picasso supports several communication primitives, including both blocking and non-blocking variants of send and receive.

3.3 Implementation Issues

Initial performance measurements of Picasso using existing benchmarks[14] showed that reducing *message latency* was a paramount consideration. Communication hardware limitations on the Ametek System/14 dictate that the size of a message be known to both the sending and receiving nodes before it is transmitted. Picasso supports variable length messages by sending a fixed size *message header* that contains the length of the full message. In addition, this header contains routing and other information. The following transmission, whose length is now known, contains the message.

Each transmission incurs a fixed overhead, independent of the size of the transmitted message. For small messages, this overhead exceeds the cost for message transmission. To eliminate the overhead of a second transmission for small messages, a portion of the message, called the **tip**, can be included in the message header. The optimum **tip** size depends on the communication characteristics of each application and can be adjusted to maximize communication performance.

As mentioned earlier, each node contains a small, static memory accessible only by the 80186 communication processor. Moving the kernel of the Picasso operating system to this faster, private memory de-

creased message latency by 30%. Because the interface between the computation and communication processors was well defined, no operating system modifications were necessary; only the bootstrap program was changed.

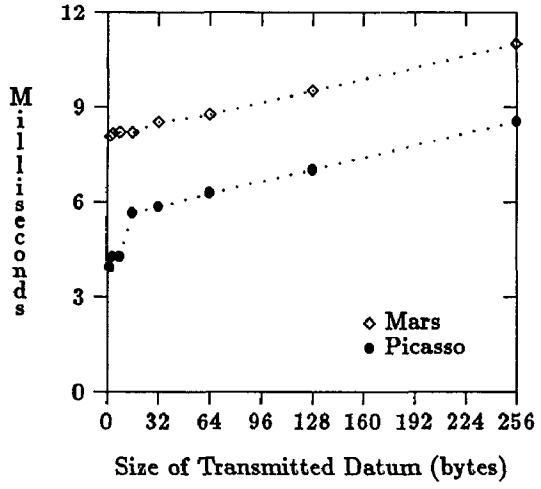

Figure 2: Simple Transfer

3.4 Performance

To provide a performance reference point, we measured the performance of Picasso using a standard set of communication benchmarks [14]. Using these same benchmarks, we measured the performance of the Mars operating system[1], a commercial operating system provided with the Ametek System/14. Although these benchmarks do not reflect *system performance*, they do reflect the efficiency of basic communication primitives.

Figure 2 shows the communication delay for simple message transfers between adjacent nodes. With the exception of small message transmissions by Picasso footnote Recall that Picasso sends small messages in the **tip.** , delay is approximated by

$$T = (t_m + t_l) + N \cdot t_b$$

using the terms in Table 1.

The message delay includes a message preparation time (t_m), incurred by the computational processor, and a communication latency (t_l), due to the message co-processor. The message preparation time is the time to *schedule* a message, whereas the communication latency is the time to *transmit* a message.

Ideally, t_m should be small, lest the advantage of a communication processor be overshadowed by the cost of scheduling messages. Similarly, a large value for t_l limits the efficiency of message transmissions.

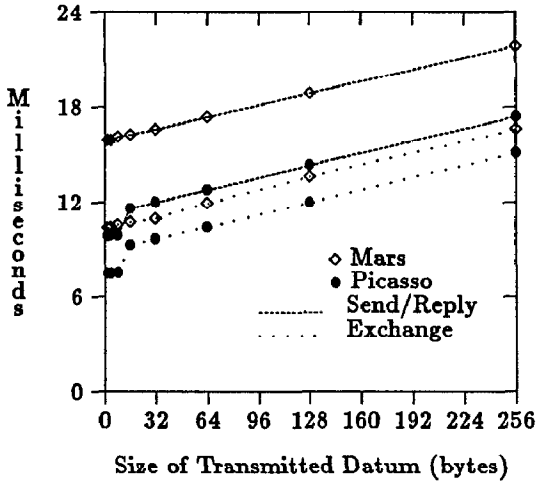

Figure 3: Simple Communication Benchmarks

Because the transmission delay is experienced at *every* node by message traveling several hops, it is particularly for deleterious non-local communication patterns.

