

GETTING CLOSER TO THE MACHINE

by: W.C. Cave† & R.E. Wassmer†

April 23, 2007

INTRODUCTION

It does not take much investigation to confirm the growing importance of the software industry. Despite huge investments, the industry lags far behind demand. Yet software technology is increasingly characterized by poor productivity, poor reliability, and slower running times, see for example [1], [2], [3], [4], and [5]. In certain sectors, demand for increased functionality contributes to reductions in speed and reliability. But in many sectors, case histories show that productivity and reliability have fallen in spite of the fact that project sizes have been decreasing over the past ten to fifteen years due to management concerns with risk of failure, [6], [7]. Repeated citations in the literature by those sincerely trying to improve the technology are (1) the apparent inability to learn from history, and (2) the layering of new code on top of old code, making it slow as well as difficult to comprehend, see [8], [9], [10], and [11].

Since the late 1970s, programming approaches have been moving further from the machine, hiding data behind layers of abstraction. The measurable results are lower reliability, lower productivity, and the need for faster processors. For over two decades Moore's Law doubled processor clock rates every 18 months, helping to conceal the software speed problem. This led to (Niklaus) Wirth's Law: "Software gets slower faster than hardware gets faster."

But Moore's Law hit the power dissipation density wall, and clock speeds have flattened. Manufacturers are now providing multiple processors on a chip to increase processing speed. As jobs move off-shore, and programming multiple processors proves much more difficult, we must consider that current approaches to programming are falling behind on three fronts - run-time speed, productivity, and quality. This paper describes a new approach to building software that solves these problems.

PERTINENT HISTORY

If one looks at the evolution of the computer, it started with electro-mechanical data processing (EDP) machines using programmable boards. One had to consider timing and synchronization to program (wire) the boards to ensure that the parallel sequences of electrical and mechanical actions provided the desired outcomes. Many jobs, e.g., sorting and collating, could be done independently - on different machines. But managers had to plan what jobs could be done *concurrently*. If a machine jammed, work on a particular job was halted until a repair was made. So the reliability of each machine had to be factored into the work plan. A good manager split up the work, taking maximum advantage of the *independence* of jobs and job steps, as well as knowledge of the machines.

† *The authors are with Visual Software International (www.VisiSoft.US)*

As the electronic computer took over, a major change occurred. Von Neumann came up with an architecture that added instructions into the same memory that originally stored only data. This forced programs to follow a sequence of instructions. Timing and synchronization were moved to the logic designers of the machine, removing a programming enigma. This great simplification led to the dramatic increases in software complexity, including Operating Systems (OS) that manage machine resources used to run independent tasks.

Today we are faced with the realization that semi-conductor expert Jim Meindl's prophecy was accurate, [12]. The Moore's curve for speed has flattened. The doubling of CPU clock rates every 18 months is gone. To make up for Wirth's law, programmers are facing multi-core processor chips. As Herb Sutter from Microsoft puts it, "The free lunch is over, ...", "because concurrent programming is hard." [13].

This has led to tailoring programs for 2 or 4 processors, and corresponding facilities for "threads" that are *not* oriented toward large scale (hundreds of) parallel processors. As a result, programmers are back to the EDP board mindset, worrying about timing and synchronization. They are not concerned with planning a huge task to run on hundreds of processors. With the EDP mind-set as a reference frame, automation has not been a concern - at least until now, as complexity multiplies, see [14]. So let's consider where we must go from here.

THE NEW ENVIRONMENT

Figure 1 illustrates the parallelization of platforms that has evolved since the early days. Most computers today may be viewed as parallel platforms communicating with each other. Each platform is managed by its own OS with tasks that are highly independent of those on the other platforms (embarrassingly parallel). From manufacturing to retail, businesses depend upon platforms tied via networks. People at the cash register do not want to wait more than a few seconds to complete a sales transaction. Time is money. This requires fast communications.

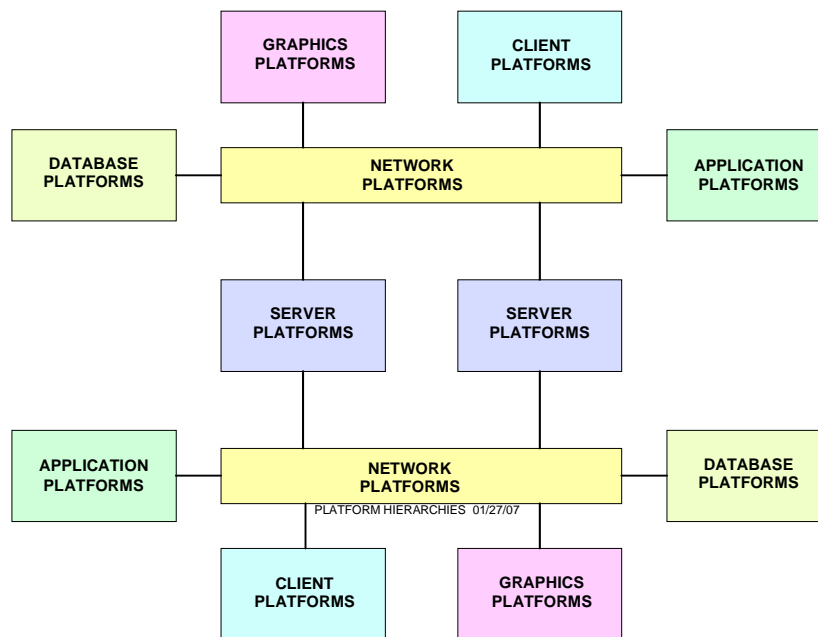


Figure 1. The parallelization of platforms.

GETTING CLOSE TO THE MACHINE

Looking behind each platform in Figure 1 we find a “machine”. There is growing agreement in the literature about where the “machines” are headed. This is depicted in Figure 2. Most of the recent literature is looking at multi-core chips, because that is what is being manufactured. The new chips offer two processors (2 cores), albeit each is slower than a fast single processor. This is different from the multiple server environment because both processors are managed using the same OS. Writing a single task to run under a Single OS (SOS) and effectively use both processors is a new problem to this market.

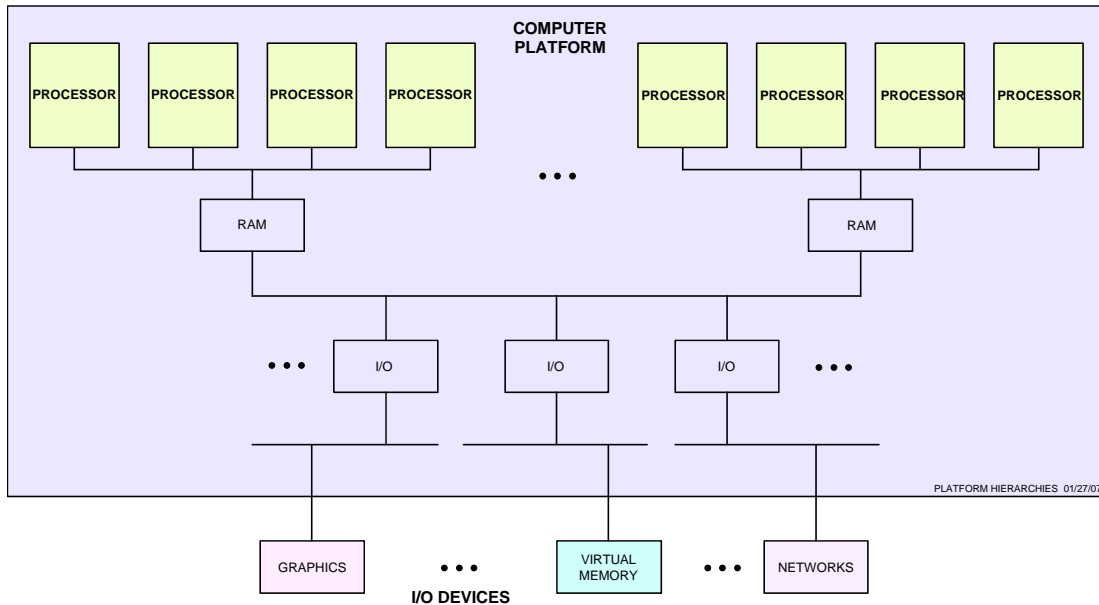


Figure 2. The computer platform of the near future.

Then there is the engineering market where hundreds of processors have been put together under a SOS to run large scale simulations. This is somewhat more difficult. One has to maintain synchronization with a *simulation clock*, a much more tedious problem than just synchronizing with a real-time clock. Unless synchronization with the simulation clock is maintained, simulation results quickly become chaotic as well as invalid, see [15].

