MODULAR, FUNCTIONALLY-DISTRIBUTED, MICROPROCESSOR-BASED SIMULATION: ONCE A CONCEPT -- NOW A FACT

By Michael B. Ash
Link Flight Simulation Division of The Singer Co.
Binghamton, New York

Summary

VLSI and ULSI technologies can support parallel, information-processing methods that only a few years back were models that generated interest among a handful of theoretical mathematicians, physicists, neurophysiologists, cyberneticists, and science-fiction writers.

Increasing performance simply by adding function-producing parts is so fundamental in nature and in human endeavors that it inevitably had to be addressed in computing applications. When specialized as well as replicative functions are performed in parallel, the result is a distributed process. Applying this principle to computing, where system parts are called modules, facilitates solving very complex problems using a divide-and-conquer method. An apparently monolithic entity like a flight simulator, for instance, can be partitioned into small, discrete, manageable modules. All distributed systems are limited by the size of the entire system and also by that of their individual components. Functional complexity is limited by the system's ability to communicate timely information between the various parts. Because of the disparity in the number of specialized modules, the primary difference between functionally distributed General Aviation Trainers and large-scale WST's is in their information-flow requirements. The factors that constrain the system are computer buses, bus networks, and I/O paths.

This paper, which examines the conceptual and philosophical approach that led to the first generation of modular, functionally-distributed, microprocessor-based simulators, deals primarily with technical, computational, and motivational perspectives and with general systems solutions that have been formulated over the last eight years. It gives dimension to, and lends insight into, the reasons for taking the critical, evolutionary leap that is necessary for economic and technological progress.

Introduction

State-of-the-art vehicle simulation technology does not take full advantage of state-of-the-art VLSI computer technology. The two industries are nonaligned with respect to technology, and the situation does not seem to be improving. Computer-product market life-cycles continue to shrink; with the most modern development techniques, the gestation period for a product exceeds its peak market activity. When a product's functionality depends largely upon its embedded computational capability, this discrepancy can be very damaging. This is true in simulation, where the actual product is many times more complex than the computer and where a much larger and longer commitment must be undertaken when a new baseline product is introduced.

This paper, which is intended to mitigate the skepticism and perceived risk that is always, and sometimes justifiably, associated with technological advancement, addresses the impact of the computer revolution on trainer design. It discusses some of the most important technical issues that apply to the entire range of Artificial Reality (AR) and provides an informational backdrop against which the technical direction of the past eight years can be seen in perspective.

We will view the problems of the simulator industry from a computer systems perspective, as a special case in a group of phenomena called Artificial Reality. Essential dimensional translation (mimicking) of a phenomenon like a jet fighter by a distinctly different one like a simulator, in such a way that the sensory inputs of a third phenomenon like me, you, or another machine cannot distinguish between them, is the characteristic challenge of all AR problems. More and more actual simulator functions are being controlled by digital computers, and the optimum approach to solving specific problems and integrating the entire product has emerged from Computer and Systems Science theory and practice.

Computational Perspective

Are Microprocessors Tiny Minicomputers?

Computer components are getting smaller, but, with a few exceptions there may be little other-than-size difference between the VLSI microcomputers and the minicomputers now used on most simulators. Although the most powerful microcomputers occupy less than a thousandth of the space of their minicomputer predecessors, they use the same basic programming models.

A visual comparison of micro- and minicomputers is startling, and the economic advantage is easy to see. There are no minicomputer precursors for microprocessors like the 16-bit Intel 80286's, 32-bit Motorola 68020's, or Zilog Z-80000's, but we can compare the DEC PDP 11-70 and 32-bit VAX CPU with their silicon equivalents, the J-11 and the *MicroVax II*, to show the staggering reduction in scale.

Although scale has been reduced in crucial ways (size, price, external complexity, and power consumption), function is either identical or improved:

- Programming is easier and software more efficient because of expanded address space, coprocessor interfaces, orthogonal instruction sets, more or equivalent numbers of general-purpose registers, and memory-mapped I/O. The newest 32-bit microprocessors optimize high-level language compilation.

  Even if a microprocessor is not so fast as a superminicomputer or mainframe, it may provide greater function because it provides the user with exclusivity at much lower cost. If the user is a simulation software module, this is true if these users can do their specific functions fast enough and if they can communicate fast enough to provide:

- Necessary information for similar coordinated units

- Process sequence-control information

A minicomputer on a chip has the advantages of:

- Size -- Hundreds of identical CPU's can replace unique analog and digital hardware.
- Price -- A CPU costs a few hundred dollars, and an entire module a few thousand dollars. Many applications never considered before for digitization are now cost-effective.
- Expandability -- Any part of a system can be expanded at very small cost. This greatly reduces risk in original procurements and configuration alterations during service. Less spare computational capability and memory need be designed for the original configuration.
- Reduced System Complexity -- VLSI technology makes it possible to reduce physical scale. Many more functions are digital, and they can be controlled with a small group of single-board modules (e.g., CPU, Memory, Intelligent I/O, and Communications). A much larger portion of the entire system comes under a single extended system.
- Reliability -- Fewer components and lower energy consumption mean longer uninterrupted operation.
- Maintenance and Repair -- Since intelligence and software capability can be provided cost-effectively throughout the system, most diagnostics can be performed automatically. Fault isolation is no longer a time-consuming, schedule-crippling inevitability, and it no longer demands the undivided attention of a full-time subsystem expert. Most hardware failures can be isolated and identified as they happen, and a technician can replace the failed unit in a few minutes.
- Automated Self-Correction -- Since the system is homogeneous, it lends itself to redundancy and rehosting of functionality (i.e., fault tolerance) -- that would be impractical with large computers because of cost.
- Inexpensive Parallelism, More Power -- Although a system that appears concatenable may have endless aggregate computing power, the limits of useful power in a time-dependent system like a simulator depend upon the maximum permissible duration of any temporally sequential group of events. Optimal use of replicated processors requires:

