The Evolution of HPC/VORX

Howard P. Katseff
Robert D. Gaglianello
Bethany S. Robinson

AT&T Bell Laboratories
Holmdel, NJ 07733

Abstract

HPC/VORX is a computing system that provides closely coupled computing between large numbers of processors. It also supports the connection of many host workstations which may be geographically distributed within the area of a large building and allows a single applications to span many processors and many workstations. We relate some of the lessons that were learned while building and using HPC/VORX and in the transition to HPC/VORX from a smaller, less capable system. The problems that we encountered included difficulties in scaling resource managers and human interfaces to large numbers of processors, the design of communications primitives and protocols, and the implementation of programming abstractions.

1. INTRODUCTION

HPC/VORX is a *local area multicomputer* system that combines the major strengths of multicomputer systems and local area networks [1]. Like multicomputers, it exhibits low latency communications, allowing the close cooperation of many processors to work on a single large application. It also provides for the connection of workstations and other resources that are distributed within the area of a large building, but with better communications performance than is usually found in local-area networks. The current system connects ten SUN 3 workstations and a pool of 70 processing nodes based on the Motorola 68020 and has been operational since early 1988. The system can easily be expanded to more than a thousand nodes by replicating the interconnect hardware.

The HPC/VORX system is based on a high bandwidth, low latency interconnect called the HPC and is controlled by the VORX distributed operating system. A conceptual diagram of HPC/VORX is shown in Figure 1. The right side of the diagram shows resources normally found in a local area network and the left side is the pool of processing nodes that is used for compute intensive or closely coupled medium grain parallel applications. Applications on the nodes may be controlled from any workstation and it is possible to build a single application that spans many workstations and many nodes.

HPC/VORX has proven to be a useful base for implementing a variety of applications. Applications implemented on HPC/VORX range from the Rapport multimedia conferencing system [2] to several circuit simulators. Because HPC/VORX allows high performance communications with workstations, it can be used to experiment with applications such as multimedia conferencing between workstations, with real-time video and high-fidelity audio transmission between conferees. Because it has enough bandwidth to transmit large bitmap images in real-time, HPC/VORX allows us to design prototypes of next-generation workstations that minimize the amount of hardware on a user's desk by distributing the workstation's intelligence over the network.

The VORX operating system is a direct descendent of Meglos [3], an operating system for the S/NET multicomputer system [4]. The S/NET–Meglos system was operational for about four years, but fell into disuse in 1988 when HPC/VORX became available. Several multiprocessor real-time applications that controlled robot arms and acquired video data were implemented on S/NET–Meglos [5]. It was also used to implement a variety of multiprocessor applications, including circuit simulators [3] and the Linda parallel language [6]. The S/NET was symmetric, providing low latency, high bandwidth communications between any pair of its processors. This was an improvement over systems with slower communications that forced programmers to consider the hardware topology while designing their