We have derived the values of t_m and t_l for both Mars and Picasso using the results of the simple message transmission in Figure 2, combined with the data of Figure 3. The latter figure shows the performance of a *message exchange* and a *send and reply*. In an exchange, both nodes first send a message; each node then then receives the message sent by the other. In a send and reply, the message is sent from one node, received by the other, and returned. Analysis of the constituent operations yields equations reflecting the performance of the simple transfer (T_{st}), send and reply (T_{sr}) and exchange (T_e)

$$\begin{aligned} T_{st} &= (t_m + t_l) + N \cdot t_b \\ T_{sr} &= 2(t_m + t_l + N \cdot t_b) \\ T_e &= t_m + 2(t_l + N \cdot t_b) \end{aligned}$$

where N is the message size in bytes. Using these equations, we can derive t_m, the message latency, and t_l, the message preparation time:

$$\begin{aligned} t_m &= T_{sr} - T_e \\ t_l &= T_e - T_{st} - N \cdot t_b \end{aligned}$$

Quantity	Definition
t_m	Message Preparation Time
t_l	Communication Latency
t_b	Transmission Time Per Byte
N	Message Size (Bytes)

Table 1: Terms Describing Message Latency

We can determine t_b, the per byte transmission time, from the data in Figure 2. Combining these results, Table 2 shows the values of t_m and t_l obtained for Picasso and Mars.

	t_m	t_l	t_b
Picasso	2.33	3.41	0.012
Mars	5.47	2.37	0.012

Table 2: Latency Components (milliseconds)

The time to transmit a single byte, t_b, is the same for both operating systems because it is largely determined by the memory bandwidth of the DMA channels and the serial transceivers. The difference in the transmission latency suggests that Picasso incurs greater overhead; this is plausible, given the additional overhead expected with an operating system designed for flexibility. However, because Mars has a significantly larger delay for message preparation, the *total* message latency is 27% lower for Picasso.

4 A Simple Experiment

The following experiment is a typical use of a research operating system. We use the modular design of Picasso to compare two message routing algorithms: *e-cube* routing and *shortest-queue* routing. The first algorithm, *e-cube* routing, was described in §2. The second algorithm is an adaptive routing method that exploits the rich interconnection of the binary N-cube. This algorithm enqueues a message on the link, drawn from the set of eligible paths to the destination, that has the shortest queue of outstanding messages. We compared the relative performance of the two algorithms by measuring the time to deliver messages using destinations drawn from two *destination distributions.*

The *uniform distribution* [14] provides equiprobable message destinations across all the nodes (i.e. each node is equally likely to be the recipient for a message). In a binary N-cube with network diameter N, there are $\binom{N}{K}$ nodes exactly K hops away.

In our experiment, each node generated three hundred messages of fixed size using the uniform distribution. Message sizes were 2, 8, 32, 128, and 512 bytes. Figure 4 shows the elapsed time for the experiment to complete.

It is clear from Figure 4 that there is little advantage to the shortest-queue routing *with this message distribution.* This is not unexpected. Because messages are drawn from a uniform distribution, and the binary N-cube is a symmetric network, queue lengths should be the same on all transmission links. Only

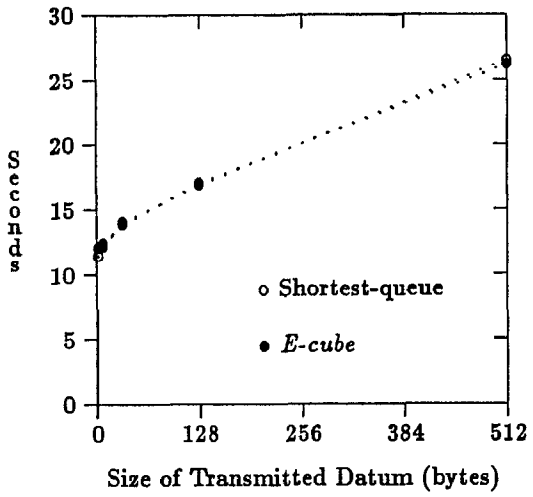

Figure 4: Uniform Distribution

for short transmission times will variability of queue lengths affect routing algorithm performance. Indeed, Figure 4 shows that the shortest-queue algorithm is marginally better for very short messages.