  -- Analysis of problems as parallel, dimensionally manipulatable phenomena

  In time, serial systems will no longer be competitive. Analyzing problems in this way is a matter of practice. We already have suitable distributed executives for network control and creation. Large-grained parallelism is already being used in most simulators. The actual performance of the newest 32-bit microprocessor (MC68020) is close enough to that of the SEL 32/77 we have been using that porting the software requires little alteration of basic flight math models and virtually no change in other areas, including the executive. We can use the hardware system before we have the software to optimize the use of its architecture. This is having a favorable effect on:

- Programming (simpler) -- With many inexpensive CPU's, high-level computer languages are used throughout the system. This reduces software engineering and maintenance costs and facilitates software transfer between systems that have different processors. If a processor upgrade becomes necessary, this can have a large positive impact.
- User Environment -- Networked microprocessor-workstations generally offer a rich array of software and more sophisticated human interface than time-sharing systems. Using portable operating systems like UNIX permits choosing permanent engineering workstations independent of the microprocessors used on the simulator and also permits using the microprocessors on the simulator in the same way.

Buses -- Information Highways That Hold the Key to an Architecture's Success

A virtually unlimited supply of tiny little minicomputers at bargain rates, which perform every imaginable function with symphonic harmony at lightning speed, has numerous advantages; but an effective computing system is not quite that simple. Buses are necessary to transfer information from one part of a system to another.

In Von Neumann computers, programs and data are stored in memory, using a program counter with an address. At some time in the execution cycle, the address is used to read the next instruction from memory. The processor puts the address on the bus, and the memory, recognizing that this address is contained within its particular preconfigured address range, places the appropriate data on the bus.

Most buses can support only one transaction at one time. (One exception is the Broadband bus, which simultaneously carries signals of different frequencies between various locations. This method of multiplexing information has been used successfully in local area networks, but it has not yet shown any potential for the much higher data rates computers need to execute programs.) Most computers have a main bus, through which all the processors communicate with each other and with memory; only one processor can use the bus at a time.

If a CPU has to share such a single-access bus with other CPU's, it spends much of its time waiting to get its instructions and data (which are stored in memory and are accessible only though the bus). If all the CPU's rely on the bus alone for all instructions and data, adding a CPU does not increase computational power. None of the many existing ways to circumvent bus contention achieves the linear vs. exponential degradation needed to use the large number of computers (microcomputer or other) needed in simulation. For large simulation applications, the method most commonly used is to have a section of memory, with multiple ports to different buses, act as a communications buffer between the CPU's that reside on the various buses. The problems of this type of solution include the following:

- Cost
- The shared memory has a limited number of ports (typically 4 to 16).
- Each bus that feeds into memory can still support only a few processors efficiently.

- Even when the system uses techniques for simultaneous access of different address locations within the memory, there is contention for shared memory.

Figure 1 -- Old Bus Layouts

Adding a Bus for Each Added CPU -- In most advanced microprocessor modules, local bus contention has been alleviated to some extent by providing each processor with its own local bus and also with access to a global, backplane bus, such as VMEbus or Multibus (I or II). The CPU has on-board memory, a local bus extension to one or more adjacent memory cards (e.g., VMX, VMX32, Rambus, LBX, LBX II), which can presently host up to 4 million bytes per card, or both.

Since this local memory almost always has two ports, one to the local and one to the global bus, other processors can access it without interfering with the owner CPU's local bus operation. The number of CPU's that can operate on any global bus has increased dramatically, mainly because another bus is added for each added CPU.

Global Backplane Bus Extension -- If there is no more room on the global backplane bus, or if we want to isolate a cluster of processors, yet still maintain direct addressability between them (i.e., have communication not through some I/O device but through a normal bus read or write), we can use a bus extender or repeater. This effectively extends the size of the computer and permits having more than one global bus. These global buses join dynamically during transactions between points that reside on both buses and operate independently at all other times. This technique is not limited to global buses of the same type; for example, VMEbus can be connected to Multibus.

To obtain better system organization and performance, the bus extension concept is used on a much larger scale. Many clusters of processors, each with a specific function, can be operated in total isolation or can be hooked up with any other cluster dynamically when necessary, as a function of the address that is placed on the global bus.

Although the architecture here is distributed, the effect is that of a single, extended multiprocessing machine. The logical ordering of the *entire system is very flexible; the place a program* is executing is less crucial than when various forms of I/O (e.g., Block Direct Memory Access, Local Area Networks) are used for interprocessor or intercluster communication. I/O methods always exact a large time overhead, and these overheads cause latencies that cannot be tolerated in high-speed applications. (The time penalties are not caused by the speed of the media that eventually pass the information but by the fact that these devices usually require programming and/or complex protocols to operate. Even when Block DMA's are automated, so that they move large groups of data synchronously between various units, to be shared, they become ponderous. Most of the information *being transferred is unchanged, and software in*volves synchronizing the transfers rather than internal or external (I/O) needs.) When a program in one location accesses a variable in the memory of another cluster 30 feet away, it is addressed directly by linking the buses of both locations. With this technique, the only delay is caused by the hardware and by signal propagation, and no extra software is needed.