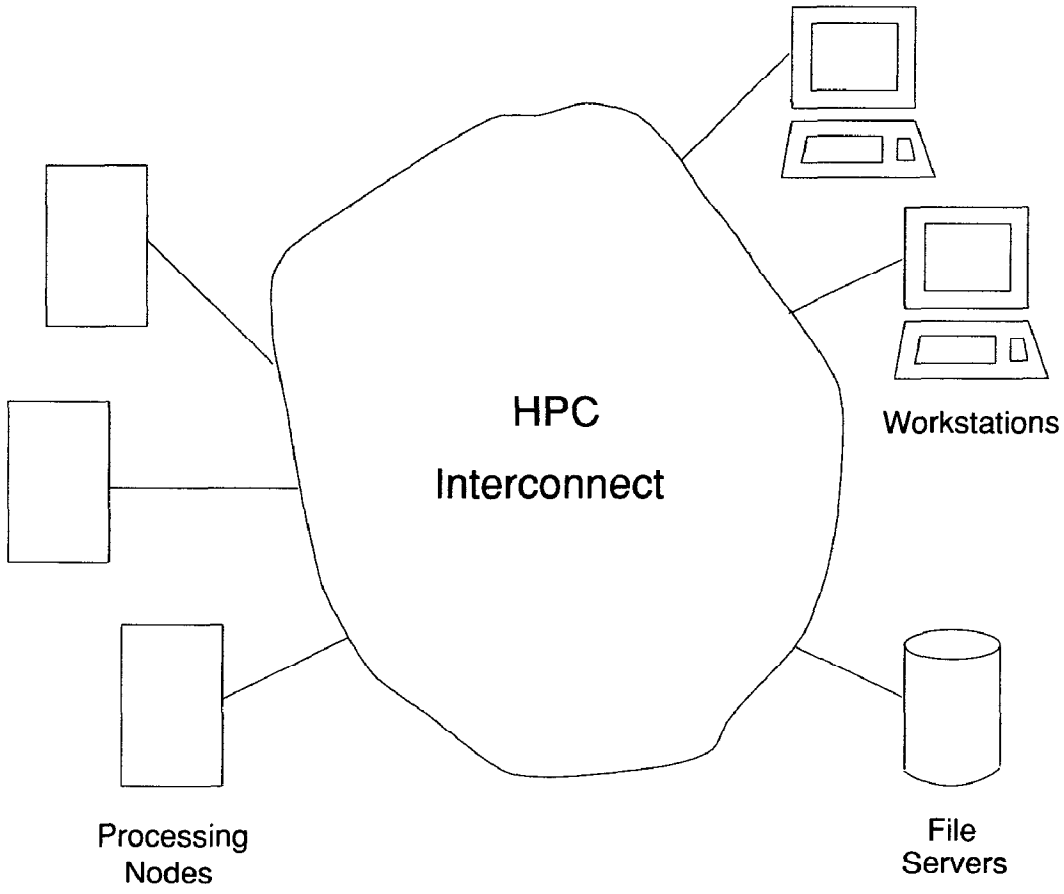

Figure 1. A Typical Local Area Multiprocessor System

applications [7].

The most significant limitation of the S/NET was that it supported only a small number of nodes. The number of nodes was limited because it used a single bus for communications between all its nodes. The largest system had 12 processing nodes and most systems had only eight. In contrast, the HPC interconnect is designed in a modular fashion, allowing systems ranging in size from two to several thousand nodes. The HPC consists of several self-routing star networks called *clusters*, each of which contains twelve ports. A port contains independent input and output sections that simultaneously run at 160 Mbit/sec and can connect to either a workstation, a processing node, or to another cluster. Fiber optic cables permit these connections to be over a kilometer in length.

A twelve node system can be constructed using a single cluster. Larger systems are built by using some port connections for processing nodes and some for connections to other clusters. While the hardware allows connections with arbitrary topologies, we have chosen to connect the clusters in the shape of an incomplete hypercube [8]. A hypercube-based system with 1024 nodes can be built with 256 clusters by using 8 of the 12 ports on each cluster for connections to other clusters and the other four for connections to processing nodes. Like the S/NET, hardware communications latency in the HPC is much smaller than the latency introduced by the communications software, so that applications programmers need not be concerned with the hardware topology.

While we planned for Meglos to eventually use large numbers of processors, it tended to be optimized for 12 or fewer processors, either to simplify its implementation or to provide a quick fix for some system bug. Some of the implementation decisions that we made for Meglos did not scale well to larger numbers of processors. This led us to consider the use of alternative schemes in VORX. This paper discusses some of the problems that we encountered while designing HPC/VORX and describes general experiences and lessons that we learned while building and using it. We begin with a discussion of problems that we encountered with hardware flow-control, follow with a description of user interface issues, and then relate our experiences with communications primitives and protocols, programming abstractions, and program development tools.

2. HARDWARE FLOW CONTROL

A major problem with the S/NET arose from problems with flow control in the S/NET interconnect. The hardware provided a fifo input buffer for each processor that could hold several incoming messages, with a combined length up to 2048 bytes. When the fifo became full, the receiver would reject messages sent to it and send a fifo-full signal to the transmitter for each rejected message. Because overflow was thought to be an uncommon event, overflow recovery was to be done in the low-level communications software by the processors that originated the rejected messages. When receiving the fifo-full signal, the originating processors were to continuously resend their message until it was successfully received [9].

We discovered that many multiprocessor applications have a natural synchronization in which many processors send a message to a single processor at nearly the same time. Instead of being a rare event, fifo overflow happened frequently, making it important for the system to continue to operate efficiently when the overflow occurred.

The original strategy of continually sending the message until it was received was found to be unsatisfactory because it could cause lockout to occur. A property of the S/NET interface hardware was that when overflow occurred, the fifo retained the portion of the message that was received up to the time of the overflow. The communications software in the receiving processor had to read and discard this initial portion of the message. When several processors were attempting to send a long message to a single processor, it was possible for the system to get into a state in which some of the messages were never received. This happened because all the sending processors were in their retransmission loop, continually sending messages that were discarded by the receiver, but the receiver could not remove words from its fifo fast enough to make enough room for an entire incoming message before a new message arrived.