Many authors have documented experiments using large numbers of processors to run a single large scale simulation. This provides a significantly different viewpoint from that of managing two processors. It is a viewpoint that magnifies the errors in approach. Much test data exists for various schemes designed to solve this problem. The results have been counter-intuitive as illustrated in Figure 3. In most cases, speed multipliers fall off quickly if not going fractional (faster to run on a single processor). This is because of the level of complexity that must be dealt with, and the totally different space of solutions one must consider. The result is that most parallel processor suppliers are no longer in that business. Except for special applications, building software to take advantage of their hardware has been much too difficult.

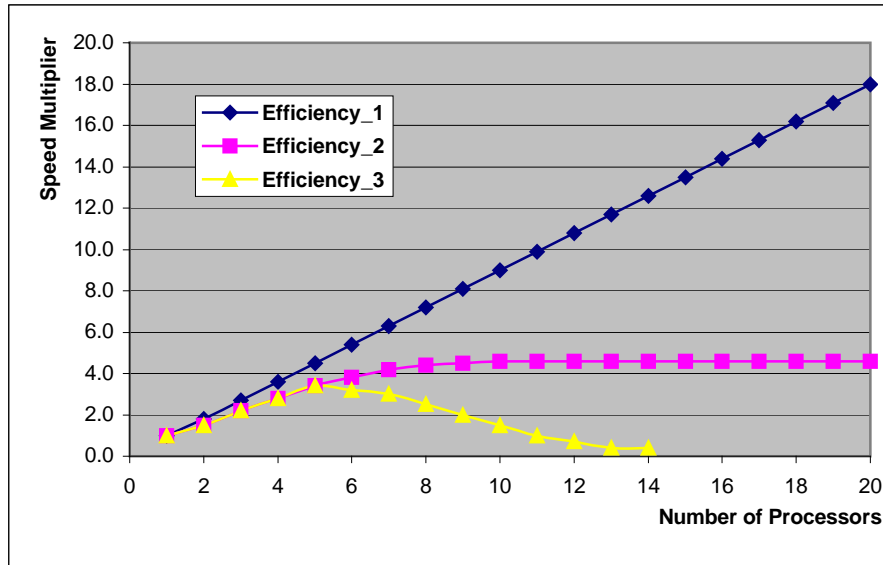


Figure 3. Efficiencies of different software approaches to SOS parallel processors.

What is more important is that programming approaches that make software slower on single processor platforms are going to make things *much slower much faster* on a SOS parallel processor (an exponential form of Wirth's Law). Piling layers of abstraction on top of the existing layers will move us further from the machine faster.

Figure 1 embodies the embarrassingly parallel case. Tasks are restricted to a single processor. Programs follow the sequential nature of the Von Neumann architecture. Transfers between processors are simplified by the use of Inter-Processor (IP) communication protocols. In the SOS parallel processor of Figure 2, one is confronted with Efficiency_3 in Figure 3. We will explain this problem and its solution below.

Before addressing the problem of SOS parallel processors, we must understand the next layer down, the basic processor itself. If we are not close to the single processor, then with N processors we are going to be much further removed. We must map software into one processor effectively, assuming that we are going to multiply it by hundreds. If we are not thinking in terms of using hundreds, we are not working in a space that will achieve an effective solution.

Understanding The Single Processor

Almost all R&D money spent on parallel processing to date has been on hardware, with software an afterthought. Going back to the Holland Machine in 1958, [16], hardware designers have tried to produce new architectures that improve on Von Neumann's original design. Instead, simplicity of programming has caused this basic design to evolve over the past 50 years to the latest chips as illustrated in Figure 4. The most significant architectural changes are the move from words to bytes (IBM 360), and the separation of data memory from instruction memory (PC on a chip). Although other benefits were realized, the driving force behind both changes was speed.

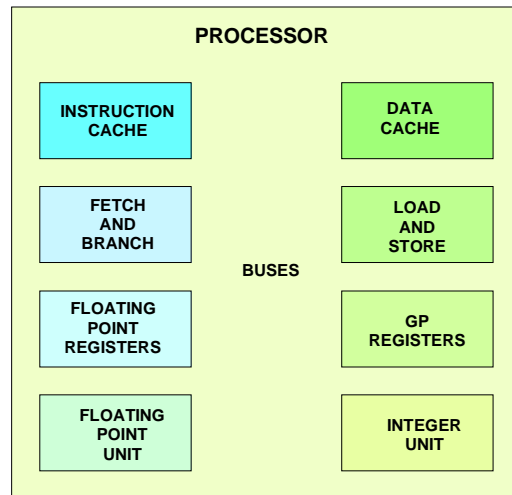


Figure 4. Illustration of the “new” Von Neumann architecture.

Behind most every platform today is a processor similar to that illustrated in Figure 4. Speed is determined by a number of factors. Referring to Figure 1, applications consist of functions that end users want to perform. Users generally work interactively on client and graphics platforms. In the client server market, application platforms process client input data using databases and return outputs to client and graphic platforms as well as updating databases. Most factors affecting speed are associated with transferring data (communications).

In band-limited systems (computers and busses), minimizing data transfer times implies minimizing the data transferred. In Figure 1, copies of databases are common, and transactions are identified by simple codes and numbers packed into records for transmission. This implies designing data structures that minimize the data transfers required to perform the desired functions. Transferring programs (instructions) is avoided when speed is important.

Using a single processor, multiple tasks may run in *virtual* concurrency, with the OS managing utilization of the processor based upon task priorities and resources requested from each task (e.g., memory, I/O devices, etc.). The OS swaps blocks of instructions in and out of instruction cache and pages in and out of data cache. Swapping and paging is done based upon where things reside (the *context*) in the memory hierarchy (cache, RAM, disk, etc.).

Beyond The Processor

At this point we must differentiate between Input/Output (I/O) transfers (memory to I/O devices) and memory-to-memory transfers. Figure 4 illustrates the processor CPU and on-chip memory only. Additional devices, including further hierarchies of memory as illustrated in Figure 2, still surround the single processor chip. Large blocks of instructions that use local memory transfers can be processed fast compared to a single I/O instruction. In these cases, the OS can determine when swapping and paging among tasks is effective. Because I/O transfer delays are easily isolated with good software architectural approaches (described later), our interest here is the speed of internal processing in a large single task.

As a single task grows in size, swapping and paging may occur when it is the only task running on the processor. It is not unusual for large tasks to use huge data spaces that require multiple gigabytes of storage. In this case, swapping and paging may occur without I/O transfers. Transfers across memory hierarchy boundaries will incur relatively significant increases in delay. These are caused by speed differences in memory types as well as transfer delays on the interchange busses. In the case of a single task on a single processor, transfers across memory boundaries have a significant impact on speed. To maximize speed, one must map out memory in a way that minimizes swapping and paging. This implies predicting what blocks of memory will be used beyond the current block of instructions. Programs containing complex sets of algorithms that deal with large blocks of data are only understood sufficiently by the programmer. With current approaches to programming, it is doubtful that the average programmer can make such predictions. Expecting such predictions from the OS is folly.

Parallel Processing And Excess Memory Transfers

If we expect to achieve speed multipliers anywhere near the Efficiency_1 curve in Figure 3, we must understand the total problem. This curve is a 90% *effective processor utilization* curve. It implies that the parallel processors were used at an average of 90% of the efficiency of a single processor solving the same problem. Anyone familiar with parallel processing, and particularly with discrete event simulation, knows that this is historically difficult to achieve except in embarrassingly parallel cases.

The applications of interest here are *partially independent*, i.e. those with substantial inherent parallelism, but with a reasonable degree of communication between otherwise independent parts. An example is the simulation of communication networks where each node has a substantial amount of internal processing, but the internal processing depends upon data transfers between nodes. This is typical of large simulations in general. Large real-time control systems also fall within this category, some requiring embedded simulations, with exceptionally fast time constraints. All of these cases are candidates for large SOS parallel processors.

When moving to a large parallel processor, transfers across memory boundaries become costly. In the case of partial independence, using multiple processors generates *excess memory transfers*, i.e., transfers between processors that do not occur on a single processor machine. These may involve instruction as well as data memory transfers. Moving threads to a different processor and sharing memory between processors are major causes of excess transfers.

To simplify sharing of memory, architectures such as the Kendall Square Research (KSR) machine provided virtual access to memory from any processor using an “ALLCACHE” system developed by Frank, [17]. The “breakthrough” of this approach was that it maintained *coherency* of the total “cache” in hardware, and used a distance measure to determine where to place blocks of data and where to run threads. These features notwithstanding, it could not solve the software architectural problem of minimizing boundary transfers while easing the burden of programming of parallel processors. As we show below, maintaining cache coherency in this manner does not contribute directly to easing the software solution, and in many cases may slow things down. Based on physical distance, the average memory transfer time for a large parallel processor is much larger than that of a single processor. Maintaining the average is going to slow things down.

