TR 00.2757
May 21, 1976

An Architecture for an Extended Machine With Protected Addressing

George Radin
Peter R. Schneider

IBM Technical Report
Poughkeepsie Laboratory
An Architecture for an Extended Machine
With Protected Addressing

by

George Radin
Peter R. Schneider

ABSTRACT
An architecture is described for a machine in which addresses are
protected, in which many conventional control program functions are
in the machine, and in which facilities are defined for extending
these functions by software. The more novel aspects of the
architecture are evaluated from both their engineering implementation
and programming usability characteristics.
INTRODUCTION

This paper describes (generally at an overview level) and evaluates some results of an exploratory system design that has some unusual characteristics. The most interesting of these are:

1. Many functions conventionally implemented in a software control program are defined as being beneath the machine interface, i.e., implemented in hardware/microcode. In particular, the management of physical resources (CPUs, storage), the distribution of these resources to logical contenders (processes, data segments), and the creation, destruction, and ownership structure of these logical contenders are inside the machine.

2. The machine interface defined by the architecture is asserted to be complete, not only for the correct functional specification of all aspects of a program, but for all pragmatic aspects as well. Thus, at the interface appear such concepts as:

   Response requirements of processes at different levels of machine availability
   Damage reports when the machine fails
   Start-up, checkpoint, warm-start definitions
   Data access time requirements

3. This interface, although encompassing a wide spectrum of services, is in fact a single architecture. A uniformity exists across these facilities of, for instance:

   Syntax
   Addressing
   Control
   Pragmatic specifications
   Damage reports
   Error handling

4. The machine is an instance of a capability architecture; that is, the generation of names (logical addresses) is protected; names include descriptions of the capabilities allowed the user of the name, and the possession by a user of a name constitutes his authority to access the object named, within the constraints of the capabilities. Thus, the machine attempts to provide a high level of security and integrity without the performance penalties inherent in checking authorization to each access of an object.

5. Operands at the machine interface are not generally defined as strings of bits (with the exception of computational data). Program modules, processes, queues, resource tables, etc., are encapsulated (that is, their representation is not architected and, in fact, not observable by programs at the interface). This provides potential cost/performance flexibility on different models, allows some changes later without
impacting programs, and reduces the synchronizing requirements between programs and the machine. It results, however, in an interface that has a large number of instructions since access and change cannot be accomplished by a single set of load-and-store instructions.

6. The machine interface is chosen at a level that can achieve acceptable performance and cost without requiring any compilation below it. It is not, however, the highest such interface. For instance, scientific subroutines, data-base managers, and network managers, although invoked interpretively by application programs, are programmed as software on the machine. Although the choice is often quite subjective, application-oriented decisions are generally left to software. Thus, sessions and users are not made part of the machines, nor are facilities that establish installation strategies such as accounting, job scheduling, or recovery.

This paper does not dwell on input/output, networking, maintenance, emulation, or computational instructions since the architecture offers relatively little that is unusual in these areas. Instead, it concentrates on the more innovative areas of addressing, control, program linkage, management queueing, etc.

There is no reasonable linear organization of the material in this paper, i.e., one in which terms are not used until they are defined. The organization chosen discusses the statics of the system first (system objects and how they are addressed), then programs and their execution, and finally the dynamic aspects (e.g., processes and resource management). This choice results in processes being mentioned before they are defined. We assume that readers have a reasonable intuition about a process (or task), which should prove quite adequate.
SYSTEM OBJECTS

Definition

The machine is defined architecturally as a collection of system objects that reside in storage and are uniquely named. The concept of storage is treated only as a resource—it is not addressable. There is no notion of where in storage an object is, only of how much storage it consumes. The architecture does not preclude having multiple machines share the same physical hardware or one machine geographically distributed.

Each system object has a unique name, called an H (for handle). This name is assigned when the object is created and is forever uniquely associated with that object. An H, once assigned, is never reassigned to another object, nor is an object ever given another H. Thus, there are no name scoping or aliasing considerations necessary in this architecture.