This problem forced us to consider other flow-control schemes. One approach was to use random-length timeouts, as is done on the Ethernet [10]. This eliminates the problem of busy loops in the kernel, but when many messages need to be retransmitted, communications runs at the timeout rate; at least an order of magnitude slower than the expected communications rate. Finally, we considered a reservation protocol, in which a processor sends a short message requesting to send its data, and does not send the data until it receives an acknowledgement from the receiver. If the receiver only authorizes one sender at a time and the receiving fifo is large enough to simultaneously hold request messages from each of the processors and one data message then this scheme would eliminate overflow. However, we rejected this scheme because the extra software and communications overhead would increase latency for all messages.

In the end, we never implemented code in Meglos to reliably recover from overflow. Instead, we required applications to limit the lengths of messages when they performed many-to-one communications, so that fifo overflow never occurred. This was feasible because the S/NET allowed only a small number of processors. For instance, 12 processors could each send a 150 byte message to a single processor without overflowing its fifo.

In contrast with the S/NET, flow-control in the HPC is implemented entirely in the interconnect hardware. This makes loss of messages due to buffer overflow impossible. Because the HPC provides reliable message transmission, the need for implementing recovery mechanisms in the communications software is entirely eliminated. Messages sent via the HPC are limited to some length (1060 bytes in the current implementation). Each HPC link, either between a processor and cluster or between two clusters, refuses to accept a message unless the hardware has room to buffer an entire message, forcing the sender to wait until the space is available. For outgoing processor links, the processor receives an interrupt when room becomes available. This scheme guarantees that messages are never lost by the interconnect and a fair hardware scheduling mechanism ensures that every sender is eventually serviced. It never deadlocks because the VORX kernel reads in messages immediately when they arrive.

3. USER INTERFACE

3.1 Processor Allocation

Because a typical Meglos system had only eight processors and two or three programmers would often want to debug their applications simultaneously, we designed Meglos to make it easy for users to share their processors with each other. Meglos allowed up to 15 independent processes to run on a processor, each with its own address space protected from access by other processes, However, we found that programmers did not want to share their processors because they wanted to balance the computational load of their application in a repeatable fashion. Realizing our mistake, we added "exclusive access" capabilities to exclude other processes from a processor.

The strategy used by Meglos to allocate processors was designed to maximize sharing between users. In Meglos, processors were allocated to an application when it started running. When the application finished, its processors were returned to the free pool and were immediately available for use by another application. This scheme met our goal of maximizing processor availability, but had other problems. While debugging

applications, programmers typically would run them, find a mistake, make changes and recompile, and then run the modified version of the application. It often happened that while a programmer was recompiling, somebody else would start their application on the remaining processors with exclusive access, so that when the programmer tried to run the modified program, he would receive the diagnostic, "processors not available." This problem was normally resolved by informal agreement among the users as to how many processors each would use. This was possible because all the users were only a few offices away from each other.

Because its users are geographically distributed, VORX formalizes the allocation of processors to users by requiring a user to allocate all the processors that he needs before running an application. The processors are not available to anyone else until they are explicitly freed by the user. This scheme eliminates the problem with processors disappearing in the middle of a program development session and appears to be better than our previous scheme.

The problem with this scheme is that users sometimes forget to free their processors when they are finished. We have considered automatic techniques to recover these processors such as automatically freeing them when a user logs off their workstation or when there is no activity for several hours. We also considered a scheme where after exhausting all the processors allocated to the user, VORX would obtain processors from the pool of unallocated processors or perhaps from other users. Since each of the alternatives that we have considered has objectionable properties, we provide a command that allows a user to free processors allocated to other users, and request that it be used carefully.

3.2 Resource Management

All resource management in Meglos was centralized on a single host. While this is appropriate for a small system, it causes a serious performance bottleneck for systems with over ten processors. The problem was most apparent in the first few seconds of execution when an application was being downloaded and then while it set up its communications channels. Both Meglos and VORX provide named communications channels that are dynamically created and destroyed during program execution [11]. Each channel has an arbitrary name, and two processes rendezvous on a channel by specifying its name in an open call. The bottleneck in setting up communications occurred because all the channel opens were processed by the single resource manager on the host.

We solved this problem in VORX by splitting the resource manager into several functional pieces and replicating the individual pieces for increased performance. One of the pieces, the communications object manager, is replicated onto every processing node allocated by the user. The object manager uses distributed hashing [12] to map a channel name to a particular processor to determine which object manager should handle the open for a particular channel. This technique ensures that two processes that open a channel with the same name always hash to the same object manager so it can perform the open. Because there are as many object managers as processing nodes, the channel opening bottleneck is eliminated.

