A PARALLEL PROCESSOR ALTERNATIVE TO THE MODULAR SIMULATOR ARCHITECTURE

Edward Kulakowski
David J. Kramer

Reflectone, Inc.
5125 Tampa West Boulevard
Tampa, FL 33614

ABSTRACT

The modular simulator concept has been proposed as a means of reducing the cost of training devices. With this approach a number of different self-contained, single function subsystems consisting of a processor, local memory, I/O, and related trainer hardware are combined. A local area network is used to link the subsystems into a complete trainer. There are a number of practical difficulties with implementing this approach. As an alternative, a parallel processor architecture is proposed in which a single parallel computer is used with a single I/O system. All processors share tasks and have equal access to shared memory and to the I/O bus. An experiment was conducted employing this architecture in an existing helicopter OFT. The results of the experiment indicate that the parallel processor approach is a more practical, cost effective solution than the modular simulator approach. A major benefit is that spare processors can be available on-line to replace failed processors automatically with no interruption of training.

INTRODUCTION

The cost of providing computing resources for today's training devices has been rapidly increasing even though the cost/performance ratio of computers has been steadily declining. There are many factors involved. Perhaps the most important is the nearly universal perception on the part of the user community that greater fidelity provides (at least the possibility of) better training. This perception has served to increase both the acquisition and life cycle costs of training devices of all types, offsetting to some degree the operational savings achieved through better training.

The modular simulator architecture has been proposed as a way to reduce the total cost of trainer computational systems. The essence of this architecture is to substitute several (and perhaps many) self-contained low cost microprocessors for traditional super-minicomputer(s). While this modular design appears to fulfill the desire for lower cost, a parallel microprocessor architecture offers most of the same advantages with fewer risks and with potentially greater cost savings.

THE MODULAR SIMULATOR ARCHITECTURE

Basic Features

With the modular simulator approach, one discrete processor is assigned to each trainer function or node. For example, there may be one flight processor, one aircraft systems processor, an instructor station processor, and so forth (1) (See Figure 1.). Each processor is physically located near the trainer hardware that it controls and is sized to perform its assigned function with reasonable spare capacity. All of these processors are connected by an extended bus or local area network. Ethernet* has been proposed for this purpose, as have other proprietary bus structures.

All data are transmitted via the interconnect bus and stored in local memory. There is no shared memory in this architecture, nor is there a general purpose operating system in each of the node processors. Stand-alone programs are the norm.

Each processor has its own I/O subsystem designed to drive only those signals which it computes. This approach is taken in order to provide sufficient I/O throughput without overloading the network.

A separate processor is usually supplied to download programs and data to the working processors, and to control execution of the system at large. This processor is often used for software development work when the trainer is not in operation.

With this architecture it is anticipated that the node processors can be obtained from different vendors, each one complete with software, I/O, and associated trainer hardware. The processors can then be integrated by the end user or a prime contractor.

Areas of Difficulty

Connecting multiple processors via an extended-length bus or network is difficult due to the need for high communication bandwidth between processors. A significant design effort may be encountered just in determining the

*Ethernet is a trademark of Xerox Corporation.

true bandwidth requirement for the system. Most of the current literature describes the requirement in terms of raw data transfer rate. In reality it is necessary to add to this figure all relevant overhead factors including controller handler execution time, message format time, channel bid time, and message decode and storage time (2).

Since all data is transferred via bus instead of shared memory, there can be significant problems with transport delay and processor synchronization. To deal with these problems, some proposed designs involve transmission of computed results to all other processors as soon as they are computed. With other designs, data are transmitted by all processors only at a fixed time within each frame. Many designs are predicated on fixed size transmission blocks. But if variable size transmission blocks are used, randomness is introduced into the data transfer times. This means that fresh data may not be available for all calculations in a frame if a "transmit when ready" scheme is used, or excessive delays may be incurred if a "transmit at the beginning of each frame" scheme is used. Common examples of variable size transmission blocks include parameter monitoring data displayed at the instructor station and performance measurement data collected during acceptance tests. The use of variable block sizes tends to increase the bandwidth requirement for the interconnection network.