Many large tasks can benefit from the parallel architecture in Figure 2. However, as proven by experiments in simulation and illustrated in Figure 3, the benefits will not be realized (speed can get worse) without a completely new approach to software design. And this approach must be mapped to the basic machine design.

The most important lesson we can learn from Von Neumann is that we must make the programmer's job easier if we expect to break the barrier of solving more complex software problems. Von Neumann removed the requirement for programmers to deal with the logical design problems of timing, synchronization, and corresponding delays and race conditions. If we put this burden back on the programmer, we are heading in the wrong direction. To deal with the increase in complexity, we must provide a solution that makes it easier to program parallel processors than the current approaches provide for single processors.

In the case of partial independence, if we expect to move from the Efficiency_3 curve toward the Efficiency_1 curve we must minimize the transfers across memory boundaries. At the same time, we must ease the programmer burden. In the ensuing discussion, we will show that, although these objectives appear to be at odds with each other, they are not. However, we must be prepared to make a leap in thinking that is equivalent to going from wiring boards to writing computer programs.

A NEW APPROACH TO SOFTWARE DESIGN

The approach described here has evolved in an engineering design environment requiring large scale discrete event simulations whose run-time constraints require a SOS parallel processor. The concepts are based upon two new paradigms. First is the *Separation Principle*, where data is separated from instructions, as in the chip in Figure 4. The second is the *Generalized State Space* framework, where software is viewed as a set of functional transformations (instructions) on state vectors (data). These new paradigms lead to *software architecture* and its visualization, with a one-to-one mapping from the architecture to the code. With a good architecture, the overall database for an application can be designed to support effective SOS parallel processing. Facilities supporting these paradigms have evolved since 1982 into VisiSoft[®], a CAD approach to building large software and simulation systems.

Given end user requirements, one must determine the software functions that are required to perform them. These may be described as transformations of state. This implies describing the functional states of the software system, and the transformations on those states. We note that the decomposition of software functions (e.g., mouse handler, keyboard handler, etc.) may be orthogonal to the decomposition of end user functional requirements (e.g., enter specified data, produce a specified report, etc.).

Inherent parallelism of an application can be measured by the independence of transformations required to implement the software functions. Transformations are independent if they share no data, and can therefore run concurrently. If software functions can be grouped into large blocks of independent transformations, there is a high degree of parallelism. If every instruction shares data with the prior instruction, then there is no independence. We assume that sharing data implies dependence, i.e., *data that need not be shared is not shared*.

Machine Factors Affecting Speed

It has been shown that the hardware and OS architecture of a multi-processor environment can have a dramatic impact on software run-time even though the processor (and clock speeds) are effectively identical, see [15]. That paper summarizes an experiment where the delays between processors were varied to illustrate the difference between a multiple platform cluster (separate OS on each platform) versus a tightly coupled SOS parallel processor. The results demonstrated that a SOS parallel processor can provide huge improvements in throughput that grow linearly as the number of processors grows.

However, achieving these improvements requires two new facilities. The first is a development environment that provides for the creation of independent modules that can run concurrently on a SOS parallel processor. The second is a run-time environment that contains “smart” scheduling and allocation algorithms that use the knowledge of inherent parallelism in the application. The aforementioned paper, [15], used a discrete event simulation - a difficult parallel programming problem - as the test. It also considered the saturated case (all processors used) versus non-saturated case (some processors free), and the use of smart load balancing algorithms to achieve further gains in the saturated case.

Figure 5 provides a generalized depiction of the VisiSoft environment used in the test. At the far left are the application requirements to be satisfied. At the far right is the operating system and hardware to be used to produce the results. Using the paradigm described here, developers must design an architecture and produce code using the development environment. The development environment generates the application run-time software along with a run-time environment tailored to run on the operating system and hardware provided.

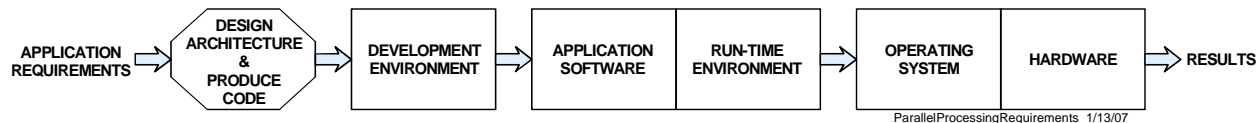


Figure 5. Depiction of the software environment.

The development environment provides the programmer interface to generate the application software compatible with the run-time environment. The run-time environment provides tailored interfaces to the OS and hardware. This paradigm provides a programmer interface that is independent of the OS as well as the hardware platform. It also allows the programmer to focus on producing software that is easily tested and enhanced without concern for run-time management issues.

Taking full advantage of the parallel processor illustrated in Figure 2 requires a space in which to portray new concepts. Rather than continue with a mathematical description, we introduce the new paradigm for software architecture. The goal of this new paradigm is two-fold. First, provide the ability to develop architectures that map directly to the machine in a way that takes maximum advantage of inherent parallelism in an application. Second, maximize the ease with which subject area experts can build software, by eliminating complicated programming techniques and maximizing software productivity. To accomplish this, we borrow concepts from control theory.

The Generalized State Space Framework

To capitalize upon the concepts of the *State Space* framework developed for control theory, [18], one must extend the mathematical definitions of vectors and transformations. These concepts are significant when translating a set of functional requirements into software. When devising mathematical transformations, selecting the best set of independent state variables is a critical part of coming up with the best solution. “Best” usually translates to speed which, in turn, translates into minimum operation counts, see Hachtel et al, [19]. This concept applies directly in software. Finding the best set of state vectors can make a huge difference in transformational burden. For example, when dealing with complex data sets, design of the tables used for transformations of the data is usually key to simplifying both the approach and resulting code. When taking advantage of inherent parallelism in a system, one must determine the best architecture of software modules that can run concurrently on parallel processors. Again, best translates to run-time speed.

Achieving Parallelism In Software Architectures

Software architectures designed using the new paradigm have no relation to control flow or flowcharts that represent code. Instead, they are based upon the concept of *independence* of modules, threads, and large blocks of instructions that may run concurrently. Achieving independence is the major component of a parallel processor software solution. Other components are: (1) passing this information to an OS; and (2) having the OS take advantage of it with effective processor scheduling, allocation, and assignment. An example of an architecture produced from this environment is shown in Figure 6. Achieving this architecture requires a new software paradigm - the *Separation Principle*.