Each of the many system object types play a unique functional role in the architecture, and each have a set of precisely defined operations that are valid for the object. The complete set of systems object types is:

<table>
<thead>
<tr>
<th>Affinity Set</th>
<th>Linkage Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Linkage Vector Table</td>
</tr>
<tr>
<td>Counter Set</td>
<td>Machine Control Table</td>
</tr>
<tr>
<td>Data Object</td>
<td>Module Set</td>
</tr>
<tr>
<td>Encryption Key Set</td>
<td>Process</td>
</tr>
<tr>
<td>Emulation State Object</td>
<td>Process Model</td>
</tr>
<tr>
<td>Emulation Storage Object</td>
<td>Queue</td>
</tr>
<tr>
<td>Event Specification</td>
<td>Resource Pool</td>
</tr>
<tr>
<td>Gate Set</td>
<td>Scheduling Table</td>
</tr>
<tr>
<td>I/O Unit Table</td>
<td>Switch Set</td>
</tr>
</tbody>
</table>

For each object type, subcomponents are defined with further constraints on the operations valid for the component. For example, the system object type Queue has components called Messages. Instructions exist to _ENQUEUE_/DEQUEUE_ messages from an addressed Queue object; instructions also exist to _PURGE_ MESSAGE for a specified Message component within a Queue.

One component found in several system object types is a Data Space, which is simply a contiguous string of bytes on which the normal computational operations can be performed. All other components of system objects are encapsulated; that is, their mapping onto a byte string is not defined or observable. Thus, special instructions are required to store and retrieve information to and from them. The storage consumed by a system object is, therefore, greater than the bytes in its Data Spaces. This delta is not defined architecturally, but is left as a model-dependent parameter.

Most system objects are created explicitly by the machine instruction CREATE. Some, however, are created implicitly as a result of other system objects reaching prescribed states (e.g., a Process may be created when a message is placed on a Queue). A few system objects are created with the initialization of the machine (e.g., Machine Control Tables).
After creation, components of a system object may be accessed by instructions that have a proper addressability, until the object is destroyed. Any reference attempted after destruction results in the termination of the instruction and the raising of an exception.

Each description of a system object includes a definition of which components, or parts of components, are guaranteed consistent during execution of an instruction. Thus, the programmer does not have to synchronize accesses to these components in a multiprocess environment by using explicit locks. The definition of each instruction in the architecture guarantees that components are properly seized, and the "architected" order of seizing guarantees that the machine does not deadlock (see "Synchronization").

Ownership of System Objects

The architecture defines an ownership structure for system objects. This structure is used for defining the scope of certain operations, for listening to actions, for distributing resources, and for implicit destruction of objects when their owners are destroyed.

A system object type, called Context, is an owner of system objects. Ownership is the only function of a Context system object and, conversely, only Contexts can own other system objects.

The ownership structure is a single-rooted tree. All nodes in the tree that are not leaves must be contexts. The root of the tree is a context called the Master Context. This object is created when the machine is initialized. It is said to own itself and is, therefore, the only allowable exception to the tree structure.

Ownership of an object can be explicitly moved from one place in the tree to another, as can entire subtrees. This is accomplished by the CHANGE OWNERSHIP instruction. When executing this instruction, the machine guarantees that the tree structure is maintained. When a context is destroyed, the entire subtree of which it is the root node is also destroyed.

Recovery of System Objects

To facilitate recovery in a system in which storage is so transparently managed, functions are provided to handle both localized failures and major system crashes.

To handle recovery from a programming error, some system objects may be explicitly checkpointed. This means that, logically, a copy is made of the object at the time the checkpoint is taken, and subsequent changes to the object do not affect the checkpointed version. Instructions are provided to fetch component information from the checkpointed version, thereby allowing selective, programmed recovery of individual objects to a previously known, valid state. Since it is assumed that checkpoints occur frequently and recovery occurs infrequently, an implementation would not copy the objects at the time of checkpoint. Instead, separate copies of parts of the objects would be generated when they are first modified. This speeds up the checkpoint function but slows down the recovery function since copies of parts of the object could be scattered throughout the implementation's real storage.
To handle recovery from a major system crash, it is possible to restore storage to a previously specified state. A checkpoint copy of a system object may be pushed to a nonvolatile, secure storage, making it impervious to almost all forms of internal system failure. Restrictions are placed on pushed objects to ensure that:

1. The set of pushed objects constitutes a valid context ownership structure.

2. The set of pushed objects includes only those objects representing a static state for the machine (i.e., Data Objects, Module Sets, etc., are allowed but active Processes, Event Specifications, etc., are not allowed).

Recovery after a system crash causes the machine storage to be brought up containing only the pushed objects. Reactivation of the system is then accomplished by programming.

Recovery from an externally caused disaster (e.g., fire) is to be handled by the normal procedures of dumping critical information to an external media that should be held in a remote location.
ADDRESSING

We have seen, so far, that a machine state consists of a set of system objects residing in storage, each having a unique name (H). These objects are addressed by means of a new data type, called a pointer. To address an object in an instruction, a pointer is loaded into a pointer register. This register is then named in the B-field of a System/370-like B, X, D instruction operand.