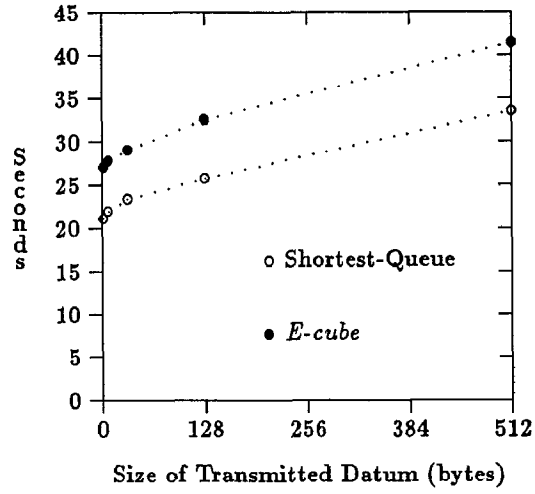

Figure 5: Hot Spot Distribution

To investigate the performance of shortest-queue routing in the presence of a network *hot spot,*[4] we modified the message distribution to send every fifth message to the node designated as the hot spot. Figure 5 shows that the skewed network distribution evinces the benefit of shortest-queue routing, yielding a 25% decrease in elapsed time. We conclude that shortest-queue routing decreases total message delay by reducing irregularities in the skewed message distribution.

This experiment suggests that shortest-queue rout-

[4] A hot spot is a region with significant network contention.

ing might be a good replacement for *e-cube* routing as a general purpose routing algorithm. However it is important to remember that shortest-queue routing can deadlock. Because it was designed to support adaptive message routing, Picasso uses a timeout mechanism to detect and recover from message deadlocks. The overhead for deadlock detection and recovery must be balanced against the potential performance gains of adaptive routing.

Although the results of this experiment illustrate the potential benefits of adaptive routing, the important point is the *ease* of conducting the experiment. The entire experiment, including writing the shortest-queue router, required less than a day. This would not have been possible were it not for Picasso's modular design.

5 The Picasso Project

Picasso is both a multicomputer operating system and a research project. Ancillary to the implementation of the Picasso operating system, we are developing support tools and investigating related issues via both analysis and simulation. Examples include the development of a multicomputer benchmark set [14], simulation studies of adaptive message routing algorithms [8], simulation studies of communication paradigms and their associated hardware support [4], and performance visualization tools [10]. In the remainder of this section, we briefly summarize these supporting research efforts.

To isolate the effects of hardware and software, and to explore their interaction, Grunwald and Reed [14] developed a multicomputer benchmark set and associated methodology. This benchmark set includes four components: simple processor benchmarks, synthetic processor benchmarks, simple communication benchmarks, and synthetic communication benchmarks. The simple processor benchmarks are, as the name implies, simple enough to highlight the interaction between processor and compiler, including the quality of generated code. In turn, the synthetic processor benchmarks reflect the typical behavior of computations and provide a ready comparison with similar benchmarks on sequential machines. Communication performance is closely tied to system software. Some hypercubes support only synchronous communication between directly connected nodes; others provide asynchronous transmission with hardware or software routing support. In both cases the simple communication benchmarks measure both the message latency as a function of message size and the number of links on which each node can simultaneously send or receive. For systems that support routing and asynchronous transmission, the synthetic communication benchmarks reflect communication patterns in both time and space.

Kim [8] has explored adaptive packet routing algorithms and their suitability for multicomputer network implementation. Five existing packet routing algorithms were studied (*NRCC, random, shortest-queue, delta, and priority queue*), and based on extensive simulation studies, a new algorithm, *hybrid weighted* routing, was proposed and validated. The most promising of these algorithms are currently being implemented as adaptive routing modules in the Picasso operating system.

To study the performance of existing and proposed network implementations, Grunwald has conducted parametric simulation studies of several message routing paradigms[4], including store-and-forward message switching, circuit switching, staged circuit switching, wormhole circuit switching, and an adaptive circuit switching algorithm developed at the NASA Jet Propulsion Laboratory. All simulations assume an equivalent hardware implementation, isolating implementation idiosyncrasies from the message routing paradigm. Using the results of these simulations, Grunwald is developing qualitative and quantitative analytic models that provide insight into network behavior.

Scientific applications on high-speed computing systems can quickly generate vast quantities of numerical data. Although the human visual system is remarkedly adept at interpreting and identifying anomalies in false color data, the importance of visual, scientific data presentation has only recently been recognized[3]. Large, complex parallel systems pose equally vexing *performance interpretation* problems. Data from hardware and software performance monitors must be presented in ways that emphasize important events while eliding irrelevant details. In collaboration with the Center for Supercomputing Research and Development at the University of Illinois, we are developing a suite of performance visualization tools. HyperView[10], one such tool designed specifically for multicomputer networks, dynamically displays node status and communication traffic.

