PARTITIONING FOR VIRTUAL MACHINE EFFICIENCY

L. D. Anderson, B. W. Manning, L. J. Slechta
and T. O. Wolff

Sperry Univac Defense Systems Division
St. Paul, Minn. 55165

ABSTRACT

The virtual machine concept, enabling individual users to concurrently access a computer’s resources, has long been accepted by business applications and utilized by software implementors. Current implementations of the virtual machine concept have been achieved almost exclusively through the use of software (e.g., virtual machine monitors). Use of the concept for high performance, application processes has been constrained by the overhead resulting from the software monitor. This includes time required for application process dispatch and communication between the associated virtual machines whenever cooperation is required to meet a system objective.

This paper discusses the advantages of partitioning the application for allocation to virtual machines. The discussion includes a description of interface standards proposed for utilization by both the designer and implementor. A machine architecture is presented that not only overcomes the deficiency in contemporary virtual machine implementations but also simplifies the design and implementation of distributed processing system architectures.

INTRODUCTION

The class of system to which the authors of this paper have turned their attention is “defense” oriented, where systems tend to be single purpose and extremely time critical. In an earlier paper by Wolff [1], it was argued that the imperatives of such systems are fundamentally different from commercial, multiuser facilities — so different that system design must be significantly altered to take them into account. We believe that the machine and system features described later in this paper reflect just such an accounting. Before exploring these design approaches; however, it is appropriate to review very briefly the objectives which helped produce them.

Performance: Those who have worked in various parts of the defense industry quickly realize that performance is almost always a crucial design issue. Performance, manifested in parameters like accuracy, stability, and speed, attracts well deserved attention in that environment where systems exist for the sole purpose of engagement in a life or death struggle. It is the ultimate acceptance criteria.

Flexibility: Defense Systems exists in an environment where ability to readily change and expand would be very advantageous. This need to accept growth and change arises out of corrections to a threat assessment, changes in threat or mission, desire to incorporate new technology, and need to remedy earlier design faults.

Readiness: Defense Systems must be available to perform their intended role at all times, in spite of a fault detection and repair environment which is difficult at the very best.

There are other requirements which may dominate a particular military system's procurement, such as compatibility, interoperability, multilevel security, environmental, and form factor demands; but the problems of performance, flexibility, and readiness are ever-present. The means of achieving these objectives must take into account two other highly influential factors. The first of these is the functionally interdependent nature of computing tasks in a single-purpose system, and the need to provide for their tight coordination. We call this factor intrinsic unity, the monolithic characteristic of dedicated military systems. The second factor is the inherent separateness of computing resources in distributed processing systems. We wish to enjoy the advantages of distributed processing architectures, but must recognize that physical separation creates problems for the operating systems and perhaps new timing and coordination constraints.

Virtual machine concepts seem to hold great promise as a technique for improving both the flexibility and the readiness objectives of military systems. Unfortunately, the implementation of virtual machine features does not come free. The cost is reduced system performance, the ramifications of which are untenable for real-time military systems. This is undoubtedly a major contributing factor to the lack of virtual machine concept usage in military systems. Performance loss in commercial systems is usually not critical when compared to human induced delays and response time.

Later in this paper, it will be shown that the performance penalties which accompany virtual machine operating systems can be all but eliminated through judicious partitioning and the addition of hardware in the computer. Once performance objections are overcome, future military systems can look forward to the benefits inherent in virtual machine philosophies. The benefits of virtual machine utilization are found in the simplicity of change. When the object code contains only references to convenient (virtual) names of computing objects (such as memory cells, channel numbers, register numbers, interrupt types, processors), then it is much easier to make hardware additions, deletions, reconfigurations, and software re-assignments. Virtual addressing has been used in many military systems and is a very small step in the right direction.

One of the biggest obstacles to reconfigurability has been in the communication area. Software modules at present must follow one set of conventions to communicate with a module in the same machine, but a different set of rules if the module is in a different machine. The rules for internal versus external communication are frequently so different that it becomes quite impractical to revise modules in order to experiment with various allocations of software to hardware. If virtual machine philosophy is to produce the benefits it promises, it must eliminate environmental dependencies from communication code. It is fairly obvious that this problem arises whenever the computing environment is partitioned into separate processing components; i.e., a distributed processing system. It may be concluded that while virtual machine philosophy may contribute benefits if performance issues can be overcome, it will still fall short if inter-process communication is not standardized across machine boundaries. Later portions of this paper introduce a set of control operators which achieve transparency for inter-process interaction for virtual resources located either locally or remotely in a distributed computer system. We believe that such a set of operators contribute to Flexibility and Readiness through attainment of the following properties.