When an object is created, a pointer is generated by the machine (with full authority) and loaded into a pointer register. Thus, the owner now has a pointer that will allow him to destroy the object, change its ownership, or extract information from the encapsulated components. Henceforth, upon command by software, the machine will copy a pointer from a pointer register to another pointer register (i.e., System/370-like Load Address), store a pointer from a pointer register into a system object (e.g., into a Data Space), move a pointer from one system object to another (e.g., Store-to-Store move), or reduce the authority of a pointer in a pointer register. The machine tags pointers when storing them in Data Spaces; loads bytes into a pointer register only if they have been tagged; and turns off the tag if a pointer is modified by a computational instruction. It never increases the authority of a pointer.

Thus, it is impossible, by programming, to obtain addressability to a system object by asserting that a byte string is a pointer, or to enhance authority.

There are seven different types of pointers each having a 16-byte length. The first field, the T-field, consists of four bits that identify the pointer type. The next four bits are called the A-field (Authority); they control the capability given by this pointer. The meaning of the A-bits varies with different pointer types. They are set to all 1s at object creation time. Henceforth, each bit may be independently set to 0 by the LIMIT AUTHORITY instruction, but once it becomes 0, an A-bit can never be reset to 1.

Examples of A-bit settings are:

A₀ in a Data Space pointer must be 1 if a pointer can be written into this Data Space.

A₂ in a Data Space pointer must be 1 if a pointer can be read from this Data Space.

(With these two bits, a programmer can further control the passing of addressability among programs.)

A₁ and A₃ similarly control the writing and reading of data into a Data Space and out of it.

A₁ in a system object pointer is for authority to destroy.

A₀ in a system object pointer allows the holder to listen for certain actions in the object (e.g., listen for attempts to destroy the object, or to write into it.)
The second byte in a pointer is the Function Code (FC). If the pointer is an extended-type pointer (i.e., to one of many system objects such as Process, Context, Queue, etc.), this field defines the system object type. If the pointer is (directly or indirectly) to a Data Space, this byte is used by programming to define software objects as extensions to those defined in the machine (e.g., User, Data Set, Job, Session). Thus, operations on software objects can be protected in the same manner as machine objects.

In all pointers to system objects (or components), the next field is a five-byte H. The remainder of the 16-byte pointer varies with the seven different pointer types:

1. **Empty Pointer** (All 0s)

   This is an invalid pointer and can only be loaded or stored. It is used, for instance, in partially filled control blocks.

2. **Token Pointer Fields**: T, A, FC, Time-of-day Clock Value, Z.

   A token does not point to an object. It is a protected, guaranteed unique password. The Authority fields allow software to set the FC- and Z-fields with software object identifiers; the TOD Clock Value is always unique.

3. **Null Pointer**

   Fields: T, A, FC, all 0s.

   A null pointer is usable in several machine instructions to specify default operations. It is generated by a GENERATE NULL instruction.


   These are the most frequently used pointers. They point to a contiguous string of bytes in the Data Space of the object named H. LB and UB are each 32-bit fields defining the lower- and-upper bounds of the string. The string can be collapsed from either end by the instructions:

   LOAD POINTER WITH ADDRESS (like System/370 Load Address) and DEVELOP SUBSET

   However, the string can never be extended. Thus range of addressability, as well as authority, can be subset. When an effective address is computed by adding an index register and displacement to LB, the result is checked to ensure that it still falls between LB and UB.

   The FC-field in a direct data space pointer is not "architected"; its setting can be controlled by software when the A-field gives the programmer this authority. A likely use of this field is describing the data type being pointed to, so that software can check argument/parameter consistency at subroutine linkage.
5. **Indirect Data-space Pointer**

Fields: T, A, FC, SW, H, LB, UB.

A direct data pointer cannot be given to a user if addressability might be withdrawn at a later time. Thus, a system object type, called Switch Set, is defined in the architecture to support indirect addressing. A Switch Set is an object that contains a vector of direct Data Space pointers called Switches.

An indirect Data-space pointer may be used anywhere instead of a direct Data-space pointer. When used, H names the Switch Set object, and SW names the Switch within that Switch Set. The Data-space pointer in the Switch is subset by adding the LB to the Lower Bound, adding the UB to the Lower Bound, and checking that the latter sum does not exceed the Upper Bound. The resulting direct Data-space pointer is then used as the required operand in the instruction.

At any instant of time, a Switch may be used by only one Process (to be defined later). This further restricts passing addressability. Also, the contents of a Switch can be modified only by the Process that is using the Switch, or by a master process that has suspended it. These restrictions allow an implementation to follow the indirection only once (loading the effective pointer into the register) without requiring inter-Process broadcasting when the Switch is modified.