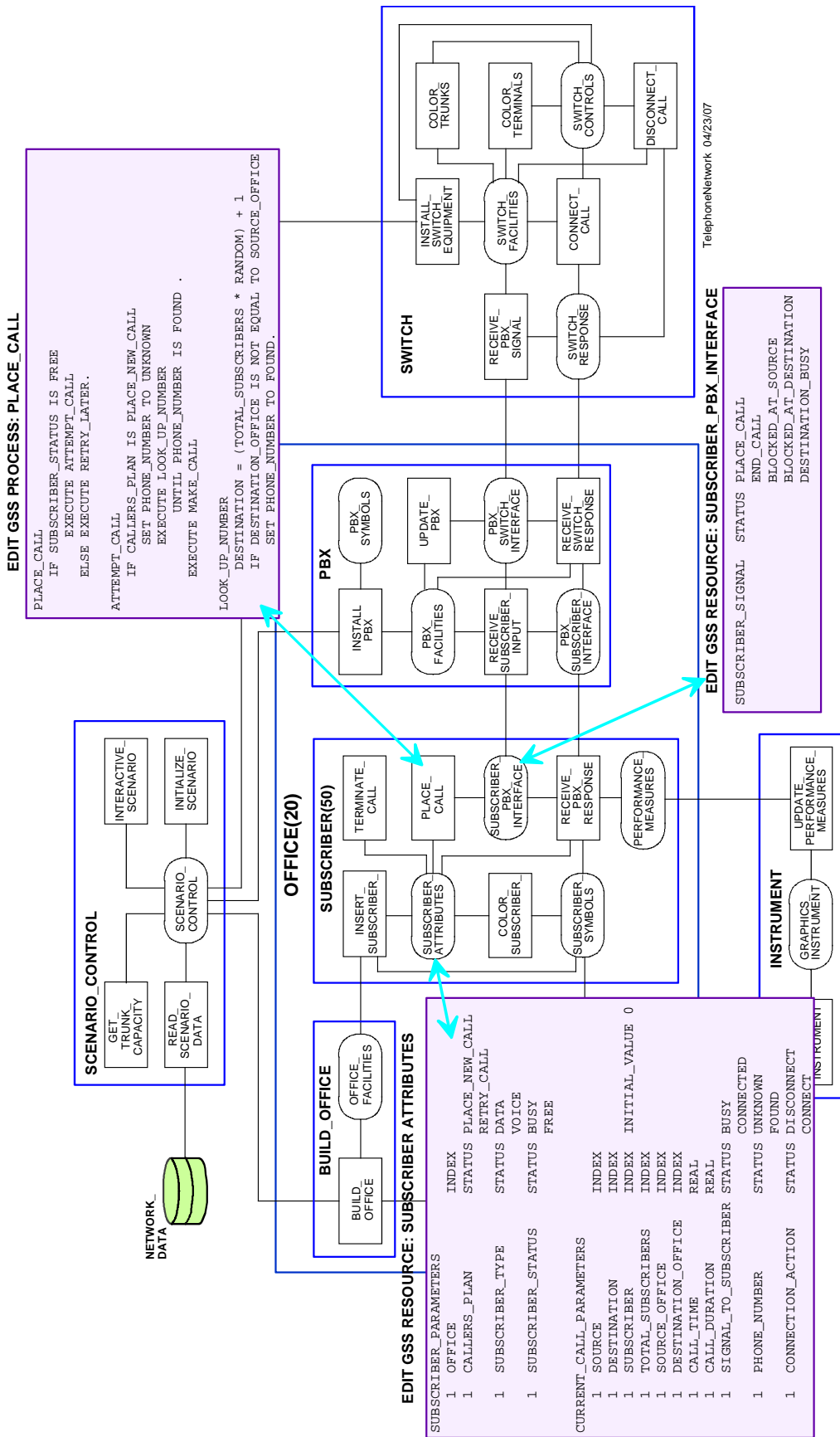
The Separation Principle For Software

For *processes*† (blocks of instructions) to run concurrently on parallel processors, they must be independent. Two processes are *independent* if they share no data. Thus, a software development environment designed to support parallel processing must provide for the creation of independent processes. Since it is up to the programmer to create independent modules based upon inherent parallelism in the system, one must separate data from instructions at the programmer interface level. This has been called the *Separation Principle*, defined by Cave in 1982 in the design of the General Simulation System (GSS), the first component of VisiSoft.

The Separation Principle is achieved in VisiSoft by storing all data in *resources* that generally contain hierarchical data structures. Resources are depicted as ovals in architectural drawings as illustrated in Figure 6. Another example is shown in Figure 7.

Processes contain hierarchical sets of rules and are depicted as small rectangles in Figure 6. Another example of a process is shown in Figure 8. Data cannot be declared in a VisiSoft process. Processes may only reference data defined in resources to which they are connected. Resources attached to a process (by a connect line in the drawing) are referenced automatically by pointer.

† *Process* as used here is similar to an assembler language subroutine that contains only instructions that reference only external data (by pointer). It has no relation to a UNIX process.



TelephoneNetwork 04/23/07

Figure 6. Telephone network models in GSS.

RESOURCE NAME: MESSAGE_FORMATS			
MESSAGE			
1	SYNC_CODE	CHARACTER 6	
	ALIAS VALID	VALUE '101010',	'010101'
1	TYPE	STATUS FORMAT_A	
		FORMAT_B	
1	CONTENT	CHARACTER 46	
FORMAT_A REDEFINES MESSAGE			
1	PAD	CHARACTER 14	
1	HEADER		
2	PRIORITY	STATUS FLASH	
		IMMEDIATE	
		ROUTINE	
2	ORIGIN	INDEX	
2	DESTINATION	INDEX	
	ALIAS BROADCAST	VALUE 0	
1	BODY		
2	LENGTH	INTEGER	
1	TRAILER		
2	MESSAGE_NUMBER	INTEGER	
2	TIME_SENT	REAL	
2	TIME_RECEIVED	REAL	
2	ACKNOWLEDGMENT	STATUS RECEIVED	
		NOT_RECEIVED	
2	LAST_SYMBOL	CHARACTER 2	
	ALIAS TERMINATOR	VALUE '\\\ ', '/// ', '<<', '>>'	
FORMAT_B REDEFINES MESSAGE			
1	PAD	CHARACTER 14	
1	HEADER		
2	SOURCE	INDEX	
2	SINK	INDEX	
1	BODY		
2	CONTENTS	CHARACTER 42	

7/13/06

Figure 7. Example of a Resource.

Using this paradigm, programmers are only concerned with indexing, not memory management. This is aided by elimination of word boundary alignment. For example, in Figure 7, MESSAGE is *redefined* by FORMAT_A and FORMAT_B; these are templates over the same area of memory. The languages are designed for understandability.

Processes and resources can be grouped into modules such as SWITCH or INSTRUMENT in Figure 3. Modules can be grouped into hierarchical modules such as OFFICE. Modules are connected when a process in one is connected to a resource in another. For example, in the INSTRUMENT module, the process UPDATE_PERFORMANCE_MEASURES has access to the data contained in the PERFORMANCE_MEASURES resource in the SUBSCRIBER module. This data is also shared with the process RECEIVE_PBX_RESPONSE. The lines connecting modules determine their independence. The INSTRUMENT module is independent of all modules except the SUBSCRIBER module. After initialization, OFFICE contains only two processes that share a resource with the SWITCH module. All other processes are independent and can therefore run concurrently with their counterparts in other instances - on separate processors.

```

PROCESS:      RECEPTION
RESOURCES:    TERMINAL_PARAMETERS          INSTANCES:  TRANSMITTER
              MESSAGE_FORMATS              RECEIVER
              TRANSCEIVER

START_RECEPTION
  IF TRANSCEIVER(RECEIVER) IS IDLE
    EXECUTE GOOD_RECEPTION
  ELSE IF TRANSCEIVER(RECEIVER) IS RECEIVING
    EXECUTE CONFLICTING_RECEPTION
  ELSE IF TRANSCEIVER(RECEIVER) IS TRANSMITTING
    EXECUTE CONFLICTING_BROADCAST.

GOOD_RECEPTION
  IF SIGNAL_TO_NOISE_RATIO(TRANSMITTER, RECEIVER)
    IS GREATER THAN RECEIVER_THRESHOLD
  AND SYNC_CODE IS VALID
    SET TRANSCEIVER(RECEIVER) TO RECEIVING
    ADD SIGNAL_POWER(TRANSMITTER, RECEIVER)
      TO POWER_AT_RECEIVER(RECEIVER).
  IF MESSAGE_TYPE IS FORMAT_A
  AND LAST_SYMBOL IS A TERMINATOR
    EXECUTE SEND_ACKNOWLEDGEMENT.
  CALL DECODE_MESSAGE

CONFLICTING_RECEPTION
  IF POWER_AT_RECEIVER(RECEIVER)
    IS GREATER THAN SIGNAL_POWER(TRANSMITTER, RECEIVER)
    SCHEDULE ABORT_RECEIVE NOW.

CONFLICTING_BROADCAST
  CANCEL END_RECEIVE NOW
  SCHEDULE START_RECEIVE IN EXPON(0.83) MILLISECONDS
    WITH PRIORITY 80

SEND_ACKNOWLEDGMENT
  XMTR = RECEIVER
  RCVR = TRANSMITTER
  MOVE ACKNOWLEDGEMNT TO TRANSMIT_MESSAGE_BUFFER
  IF DESTINATION IS BROADCAST
    SEARCH RECEIVER_CONNECTIVITY_VECTOR OVER RCVR
      EXECUTING TRANSMISSION
      WHEN LINK(XMTR, RCVR) IS GOOD
  ELSE EXECUTE TRANSMISSION.

TRANSMISSION
  SCHEDULE RECEPTION IN LINK_DELAY(XMTR, RCVR) MICROSECONDS
    USING XMTR, RCVR

```

Figure 8. Example of a Process.

THE IMPORTANCE OF SOFTWARE ARCHITECTURE

Figure 6 represents an example of a very simple software architecture. Note that there is no concern for the concept of “flow” in this type of architecture. Instead, the concern is for *connectivity*, i.e., what processes share which resources, depicting the property of independence. In addition, modules may be instanced, where each instance may run independently on a separate processor. For example, the OFFICE(20) module is instanced 20 times. Each instance of OFFICE contains 50 instances of the SUBSCRIBER module - a total of 1000 instances. In a large system, each instance of a hierarchical module may contain large numbers of processes. Clearly the topic of instances deserves more consideration than can be afforded in this overview. It is further explained in [20].

Benefits Of Separating Data From Instructions

The separation of data from instructions provides significant benefits. The connection of processes and resources is represented graphically using icons and lines. This provides the ability to produce an engineering drawing of the *architecture* of a software system.