Even without the inherent overheads, if the intercluster bus were used only for point-to-point reads and writes, linked systems would soon degrade with the amount of crosstalk incurred in large simulation applications. All the groups or their *subsets need common information.* To increase the efficiency of such an extended memory-mapped system, we have combined some concepts used in other computing areas:

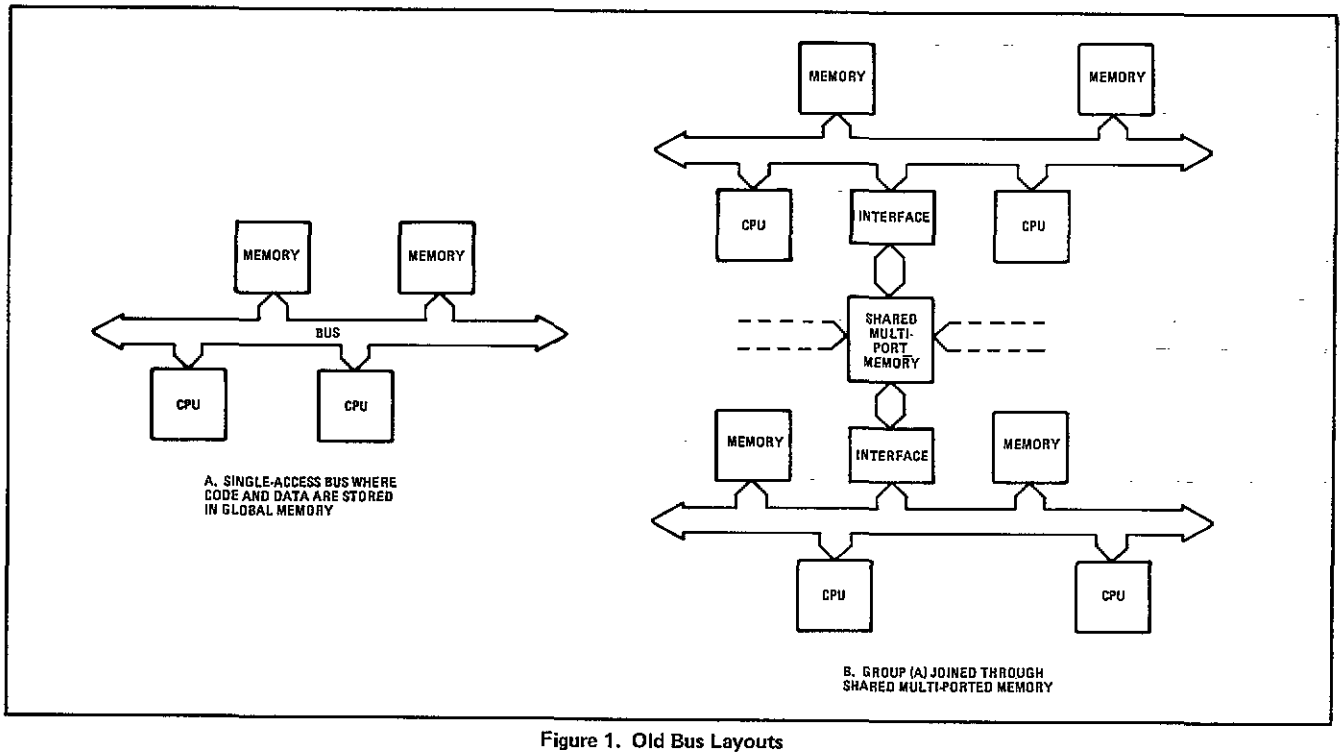

Figure 1. Old Bus Layouts

1) The private bus. We have mentioned this above as significant in isolating the activities of processors on the same global bus.

2) Write-through memory (used in high-speed cache memory systems). Data in the high-speed cache are copies of data in the slower main memory. When the high-speed cache data are changed, the new values are passed back to main memory so that it reflects the most up-to-date value, for access by another processor.

3) Broadcast, multi-target writes. A single write transaction can be received simultaneously by all or some subset (then termed multi-cast) of the devices that reside on the global interconnect bus.

Each bus extender module has its own private bus, which has local memory that resides either on the extender, on adjacent memory cards, or on both -- just like a microprocessor card. The local memory can be accessed either from the local, global backplane bus or from the external connecting bus through the extender's local bus. Depending upon the memory address for which it is targeted, a write will:

- Stay on the processor's local bus
- Access the global backplane bus
- Be routed directly to a target location on another local global bus
- Be written simultaneously to the extender's local memory and broadcast to the memories of all other preselected extenders

This technique makes the datapool dynamically changeable and replicative. Since it uses the local-bus concept, most transactions do not involve the global backplane buses of the targets. It reduces bus activity between clusters, because most of the writes are broadcast and because all reads from the replicated datapool are local. Intercluster activity is also reduced, since the only data that need involve the interconnect bus to ensure complete replication are those actually written to in the datapool.

In a simulator, almost all reads are local, except those that involve point-to-point diagnostic, fault-tolerant checks and occasional remote memory-mapped I/O. The writes are also mostly local or to datapool, except in the last circumstances mentioned for reads. An added capability is direct program loading anywhere within the entire system. This makes the loading procedure direct and also makes dynamically rehosting and reconfiguring software for fault-tolerance a less complex procedure.

The global backplane bus for a given extender can vary. (The back-end is always the same, and the global interconnect bus is always the same for a given generation of these boards, but the front end can be different from one bus to another.) The actual nature of the interconnect bus, however, is not important, just so there is a memory-mapped