6. **Extended System Object Pointer**

Fields: T, A, FC, H, D, Z.

An ESO pointer is provided as a primitive to permit programming of additional system objects; it functions on top of the machine architecture. The instruction DEVELOP ESO POINTER takes a pointer to a Data Space and develops an ESO pointer in which H, D (displacement) points to a field in the Data Space containing two contiguous pointers:

- **Password Pointer, User Pointer**

The instruction, LOAD ESO USER POINTER, names one pointer register as the target of the instruction, one pointer register containing an ESO pointer, and a third pointer register containing a password pointer (generally a token). If the password pointer exactly matches the password pointer pointed to by the ESO pointer, the user pointer is loaded into the target.

It can be seen that a programmer can freely pass out ESO pointers, or store them in control blocks that are available to others. But only the extended system object manager, who has the password, can extract the user pointer.

For instance, suppose the data management component that establishes authority (e.g., OPEN) decides to grant user A authority to access data set B. This component would develop an ESO pointer to a Data Space containing:
Password Pointer for User A
Control Block Pointer for Data Set B

User A would be given the ESO pointer, which he can store freely. But he cannot use the password pointer, and, unless he already has this pointer, he cannot read the control block pointer. At any time, any user can issue a request to data management to access the data set (e.g., READ or WRITE), passing the ESO pointer as a parameter. The data management component would use the password of the requester to load the control block pointer. Thus, only if the requester was the same user to whom authority was originally granted (i.e., user A) would the LOAD instruction work.

7. **Extended-type Pointer**

Fields: T, A, FC, H, (Object Dependent Fields)

These are used to address system objects and also to provide special pointers to Data Spaces.

The system object/pointer model just described is the result of many iterations required by recognition of serious problems in architecture, in programming, and in engineering. Its major advantages are:

1. The system can accommodate many users, sharing some things by non-nested subsets, and ensuring privacy and integrity when desired. Moreover, it can accomplish this without the run-time overhead of an active manager program, which guards each shareable object and determines eligibility. Checking is factored back to the granting of addressability because the architecture ensures that this addressability is protected.

2. It is a system in which user programs remain valid across many significant changes in system implementation. Storage devices, level of multiprocessing, representation of systems objects, dispatching algorithms, and software control blocks can all change without invalidating applications.

The problems that caused the changes (and which are still present in some degree) are primarily of three types:

1. An overhead associated with this architecture, even when its advantages are not needed. Addresses (pointers) are large, and loading them into registers is slower than loading a four-byte address into a general-purpose register. Tagging memory to protect pointers requires several bits per 16 bytes as well as checking logic. Checking, which must be performed at effective address calculation time, adds several machine cycles to every instruction. These penalties must be weighed against the performance advantages gained by eliminating the need to check authorization interpretively.

2. High-level facilities generally have a built-in risk. Namely, their advantages are only real when they exactly match the users' requirements. When they fail to match, the overhead of programming around them is high.
For instance, this machine architecture provides Process-local Switches to guard against the passing of addressability to unauthorized users. One use of this facility would be by a Data Base Management component. A Switch would be established at OPEN time, and subsequently, the authorized user Process could simply address records via the Switch. The Process could not embed the Switch pointer in a message to another user's Process, thus violating the Data Base Manager's control.

However, suppose the application required the use of the machine's parallel processing capability to create several cooperating Processes in one job. Now, an OPEN does not make the data set available to the job, but only to the one Process. Switch pointers cannot be passed as parameters between processes. Another software address-controlling facility needs to be built on top of the architecture, and the Switches may not help; they may even get in the way.

Examples abound in such a high-level architecture where a miss is as good as a mile.

3. Finally, as can be seen by the several "bumps" on top of pointer addressing in this architecture (e.g., Switches, protection against writing pointers), the prospect of restricting authority checking to address-granting time, and basing the integrity of the entire system on it, is not one that leaves many people relaxed. What if the machine malfunctions, and a critical address gets out? How can the machine ever be trusted again? Is it even possible to change all addresses without totally reinitializing the entire data base?

In summary, a capability architecture has some appealing characteristics and affords many system simplifications. But it remains an open question whether a real online system would rely heavily on such protection facilities for its security and integrity.
PROGRAMS AND LINKAGE

Modules

The unit of program that is analogous to a procedure in ALGOL or PL/I, or a subroutine in FORTRAN, is called a Module. A Module is encapsulated and stored in a Module Set system object.