Since each modular processor is sized to perform its dedicated function, there may be significant processing capacity inefficiencies due to the allocation of functions. For example, suppose that a functional analysis resulted in the determination that a given trainer consists of three functions: F1, F2 and F3. Each function executes at a different rate but consumes the following time (with spare) per second: F1 = 960 msec., F2 = 420 msec., and F3 = 540 msec. Clearly, two processors are capable of the task (see Figure 2A). However, the functional allocation has determined that three processors are required (see Figure 2B), so an unneeded processor is included in the trainer. Unfortunately, this unneeded processor cannot serve as a spare.

There can also be a significant duplication of hardware with the modular simulator approach. Most obvious is the requirement for multiple I/O systems with

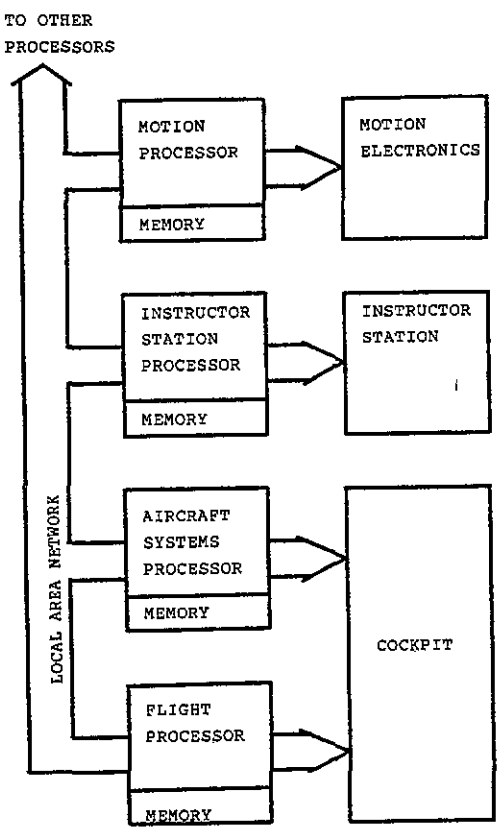

FIGURE 1. BASIC MODULAR SIMULATOR ARCHITECTURE

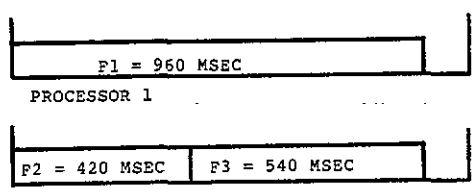

FIGURE 2A. OPTIMAL ASSIGNMENT OF FUNCTIONS TO PROCESSORS

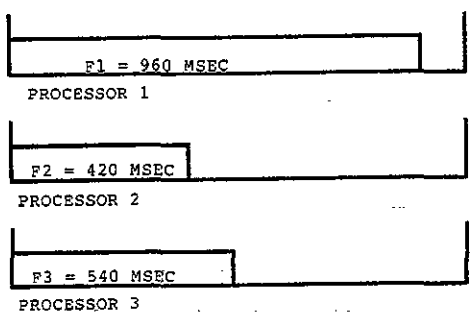

FIGURE 2B. MODULAR ASSIGNMENT OF FUNCTIONS TO PROCESSORS

their associated controllers, power supplies, and cabinetry. Less obvious is the proliferation of power supplies and cabinetry needed for the processors themselves.

Software development issues become significant when a separate host processor is supplied to do development and configuration management work. There will usually be a different operating system used for development (perhaps UNIX**) than for real-time operation (perhaps a custom executive) (3). The integration staff and maintenance personnel will need to be familiar with both systems. Effective on-line testing may not be possible on the target system since there may be no system services available. In addition, it may be difficult to debug on the development machine since it has no access to the I/O equipment of the various real-time processors or to their internal memory.

Logistic support factors to be considered include spare parts provisioning, training, documentation cost, and finally, special test and support equipment. Logistic support can be very expensive since there will likely be a number of different processor, I/O, and power supply types included in a trainer developed under the modular simulator approach.

A PARALLEL PROCESSOR ALTERNATIVE

Proposed Architecture

Having become aware of some of the potential difficulties with the modular simulator architecture, it occurred to us that there is a somewhat different way to structure a trainer computational system to reduce its cost and complexity even further: use a parallel computer to provide the total computing resource.