Resources and processes may be grouped into elementary modules. Elementary modules may be grouped into hierarchical modules. The architecture can be drawn and changed, and the resources and processes edited directly on the CAD drawing as shown in the boxes designated by the arrows in Figure 6. Modules (blue boxes) may be connected to each other by connecting a process in one module to a resource in another. *Independence of modules can be inspected, visually, simply by looking at the lines connecting them.*

Another critical property of this approach is the *Connectivity Matrix* shown in Figure 9. It is maintained by the development environment and used by the run-time environment to determine if two processes are independent and therefore can run concurrently on parallel processors. For each resource, one can see the processes that are connected to it (denoted by an X), and therefore share data. The more sparse the matrix, the greater the independence, and the more simple the transformations, a concept taught to engineers in linear system theory. The blue boxes indicate the module boundaries for elementary as well as hierarchical modules.

Conceptually, this is no different from the problem of picking the best set of variables or coordinate system to simplify a set of partial differential equations. In VisiSoft, one selects the best breakout of resources (state subvectors) to simplify the processes (transformations of state).

The connectivity matrix has additional information. Where there is an I, the resource is shared during initialization only, not during run-time. The O denotes that the resource is shared when outputting data. After initialization, OFFICE contains only two processes that share a resource with the SWITCH module. During run-time, it only shares two resources with the rest of the system, except when producing performance data, a one-way output. All other processes are independent and can therefore run concurrently with their counterparts *in all other instances* in separate processors.

Independent Module Architectures

Using this environment, one can view well designed architectures of large independent modules and visually estimate parallel processing efficiencies based upon the ratio of internal processes to those shared by a connecting module. Conversely, modules can be designed to be independent by virtue of architectural design rules that are checked by inspection of a drawing. This is illustrated in Figure 10. Finally, using this paradigm, the *independence of modules* is an important property of software architecture - whether or not one is concerned about parallelism.

CONNECTIVITY_MATRIX_ 01/16/07		RESOURCES													
MODELS	PROCESSES	SCENARIO_CONTROL	OFFICE_FACILITIES	SUBSCRIBER_SYMBOLS	PERFORMANCE_MEASURES	SUBSCRIBER_ATTRIBUTES	SUBSCRIBER_PBX_INTERFACE	PBX_SUBSCRIBER_INTERFACE	PBX_FACILITIES	PBX_SYMBOLS	PBX_SWITCH_INTERFACE	SWITCH_RESPONSE	SWITCH_FACILITIES	SWITCH_SYMBOLS	GRAPHICS_INSTRUMENT
		READ_SCENARIO_DATA	X												
INITIALIZE_SCENARIO	X														
INTERACTIVE_SCENARIO	X														
GET_TRUNK_CAPACITY	X														
BUILD_OFFICE	I	X	X												
INSERT_SUBSCRIBER		X	X												
PLACE_CALL					X	X									
TERMINATE_CALL			X	X											
COLOR_SUBSCRIBER			X	X											
RECEIVE_PBX_RESPONSE			X	X	X	X	X								
RECEIVE_SUBSCRIBER_INPUT						X	X	X	X						
INSTALL_PBX	I							X	X						
UPDATE_PBX								X	X						
RECEIVE_SWITCH_RESPONSE							X	X	X	X					
RECEIVE_PBX_SIGNAL										X	X	X			
INSTALL_SWITCH_EQUIPME	I												X	X	
COLOR_TRUNKS													X	X	
COLOR_TERMINALS													X	X	
CONNECT_CALL											X	X	X		
DISCONNECT_CALL											X	X	X		
UPDATE_PERFORMANCE_MEAS				O											X
INSERT_INSTRUMENT															X

Figure 9. Resource, Process, and Module Connectivity Matrix.

In large systems, module hierarchies may exceed nine layers, with hundreds of resources and processes contained in mid-level modules as shown in figure 11. A good architecture provides a high degree of control over software. After designing many types of architectures, it becomes clear that coding is more focused at a lower implementation level. Architecture requires a higher level of knowledge and expertise. As in other fields, coders may be prohibited from changing the architecture and producing new drawings.

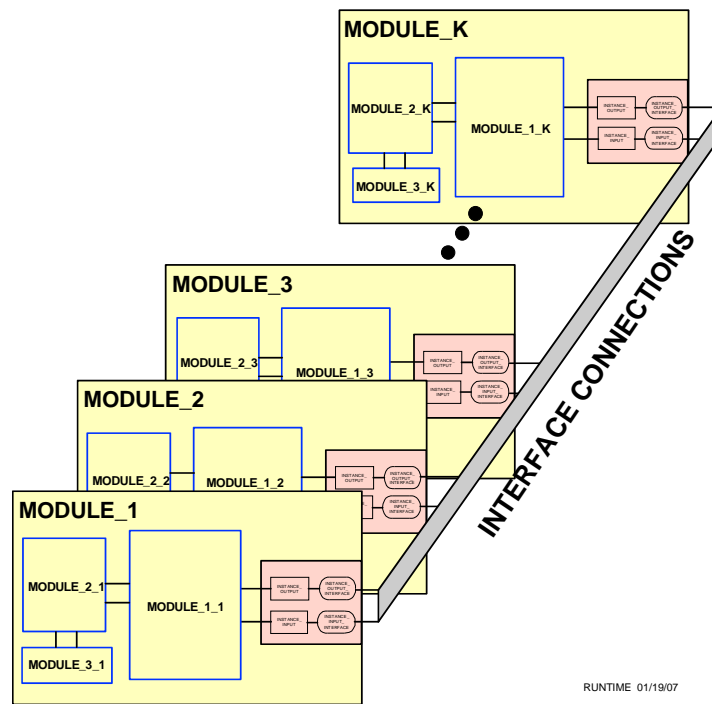


Figure 10. Illustration of independent modules.

The modules shown in Figure 10 may be completely different modules or instances of the same module. What is important is that connections to other modules are implemented using separate *interface modules* (pink). The gray band (interface connections) is symbolic, as are the interface modules, representing interconnections to an Inter-Processor resource manager in the run-time environment discussed below. Although Figure 10 shows single (pink) interfaces in each module, the extension to multiple module interfaces follows directly. The critical aspect of this module architecture is that *all submodules other than interface modules are independent of the other modules*. This rule may be checked by visual inspection.

The concepts and architectural rules covered so far are sufficient for single processor systems. They lead to dramatic improvements in productivity in the development and particularly the enhancement stage of the software lifecycle. In the case of parallel processing, there are additional rules for the programmer interface. They are equally simple and productive.