Because this encapsulation prevents users from modifying programs while they are being executed, the machine is free to optimize its execution within the semantic and synchronization constraints of the architecture. In fact, since Modules are explicitly created by the EXTEND MODULE SET instruction, early versions of the architecture were defined to allow this instruction to perform many of the sophisticated optimization techniques of a compiler. Concurrently, with this potential optimization in mind, the instruction set, was placed at a rather high level, incorporating, for instance, generic operators, array operators, and mixed-mode arithmetic.

As the evaluation of the architecture progressed, it became clear that any significant optimization by an implementation conflicted with the carefully defined synchronization constraints in the architecture. The result was a continuous erosion of the optimization capability of the EXTEND MODULE SET instruction. Because this was to be a machine, not a programming language, the need for the state of the machine to be precisely predictable at any possible time of observation was critical. Debugging could never proceed at a lower level, and performance must be acceptable for everyone, including those who needed to code very tight loops. As the capability of this compilation facility was reduced, the level of the instruction set necessarily was lowered, finally arriving at a computational set roughly similar to System/370.

The final version of EXTEND MODULE SET does nothing but copy a program from a Data Space to a Module Set. It does not even check syntax. However, the encapsulation aspect remains, allowing the machine to do some lookahead and parallel execution.

Debugging

A Module, once created, can be debugged by using a set of special instructions. The debugger and debugged programs execute in two different Processes called master and slave. The master Process begins by issuing an ENABLE DEBUG instruction. From then until it issues a DISABLE DEBUG instruction, the master can trace execution of the Module being debugged (or, more precisely, of any activation of this Module in the slave Process [see below]). The master can trap and gain control when specific data areas are accessed, or when instructions are executed.

When the master gains control, the slave Process is suspended. The master can thus inspect and/or change registers, instruction counter or data areas. It cannot change the code, but it can insert Patches that remain effective during this debugging session.
Linkage

A collection of Modules can be organized into a unit that is analogous to a program in PL/I (i.e., to the output of a conventional Link Editor). This collection is defined in a system object called a Linkage Vector. Because the Linkage Vector contains not the Module itself, but only a pointer to it, the same Module can participate in several Linkage Vectors. The Linkage Vector also contains a Data Space holding all constants and pointers addressable by subsets of its modules.

Finally, during the execution of a single Process, many Modules may be executed, invoked directly or indirectly via different Linkage Vectors. These Module Activations need a common Data Space whose scope is not the static one of a Module Set or a Linkage Vector but the dynamic scope of the Process itself. Thus, there is need for yet another system object: the Linkage Vector Table. The LVT contains entries of all Linkage Vectors and Modules that can share addressability to its Data Space; and it contains the Data Space itself. An LVT, at any point in time, can be associated with no more than one Process. This association is accomplished by the issuing of the CONNECT LVT instruction.

In summary, there are three non-nested kinds of collections of Modules; the first physical, the latter two logical:

1. Module Sets for space conservation
2. Linkage Vectors for static link editing
3. Linkage Vector Tables for sharing a Process-local Data Space

A Module is executed by the issuing of a CALL instruction. This creates a new component within a Process system object, an Activation. Because the CALL instruction is itself issued during execution of some Activation, there is a chain of Activations linked by CALL instructions. A RETURN instruction destroys the current Activation, and the Process executes the instruction following the CALL in the last Activation up the chain. The roles of CALL and RETURN are similar to those they play in most programming languages.

An Activation contains (among other things not discussed here) a save area to link back properly on RETURN, an instruction counter, and a local work space private to this Activation of this module (similar to PL/I AUTOMATIC).

The CALL instruction has a single operand, a register, in which may be stored a Module pointer, an LV entry pointer, or an LVT entry pointer, thus requiring 0, 1 or 2 levels of indirection before the called Module can be determined.

This syntax allows a Module to be written and encapsulated before decisions are made about how dynamic its linking to other Modules will be. On CALL, machine registers are loaded with the Module pointer, a pointer to the static Data Space (from the Module, the LV, or the LVT), and a pointer to the Activation-local work space.
Of all the aspects of this architecture, CALL is the one most difficult to define satisfactorily. The reason is primarily that CALL is aimed at solving many self-conflicting problems. The following is an example of such conflict:

Unless the Activation-local work space is very quickly allocated and initialized, CALL becomes too slow. This implies a reused stack for its implementation. But if pointers can be generated to this work space and stored in Data Spaces that persist across CALLs, the integrity of the system can be compromised (that is, one Activation or Process can access the local space of another by secreting a pointer and using it at some later time). Further, if the machine stores its linkage information in this same stack, it is possible that a program may even break the machine.