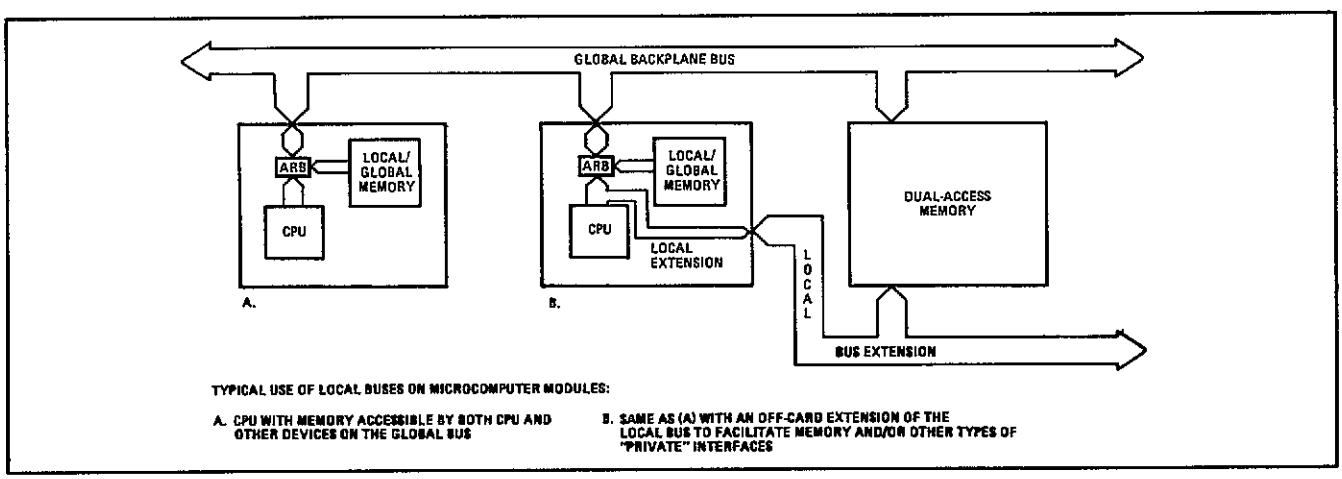

Figure 2. Microbus Layout

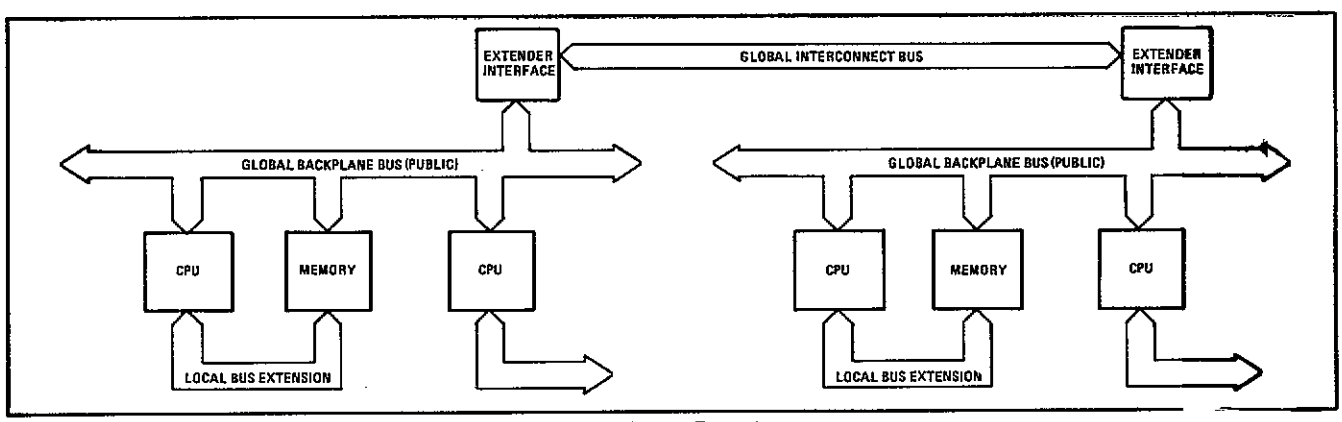

Figure 3. Bus Extenders

interface for all the industry-standard buses used throughout the entire extended system. For practical reasons, the bus is fast enough to achieve as close an approximation of global backplane bus speeds between points in the system as logic and propagation delays permit. (Expansion of physical address space to 4 gigabytes (32 bits) in the newest microprocessors and buses, e.g., MC68020, the forthcoming IAPX80386, VMEbus, and MULTIBUS II, is timed perfectly for implementing, such extended memory-mapped systems.)

I/O-bound methods of interfacing many processors limit the solutions of computational problems to serial and large-grained parallel techniques. Any large-grained solutions attempted under these conditions are still extremely dependent upon the serial-processing speed of the CPU's used. With extended memory-mapped systems, there are the same logic and physical-signal propagation delays, but host-to-controller overhead is eliminated. To alleviate contention, we can add additional intercluster buses selected specifically for their address content or by hardware detection of the first inactive pathway.

Public Buses

The minicomputers used now in many simulators are not very prolific, and they have their own proprietary buses. The number of vendor sources for hardware modules is severely limited. Usually the sources are the computer manufacturer and a few specialty houses.

Using public buses makes it possible for designers to choose processors that suit the needs of their applications and to make changes without disastrous impact. Most vendor micro-CPU's already run or are planned to run on public buses.

When custom modules are designed to perform special functions for any application, they are also being designed to operate on our industry-standard bus. This eliminates one of the major disadvantages in their use, since these modules may now be multi-sourced throughout industry.

Microcomputers Enable State-of-the-Art Technology

Because it has no common denominator with the computer industry's mainstream, the simulation industry has largely been bypassed by the massive proliferation of microcomputing devices and the resultant innovation revolution. Making innovations with the esoteric devices now used would mean massive implementation costs, because there is a noncompetitive situation among only a few vendors. Few third-party manufacturers will put themselves into the high-risk situation of designing a bus extender for the superminicomputers we use so extensively. Although one vendor might design or license someone to design such a device to gain a competitive edge over another, the price would be exorbitant. Even though these companies design some new devices at our request, so long as we base our simulators upon mini- or superminicomputers, we can never dip into the vast pool of technology from the hundreds of independent manufacturers who are specializing in