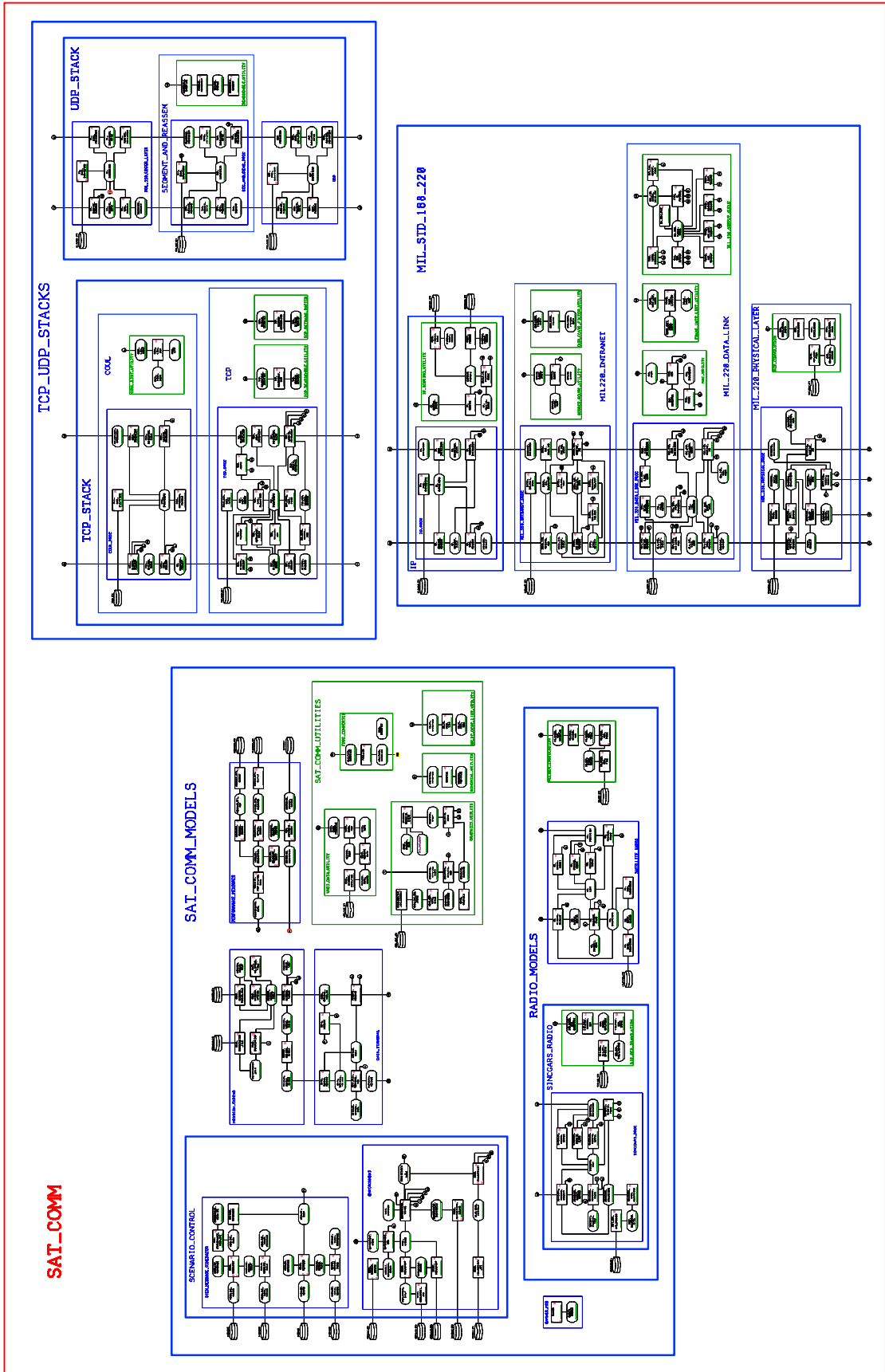


Figure 11. Engineering drawing of software.

PARALLEL PROCESSING ARCHITECTURES

Those familiar with parallel processing have been exposed to the difficult process of achieving efficiency as the number of processors increases. As illustrated in Figure 3, achieving a speed multiplier anywhere near a factor of N , where N is the number of processors, has been the unachieved goal of many computer companies and investors, except for embarrassingly parallel problems. What is apparent is the lack of understanding of the property of independence and the inability to marshal inherent parallelism. Without the Separation Principle and the corresponding CAD architectural facilities of VisiSoft, developing large software systems that take advantage of a high degree of inherent parallelism becomes extremely difficult, if not intractable.

Before describing the ease with which one can develop parallel processing software, some background information is necessary. We describe only those details required to understand the programmer interface.

Run-Time Considerations

In VisiSoft, any process may SCHEDULE itself or other processes while it is running. The SCHEDULE statement causes the name of the scheduled process to be entered into a VisiSoft run-time *schedule queue*, with control passed to the next statement in the original process. Processes may also CALL other processes in which case control is transferred directly to the CALLED process. When execution of a called process has completed, control is returned to the next statement in the calling process. A VisiSoft *thread* contains a lead process initiated by a SCHEDULE statement. When the lead process runs, it may initiate CALLs to other processes that, in turn, may CALL others. Processes that are part of the calling sequence are part of the thread. We note that two *threads are independent if they share no resources*; i.e., none of the processes in one thread share a resource with those in the other. Also, a thread is *contained in a module* if all of the processes in that thread are contained in that module. If two modules are independent, then all of the threads contained in one module are independent of those in the other module. Note that, in Figure 10, threads contained within the submodules of a module are independent of those contained in the other modules.

In a SOS parallel processing environment, the VisiSoft run-time system determines the *threads* to be run on a particular processor. To do this, the run-time system allocates processors to independent modules. It then assigns threads to processors based upon the modules they are contained in. Because these threads are independent, the resources they need are mapped into the local cache of the assigned processor. Threads in the same module need not be independent since they are assigned to the same processor and cannot run concurrently. Threads contained in modules that are independent are assigned to separate processors when available, since they can run concurrently with threads in other independent modules. Once assigned to a processor, they remain there so the context effectively remains the same (an exception is migration of modules, discussed below).

Interface modules contain processes that share resources with modules that are not independent, but may reside on separate processors. When on different processors, the shared resources are treated as Inter-Processor resources (IP resources). Processes attached to IP resources are managed by the run-time system to ensure coherency of the IP resource data.

Architecturally, this is limited to a single process in each processor. In Figure 10, each processor has two resources that are shared across the boundary and correspondingly two processes, providing for two-way communications.

Concurrent Software Architectures

An important feature of this approach is visualization of the architecture relative to the number of independent processes/threads contained within the module versus those at the interface. This simplifies the programmer interface relative to optimizing the ratio of processes that can run concurrently versus those that must be governed by coherency checks at the boundary. In either case, the programmer has no concern with coherency checks. Those that are required can be detected automatically based upon the architecture, and dealt with by the run-time system. Retry protocols to combat race, livelock, or deadlock are unnecessary since these conditions are prevented by the system when simple design rules are followed.

Figure 12 illustrates an application where transformations can be performed in parallel. The case of interest is where module P provides input vectors to modules 1 through K. These modules may or may not be performing the same transformations. If they are the same, they may be instanced. In either case, module P likely requires separate interface vectors for each of the parallel modules at the interface. By virtue of the application transformations to be performed, P is in control relative to what module gets scheduled and when. Again, the gray bar in the figure is symbolic, representing IP resource connections through the run-time environment. This is a communications channel where, in this case, P provides a vector input and expects a vector back from each module. This is addressed further below.

Figure 13 illustrates a set of serial transformations. Module S may provide input vectors from a large database, e.g., in a geo-spatial or intelligence application. If data is fed in successive vectors so that each transformation works independently on the next input vector, passing it down the line, then these sequential steps can run effectively concurrently. Again, with each module on a separate processor, vectors are communicated using one or two IP resources for communications.

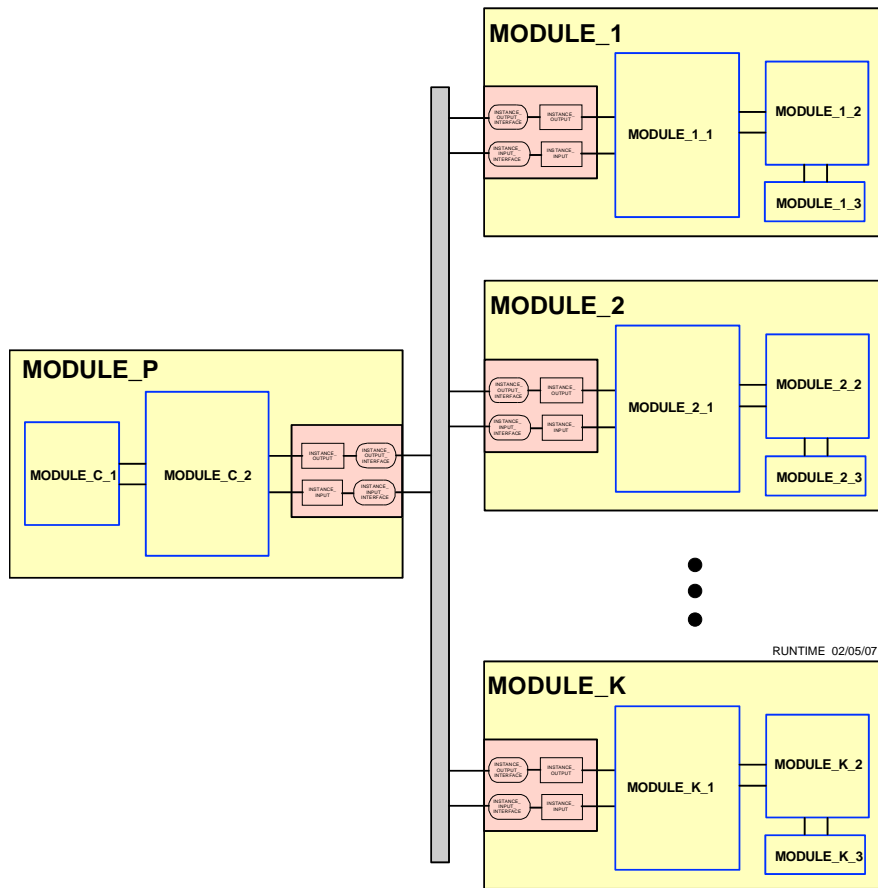


Figure 12. Illustration of parallel transformations.

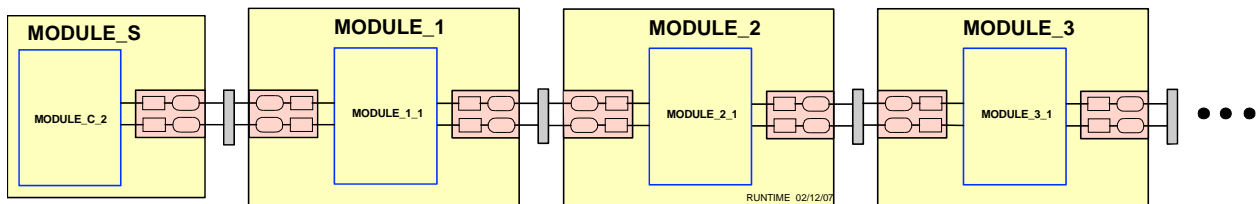


Figure 13. Illustration of serial transformations.