A parallel computer consists of numerous identical processors connected to a large shared memory via a high speed parallel bus (see Figure 3.). Each processor operates independently, working from a central dispatch queue to select a task to perform, until all tasks have been executed. Local processor cache memory is used to reduce the required bus bandwidth. A separate I/O bus is accessible to all processors. A single set of I/O hardware is used to communicate with all parts of the trainer. All processor hardware is controlled by an operating system at all times, regardless of whether software development or real-time training (or both) is in progress.

Relative Merits

The parallel processor architecture preserves most of the benefits of the modular simulator concept and provides the following additional benefits.

Since the interconnecting bus is physically very short, it can be a parallel data bus operating at very high speed, providing excellent bandwidth. The overhead for this type of bus is much lower than for a communication network, providing more usable bandwidth.

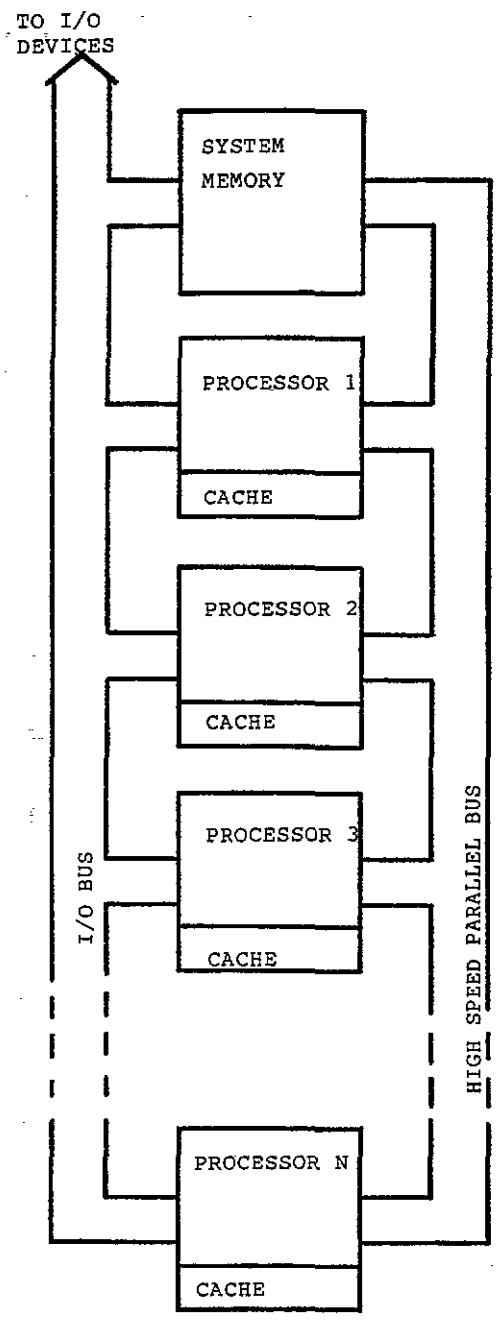

FIGURE 3. PARALLEL COMPUTER ARCHITECTURE

**UNIX is a trademark of AT&T Bell Laboratories.

Each processor is able to perform every task since each has access through shared memory to all instructions and data and through the I/O bus to all trainer hardware. The system can be sized so that very little processing time is wasted. Taking our previous example of the three functions (F1 = 960 msec., F2 = 420 msec., and F3 = 540 msec.), it is possible to use just two processors instead of three while retaining the same functional software allocation.

Since all data is available in shared memory, the problems of transport delay and processor synchronization become much less significant, although they never disappear entirely with either of these architectures since simulation software is by nature highly interactive.

The parallel approach eliminates the proliferation of power supplies, chasses, and other hardware that occurs with the modular simulator architecture since there is a single I/O system and a single computer system.

Software development and debug activities are greatly simplified. Software can be developed on the same machine used to run the trainer. A separate development system is not required. With an appropriate operating system, development can be done in background while the trainer is running. Debug is facilitated since all processors have access to all data and I/O hardware, and a spare processor can be used to debug the full software load in real time.