- Simplified design, hence lower costs and shorter lead times in system development.
Suitability for change and growth, hence accommodation to the changing mission and technology environment.

Suitability for dynamic reconfiguration, hence the ability to provide fault tolerance and graceful degradation.

The authors of this paper contend that the above properties can be attained along with protection and security safeguards without a performance penalty. In fact, hardware support for the virtual machine concept does more than compensate for performance loss, it actually produces a performance gain of very significant proportions.

**DESIGNER VISIBLE SYSTEM INTERFACES**

The normal design and development cycle for a mission-oriented data processing system is comprised of several phases, involving numerous disciplines and technologies. The phases address concept formulation, system definition, performance prediction, equipment selection, equipment and software design, prototype and product build, system integration, production, and testing.

The conceptual and definition activities have traditionally followed a sequence similar to that shown in Figure 1. Application of very high-level system sizing estimates (e.g., total storage, total throughput, MTBF requirements, etc.) resulted in the selection of hardware and control software architectures which were then combined to form a candidate system architecture. Specific functional requirements were applied much later in the cycle to populate the candidate system architecture. This general design methodology has been justified for contemporary systems because of the high cost of hardware and of operating system technology.

**Figure 1. Rejected System Design Methodology**

The design methodology of Figure 1 was ultimately rejected in a study reported by Freeman and Christiansen [2]. They noted that a decision to change a system's physical requirements and obtain the benefits available through the use of new technology could not be achieved without detailed knowledge of the current hardware and software implementation techniques. In effect, the full impact of change, both cost and performance, could not be seen until the entire system was redesigned, an inherently costly exercise. As a result, they recommended a new set of design guidelines which were traceable to system requirements. It is the logical partitions or functions which are key system ingredients or system building blocks. The guidelines establish a chronological sequence of design activities. The guidelines appear in Figure 2a as a subset of what the authors believe to be a broader, more inclusive representation of the design process as shown in Figure 2b. The major difference between the rejected and recommended approaches is that the recommended approach leaves selection of implementation media (hardware, software, or hardware) until the final stages of the design process. This avoids the potential ineffective mapping of logical solutions onto physical architectures.

The broader but compatible view of system design is illustrated in Figure 2b. System development begins with the definition of requirements; i.e., the physical environment with which the system is to interact and the impact desired. The desired impact is obtained by partitioning the "requirements" into logical (functional) descriptions. The partitions are later mapped onto physical representations; i.e., software and/or hardware.

The complexity of partitioning the requirements into functions has caused system designers to establish system structuring rules — rules which define the relationships intended between individual partitions [3, 4, 5]. Earlier

The authors of this paper contend that the criteria used to interface logical partitions should be functionally equivalent to machine primitives (hence, criteria) for interfacing the physical representations of these partitions. The key system ingredients or system building blocks are logical partitions which are traceable to system requirements. It is the logical partitions or functions which ultimately drive the data processing equipment. Three factors constitute an adequate description of a logical partition.

The first factor is a description of the functional logic or transformation "f". The second factor is the data space "S" which can be accessed. The
third factor, which uniquely determines the results of having executed a function, are the initial values or state of the data base accessible by the function. The set of resources identified by this triple $(S, f, s)$ has been formally defined by Horning and Randell [6] as the process. This set of resources is also said to be a virtual machine description. A process requires a real machine as defined by these resources. The allocation of virtual machines to real machines, the interface (or communication) between virtual machines and process execution sequence control are all operations which the authors believe are necessarily included as primitives of the real machine.

Similar observations have been made in a paper by Lauer and Needham [7]. In their paper they reached the conclusion that effective use of the operating system properties, process, synchronization, and interprocess communication, were more dependent upon the machine architecture selected than on the basic operating system structures utilized; e.g., message-oriented or procedure-oriented. In other words, proper partitioning and implementation of hardware capabilities are more effective than judicious use of operating system structures provided by software.

A system is defined as a set of concurrently executing processes cooperating through resource utilization relationships and authorities (system structure). Each process is defined to independently drive digital processing equipment. The elemental resources, called entities, have been grouped into four types: virtual machines, storage, communication and synchronization. Entity naming conventions are all virtual at the system level. This allocation of entities is based on the structuring rules outlined earlier. Characteristics of the entities are referred to as attributes; e.g., the virtual machine type, the length of a memory segment, the access permission associated with a memory segment, the priority of a process, etc.