Applications with inherent parallelism can be decomposed into architectures containing combinations of parallel and serial transformations within independent modules that may run concurrently. Independent modules may be assigned to different processors such that communication between them is through IP resource connections. This is addressed in more detail below.

Control Considerations

Independent modules are composed of one or more threads as defined above. Threads within the same independent module need not be independent. Information regarding hierarchies and connectivity of threads and modules is maintained in the development environment and passed to the VisiSoft run-time system. Using this information, the run-time system allocates processors to the processes and resources of independent modules. This ensures that memory transfers are minimized, and that threads that are not independent cannot run concurrently. Software design within an independent module is the same as that on a single processor.

VisiSoft modules can only appear once in a module hierarchy; this includes the top level task. This ensures that only one copy of the resources exist during run-time. In the case of instanced modules, each instance has a separate copy of the resources as well as the processes within the instance. Instances that only share IP resources are independent modules. In the case that a designer wants to share a non-IP resource with more than one other module, other module types, e.g., utilities and libraries may be used (these are beyond the scope of this paper). Independent modules may contain independent modules.

A thread in an independent module may be started by a SCHEDULE statement contained in a process outside that module. Data shared between the independent module and the starting process must be through an IP resource implying a communication channel between processors that ensures data coherency (the scheduled process cannot run until all processes sharing the IP resource have stopped). Information may be exchanged over this channel to control communications as well as transformations.

One of the best ways to understand process scheduling and concurrency is through the use of discrete event simulation using the General Simulation System (GSS). The language facilities used in VisiSoft have evolved from GSS.

Communications Considerations

There are basically two types of communications: *asynchronous* and *synchronous*. When using asynchronous transfers, the information may be passed to a shared resource without regard to the state of the rest of the system. Or, it may be passed based upon knowledge of the state. For example, a shared data element may be set and reset based upon the allowance of change. This is fast and simple, since the state can be set by the sender and reset by the receiver. In real-time control systems where information is exchanged based upon real time clocks or slots, it may have to be synchronized. In this case, one may have to wait for the next “tick” before making a change to a shared resource. This is also handled easily with one or two shared data elements.

In both cases, one must be wary of approaches that prevent other independent threads from running. An example is tight polling loops that waste processor time waiting for a state change. This is avoided by scheduling a thread within some time increment. The language facilities provided by VisiSoft support these architectural needs.

Information transfers among many processors also require routing. Hence, high speed routing technology is used by parallel processor manufacturers today.

Generalized Transformations

It is the programmer's job to develop an architecture that takes maximum advantage of the inherent parallelism of a system. This is best accomplished by focusing on separating concurrent system functions into independent modules. Generalized state space concepts can be used to delineate the architecture. By viewing software functions as generalized transformations on generalized substate vectors, one can determine which transformations can be performed concurrently. This is often determined by the dynamics of the data that must be transformed. If the transformations can be designed to take in data vectors and produce different data vectors, the transformations can be performed in sequence - concurrently. Although this may appear to waste memory, using memory to gain speed is an important design technique.

The Large Memory Model

As indicated in the Introduction, the software field is driven by major changes in hardware design. One of the continuing trends is the increasing size of cache memory space. This is a key property of hardware that can be used to increase processing speed for software. We refer to this as the *large memory model*. Using this model, we are constantly looking to trade increased use of memory for speed. This is accomplished by having copies of data vectors in each processor where they are needed - instead of "saving memory!" Obvious examples are copies of library modules (they are typically memoryless) and their associated data vectors, and copies of databases that remain unchanged. The recommended rule for the foreseeable future is - *use memory to gain speed*.

ARCHITECTURAL RULES FOR ENSURING SPEED

To lay down rules for speed, we must stay close to the machine. Looking back at Figure 3, one is concerned with the overhead encountered when using multiple processors to increase speed. To gain close to 100% effective use of the processors, the application must be embarrassingly parallel, just as running separate tasks on separate computers as in Figure 1. In this case, as separate tasks are spread on more processors, the speed multiplier can increase linearly until the number of processors is greater than the number of independent tasks.

On the other end of the spectrum, if instructions in the task generally depend upon results from the prior instruction, there is no inherent parallelism. In such cases, it should be apparent to the architect that the use of more than one processor will cause the speed multiplier to be less than one. But this will not be apparent to the OS. In practice, only an architect who understands the application's transformations knows if and where parallelism exists.

Minimizing Memory Transfers

To maximize the speed multipliers, one has to minimize memory transfers between processors. This includes instruction transfers (when assigning a thread a processor) as well as data transfers (when assigning memory to support a thread as well as data transfers between threads on different processors). In addition, transfers cost more time as they cross more memory hierarchy boundaries. The fastest transfers are those within the cache closest to the processor.

Even on a single processor, the inability to predict the dynamics of the huge unfolding event space is obvious, and the OS must rely on I/O requests, page faults, etc. Allocation of a large number of processors to threads is a particularly difficult optimization problem, requiring prediction of which threads are going to have the most data exchanges. Accuracy of prediction models increases with the information used to condition the probability statements. Using VisiSoft, a system designer produces architectures that maximize module independence. This architectural information is used to significantly decrease memory transfers, enhancing effective processor utilization. Given this information, processor allocation and assignment is far better handled by the run-time system, making smart OS calls.

The VisiSoft run-time system is provided with a map of threads, similar to that shown in Figure 9. To minimize memory transfers, all of the processes and resources in an independent module are assigned to the allocated processor. Furthermore, unless load balancing is required (discussed later), the module will remain on that processor. Under these circumstances, neither instruction nor data memory within the independent module are ever reassigned (never transferred). The only transfers are between IP resources shared between modules on different processors.

Maximizing Speed

Effective processor utilization, as defined above, depends upon the ratio of internal module processing to communication processing (data transfers between processors). It also depends upon the percentage of time that the processors are busy. This implies a trade-off between processor utilization and overall speed. There are two cases of concern. In the first case, there are more processors than independent modules and these modules are distributed such that overall data transfers are minimized. In this “unsaturated” case, utilization efficiency may be low for some processors while high for others. Since there is ample processor capacity and overall data transfers are minimized, speed multipliers are maximized.

In the second case, there are more independent modules than processors and multiple modules must be assigned to a single processor. In this “saturated” case, unbalanced loading may provide room for increasing speed. The run-time system can use knowledge about the unbalanced conditions to reassign independent modules to different processors. Again, the additional architectural information can be used to migrate all the threads belonging to an independent module to the new processor. This is an area where information may be gathered by the hardware and SOS during run-time to improve the accuracy of run-time system predictions regarding when and where to migrate independent modules to maximize speed.

GETTING CLOSER TO THE MACHINE

By getting closer to the machine we imply that the development environment is designed with a sufficient understanding of the hardware to support maximizing run-time speed. In practice, this implies that software architects must be able to produce designs that take maximum advantage of the inherent parallelism of a system. We must also ensure that the software architecture is mapped effectively onto the SOS parallel processor hardware.

It is currently claimed in the literature that this is a hard problem to solve, even for the most experienced programmers. We see this as a paradigm issue. The OOP paradigm (using OOP and C based languages, e.g., C++, Java, etc.) for building software flies in the face of the industry's most important history lessons, e.g., that of top down design, one-in one-out control structures, methods of typing data, etc. The OOP paradigm with abstractions and data hiding makes it difficult to take advantage of the properties discussed above. Given the proper paradigm, developing software for parallel processors can be easier than for single processors.

Getting close to the machine to improve speed also implies mapping memory directly - What You See Is What You Get (WYSIWYG) - in huge chunks. This applies to single processors as well as parallel processors. Mapping large data vectors so they are shared by large transformations improves understandability as well as speed. This implies moving data in and out of hierarchically organized structures. WYSIWYG memory hierarchies provide huge improvements in software productivity as well as speed.

Understandability

Assembler provided a productivity leap over mnemonic or binary coding. FORTRAN provided a leap over assembler, aiding run-time speed as well as productivity. COBOL was a huge leap for processing data. Because of the ease with which one can organize and access data hierarchically in COBOL, and WYSIWYG mapping into memory, processing large complex data structures is very fast, getting closer to the machine than FORTRAN. Yet COBOL was the easiest for programmers to understand. We see no fixed relation between run-time speed and ease of programming. Historically, languages that are more understandable also run faster.