Logistic support is much simpler and cheaper since there is one type of processor and I/O system, and fewer components overall. The need for different types of test and support equipment is reduced. Maintenance documentation complexity and training time requirements for maintenance personnel are also reduced. In addition, a parallel computer can include extra processors to serve as on-line spares. Whenever a processor fails, the operating system simply takes it off-line and its workload is automatically assumed by a spare processor. At the next convenient maintenance period, the failed processor can be replaced or repaired. There are some potential failures that would require immediate attention, such as memory or I/O failures, but the most complex electronic element in the system, the CPU, can be spared on-line.

Expansion of processing capacity to incorporate trainer modifications after delivery can be accomplished more easily with the parallel processing approach than with any other architecture: simply add one or more additional processors to achieve the desired performance. No major hardware or software changes are required. The ability to easily and inexpensively add processors at any time virtually eliminates the risk that a trainer under development will fall short of its spare processing time requirements. Considering the ease of expansion of the parallel processor approach, less spare capacity needs to be delivered initially, reducing costs even further.

The one potential disadvantage of the parallel processor approach is that it is not easy to procure various parts of the trainer, such as instructor station, flight module, avionics module, etc., from various independent manufacturers as complete, self-contained subsystems. This has been stated as one goal of the modular simulator architecture. However, it seems doubtful that this goal can ever achieve real acquisition cost savings because of the difficulty of properly testing and integrating the many intimately connected elements required to produce a sophisticated trainer (4). Further, the total life cycle cost of a parallel processor trainer will surely be much lower than that of a similar modular trainer obtained from a collection of vendors.

A PROOF-OF-CONCEPT EXPERIMENT

Description of the Experiment

Given the above observations of the modular simulator architecture and a parallel processing alternative, an experiment was planned to use a microprocessor-based commercially available parallel computer as the computational resource in an actual trainer. The experiment was carried out within the past year as an internal research and development project. The plan for the project was to convert an existing minicomputer-based trainer to run on a parallel microprocessor system.

The TH-57 helicopter OFT was selected as the test case. There were several reasons for this choice:

- The trainer is relatively simple in terms of overall complexity, yet all important systems are represented.

- The trainer is designed using the most recent modeling concepts, and coded almost exclusively in Fortran-77.

- The TH-57 development schedule fit well with the experiment plans, with Unit #3 available for use on an off-shift, non-interference basis.

This trainer was originally developed using a Gould Concept 32/6780 computer. The computer was interfaced using a Computer Products RTP I/O system. Trainer subsystems include motion, digital control loading, digital aural cue, and an instructor station with a single CRT display. The Gould computer contains two processors, a CPU and an IPU, for a total

processing capacity of about 3 MIPS. The basic frame rate for the trainer is 30 Hz.

A survey of available micro-based parallel computer systems led us to Sequent Computer Systems of Portland, Oregon.

The Sequent Balance 8000 architecture is shown in Figure 4. The Balance 8000 consists of up to twelve processors (other Sequent models offer up to 30 processors) connected to a large shared memory via a high-speed parallel bus. There are very few circuit cards in the system. Each CPU card contains two processors with a floating point co-processor and 8 KB cache memory for each processor. There is one bus controller card and a Multibus*** interface card. Interface to the external world for I/O, peripheral access and other needs is achieved through a separate Multibus chassis housed in the same cabinet. The system memory supports a logical address space of up to 28 MB.

The Balance 8000 computer was the logical choice to use in this experiment, even though the system had not been designed specifically for this sort of real-time application. Both hardware and software were on the shelf and available. The configuration used in the experiment consisted of 10 processors (5 cards), 8 MB memory (2 cards), a diagnostic, Ethernet, SCSI interface (1 card), and a Multibus interface (1 card) in the main chassis. The Multibus chassis contained one DR-11 card used to interface with the flight controls subsystem, one RTP interface for the main trainer I/O, one disc interface for the 400 MB winchester disc, one 9-track magnetic tape controller, and a two-card RS-232 interface for terminal support. A 1/4-inch cartridge tape and 40 MB winchester disc were connected to the SCSI interface.