The approach to hardware development has centered around the identification of the "necessary and sufficient" set of primitives required for process or virtual machine interface. The set consists of subsets, each of which may be selected for implementation after cost and/or performance tradeoffs. The original set of primitives [8] was defined and documented using ISP notation; this initial set has been refined as reflected in Figure 3. The set has been divided into two parts: system structuring primitives and progress management primitives.

The system structuring primitives are shown as a symmetrical hierarchy of operations which are concatenated to form the individual primitives. For example, the top level of the hierarchy provides the means to selectively enable or disable any or all of the primitive operations at lower levels; the second level provides for the definition and subsequent release of resources and attributes required to support process execution. The lowest level of the hierarchy defines the fundamental units required for the definition of system structure.

**PROGRESS MANAGEMENT PRIMITIVES**

The primitives shown in Figure 3 have been implemented on a target machine using a laboratory software test vehicle [8]. However, the subset of primitives selected for target hardware implementation must be based upon cost and/or performance tradeoffs. For systems where a static system structure is sufficient, the system structuring primitives can be implemented.
offline using a software system generator. If the system requires a dynamic structure (as illustrated by a multiterm time share system), it may be desirable to implement the system structuring primitives online using either hardware or software. In either case, the target machine must be capable of enforcing the structure so defined. For the hardware subsequently defined in this paper, the system structuring primitives are assumed to be offline. Only the progress management primitives are included in the hardware architecture.

The system structuring primitives and their purpose (function) are defined in Table 1. In actual implementation it has been assumed that all processes form a tree structure. The enable/disable primitives permit parent processes to optionally pass the authority for resource utilization on to their descendants. If properly enabled, a named descendant may define/release an entity, entity sharing, or entity attributes. In effect this permits all resource usage conflicts to be identified and potentially resolved by a common parent (process).

| TABLE 1. SYSTEM STRUCTURING PRIMITIVE DEFINITIONS |
|-------------|-------------------------------------------------|
| PRIMITIVE   | FUNCTION DEFINITION                              |
| ENABLE/DISABLE ENTITY DEFINITION | ENABLES OR DISABLES STRUCTURED USE OF SYSTEM RESOURCES. |
| DEFINE/RELEASE ENTITY SHARING | REQUESTS OR RELINQUISHES CONTROL OF A SYSTEM RESOURCE. |
| ENABLE/DISABLE ENTITY SHARING | ENABLE OR DISABLES THE ESTABLISHMENT OF RESOURCE SHARING. |
| DEFINE/RELEASE ENTITY SHARING | REQUESTS OR RELINQUISHES RESOURCE SHARING. |
| ENABLE/DISABLE ATTRIBUTE CHANGE | ENABLES OR DISABLES PERMISSION FOR A PROCESS TO CHANGE CHARACTERISTICS OF RESOURCES (E.G., SEGMENT LENGTH). |
| DEFINE ATTRIBUTE | REQUEST CHANGE TO RESOURCE CHARACTERISTICS. |

The remainder of the primitives illustrated in Figure 3 are called progress management primitives. Progress management primitives exist to enable application strategy to be achieved through process cooperation while retaining the established system structure. The logic required to implement progress management operations is referred to as the kernel. Two distinct categories of logic exist. Primitives available for invocation utilizing the macro-level instruction repertoire are called explicit primitives. Operations enabling the hardware to recognize the concept of the process but which cannot be invoked through macro-level instructions are referred to as implicit primitives. Both implicit and explicit operations are described in subsequent paragraphs.

Implicit Primitives

DISPATCH

Each active process is assigned a priority, which is used by the kernel to sequence processes eligible for dispatch on the processor. Several processes may have the same priority, in which case, sequencing is First In/First Out (FIFO). Dispatch involves the assignment of the processing element to the highest priority process on the dispatch list.

EVENT HANDLING

Event is the term used to denote an instance of cooperation, hence the need for synchronization between two asynchronously executing processes. The act of synchronization is not visible to the macro level process directly. For example, a process which performs a WAIT operation on a semaphore will ultimately execute the next instruction regardless of whether it is temporarily suspended or continues execution. Events observed as interrupts on conventional machines will create kernel-to-process messages using the standard logical destination mechanism (see explicit primitives). Thus, events are not explicitly defined using progress management primitives. All synchronizing operations are performed on semaphores by means of the WAIT and SIGNAL primitives. Each instance of synchronization is represented by the assignment of a semaphore to each event (called event registration) coupled with the asynchronous execution of the WAIT and SIGNAL operations by different processes.