Comparisons of discrete event simulations show that large systems built using VisiSoft run much faster than the same systems built using C based languages. Yet, the ability to understand huge complex models is relatively easy compared to C based approaches. Experience in modeling also shows that the use of abstractions can slow down run times. This is because most abstractions ignore the inherent independence properties of a system (often to save memory), therefore ignoring the reality of the hardware.

Productivity is a major problem in software that is consistently shielded from view. It is affected by many factors, as described in [21]. One factor that is especially important in parallel processing software is understandability. In any control system, one must be able to identify who has control and how it was transferred. To manage a large set of human or mechanized resources effectively, control is generally distributed hierarchically so that its transfer is easily understood. This provides a clear chain of control.

Experience in software supports this observation. When control is transferred in an unstructured manner, it becomes hard to understand the chain of transfers, and how control arrived at a given point in a complex program. The concept of one-in one-out control structures was one of the major conclusions of the structured programming revolution, see Mills [22]. It ensures that when control is passed to a given element of a program, it is always transferred back when that element completes execution. This does not inhibit an element from passing control further down the line. It merely ensures that control is always passed back up the chain. This contributes to understandability.

An English-like source language improves understandability and reliability. With smart translators, CAD systems, and language facilities that support architectural needs, run-time speeds can be improved concurrently.

Mapping Memory

Taking advantage of the inherent parallelism of a system implies taking advantage of the independence properties of that system. This implies designing transformations on vectors that contain only the information needed by that transformation, and grouping transformations so that the data vectors can be shared directly. To do this, one must design modules containing shared resources at the boundaries, shared resources between internal processes that perform the transformations, and dedicated resources for each process so that independence is maximized. Having done this, data memory is mapped directly by the resources, instruction memory is mapped directly by the processes, and the independence properties can be observed visually by looking at the architectural drawings.

THE PROGRAMMER INTERFACE

Having used this system, one may deduce that developing software for parallel processors with VisiSoft is easier than building single processor systems using conventional approaches. Many programmers can work with the CAD interface to build different parts of an architecture for a large system, and at the same time create independent modules that can take advantage of inherent parallelism in the system. Having flagged these modules at the highest level, the run-time system effectively takes care of the rest. The programmer need not be concerned with the number of processors available. Whether it is one or one thousand, the run-time system will take maximum advantage of what is available.

Behind the scenes, the management system in the development environment tracks all of the process and resource connections and uses this information to build a matrix of process (thread) connectivity. This matrix is effectively diagonalized by following simple architectural design rules that are checked easily by visual inspection of a drawing.

In addition to creating architectures that take maximum advantage of inherent parallelism in a system, there are other factors that can affect the ability to achieve significant increases in speed. One of these is the manner in which communications is handled. A final consideration is ensuring that systems get into production. This is affected by productivity and testing. VisiSoft language facilities have been designed specifically to support the architectural features while improving productivity.

Testing

One of the best ways to design and test complex event driven systems is to use discrete event simulation. The first component of VisiSoft, the General Simulation System (GSS), was devised for this purpose. It can be used easily to test software since modules in the software environment look identical to models in the simulation environment. The architecture and language facilities are virtually identical. Moving modules in and out of GSS greatly simplifies otherwise complex software testing.

PARALLEL PROCESSOR SCHEDULING AND ALLOCATION

The VisiSoft run-time system is somewhat more complex than implied above. It maintains resource coherency by ensuring that processes that share resources at interfaces with modules in other processors wait until the resource is free. With reasonable inherent parallelism, these processes generally represent a miniscule percentage of processing that goes on within an independent module, providing highly efficient use of parallel processors. For the interested reader, reference [20] provides more detail on how this is done.

The run-time system also optimizes processor and memory allocation. It can be interfaced to any SOS parallel processor that supports the required thread to processor assignment and cache controls. Thus the run-time system transforms the knowledge of module independence designed in the development environment into an effective allocation of processor resources. In the case of discrete event simulation, synchronization of simulation clocks across processors is also automated.

SUMMARY

Problems with current approaches to software, namely poor productivity and slower speed, have been concealed by hardware upgrades when computer clock rates followed the Moore's Curve. Now that this era appears to have ended, chip manufacturers are increasing the number of CPUs in a computer to provide more power. The problem faced by software developers is how to take advantage of parallel processors that are improving every year, as newer designs decrease distances and transfer delays between processors. The impediment to capitalizing on parallel processors is the great increase in difficulty faced by programmers using current programming approaches.

This paper describes new paradigms that allow programmers to: (1) develop software architectures using a CAD interface that reveals module independence; and (2) capitalize upon knowledge of the inherent parallelism in a system automatically at run-time. By following relatively simple architectural design rules, checked by visual inspection of drawings, module independence can be maximized. Thus a programmer's knowledge of the inherent parallelism in a system is captured and stored for use at run-time. This information is then used by the run-time system to schedule software processes and allocate CPU and memory resources in a way that maximizes the effective use of parallel processors. The programmer interface is independent of the number of processors available at run-time. Developing software for parallel processors using this paradigm is considered easier than developing complex software for a single processor using conventional approaches. Conversely, the properties of independence, the Separation Principle, and the CAD approach to architectural design are as helpful in improving programmer productivity and run-time performance for a single processor as they are for parallel processor systems.

The essential ingredient of this approach is the Separation Principle. It follows the internal paradigm of advanced chip design, where instruction memory is managed separately from data memory. It is the Separation Principle that provides the ability to draw software architectures with a one-to-one mapping from drawing to code. This, in turn, supports the VisiSoft CAD approach, providing a greatly simplified programmer interface.

REFERENCES

- [1] Armour, Phillip G., *Software: Hard Data*, Comm. of the ACM, Vol. 49. No.9, Sep. 2006.
- [2] Anselmo, Donald, *Why Hasn't Software Productivity Improved?*, Software Summit, Washington, D.C., May 2004.
- [3] Groth, R., *Is the Software Industry's Productivity Declining?*, IEEE Software, Nov/Dec 2004.
- [4] Guth, Robert A., "Battling Google, Microsoft Changes How It Builds Software," The Wall Street Journal, Sept 23, 2005, page 1, column 5.
- [5] Ledgard, Henry F., *The Emperor with No Clothes*, Communications of the ACM, Vol. 44, No. 10, Oct 2001.
- [6] "Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%," Press Release, The Standish Group International Inc., West Yarmouth, MA, 2003.
<http://www.standishgroup.com/press/article.php?id=2>
- [7] "Standish: Project Success Rates Improved Over 10 Years" Software Magazine, January 2004.
<http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish>
- [8] Broy, Manfred, *The "Grand Challenge" in Informatics: Engineering Software-Intensive Systems*, Computer, IEEE Computer Society, Oct. 2006.
- [9] Cusumano, Michael A., "What Road Ahead for Microsoft and Windows?," Communications of the ACM, July 2006, pg 21-23.
- [10] *A Tale of Three Disciplines and a Revolution*, Jesse H. Poore, IEEE Computer Society, Jan 2004.
- [11] Ranum, Marcus J., *SECURITY - The root of the problem*, ACM QUEUE, June 2004.
- [12] Meindl, James D., Keynote Address, 1997 Hot Chips IX Symposium, Stanford Un., Palo Alto, CA.
- [13] Sutter, Herb, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software, Dr. Dobbs's Journal, 30(3), March 2005.
- [14] Sutter, Herb, and J. Larus, Software and the Concurrency Revolution, ACM Queue, vol. 3, no.7, September 2005
- [15] Cave, W., et.al, " The Effects of Parallel Processing Architectures on Discrete Event Simulation," Proceedings: SPIE Defense & Security Symposium, Mar/Apr 2005, Orlando, FL.
- [16] Holland, J.H., "A Universal Computer Capable Of Executing An Arbitrary Number Of Sub-Programs Simultaneously," Proc EJCC, pps 108-113, 1959.
- [17] Frank, S., "Tightly Coupled Multiprocessor System Speeds Memory Access Times," Electronics, pps 164-169, 1984.
- [18] Cave, W.C., A Generalized State-Space Framework for Software, Visual Software International Technical Report, Nov 2007, Spring Lake, NJ.
- [19] Hactel, G.D. et al, *The Sparse Tableau Approach To Network Analysis And Design*, IEEE Transactions on Circuit Theory, Jan 1971, pp 101.
- [20] *High Efficiency, Scalable, Parallel Processing*, DARPA Contract SF022-035, Final Report, Prediction Systems, Inc., Spring Lake, NJ, June 2003.

- [21] Anselmo, Donald and Henry Ledgard, *Measuring Productivity in The Software Industry*, Communications of the ACM, Vol. 46. No.11, Nov 2003.
- [22] Mills, H.D., *Mathematical Foundations of Structured Programming*, Technical Report FSC 72-6012, IBM Federal Systems Division, 1972.