For simplicity, we decided not to interface the Sequent computer to the digital aural cue system since the aural cue system uses a Gould parallel interface (HSD) for which no direct replacement was immediately available. The aural cue system software was, however, included in the test load. For safety, we decided not to activate the motion system, but we included the motion software in the load. Finally, lacking a replacement for the HSD interface, the instructor CRT was activated via a serial interface. All instructor software was included, and trainer control was exercised via normal instructor station commands.

The software environment provided with the Balance 8000 is a parallel processing derivative of Berkeley UNIX 4.2bsd, named DYNIX, as supplied by Sequent. DYNIX has several key features:

- Automatic distribution of processing tasks to hardware processors.

- Support of shared memory partitions between tasks.

- The ability to assign a task to a specific processor.

UNIX is usually considered undesirable for a real-time environment due to its time-slice nature and large time overhead. In order to make a real-time application feasible, a way had to be found to eliminate the time overhead while still retaining the convenient UNIX features for software development and debug work, but this was judged to be possible.

One of the most important considerations in using any computer system is ease of software development. Good development tools provide schedule benefits. DYNIX offers such tools. There are full screen editing capabilities, a sophisticated file system, facilities to compare files and control access at many levels. Use of a symbolic debugger such as Sequent's dbx offers greatly enhanced capability which results in more rapid development. For example, the manual method of determining the location of a run time error in the source consists of finding the abort address, subtracting the program bias address, finding the address in the program link map, and looking at the assembly language printout of the high level language program. With dbx, the source line is immediately printed at the terminal. This capability made the Balance 8000 an especially good choice for this experiment.

Design Considerations

Our initial design concept was to create an executive system which was subordinate to DYNIX, and yet was able to use the full power of each hardware processor without interruption. This idea grew into a system design with the following features:

- Control of simulation software execution centered around a trainer subroutine dispatch queue. This queue would reside in shared memory and be accessible by all tasks. It would contain the usual dispatch queue data (e.g start address, iteration rate, next execution frame number, etc.).

- Use of the DYNIX fork service to make as many identical copies, referred to as child processes, of a simulation task as possible. For an N-processor system there would be N-2 processes, leaving one processor for I/O operations and a second processor for DYNIX. In normal operation the parent and each child process (referred to from here on as a logical process) would scan the subroutine queue from the top and select for execution the first available subroutine that was