Another bottleneck in Meglos was that all program developers and users ran their applications from a single host. VORX eliminates this problem by allowing programs to be run from different hosts. Each host has its own process resource manager that is responsible for applications started on that host and for keeping track of the mapping of applications to processors. Program downloading, file access, and other system services are also spread among the host workstations.

3.3 The Execution Environment.

Each process running on a processing node has a stub process running on the host. The stub is responsible for initially downloading the process and for providing a UNIX® operating system environment while the program is running. Each time a system call (such as a `write` to a file) is executed on the processing node, it sends a message to the stub. The stub then executes the system call and passes the results back to the node. This method perfectly replicates the host environment on the node. Unfortunately, it is slow to start applications with many processes. For instance, it takes 12 seconds to download and initialize a process on each of 70 processors. Most of this time can be attributed to work centralized on the host: the host creates 70 stub processes, channels are set up between each process and its stub, and each stub independently downloads a copy of the program.

For the case where all the application's processes use the same object code as each other, VORX offers an alternative method in which one stub services all the processes of the application and uses a tree scheme in which the stub downloads only one processing node. That processor copies the text to be downloaded to two other processors as the text is being received. Each of these processors copy the text to two other processors. The fanout continues in a tree-like fashion to reach all the processors that are to run the application. With this method, it takes only two seconds to download and start 70 processes.

The problem with this method is that the host environment is not duplicated correctly. For instance, if one of the processes issues a UNIX system call that blocks, such as a read from the keyboard, then the stub does not process system calls from any of the other processes served by that stub until the original system call completes.

When each process has its own stub, the system calls issued by one process are processed independently of the other processes. Another problem is that the stub process is limited by the SunOS kernel to 32 open file descriptors, imposing a limit of 32 open files for all the processes of an application combined. With one stub per process, each process can open 32 files.

Because VORX allows the programmer to specify how processes are allocated to stubs, an application with processes that use blocking system calls or opens many files can arrange for those processes to each have their own stub. We are working on a better solution to these problems that will alleviate the bottleneck of using a single host for all the system calls of an application. It uses a decentralized scheme that distributes the overhead of system calls by allowing a process to direct system calls to any of the host workstations.

4. CHANNELS

Channels provide low latency, high bandwidth message passing communications between processors. In VORX the software end-to-end latency between application programs running on separate 25 MHz Motorola 68020 processing nodes for four byte messages is 303 µsec and 1024 byte messages can be sent at the rate of 1027 kbyte/sec, substantially better than performance of other systems with comparable processors but other interconnects [13]. Channels are also easy to use: they are set up with a single `open` call and data is transferred with `read` and `write` calls [11]. There are also specialized calls for operations like multiplexed read in which a process blocks until data arrives from one of several channels and a mechanism that allows servers to continually reuse a single channel name. Our hope was that channels would be sufficiently fast and flexible to suit the communications needs of all applications.

Channels are implemented in Meglos with a stop-and-wait protocol [14]. When a process issues a `write` call, that processor's kernel sends the data to the destination processor, and blocks the process until an acknowledgement message arrives from the receiving kernel. If there is no buffer space available (a rare occurrence in VORX because the kernel has many side buffers to hold these messages), the receiver requests retransmission when buffer space becomes available. The most important attribute of this protocol is that it implements flow-control by preventing a second message from being sent until this first one is processed. This prevents the reader from being inundated with messages.

A stop-and-wait protocol was chosen because it is simple to implement and has little processing overhead. For networks with low latency, like the S/NET and the HPC, such protocols work well because the acknowledgement arrives with little delay [14]. In our early work with the S/NET, we were unsure of its error characteristics and implemented error detection and recovery in the channel protocol. This was done efficiently with stop-and-wait because the sending process blocks until the message was successfully received, eliminating the need for the kernel to make a copy of the message before sending it.

4.1 Alternative Communications Protocols

While most of our users are happy with channels, some are not. For example, when using Meglos, the implementors of Linda [6] needed a different type of semantics: multicast with no explicit flow control, and the CEMU group [15] wanted to experiment with various low-level communications protocols for their circuit simulator. Meglos allowed these applications to circumvent the channel communications mechanism by directly accessing the communications hardware to send or receive messages and by responding to communications interrupts when messages arrived.