6 Future Directions

Multicomputers have only recently begun to mature. This maturity is reflected by the improved performance of their interconnection networks and their support for peripherals and heterogeneous networks. Early network implementations constrained the space

of feasible algorithms, requiring strict communication locality to achieve good system performance; second generation networks eliminate most constraints on the algorithm design space by reducing average message latency. On early hypercubes, nodes could only access peripherals via a host processor; second generation systems embed peripherals in the fabric of the multicomputer network.

The increased performance and flexibility of second generation multicomputer networks provide fertile ground for operating systems research and development. Developing a message passing application currently can be likened to writing an operating system. The user must partition his or her computation and assign the tasks to individual nodes in a way that balances the computational load while attempting to minimize communication costs. Conventional operating systems do not encourage or even permit users to provide page replacement algorithms on virtual memory systems; it is unreasonable to expect users to assign tasks to processors in a message passing system. Dynamic load balancing will permit applications to create new tasks as they execute. The operating system will assign these tasks to nodes as appropriate. Similarly, adaptive routing paradigms, either in software or hardware, will avoid congested areas of the network when transmitting messages and accessing peripherals. Global, virtual memory will permit efficient data sharing across network nodes. Performance visualization tools will permit the operating system developer and application programmer to tune the system, maximizing performance.

To reach these goals, a malleable, portable *research operating system* is needed. The Picasso operating system was designed to fill this need.

References

[1] AMETEK COMPUTER RESEARCH DIVISION. *Ametek System 14, Mars System Software User's Guide Version 1.0.* Arcadia, California, 1987.

[2] FLOWER, J., OTTO, S., AND SALAMA, M. *A Preprocessor for Irregular Finite Element Problems.* Tech. Rep. C3P-292, California Institute of Technology, June 1986.

[3] FRENKEL, K. A. The Art and Science of Visualizing Data. *Communications of the ACM 21*, 2 (Feb. 1988), 110–121.

[4] GRUNWALD, D. C., AND REED, D. A. Networks for Parallel Processors: Measurements and Prognostications. In these proceedings.

[5] GUSTAFSON, H. L., HAWKINSON, S., AND SCOTT, K. The Architecture of a Homogeneous Vector Supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing* (Aug. 1986), pp. 649–652.

[6] HAYES, J. P., MUDGE, T., STOUT, Q. F., COLLEY, S., AND PALMER, J. A Microprocessor-Based Hypercube Supercomputer. *IEEE Micro 6*, 5 (Oct. 1986), 6–17.

[7] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (Aug. 1978), 666 – 677.

[8] KIM, C. *Integrated Switching Networks; A Performance Study.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1987.

[9] KUCK, D. J., DAVIDSON, E. S., LAWRIE, D. H., AND SAMEH, A. H. Parallel Supercomputing Today and the Cedar Approach. *Science 231* (Feb. 1986).

[10] MALONY, A. D., AND REED, D. A. HyperView: A Tool for Performance Data Visualization. In preparation.

[11] POPLAWSKI, D. A., AND RICH, D. O. Code Paging on Hypercubes. In *Proceedings of the 1987 International Conference on Parallel Processing* (1987), pp. 710–716.

[12] RATTNER, J. Concurrent Processing: A New Direction in Scientific Computing. In *Conference Proceedings of the 1985 National Computer Conference* (1985), AFIPS Press, pp. 157–166.

[13] REED, D. A., AND FUJIMOTO, R. M. *Multicomputer Networks: Message-Based Parallel Processing.* The MIT Press, 1987.

[14] REED, D. A., AND GRUNWALD, D. C. The Performance of Multicomputer Interconnection Networks. *IEEE Computer 20*, 6 (June 1987), 63–73.

[15] SIETZ, C. L. The Cosmic Cube. *Communications of the ACM 28*, 1 (Jan. 1985), 23–25.

[16] WHITBY-STREVENS, C. The Transputer. In *Procedings of the 12th International Symposium on Computer Architecture* (Boston, Mass., June 1985), pp. 292–300.

[17] WOLFE, M. J. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1982. Report No. UUCDCS-R-82-1105.

[18] WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., AND POLLACK, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM 17*, 6 (June 1974), 337–345.