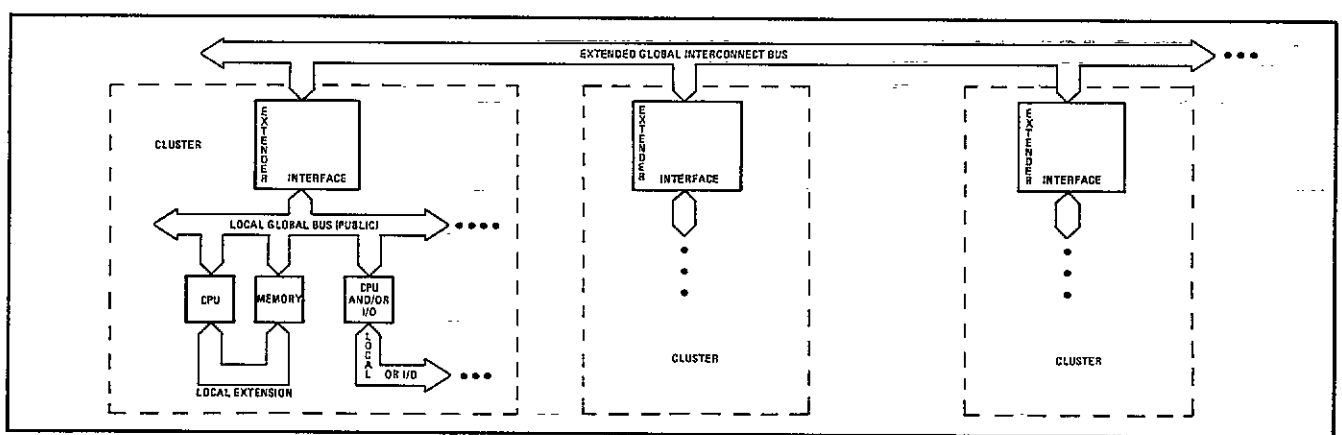

Figure 4. Extended Memory-Mapped System

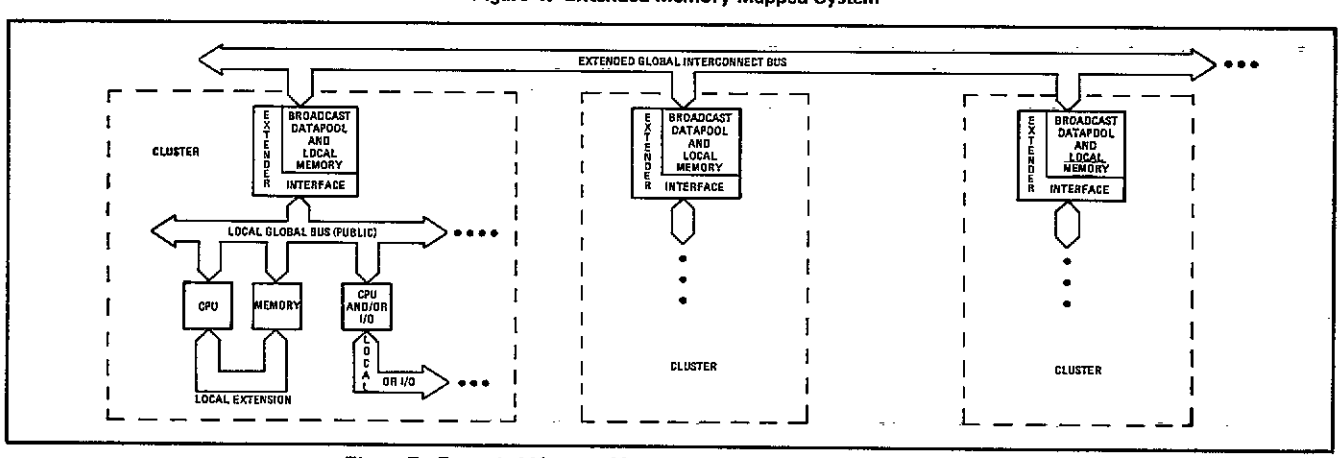

Figure 5. Extended Memory-Mapped System With Broadcast Datapool

making both mundane and exotic devices for a few standard buses such as VME and MULTIBUS.

However, with microcomputers we can make more frequent use of state-of-the-art technology. During our Microsimulation Technology (MST) research and development effort, we often discovered solutions to complex and costly problems in trade-magazine advertisements. In one case, we were going to configure an early prototype system with a certain company's graphics subsystem. To interface the public-bus system to the graphics system, it seemed necessary to design the interface ourselves or ask the subsystem manufacturer to produce a custom interface at a cost of $250,000. How fortunate it was that another employee discovered a usable $2,000 interface card while "wasting time" reading a trade magazine! This serendipity has become a frequent occurrence. With this technology, we no longer have to depend upon proprietary product lines, and we now take our "time wasting" more seriously.

Serial vs. Parallel Computing

The Speed of Light and Serial Processing -- Nature's Bottleneck

So far as we know, information can move no faster than the speed at which electromagnetic radiation propagates in a vacuum, i.e., about 1 foot every billionth of a second. For phenomena such as elections and people, which have mass, speed is restricted even further to a domain always some fraction of, but never equivalent to, that of their photonic coresidents of space and time (Special Theory of Relativity). Because information cannot travel instantaneously in space from point to point (i.e., much faster than light), parallelism is probably not even verifiable as a physical phenomena.

Devices currently in development can process information in trillionths of a second, far less than the time for current to move between chips. If, when optoelectronic logic is available, photons are substituted for electrons, and the distances between chips are replaced with submicron distances, there is still an I/O delay. Natural barriers imposed by known relativistic and quantum mechanical physics with respect to velocity and spatial proximity are now becoming appreciable. Devices and their information are light-speed limited. As these barriers are approached with teraHertz frequency devices, the distances between which are measured in tens of atoms, serial information processing will approach a natural barrier. When we reach this barrier, the only way to increase computation throughput substantially for a given period of time is by using many computing devices simultaneously. Even then, the entire system can exist as an identifiable phenomena only within the limits that information can propagate between various cells. This explains the preoccupation with buses.

Single, Sequential Instruction Stream Machines

No matter how instructions are accessed, they are usually executed in a linear, sometimes overlapping sequence. This has driven the evolution of our present computer language, and habitual use has led most of us to believe it makes software sense. This linear execution has limited the way we approach problems and generate algorithms. Our current structured programming practices are heavily influenced by a sequential approach to problem solution. This mindset is the result of a particular computing method, but with our present replicative microcomputational capabilities, there is no reason why we should not seek uniquely parallel solutions.