In VORX a general interface for user-defined communications objects is provided. As in Meglos, processes can access the hardware registers from their applications, eliminating the overhead of supervisor calls into the kernel and can specify interrupt service routines to handle incoming messages. This allows the programmer to use whatever low-level protocols are appropriate for the application. Other application-specific input and output techniques, such as scatter/gather may also be implemented. VORX allows user-defined communications objects and channels to coexist and permits several user-defined objects, each with its own protocol, to be simultaneously used. User-defined communications objects are integrated with the object manager, allowing these objects to use the same rendezvous mechanism as channels.

We have seen two ways in which users can write protocols with better performance than channels. One is to use sliding-window protocols [14] and the other is to use no flow-control protocol at all.

Guided by the experiments done with the CEMU simulator using sliding-window protocols [15], we have seen that a sliding-window protocol can be more efficient than a stop-and-wait protocol, even with very low latency interconnects like the HPC. In a sliding-window protocol, the reading processor starts by sending a protocol message to the sending processor indicating the amount of free space available in the input buffer. The sender then can send as many messages as would fit in the buffer without further protocol messages from the receiver. As the reader removes data from the buffer, protocol messages are sent to the sender updating the amount of buffer space available. To obtain improved performance, the number of update messages should be kept small, but should be sent often enough to maintain concurrency between the sender and the receiver.

Unfortunately, tuning the protocol to find a proper update rate must be done in an application-specific manner.

Unlike the stop-and-wait protocol, error recovery in the sliding-window protocol requires significant processing overhead. Either the kernel must copy messages into a safe place until it is assured that the messages have been correctly received or the sending process must delay reusing its buffers until it is notified by the kernel that they have been correctly received. However, none of this is necessary in HPC/VORX because the HPC hardware guarantees correct message delivery.

To determine the efficacy of sliding-window protocols, we benchmarked a sliding-window user-defined protocol that allowed messages of some fixed length to be sent between two processors. Both the sender and receiver know the length of the messages. The receiver initially sends k buffer-available messages to the sender, where k is the maximum number of messages that fit in its available buffer space, and thereafter sends one buffer-available message each time a message is received. The sender keeps its own count of the number of receiver buffers available. The count is initially zero, is incremented for every buffer available message received, and decremented for every message sent. If the count is greater that zero, the sender can send a message immediately, otherwise it blocks until the count becomes greater than zero. For our benchmark, the sender transmitted 1000 messages and the resulting communication latency is computed by dividing the elapsed time by 1000. We varied the number of available input buffers to determine its effect on communication latency. The data are shown in Table 1. For comparison, we benchmarked channel communications for the same message sizes and obtained the latencies shown in Table 2.

Even with a simple protocol and two buffers, a sliding-window protocol obtained better latencies than the highly optimized channel protocol. The performance gain is obtained because the sender can send a message immediately whenever extra buffers are available. With channels, a message can never be sent until a software acknowledgement is received for the previous message. This result suggests that we should consider the use of a sliding-window protocol for channels and strengthens our belief that programmers can obtain better communications performance when they tailor a communications protocol to their application.

An alternative available to some applications is to use no flow-control protocol at all. Consider an application with two processes that alternately send a message back and forth. If each process ensures that it has enough buffer space to hold an incoming message before it sends a message, then when either process sends its message, it is assured that the message will be received. The message always arrives because the hardware provides reliable communications and the application guarantees that buffer space is available. Such a scheme can be used by most applications with natural synchronization between their processes. User-defined communications objects were successfully used in a parallel implementation of SPICE that needed very low latency communications to solve large sparse linear systems [16]. It was able to obtain 60 μsec software latencies for 64 byte messages with direct access to the communications hardware and no low-level protocol.

In our experiments with transmitting real-time bitmap images to workstations, we wanted to obtain the maximum possible communications bandwidth from the HPC. We did so by having the processor originating the bitmap image send it to the HPC interconnect as fast as it could and for the workstation receiving the bitmap to copy it from the HPC directly to its frame buffer. Because all flow control was done by the HPC hardware, the protocol overhead was only the few statements needed to determine where to place the incoming bitmap data in the frame buffer. With this simple technique, we obtained a rate of 3.2 Mbyte/sec, sufficient to refresh a 900×900 pixel portion of a monochrome (bi-level black and white) display 30 times per second from a remote processor.

4.2 Multicast is Inappropriate

When formulating multiprocessor applications, many programmers design their applications to make use of a multicast mechanism in which each process sends the identical message to many other processors. We therefore designed the HPC hardware to be able to implement multicast efficiently and devised a flow-controlled multicast primitive that is integrated with channels [17].