***Multibus is a trademark of Intel Corporation.

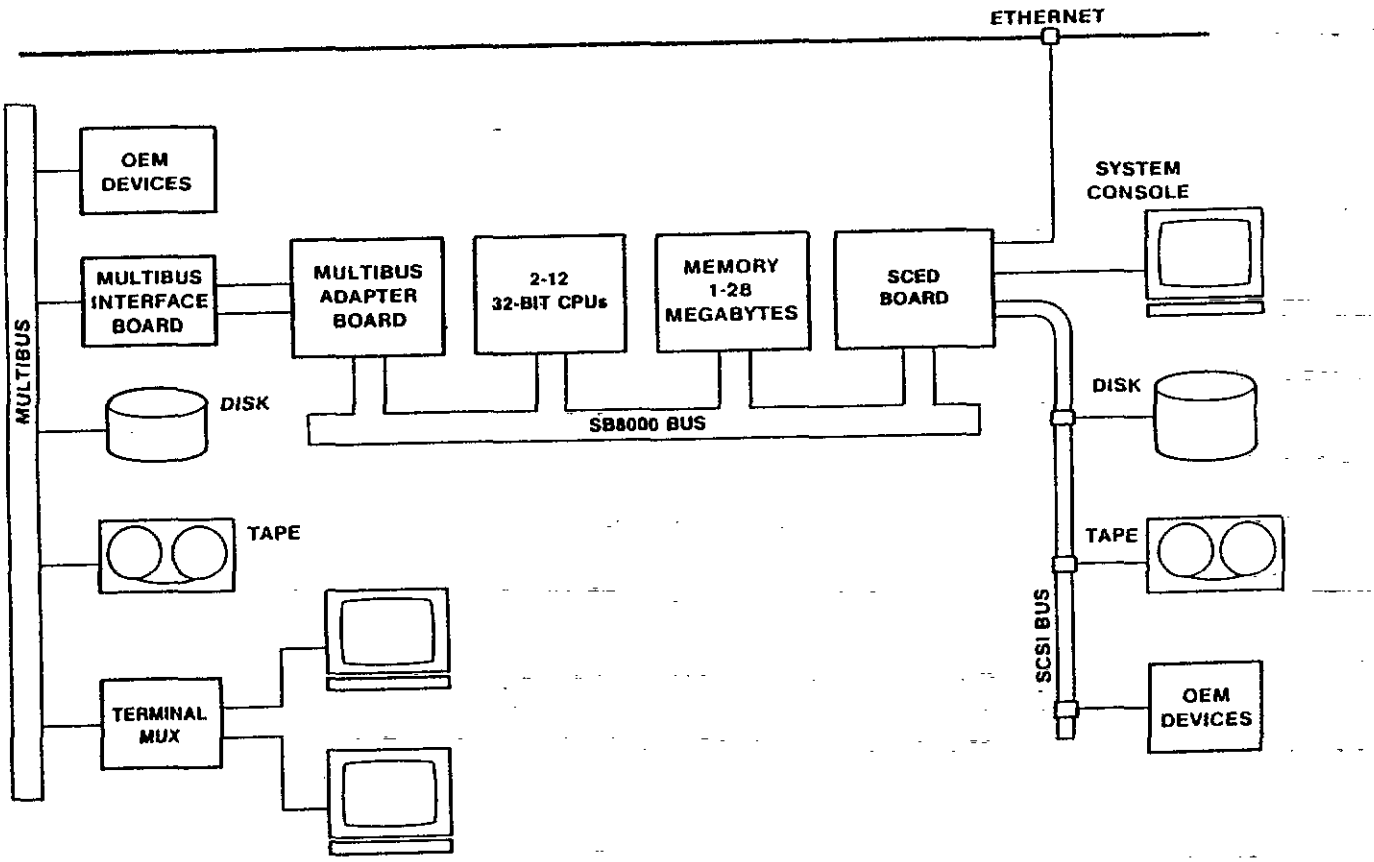

FIGURE 4. BALANCE 8000 ARCHITECTURE

ready to run. After the subroutine completed, the process would again scan the queue and select another subroutine for execution. Queue scanning and subroutine execution would be continued until all subroutines scheduled for execution in the current frame were done. Each logical process would then idle until the beginning of the next frame. Suitable memory locks would be provided to ensure dispatch queue integrity.

- Use of a shared memory partition containing those data which would be passed between modules.

- A unique executive task which would keep track of time and signal the beginning of each frame to the logical processes. This executive would also detect processing overruns and other error conditions.

The major design challenge of any parallel computing environment is program partitioning. In most cases functional subroutines execute in a short time compared to the iteration rate, and are sufficiently independent to avoid sequencing problems. There are some cases though, where the execution order of subroutines is of extreme importance. This is due to the nature of the integration (and sometimes prediction) algorithms. For example, a particular formulation of aerodynamic model calculations requires the presence of both acceleration and velocity terms from the current frame in a single equation. Use of previous frame data does not give adequate performance. A way must be provided to synchronize subroutine execution so that both acceleration and velocity from the current frame are available when needed. In the experimental implementation this was achieved by the use of a sequential execution flag in the subroutine dispatch

queue which prevented any processor from executing a sequential subroutine until its predecessor had been completed.

The degree of difficulty experienced in partitioning the software in a parallel computer depends upon the speed of the chosen processor. A slow processor will require that the problem be broken down into many small pieces. In some cases there may not be sufficient time available to do the job with a reasonable number of processors. One difficulty with the specific Sequent computer used was that it had poor floating point performance. More powerful processors now becoming available should eliminate any problem.

I/O operations presented a special design problem since no-wait I/O is not supported in DYNIX. To overcome this limitation a design consisting of an I/O queue and a number of I/O processes was developed. In this design the parent I/O process is forked until the number of logical processes equals the number of supported I/O channels. All the I/O processes are assigned to a single hardware processor. Each I/O process scans the I/O queue, begins the requested I/O operation, and then is suspended by DYNIX. While this process is suspended pending I/O completion, the other I/O processes are free to scan the queue and begin the next requested I/O activity.