Explicit Primitives

(i) PROCESS MANAGEMENT

BLOCK — Causes either the specified process (if any) or the invoking process to be removed from the dispatch list. It remains ineligible for dispatch until a RELEASE operation is performed.

RELEASE — Causes the named process to be made eligible for dispatch, if blocked. A RELEASE operation will be retained (as a SIGNAL to an implicit private semaphore) pending execution of the next BLOCK operation on the named process.

PRIORITY CHANGE — Causes a change of process priority for the named process (if any) or the initiating process. Priority may only vary within the limits specified in the state vector (currently system generation parameters).

(ii) STORAGE MANAGEMENT

SHARE — Causes a specified memory page (or pages) to be shared between the invoking process and a named process. This operation permits dynamic memory sharing between processes, if established by the system structure. (Static sharing can also be established during system generation.)

(iii) COMMUNICATION MANAGEMENT

SEND AND RECEIVE — All communication is performed by reference to a logical destination, as opposed to conventional real or virtual computer channels. This mechanism allows a mix of destinations, located on shared busses and point-to-point channels, to be transparent to the software. Logical device transparency is particularly important in a tactical situation, where functional allocation frequently changes due to hardware reconfiguration. Reallocation can be achieved in this environment by changing the logical destination tables in each computer. The software remains unchanged even when functions are reassigned to different computers.

Communication is in the form of process-to-process messages. The SEND and RECEIVE primitives are invoked regardless of the location of producer and consumer processes.

(iv) SYNCHRONIZATION MANAGEMENT

Synchronization is achieved using the semaphore, a data structure composed of a counter and a queue. The semaphore operations are the F (WAIT) and V (SIGNAL) operations originally proposed by Dijkstra [9]. The WAIT operation decrements the counter and, if the counter becomes negative, adds the associated process to the semaphore queue. The SIGNAL operation increments the counter and, if the counter remains nonnegative (includes zero), removes a process from the semaphore queue for dispatch.

In distributed processing environments, a special form of the semaphore is required to handle synchronization between processes on different computers. This semaphore, called a multi-copy global semaphore, responds to SIGNAL and WAIT primitives, as though it were a local semaphore (providing hardware transparency). However, this global semaphore requires special implementation techniques for queueing and updating.

HARDWARE IMPLICATIONS

The merits of using virtual machine concepts (i.e., processes) as the basis of system design have been argued in the previous sections. However, the implementation of process structures on conventional computer architec-
tures poses performance problems because of the major differences between the structure of real machines compared to the structure of virtual machines. Traditionally, this gulf has been bridged by a large layer of software called the Virtual Machine Monitor (VMM). However, the overhead of this software layer has been so excessive that implementation in high performance systems has been impractical to include. Before the advantages of the Virtual Machine (VM) concepts can be realized in a real-time environment, the VMM overhead must be significantly reduced.

One way to reduce the VMM overhead is to make the structure of the real machine more closely resemble the structure of the virtual machine. If in fact the real machine and virtual machine structures and interfaces were identical, no software layer would be required at all to make the two compatible. This obvious solution assumes that a single set of conventions, practical for use in both system synthesis and machine structure, has been defined for hardware implementation. The user visible interface, defined in the previous section, is our recommended definition.

A further important means of increasing performance is to improve hardware speed in process to process, and process to machine interfaces. Additional hardware invested in frequently used VMM operations more than pays for itself in increased useful throughput. Most military systems consist of only a fixed set of processes; i.e., there is very little dynamic creation of new tasks. For this reason we have chosen to implement in hardware only the implicit and the process management set of explicit primitives. The more infrequently used system structuring primitives are left for runtime software implementation or for system generation software.