To guard against this, the architecture defines a constraint on the use of the Activation-work space that guarantees that a pointer to it can never be generated and stored. This constraint, however, makes the work space unusable for PL/I AUTOMATIC, because that language allows pointers to this type of data storage. A common space-management convention across all languages subsequently results in one hardware stack and also one software stack per Activation chain. The usefulness of the hardware stack is then in question.
ASYNCHRONOUS PROCESSING

Processes

A Process is the fundamental unit of computing parallelism at the machine Interface. Each Process generally runs independent of any other process, and in parallel with it. In addition to allowing multiprogramming, the coexistence of multiple Processes at the machine interface is the means for exploiting multiple processors within the machine implementation. Each Process appears as a last-in, first-out stack of Module Activations, with each Module specifying the sequence of instructions to be executed (or next Module to be called), as described under "Modules". A description of Process creation requires first that message Queues be defined.

Message Queues

A basic means of communication among Processes is through the use of system objects called Queues. The existence and lifetime of a Queue is independent of the Processes using the Queue.

Messages are placed on a Queue either explicitly, by a Process issuing an ENQUEUE instruction, or implicitly, by the machine's signaling the occurrence of some predescribed asynchronous action (see Event Signaling). To remove messages from a Queue, a Process issues an explicit DEQUEUE instruction.

A Message consists of fixed-form system data and free-form user data. The system data part contains:

1. A pointer identifying the Message.
2. A pointer identifying the enqueuer.
3. Scheduling and ownership pointers to be used if a Process is to be created (see "Process Creation").
4. Dequeuing priority.

In general, the user data portion of a Message consists of a mixture of bytes and pointers. When the Queue is created, the user data area of Messages is declared to be either fixed size or variable with a fixed maximum size. This is done to simplify the machine's space management structure. When the Queue is created, it can also be restricted to disallow pointers. Restriction provides a means for Processes to communicate while allowing some control over the flow of pointers, hence the flow of addressing capability.

The algorithm that selects which of several messages in a Queue is to be dequeued may be specified in the DEQUEUE instruction or taken from the default given when the Queue was created. Three built-in algorithms are provided: FIFO, LIFO, and priority. In addition, the dequeuer may also specify an additional selection criterion based on a comparison of the contents of a particular field in the Message with an argument provided by the dequeuer. The comparison can be against the relations: =, \#, <, >, \%. The dequeuer also has the option to go into wait on the Queue when no Message satisfying his criteria now exists on the Queue.
The definition of Queue structure and dequeueing criteria at Queue creation addresses a classic trade-off between function and performance when the function is not desired. It is relatively clear that specifying the Message size and default dequeueing algorithm at Queue creation time allows the implementation to optimize the Queue structure and achieve high performance when this algorithm is actually used in dequeueing. However, when the dequene specifies another algorithm or uses field selection criteria to obtain added function, the performance erodes.

A second problem involves the decision to use Queues in Process creation (see following paragraph). This increases the size of the system data and forces the architecture to double the number of ENQUEUE/DEQUEUE instructions: one set supplies only user data, uses the Queue's default system, and has good performance; while the other set has the full function of supplying system data, but has poorer performance.

Process Creation

The Process system is unique in that it is not explicitly created. A Process is created only as the result of the appearance of a Message in a Queue that has been connected to a system object, called a Process Model. The Process Model contains optional scheduling and ownership specifications, a pointer to the Module which is used in the initial Activation, and a list of connected Queues.

The order of determining scheduling and ownership is first from the Process Model if it exists, next from the connected QUEUE, and finally from the system data in the Message itself. This order is selected to allow the creator of the Process Model to maintain maximum control, and force the criteria if he so chooses. When the Process is actually created, the user data in the Message is placed in a data area within the Process object where the initial Module Activation can gain addressability to it.

Implicit Process creation, via Process Models and Queues, is an extremely controversial aspect of the machine. With implicit creation, it is troublesome for the systems programmers building the control program to gain addressability to Processes, and hence gain what they feel is necessary control -- especially for error recovery. Thus, the architecture is forced to provide support for "walking down the ownership tree structure" to obtain fully authorized Process pointers, thereby sacrificing some system security. The advantage of implicit creation is the independence of lifetime and even existence between the Process requesting creation and the Process actually created, a feature valued by those concerned with machine parallelism. This allows applications to be written which react to the arrival of data in a work Queue without having an existing Process available to poll or wait on the Queue. The counter force is the cost of creating a Process, which appears to be considerable in most machine implementations.
ATOMICITY

The machine has a very precise formulation of the rules for referencing storage and for interaction among multiple Processes. This formulation is considered the backbone of the multiprocessing architecture.