The design of system control for this experiment was simple and direct since DYNIX provides all software needed to load and control program execution for every processor in the system. Hooks to allow processes to float between processors or to be tied to a specific processor did not need to be developed; they are standard features. Processor synchronization was achieved through a simple software flag structure rather than through hardware interrupts or through a communication network. The overall result was reduced development effort and risk.

Problems Encountered

As this experiment progressed, several facts emerged that caused substantial design changes. The first and probably most important was that we had to abandon the strict subroutine dispatch queue idea. The reason was that private or local data is used extensively in the current version of the TH-57 software to store intermediate results from frame to frame. Since each child process created by the DYNIX fork service has its own local data, subsequent iterations of the same subroutine code in different logical processes might produce different and discontinuous results. The correct solution to this problem would be to re-code all the TH-57 software to eliminate the local data. Unfortunately, we judged that re-code was too large a task for this experiment, and we decided to fall back to statically partitioned tasks for all simulation code.

In this fall-back static scheme, each simulation subroutine was permanently assigned to a specific process. Each process was compiled and linked separately, and loaded into system memory as a complete and independent program rather than as a parent/child. However, since the I/O code design was new anyway, we retained the concept of having multiple, forked I/O tasks which utilize I/O queue scanning.

A second order problem was created by static partitioning. In a system where child processes are created by the fork service, all executable images are identical, and all data reside at the same logical address. When the switch was made to static partitioning, the data addresses became different due to the differing task sizes. For normal program execution this causes no problem, since all addresses are consistent within a task, and all addresses point to the correct physical memory locations. It rapidly became apparent that there was a significant problem for those tasks which exchange pointer data. This problem was resolved by adding directives to the DYNIX loader to relocate the shared memory block to a statically defined address for all tasks. To avoid overlap between the task code space and shared data space the shared data was assigned to an address range above the largest task address.

Another problem we encountered was in the area of program debugging. In a sequential system accesses to a particular variable are controlled in such a way that the programmer can determine which subroutine was last to update a value. In a parallel system there is generally no way to single step all processors in unison. Any subroutine can change any shared memory variable at any time without restriction. Error detection becomes a process of elimination, where subroutines are individually removed until the offending one is found. While excellent source level debug aids are available, debug control over multiple processors is not yet a reality. This problem, however, would also be encountered with the modular simulator approach.

The final problems we encountered were with byte ordering and common block allocation. The byte ordering sequence in the Balance 8000 is the reverse of that in the Gould computers. Programs which received integer word input and then used it as an array of bytes had to be re-designed. Special attention was also paid to converting the Gould DATAPOOL common block to a standard ANSI Fortran common block. While neither of these had anything to do with the design approach we were testing, a fair amount of time had to be spent converting code.

RESULTS

The experiment has yielded very encouraging results. Using a parallel computer with the operating system assigned to one processor, the operating system's overhead had no negative impact on the real-time trainer software and allowed us to use its convenient development and debug tools while the trainer was on-line. Software development activities off-line were very efficient using DYNIX.

The trainer software executed properly as anticipated. No problems with throughput delay or bus bandwidth were observed. This was undoubtedly due in part to the fact that each processor has local cache memory.

The I/O design worked well, verifying the approach of using parent and child processes working from a central queue and allowing us to successfully bypass the problem that a UNIX-style operating system does not support no-wait I/O. Real-time debug was greatly facilitated because of the use of shared memory for all exchanged data. This allowed us to use a background utility to monitor and set memory data in real time in order to isolate problems. However, there is no avoiding the fact that debug in any type of multiprocessor environment in real time is challenging.

The net result of the experiment is that we were able to accomplish the goal of using a parallel processor computer to control a helicopter operational flight trainer using a computer costing about two-thirds as much as an equivalent minicomputer and having much lower support costs. The amount of effort required for the experiment convinced us that the parallel processor software design approach is not any more complex than the traditional approach.

RECOMMENDATIONS

The parallel processor architecture represents a way to reduce significantly the life cycle cost of training devices and simultaneously improve their reliability and expandability. However, there are several developments we would like to see occur in order to reduce engineering cost and risk:

- Parallel computers need to be based on processors having fast floating point capability in order to serve as the basis for complex training devices. Such improvements are currently underway.