However, we discovered that when programmers implement algorithms that were originally formulated with multicast, they often find multicast to be inappropriate. The problem with multicast is that as the number of processors is increased, the number of messages received by each processor grows and each process spends more and more time reading data that it is not concerned with. It is usually better for the sender to produce a different message for each receiver that contains only the data that it needs.

For example, consider the calculation of a two-dimensional Complex Fast Fourier Transform (2DFFT), a computation used by image processing applications. The 2DFFT of a 256×256 grey scale image is computed as follows:

- Compute a 256-point one-dimensional Complex FFT (1DFFT) for each row in the original image, producing a row of 256 complex numbers for each row in of the original image.

Number of Buffers	4 Byte Messages µsecs/msg	64 Byte Messages µsecs/msg	256 Byte Messages µsecs/msg	1024 Byte Messages µsecs/msg
1	414	451	574	1071
2	290	317	412	787
4	227	251	330	644
8	196	218	289	573
16	179	200	267	535
32	172	192	257	518
64	164	184	248	504

Table 1. *Message Latency for Reader-Active Communications Protocol.*

4 Byte Messages µsecs/msg	64 Byte Messages µsecs/msg	256 Byte Messages µsecs/msg	1024 Byte Messages µsecs/msg
303	341	474	997

Table 2. *Message Latency for Channel Communications.*

- Compute a 256-point 1DFFT for each column from the 1DFFT's computed in the first step, producing a column of 256 complex numbers for each column computed in the first step.

Computing the 2DFFT with multiple processors is straightforward. Since each 1DFFT is computed independently of the others, the 1DFFT's may be computed concurrently on separate processors. After the first step, the processors distribute the results of their computation to each other so that all processors have a column of data for the second step. The processors compute the 1DFFT on their column of data, and the 2DFFT is completed. Assuming that 256 processors are available and that communications has zero cost, the 2DFFT can be computed in the time it takes to compute 2 1DFFT's.

One approach for distributing the results of of the first step is for each processor to multicast its entire row to all the other processors. The problem with this approach is that each processor reads 65536 numbers of which only 256 are needed. A better approach than using multicast is for each each processor to send a different number to every other processor. By sending a single message containing one number to each processor. The latter technique requires the receiver to process only the 256 numbers it needs.

We have found some limited uses for multicast. For instance, it may be necessary for a process to multicast initial values to all the other processes when the application is first started. Other applications (especially local area network servers such as distributed file servers) sometimes need multicast, but only to a few receivers. This can be done with reasonable efficiency by issuing multiple writes.

5. SUBPROCESSES

Both Meglos and VORX allow a process to be subdivided into subprocesses. Like threads in Mach [18], subprocesses are parts of a process that execute asynchronously with each other. Each subprocess is an independently scheduled thread of execution that may block for communications or other events without affecting the execution of the other subprocesses of an application. All the subprocesses of a process share the same address space but each subprocess has its own stack for its local variables and for the kernel's machine state information.

Subprocesses were originally included for real-time applications that controlled hardware devices, such as robot arms and cameras connected to the processing nodes. Because distinct execution priorities can be specified for each subprocess and the scheduler is preemptive, the programmer had enough control over switching between and scheduling of subprocesses to be able to effectively implement real-time applications [5].

We found that subprocesses provide a useful way to structure applications that had no real-time or device control aspects to them. A common way to structure applications is to have at least three subprocesses for each process: one for input, one for output, and one or more to do the actual computation. The subprocesses communicate with each other by semaphores (provided by VORX). This subdivision of work makes it easy to do the computation concurrently with input arriving or output departing.

Because the VORX scheduler is preemptive, a subprocess can be interrupted to service interrupts or to start higher priority subprocesses. To allow for preemptive scheduling, VORX saves all a processor's registers when it does a context switch between subprocesses. A context switch, which includes saving both fixed and floating point registers takes 80 μsec using a 25 MHz Motorola 68020 with a Motorola 68882 floating point coprocessor.

Because context switching is too slow for some applications, program structuring techniques other than subprocesses have been used. One approach is to use a single subprocess that never switches context. Communications interrupts are disabled and user-defined objects are used to test for input at convenient places in the program. If input is available then the program can read the incoming data without blocking. This scheme was also used in the parallel SPICE work [16].