In order to gain the maximum performance for kernel primitives, a modular addition called the Control Unit (CU) is introduced into the traditional VonNeumann architecture, as shown in Figure 4. The CU is a hardware version of the kernel which, when added to a conventional computer, results in a process controlled computer whose structure is identical to the structure of the VM's (i.e., processes) that will execute on it. The CU approach to building process-controlled computers has definite advantages over other approaches. First, when the kernel is implemented in an independent module, that module may be hardware optimized to perform efficiently. The kernel functions execute several times more rapidly than they would in the CPU. Secondly, placing the kernel in a separate module allows the computer designer to introduce overlap or parallels in between the kernel functions and application functions in the CPU. This further increases the task throughput capacity of the CPU. Finally, a third advantage of the CU approach is that it allows the kernel to be introduced without major redesign of an existing CPU. Thus a process-controlled computer can be developed from an existing computer, for a modest investment.

The CU consists of a microprogrammed controller for execution of the kernel primitives, high-speed local storage for the kernel data structures, and special hardware logic to boost execution speed of the kernel primitives. The kernel data structures are retained in local CU storage and include the CPU Process Dispatch List, the set of Process State Vectors, the Logical Destination Table, the I/O Port Table and the Semaphore Data Structures. Retention of the kernel data structures in the CU improves performance because there is no contention with main memory users and local storage, faster than main memory, can be used. Special execution logic in the CU includes state switch hardware to facilitate rapid process switching in the CPU, virtual memory address translation logic to provide hardware enforcement of process namespace separation [10], and custom logic to manipulate the global semaphore data structures at high speed.

As shown in Figure 5, the CU has control paths to the CPU and the Input/Output Controller (IOC) as well as a data path to main memory. When the CPU encounters an explicit primitive in the process instruction stream, it sends a control packet to the CU identifying the primitive and its parameters. The CPU then enters a PAUSE state. The CU will receive the control packet and decode the primitive operation. Depending on the primitive, one of three possible control cycles will be followed.

1. If the primitive is one which requires only a brief amount of CPU time, the primitive will be executed by the CU and the CPU will then be restarted at the next instruction of the process.

2. If the primitive is one which calls for suspension of the executing process, the CU immediately "saves" the CPU state, selects the highest priority process from its CPU dispatch queue, initializes the CPU with the new process parameters, and restarts the CPU. The CU then completes execution of the primitive which caused the suspension.

3. If the primitive is one which requires a longer amount of CPU time, the CU immediately "saves" the CPU state, selects the next process, initializes and restarts the CPU. The primitive is then executed by the CPU, after which the suspended process is re-entered into the dispatch queue at its current priority.

![Figure 4. Control Unit (CU)](image)

![Figure 5. Control Unit Interfaces](image)
can be accessed and/or changed only by the CU in response to the execution of primitive operations on the CPU. The process state vectors themselves are part of the application structuring procedure and are generated offline during system generation. A process state vector contains the following information:

- **Current State**: Target machine general registers, program counter, status register, etc.
- **Authorities**: Logical destination usage rights, semaphore usage rights, and allowable priority range.
- **Event Registrations**: Hardware faults, software faults, and clock message.
- **Namespace Description**: Page descriptors.
- **Structural Relationships**: Link to parent process.

The fact that the process is entirely defined by the process state vector allows a process to be dispatched on the CPU in a very efficient manner. The contents of a register located in the CU called the Process Identifier (PID) is used to specify which process of the allowable set of processes is executing on the CPU. This PID can be thought of as an index that selects one of “n” process states. It specifies which set of general registers, program counter, etc., are to be used by the CPU, which set of page descriptors are to be used by the Virtual Address Translator [11, 12, 13, 14], which set of authorities are to be used by the CU in verification of access rights during primitive execution, and which set of event registrations are to be used during the processing of events. To change the process running on the CPU, the only thing that is required is to change the contents of the PID and to perform a small amount of housekeeping in the CPU. The PID mechanism not only allows process state changes to be performed in a few microseconds, it also effectively enforces hardware separation between processes. Since the process state vector defines the entire set of machine resources and access rights available to its process and since the PID reflects only the relevant state vector, there is no way that one process can interact (i.e., interfere) with another process unless allowed to by the parameters in its process state vector. As a result, the process to process interfaces are hardware enforced. This insures reliable and predictable operation of a system using process controlled computers.

**SUMMARY**

The object of this paper has been to establish the relationship between application partitioning and hardware partitioning and assess the impact on system performance. It has been shown that efficient, high-performance systems are easier to construct when the hardware partitioning most closely resembles the application partitioning. A solution has been presented which establishes a hardware structure directly related to the logical system structure, produced by means of a suggested system design methodology. The authors believe that this approach satisfies the performance, flexibility and readiness objectives of future defense systems.

**REFERENCES**