- Development of an operating system having the tools of UNIX but with better real-time performance would be a big plus. However, even with current performance, a UNIX implementation supporting parallel processors, such as DYNIX, is suitable for use at the cost of one processor. We feel this cost is worth the benefits obtained.

- Development of debug tools that allow better control of multiple processors would be a definite asset, although clearly a difficult undertaking.

The concept of forking a process and using a central dispatch queue as a means of running a parallel computer has been shown to work effectively. However, this concept requires that shared memory be used to store all computational results that will be retained from one iteration to the next. Local storage can only be used for loop counters and such. This needs to be kept in mind when designing software.

The ability of a parallel computer system to replace a failed processor on-line is perhaps its most significant benefit. However, not all faults are sufficiently catastrophic to cause the operating system to recognize the faulting processor and take it off-line.

The parallel architecture suggests a very useful design approach to minimize the effect of such minor faults. A fault tolerance scheme could be implemented whereby subroutine execution is rotated between processors, including the spares. For hardware failures that are not catastrophic, the affected system changes from frame to frame. The natural inertia of the simulated system will tend to diminish the importance of the error for any specific subroutine within any one frame (5). Further, by including a diagnostic subroutine in the real-time software, the faulted processor could be identified to maintenance personnel and to the operating system. Upon recognition of such a processor as faulted, the operating system could take it off-line automatically.

Even though the design problem of dividing the heavily sequential process of simulation into parallel units seems destined to remain with us, we recommend pursuing the parallel approach and encourage computer vendors to develop machines with even higher degrees of parallelism as a means of further reducing cost and development complexity for training devices.

References

1. Steve Seidensticker, "Modular Simulators: A Comprehensive Unified Architecture", Proceedings of the IEEE National Aerospace and Electronics Conference, 1985, pp. 1074-1079.

2. Donald L. Johnston, "Modular Simulators: Is A Local Area Network Sufficient?", Proceedings of the IEEE National Aerospace and Electronics Conference, 1985, pp. 1086-1093.

3. Edward M. Holler, "Modular Microcomputers for Trainers", Proceedings of the Sixth I/ITEC Conference, Volume 1, October 1984, pp. 353-357.

4. Mary-Ellen Hecker, Ph.D., "Modular Simulators: How To Make It Work", Proceedings of the IEEE National Aerospace and Electronics Conference, 1985, pp. 1080-1085.

5. Stanley J. Larimer and Scott L. Maher, "A Continuously Reconfiguring Multi-Microprocessor Flight Control System", Final Report, 1 Aug 1979 - 30 April 1981, Air Force Wright Aeronautical Laboratories (AFSC), Wright-Patterson AFB, Ohio. AD-A101412, AFWAL-TR-81-3070.

ABOUT THE AUTHORS

MR. EDWARD KULAKOWSKI is the Manager of Research and Development at Reflectone, Inc. In this position he is responsible for all aspects of internal R&D activity from concept development through technical and administrative management. He has led both the Computer Systems and Instructional Systems Groups at Reflectone. Previously Mr. Kulakowski worked as a software consultant and, while at General Electric, on training devices for sub-surface Navy tactical equipment. Mr. Kulakowski holds a Bachelor of Engineering (Electrical) degree from Pratt Institute, Brooklyn, N.Y.

MR. DAVID J. KRAMER is a Project Engineer with Reflectone, Inc. As Project Engineer he is responsible for the design, development, and integration of all hardware and software for state-of-the-art flight simulation training devices for military and commercial aircraft. He has served as Project Engineer on an EA-6B Operational Flight and Navigation Trainer, a C-5A/C-141B Air Refueling Part Task Trainer, and on B-747 and A-310 Phase III Operational Flight Trainers. He was formerly Leader of the Computer Systems Group at Reflectone, having joined the company in 1977. Prior to joining Reflectone, he directed the development of real-time computer-based systems for ComGeneral Corporation in Dayton, Ohio. He holds a Bachelor of Science Degree (Summa Cum Laude) in Computer and Information Science from The Ohio State University. He was a NATO Fellow in 1975, investigating large-scale simulation models.