Multiple Sequential Instruction Streams, Very Large-Grained Parallelism (What Happens When Your CPU Runs Out of Time)

In the simulation industry, we used large-grained simulation first when sequential application could not execute fast enough. Since the original software was comprised of many parts that had little or nothing to do with each other, we could build two or more programs that ran simultaneously in synchronized processors and use some method of sharing mutual data in as timely a fashion as possible. Only when any single, serial sequence of code could not be executed by a CPU, or when it could not be divided into parallel programs that overlapped enough to meet the time constraints, did we look for a faster CPU.

With a serial frame of reference, where it is natural to optimize algorithms sequentially, we believed that a specific sequence of events must take place. For example, in flight simulation, we have solved multidimensional equations of motion sequentially, forcing the mathematical approach. In reality, the differential equations of motion and dynamics of a rigid body in space can be expressed in terms of three translational and three rotational equations, which can be solved in parallel. (In addition to this gross parallelism, each of these equations is actually a set of arithmetic manipulations and operations, which are factorable into simpler expressions that can be executed in parallel by many computers.)

From a software perspective, current superminicomputer programming for simulators, using multiple CPU's, is not true parallelism but a form of spillover serial processing, made to fit in a parallel environment through complex manual positioning of modules running in multiple CPU's.

True Large-Grained Parallelism and Functional Distribution

One of the reasons for the spillover serial design phenomenon was and still is the cost of minicomputer technology. In the past, it has been an objective to use as few of these monetary behemoths as possible, because of the large cost of hardware expansion and the software upheavals that ensue when a load that is balanced (i.e., sequenced properly) for nCPU's must then be rebalanced for n+1 CPU's.

VLSI technology economics and performance can virtually eliminate these difficulties. Applications engineers work on parallel solutions to application problems without considering the more demanding design techniques required by small-grained parallelism. When tradeoffs favor parallel solutions, they are designed that way from the start. This reflects a subtle, important change. The hardware configuration now reflects the original intent of the software design. From the beginning of the process, optimizing the software function is a major influence in designing the overall system hardware. The application can use many CPU's that process simultaneously and take full advantage of even the grossest, large-grained techniques. Spillover is a rarity instead of an occurrence so highly probable that we must guarantee 50 percent spare.

Large-grained systems risk spillover because large portions of algorithms are executed serially. They are more vulnerable to the finitude of a processor's instruction bandwidth. However, the impact of this deficiency is far less costly if the physical architecture is not overburdened significantly by data cross-talk when extra processors are added. Two large-grained multi-microprocessing architectures that have different methods of interfacing internal processors are the hypercube and extended-memory system under discussion. The interface methodology can have dramatic impact when high data communication rates are necessary:

- In hypercube architecture, the only way CPU's exchange data is through dedicated ETHERNET channels, processes that require moderate-to-high amounts of data exchange can easily become I/O-bound.
- In the extended memory model, there is only a slight increase in bus degradation.

Functional Distribution

In functional distribution, when hardware is allocated specifically for a subsystem, such as Flight, Navigation, Instruments, or Aural Cues, an entire cluster of computers is available from the very beginning of implementation, and there is greater freedom for software design optimization and system development and maintainability. Each module is a small-size computer system with intelligent I/O. It can host an entire subsystem throughout the product's life-cycle and permit developing and testing real-time subsystems or blending them into a single supercomputing complex, through bus extension, for fully integrated functionality. In such a system, development is also parallel, and schedules can be reduced.

Smaller-Grained Parallelism

With the same hardware configuration as that in Figure 5, a different software approach for real-time application can produce immediate, recognizable advantages.

In a given subsystem, if a real-time module execution is limited, for example, to 1 millisecond, if every CPU's local memory has the same code, and if the static data (data that must not change over an iterative (frame) boundary) for all modules are in a node's global location, any CPU can execute any module. Time usage can be optimized with respect to sequencing and scheduled by predetermined, structured manual methods or by software. Such methods are currently used for flight controls, where fault-tolerance is a key objective. (See CRMmFCS.) Spare CPU's can be assigned to modules as soon as there is any sign of trouble. There is no central executive mainly for fault tolerance. For simulation:

- Spillover can be almost eliminated. A distributed scheduler for parallel processing permits sizing the system dynamically in terms of the number of processors. If time runs out, a CPU can be added with no software redesign or load rebuilding.
- Only a single, structured task must be built for the entire subsystem. This eliminates much development and maintenance overhead and maintains the advantages of functional distribution.
- If there is a CPU failure, N-1 processors can take up the slack immediately, and if necessary, fidelity can be slightly degraded, temporarily, by decreasing the synchronous iteration rate, or a spare CPU can immediately take over. For a new, faster microcomputer upgrade, requiring fewer processors, software need not be redesigned, although it may be desirable to rebuild the sequencing network for scheduling optimization.
- Software configuration is then as flexible as the hardware.

Modularity

Modularity means different things in different disciplines. In simulation, definitions differ among instructors, system designers, and electrical, mechanical, software, and aerospace engineers. In a given field, there are different perspectives, and interdisciplinary projects involve a variety of perceptions of a module. Here we define a module recursively as a self-contained unit that implements a specific function (or class of functions) that constitutes a subset of another larger unit or complete system.

A system comprises well-defined subsystems, and these may also be divided into discrete building blocks down to primitives. All intermodular connections (mechanical, electrical, parametric, or logical) use specialized modules. This encourages designing standardized interfaces; these can cause great performance inefficiencies, which may be offset by large compensations in other areas.

In many systems, the primitives are numerous and may also be microscopic. Design that manipulates these directly is extremely complex and time-consuming. This type can approach maximum optimization, and this may make the product too complex to copy.

In sacrificing optimization of various performance and component-cost-related design elements (to increase it in other areas), we can use, as design building blocks for even higher order functions, logical groups comprised of relationally ordered physical primitives that define some higher-order function.