At the machine interface, there is exactly one representation of a given system object in storage. With respect to storage references, all instructions in the machine are defined in terms of six primitive meta-functions:

1. Fetch byte
2. Store byte
3. Fetch pointer
4. Store pointer
5. Seize system object component
6. Release system object component

The fetching and storing of bytes and pointers are atomic at the machine storage, but their order, as observed by another Process, is not specified. Thus, if one Process issues an instruction that stores two pointers, another Process sees either the new value or the old value for each pointer (never a mixed result in the 16-byte field), but may see them stored in either order. The machines implement the atomicity requirement by exclusively locking data lines into their caches.

When more complex system object components are to be accessed in an atomic fashion, the instructions specify seizing (exclusively locking) the component, fetching and/or storing the information, and then releasing (unlocking) the component. Because the internal representation of system objects is encapsulated, the machines implement these functions by placing component lock fields (which are not directly "architected") within the objects. All instructions using seize/release are "architected" so that no circular dependencies exist; hence, no deadlocks can occur within the machine.

Atomic access to data strings of bytes and pointers is not provided directly by the machine and must be programmed by use of Gates. Byte and pointer atomicities are considered necessary and easy to provide, but solving the generalized problem of updating diverse data is not amenable to precise architectural specification.

Synchronization

The logical sequence of a Process is precisely defined to be the operation order determined by both the Module text and the "architected" definition of the instructions. To itself, a process appears to follow its logical sequence.

However, one Process could observe another as it followed some other sequence, as was noted in the previous example of multiple pointer stores. Such phenomena frequently occur in pipelined, overlapped multiprocessors with lookahead buffers. To ensure that a Process appears to all other Processes to have reached a specified point in its sequence, the notion of synchronization is introduced. Synchronization establishes an unambiguous ordering between the logical sequence and the activities at the machine storage or the time-of-day clock. If a Process is at a point of synchronization, then:
1. All store-and-fetch metafunctions in the logical sequence prior to the point of synchronization are completed at the storage.

2. No store-or-fetch metafunctions in the logical sequence after the point of synchronization have begun at the storage.

3. All reads of the time-of-day clock in the logical sequence, prior to the point of synchronization, are completed at the time-of-day clock.

4. No reads of the time-of-day clock in the logical sequence after the point of synchronization have begun at the time-of-day clock.

5. If, in the logical sequence prior to the point of synchronization, there are multiple store metafunctions to the same byte, pointer, or system object component, the last store metafunction in the multiple set is completed last at the storage.

In addition to a specific instruction to allow the programmer to cause synchronization, the machine defines synchronization points to occur in most system object manipulating instructions: e.g., ENQUEUE a Message, LOCK a Gate, etc. Although, in heavily overlapped machines, this can cause some performance loss, it appears necessary to define precisely the state of a multiprocessing system. General-purpose computational instructions (e.g., ADD) never cause synchronization.

Gating

Frequently, it is necessary for one Process to control the state of shared objects for more accesses than are provided by atomicity. Gates (locks) are provided by the machine to allow Processes to serialize their references to such shared objects. It is important to note that, as defined in the architecture the use of Gates is a programming convention with no enforcement by the machine. Enforcement would require the machine to maintain a correlation between a Gate and a set of diverse object components and data fields and then monitor access to them to verify the accessor's rights. It is generally felt that this is too great a performance burden for the machine to bear.

Gate Sets are system objects containing a collection of Gates packed together for reasons of storage efficiency, common lifetime, and common ownership. Each Gate can be independently locked and unlocked. An unlocked Gate may be placed in one of three locked states: Exclusive, Shared State 1, or Shared State 2. No more than one Process at a time may hold a Gate in the exclusive state. Several Processes may hold a Gate in either the Shared State 1 or the Shared State 2 at any one time, but these two states are mutually exclusive of each other and of the exclusive state. The two shared states exist in order that two different algorithms with multiple holders may be programmed for a single Gate. Contention for Gates can be resolved either by the machine in a manner defined in the architecture, or by explicit programming. The choice is specified when the Gate is created.
A Process requests a Gate by issuing a LOCK instruction and specifying whether or not it wants to wait if the Gate is unavailable. The Gate is available and assigned to the Process if it is:

1. Unlocked.
2. Locked shared state 1 or 2; the request is for the same state, and no Processes are Queued in lock wait state for the Gate.

If the Gate is unavailable, and machine management is specified for the Gate, the Process is Queued against the Gate if it had requested this option. If programmed management was specified for the Gate, the LOCK instruction is terminated; an implicit CALL is generated to a Module specified in the Gate Set. This Module Activation receives a fully authorized pointer to the Gate and is expected to program the Gate-Locking management.