It is possible to use coroutines to provide multiple threads of execution within a subprocess, as was done in CEMU [15]. Coroutines have less overhead than subprocesses because coroutine switches occur only at well defined places in the application code, so that most registers need not be saved when switching between coroutines. Another alternative is interrupt level programming. Here, a single subprocess starts application-specific input and output interrupt service routines and then suspends itself. The entire computation is done by the interrupt service routines. This technique runs efficiently in VORX because it does not incur the overhead of restoring or saving registers when switching to or from a suspended process.

6. *PROGRAM DEVELOPMENT TOOLS*

The only debugging tool available under Meglos was *vdb*, a symbolic debugger derived from the *sdb* [19] debugger. *Vdb* includes a few enhancements, such as the ability to switch between subprocesses to examine their local variables, but is basically a single process debugger. When used on a workstation with a window system, it is possible to do breakpoint debugging on a multiprocess application by starting several copies of *vdb* in separate windows. Each copy of *vdb* controls the execution of one process of the application. By switching between windows, the programmer can simultaneously debug all the processes.

When there are more than a few processes, this method becomes unwieldy because the programmer cannot remember what he is doing in each window. In practice, programmers usually run one or two processes with the debugger and run the other processes normally. Because the programmer may not know in advance which process needs to be debugged, VORX makes it possible for the programmer to attach *vdb* to any process that is running and to switch between the processes of his application. Despite the problems of dealing with many processes, *vdb* is still a popular tool for debugging multiprocessor applications.

6.1 The Communications Debugger

While we do not know how to deal with the complexity of large numbers of processes in general, we have created a tool that is useful for dealing with a type of program bug that is surprisingly common in applications with many processes. The symptom, which is caused by a programming error, is that the application stops running with each process waiting for input from another process. The VORX communications debugger, *cdb*, helps debug such deadlocked applications by allowing the programmer to examine the communications state of the application [20]. This information can be used to determine which messages caused the problem and often can help isolate the process that caused the deadlock to occur.

For each channel, the state reported by *cdb* consists of the name of the channel, which two processes it connects, how many messages have been sent in each direction on the channel and most importantly, the state of each end of the channel. The channel state includes information such as whether an application is blocked waiting for input or output on the channel. Because an application may have a large number of channels, *cdb* includes several filters to help isolate the channels of interest. *Cdb* was easy to implement because most of the information that it needs was already encoded in the communications driver. *Cdb* has proven to be easy to use and has become a useful tool for examining deadlocked applications.

6.2 The Software Oscilloscope

The *prof* profiling system [21] available in VORX can be run on a process to show how execution time is divided up among different parts of the program. Typically one finds that a large portion of the execution time is spent in a small section of the code. This part of the program can then be examined and carefully rewritten.

While a process profiler can help improve the performance of a multiprocessor application, execution characteristics that a process profiler does not measure are also important. The major problem is one of improper load balance in which processors spend time waiting for data from other processors instead of doing useful work. A related problem is that communication between processors is often more expensive than envisioned by the designers of an application, exacerbating the load balancing problem.

VORX includes a tool called the *software oscilloscope* [20] that helps the programmer visualize how well processors of an application are utilized and how well the computational load is balanced. It runs on a color

workstation and displays a graph for each processor indicating CPU time usage with different colors used to partition time into several categories. Two of the categories are quite standard: *user* time in which application code is executed and *system* time in which operating system code is executed. The remainder of the time is *idle* time in which the processor is doing no useful work.

Because programs use messages to communicate, idle time can be further partitioned to provide more information. The processor may be idle because the program is waiting for input or it may be idle waiting for output. Because the kernel allows multiple threads of execution within a processor, a third possibility for idle time is that some threads are waiting for input and others are waiting for output. Finally, the processor may be idle for some other reason, such as waiting for access to a local disk.

Execution data is recorded while the application is running and later the software oscilloscope is used to display the data. The software oscilloscope synchronizes all the graphs with each other, so that when several graphs are displayed, each shows the same interval of execution time. It is possible to freeze the display, run faster or slower than real-time, or seek to any moment in execution time. This tool works well when the application has few enough processors so that all the graphs fit on the screen. We are studying ways to effectively display data for more processors.

7. CONCLUSIONS

VORX has proven to be a useful testbed for experimenting with multiprocessor applications and for experimenting with different techniques for their implementation. Some of our success resulted from being able to build on experiences from a previous generation of hardware and software. Because some of our designs for the earlier system, particularly in the areas of user interfaces and programming tools, did not scale to large scale systems as well as we hoped, we developed better techniques for VORX.