Although we want to decrease the number of operational primitives as much as possible, we want to do this without making the system modules so general-purpose that we need too many, so that they are largely wasted. This can happen if:

- Modules are overgeneral. (Single modules with many capabilities are used too often for only a small capability subset, and much functional capability is wasted.)
- A system with fewer degrees of freedom (primitives) makes excessive manipulation necessary to express a complex pattern that possesses a natural set of descriptive and functional parameters.

Complexity can be reduced both logically and physically, but any system that has even a few primitives has ordering permutations that vary factorially. We can prevent chaos by making further rules of structuring. This approach reduces complexity and impacts design, procurement, maintenance, and manufacturing costs positively and

makes it possible to handle previously unsolvable design and research problems.

The level at which a targeted system or subsystem becomes overburdened depends upon price and performance expectations. For computational systems, this level depends upon the availability of hardware and software technology. Different subsystems based on different technologies often have optimization and modularity cutoff points at hierarchical levels that are not parallel. For example, current software technology permits much more modularity than that permitted by the technology of the computer hardware it normally runs on. The reason is that extremely modular software requires more processing, bus, and memory overhead, which require very powerful minicomputer or mainframe computer resources, and these are much less modular and far more costly to acquire or to design, maintain, expand, and interface to custom hardware. To produce modular, functionally-distributed systems, we must match the modularity levels of software and hardware so that we can combine the advantages of dedicated hardware and general-purpose devices.

All our simulation processes still represent multidimensional events serially. Defining modules, determining where they will reside within a finite array of sequential processors (partitioning), and sequencing them is a logical process rather than one that is constrained naturally. This causes abstractions that optimize some but never all problem aspects.

If we sacrifice the traditional view of serially oriented modularity, a module can reside on a group of processors in a way that mirrors space and time more accurately, and what we saw earlier as fragmentation, or spillover, may really be the optimum solution to a specific problem. It may result in parallel processes that compute and send data to adjacent processors while they simultaneously receive information from other adjacent processors. Such a solution may require more than one information path, and it may seem complicated or convoluted to people conditioned to sequential problem solving.

Some existing computer architectures share linear instruction streams on a limited basis among more than one processor. In such systems, nondependent instructions from a traditionally linear stream can be executed in parallel. With more intelligent compilers, we can optimize code for this kind of environment, but, unless we change the basic approach to process definition and the resultant algorithm definition to encompass parallelism at the fundamental logical level, we will have accomplished only multiple linearity, and a great deal of capability will be unused because of between-process waiting for sequential events. The new algorithms must be modular in their own right.

Motivational Perspective, Goals, and Some History

A modular, functionally-distributed, memory-mapped system is one of two functioning alternatives. However, one leads to an eventual performance and economic impasse and the other to the rapid, extended growth we need. Flight simulation will always be performance-driven. We must continue to devise various methods to achieve required performance levels and will seek more economical ways of doing so that do not sacrifice performance. Parallel computation is the answer.

Basic Challenges: The Digital Feedback Syndrome (Simulation Performance and Computational Performance)

Simulator functionality channeled into intelligent digital devices is related directly to the simulator's requirements and also to those of the devices being emulated. There is a feedback relationship, largely because our dependence upon digital technology continues to increase and because the vehicles and systems simulated are so complex.

The performance of training systems is also increasing because of the advances in software technology, and these same advances make target vehicles intelligent, another feedback situation. The increased automation and machine-thinking (artificial intelligence (AI)) used on vehicles must be reproduced on simulators, and this automation and AI must also be hosted by an intelligent training system.

There will be an increased requirement for computational horsepower. The development environment and its control systems are driven by the marketplace. Automated engineering techniques are commonplace. If we do not use them, we will not achieve at break-even cost the technical sophistication that our increasing product performance specifications require.

The Problem Eight Years Ago (The Hardware Years)

Eight years ago, we viewed and dealt with this problem largely from a hardware perspective. In establishing performance specifications, hardware designers and software engineers communicated little, except for requests like, "Make it fast enough to run FORTRAN instead of assembly language." We designed the hardware first, and software was generated to fit the system. In this situation:

- Although various forms of replicated, parallel processing seemed possible as solutions to high-speed applications through aggregation of inexpensive VLSI computers, there were no buses available that supported this environment. It appeared that it would be necessary to invent special-purpose proprietary buses.
- LSI and VLSI processors at this time did not have enough serial processing power to support large-grained parallelism. We conceptualized hardware schemes with too many processors, and the mostly serial software was unsuited for distributed processing.
- Experimentation with distributed processing software management theory was usually in areas where many small, unrelated, or self-contained processes were distributed over a network. This resulted in a migration from centralized time-sharing systems to networked intelligent workstations.
- Insufficient attention and funding were given to implementing very large, tightly-coupled, real-time software applications on many microprocessors.
- Supposedly, the Industrial Revolution revealed the value of having interchangeable parts. However, eight years ago in simulation, the use of custom analog and digital portions of the product was still economi-

cally viable, and general-purpose interchangeable parts did not seem so important. We realized that most custom analog and digital functions would eventually be implemented with general-purpose digital logic, but it was still feasible to build systems from electronic components. Now that Multibus, Qbus, VMEbus, etc., have emerged, *this is no longer true.*

Today -- Software is the Key Issue

Major design emphasis has shifted from hardware to software; many systems designers now have software backgrounds, because:

- Software requirements are greater -- A growing number of functions now use microcomputer-controlled systems. This provides numerous advantages in all areas, but it also creates more software. A function *formerly implemented with an analog circuit* or a digital combinational logic sequence may now use a high-order-language computer program that runs on a general-purpose processor, which stimulates target devices, servos, or transducers through direct interfacing over a standard digital bus or general-purpose I/O system module. We suddenly have new, unprecedented concerns about design, development, and documentation for software and firmware. With all other software, the new software requirement causes more monetary outlay than all non-software areas combined. The sheer volume of software and its proportionate cost is growing, not only because software is replacing hardware, but also because of new digitally implemented functions.