We found that the designers of programming languages and some applications programmers want the ability to experiment with low-level communications protocols, so VORX provides an extensible environment with user-defined communications objects. Even for these users, the standard VORX communications environment is useful for initial implementation and debugging. Program monitoring tools can then be used determine the performance requirements for user-defined communications objects. It is often the case that the standard environment provides adequate performance, saving the programmer from having to design his own protocols.

The local-area multiprocessor approach is an improvement over traditional multiprocessor systems that are controlled from a single host. It allows the user community to be split over many host workstations, distributing the burden of administering the multiprocessor. It has the further advantage of supporting applications that span multiple host workstations and processing nodes by making use of the high performance communications available to the host workstations.

REFERENCES

1. Gaglianello, R. D., et. al., "HPC/VORX: A Local Area Multicomputer System," *Proc. Ninth Internat. Conf. on Distr. Comput. Sys.*, June 1989, Newport Beach, 246-253.

2. Ensor, J. R., et. al., "The Rapport Multimedia Conferencing System—A Software Overview," *Proc. Second IEEE Conf. on Comput. Workstations*, Washington, March 1988, 52-58.

3. Gaglianello, R. D. and H. P. Katseff, "Meglos: An Operating System for a Multiprocessor Environment," *Proc. Fifth Internat. Conf. on Distributed Comput. Syst.*, Denver, May 1985, 35-42.

4. Ahuja, S. R., "S/NET: A High Speed Interconnect for Multiple Computers," *IEEE J. on Selected Areas in Communications* **SAC-1,** 5, November, 1983.

5. Gaglianello, R. D. and Katseff, H. P., "A Distributed Computing Environment for Robotics," *Proc. 1986 Internat. Conf. on Robotics and Automation*, San Francisco, April 1986, 1890-1896.

6. Carriero, N., and Gelernter, D., "The S/Net's Linda Kernel," *ACM Trans. on Comput. Syst.* **4**,2, May 1986, 110-129.

7. Seitz, C. L., "The Cosmic Cube," *Comm. ACM* **28**, 1, January, 1985, 22-33.

8. Katseff, H. P., "Incomplete Hypercubes," *IEEE Trans. on Comput.* **37**,5, May 1988, 604-608.

9. London, T. B., et. al., "Performance of an Interconnected Microprocessor System Designed for Fast User-level Communications," in *Concurrent Languages in Distributed Systems, Hardware Supported Implementation*, edited by G. L. Reigns, and E. L. Dagless, Elsevier Science Publishing Company, New York, March 1984, 125-134.

10. Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM* **19**, 7, July 1976, 395-404.

11. Gaglianello, R. D. and Katseff, H. P., "Communications in Meglos," *Softw. Pract. and Exper.* **16**,10, October 1986, 945-963.

12. Andrews, G. R., et. al., "Distributed Allocation with Pools of Servers," *Proc. ACM SIGACT/SIGOPS Symp. on Principles of Distrib. Comput.,* Ottawa, August 1982, 73-83.

13. Renesse, R. van, et. al., "Performance of the World's Fastest Distributed Operating System," *Operat. Syst. Rev.* **22**,4, Assoc. Comput. Mach., October 1988, 25-34.

14. Tanenbaum, A. S., "Network Protocols," *Comput. Surveys* **13**,4, December 1981, 453-490.

15. Ackland, B. D., et. al., "MOS Timing Simulation on a Message Based Multiprocessor," *Proc. IEEE Internat. Conf. on Comput. Design,* Port Chester, N. Y., October 1986, 446-450.

16. Narayan, S., personal communication, 1988.

17. Katseff, H. P., "Flow-Controlled Multicast in Multiprocessor Systems," *Proc. Sixth Internat. IEEE Phoenix Conf. on Comput. and Communicat.,* Scottsdale, February 1987, 8-13.

18. Accetta, M., et. al., "Mach: A New Kernel Foundation for Unix Development," *Proc. 1986 Summer USENIX Technical Conf.,* July 1986.

19. Katseff, H. P., "Sdb: A Symbolic Debugger–Version 3.0," part of documentation for the UNIX® System V Operating System, AT&T, 1980.

20. Katseff, H. P., "Debugging and Performance Monitoring in HPC/VORX," *Proc. First Usenix/SERC Workshop on Exper. with Building Distrib. and Multiproc. Syst. (WEBDMS),* Ft. Lauderdale, October 1989, 255-268.

21. 'PROF (1),' in *UNIX Programmer's Manual,* 4.2 Berkeley System Distribution **1,** Computer Science Division, University of California, Berkeley, CA, August 1983.