- Software functionality is increasing -- Because the range of function computers now perform is so great and because the software design to provide these services is so complex, the performance expectation levels in competitive products are very high and will continue to rise. There has been a major shift in the areas where engineering dollars are spent. "Magic" is now commonplace. This will continue to affect all human interfaces, especially training.

- Increased software functionality will cause direct exponential performance demands upon hardware systems -- When AI technology produces subcognitive software intelligent enough to produce code from specifications, there will be a software explosion that will dwarf our present conceptions of "big."

- Almost all diagnostic and maintenance functions in our products will soon be automated -- This will drastically reduce the amount of money spent on these products during their useful lives.

- Technology life-cycles are growing shorter -- We are looking for solutions that stretch the use of an overall topological configuration, e.g., the use of public buses. The microcomputer industry is cooperating in this situation by implementing such buses; the competitive arena can concentrate on improving products that run on them.

Cautions Against Buzzwords

Distributed -- Should imply the use of parallelism for increasing both the power and modularity of a system. The presence of many processors does not imply this for real-time applications.

Non-Von Neumann -- Most CPU's use program counters and programs stored in memory...that is Von Neumann, even if there are 100 of them! Even *Dataflow machines are using 2 National Semiconductor* 16032 Von Neumann microprocessors for each CPU card. The assumption that non-Von Neumann implies a quantum leap in technological expectations is untrue.

MIPS, FLOPS, WHETSTONES, # of VAX 11/780's -- Run your application, and then get disappointed. Use a stopwatch...not an instruction counter.

Concurrent -- Currently, this term describes both serial and parallel systems. One CPU can run a number of related or nonrelated processes at the *same time. The tasks or processes can continue* indefinitely, while the computer switches back and forth between serial execution of any one of them according to some event-driven scheduler. This is not real concurrent processing.

Goals

Modest goals largely met in the laboratory and now or soon to be disclosed include:

- Truly distributed architecture to meet future demands

- Intelligent human interfaces with an entire range of programmability. The system can seem like one large serial processor or like a specifically accessible array of individual units. Serial and parallel computing processes can both be designed to run on such systems. system design takes full advantage of the linear computational expansion the hardware offers and of the parallelism that results, which itself encourages partitioning.

- Functionality generic (i.e., machine- and language-independent) enough to withstand and exploit all forms of hardware and software advancement. Use of portable, widely used languages and operating systems with memory-mapped communication techniques is fundamental to achieve this.

- Automation and fault tolerance to reduce costs. Reduction of the numbers and types of physical modules that comprise the system, faster and simpler communication, and I/O cards that are intelligent and interface directly to the global backplane bus are making this goal a reality.

Conclusion

Maximizing use of system-level, off-the-shelf products for public buses has given tremendous impetus to our work in Binghamton by permitting us to concentrate on the hardware modules needed to meet our goals. It also makes possible a much larger, and also less complex, software effort, because we can use high-order languages throughout system software implementation.

We are currently realizing these goals. Parallelism is being applied not only to the computational portions of simulation design but also to coordinated work in all other areas.

Because of its size and its roots in systems theory, the computational problem has yielded solutions, and we can apply these generally to almost all engineered subsystems. A proof of concept for any of the several variations of modular, functionally-distributed, microprocessor-based simulators will have to be retrospective. This approach works and is viable. The length of time between now and the day that thinkers and planners of the simulation industry no longer consider this new approach modern will be the proof of concept.

References

1. Satyanarayanan, M., Multiprocessors: A Comparative Study, Prentice-Hall, 1980, Chapters 8,9,10.

2. Leighton, F.T., Complexity Issues in VLSI, MIT Press, 1983, pp. 12-16.

3. Abelson, H., diSessa, A., Turtle Geometry, MIT Press, 1980, Chapter 9.

4. Moto-oka, T. (Editor), Fifth Generation Computer Systems, International Conference on the Fifth Generation Computer Systems (1981: Tokyo), North Holland Pub., pp. 93-106, 121-130, 147-158, 189-222, 277-282.

5. Law, A.M., Kelton, W.D., Simulation Modeling and Analysis, McGraw-Hill, 1982, pp. 59-101.

6. Rosenzweig, M.R., Bennett, E.L., Neural Mechanisms of Learning and Memory, MIT Press, 1976, pp. 57-66, 79-96.

7. Larimar, S.J., Maher, S.L., "A Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMmFCS)," AFWAL-TR-81-3070, 1981, pp. 1-81.

8. Symon, K.R., Mechanics, Addison-Wesley, 1971, Chapters 5,7,8.

9. Padua, D.A., Kuck, D.J., Lawrie, D.H., "High-Speed Multiprocessors and Compilation Techniques," IEEE Transactions on Computers, Vol. C-29, Sept. 1980.

10. Blech, R.L., Arpasi, D.J., "Hardware for a Real-Time Multiprocessor Simulator," NASA Technical Memorandum 83805, Jan. 1985.

11. Bosworth, L.K., Connors, J.J., Goad, D.E., et al., Modular Simulator Concept Definition, Logicon, July 1984, Vols. I and II.

12. Bell, G., Snyder, F., Standard Modular Simulator Systems Program -- Phase II, Boeing Co., July 1984.

13. Holtsman, W., MicroSimulation Technology, Singer LFSD -- LR1069, Feb. 1982, Vol II.

14. Marini, L.G., Finn-Hawk LJT Hardware Ref. Manual, Singer Co. UK Link-Miles (Internal), 1981.

15. Muchmore, S., Functionally Distributed Simulation Architecture and Systems Overview, Singer Co. UK Link-Miles -- LMR 497, Dec. 1982.

16. Technology Report, Electronic Design, Feb. 21, 1985, p. 166.

Biography

Michael Ash is a research and development staff engineer at the Link Flight Simulation Division of The Singer Company in Binghamton, New York. During the past three years, he has been involved exclusively in systems design of distributed microcomputational environments for simulation.

Mr. Ash received a BA in Physics and an MS in Computer Science from the State University of New York at Binghamton. He has coauthored six books.