In similar fashion, when an UNLOCK instruction is issued against the Gate and programmed management was specified, an implicit CALL is generated to another Module specified in the Gate Set that is expected to program the Gate-unlocking management. Otherwise, machine management causes the Gate to be unlocked with respect to the issuing Process. If the Gate now has no holders, it is placed in the unlocked state and may be granted to a waiting request or according to a priority determined by scheduling criteria (see "Scheduling").

The Gate architecture was selected to provide good performance through direct machine management for the most common uses, while still allowing almost unlimited functional flexibility through the programmed extension mechanism. One price paid for this flexibility is the addition of a dozen instructions to allow the programmed extension managers suitable functional access to the encapsulated Gate objects.

Event Signaling

Actions describe the occurrence of various machine states during instruction execution or during other machine activities. They trigger asynchronous event signaling of the occurrence of the specified state. Any given state can cause one or more actions to be raised. The total number of actions defined for the machine exceeds 100. Examples of actions are:

1. A specified instruction exception occurs (such as a protection violation).
2. Time-of-day clock reaches a selected value.
3. Storage consumption is exceeded.

In addition, an instruction is provided to enable the explicit raising of some processing actions to provide a method of simulating actions for program checkout.
The user of the machine declares his intent to listen for actions by creating a system object, called an Event Specification, which defines:

1. The action of interest (or a generic set of action types).
2. The domain in which to listen for the action (this is usually a subtree of the Context ownership structure).
3. Whether a Message describing the action is to be placed in a Queue or a machine Counter is to be incremented.

When an Event Specification has a Counter as the signal target, the Event Specification contains an integer value that is added to the Counter. When a Queue is the target, the Event Specification contains the fixed system data; the user data is generated in a manner uniquely "architected" for each action. This latter data describes what action occurred, when it occurred, and why it occurred. An auxiliary function is provided for some actions involving event queueing that allows the Process causing the action to be suspended. This permits the Process dequeueing the event signal to proceed in a synchronized fashion from the point the original Process caused the action. The recipient of the event Message is responsible for the resumption or destruction of the original Process.

When the architecture was defined, it was recognized that indiscriminate use of event signaling could seriously erode performance. The cost of checking for actions, locating all Event Specifications "listening" in the given domain, and performing the specified signaling is significant. For this reason, event signaling is viewed much like Processes; it is made a function that can be scheduled under the control of resource management (see "Resources"), thereby giving an operating system a great deal of control over its use. However, there appears to be no way to achieve significant reduction of the overhead of the function. But, in spite of those performance problems, event signaling is considered such a critical function for monitoring system behavior that its use within an operating system would be pervasive. This heightens the need to discover more efficient architectures and implementations.
RESOURCES

Resource Function

The machine architecture allows for creation of any number of objects, limited only by the capacity of the pointer H generator; it also allows for the parallel execution of any number of Processes. The resource architecture of the machine provides a mechanism for modeling the finite storage and processing capability of a given implementation. Sufficient function is provided to allow the programming of resource allocation algorithms above the machine without having to expose the explicit structure of the machine implementation in terms of its memory hierarchy, number of processors, etc. The two resources defined by the resource architecture are storage capacity and processing rate.

Storage Resources

Storage is the resource medium occupied by system objects from the time they are created until they are destroyed. The exact amount of storage occupied by a given object can vary with different implementations. To improve performance by reducing the reporting rate, the use of storage is reported in units of one kilobyte, as opposed to individual bytes. The total amount of storage resource available in a given machine implementation for allocation to objects is determined by its configuration, the efficiency of its recovery procedures (e.g., how much real storage is used for backup copies), etc. A reasonable approximation of this total is that storage available for programmer use at the highest capacity levels of the hierarchy.

The use of storage by an object is independent of the number and frequency of accesses made to that object. For example, the fact that frequent reference to an object causes it to exist in the more costly high-performance levels of the implementation's storage hierarchy is accounted for by the consumption of processing rate associated both with the referencing and any resulting paging.

To allow a more efficient storage management system, several storage use instructions were introduced into the architecture:

DECLARE OBJECT AFFINITY specifies a set of objects that will generally be referenced together, hence might be staged together.

DECLARE NEXT REFERENCE specifies an expected time to next reference of a set of objects, hence it can be used to control page-out/page-in.

DECLARE SEQUENCE specifies intent to reference a Data Space in a sequential manner, hence it can be used to control page-ahead/page-behind.

DECLARE PREFERENCE specifies that a specific Process intends to access a set of objects frequently; hence it can be used to control swapping and/or defeat normal aging criteria.
Correct machine operation does not require that these declarations be issued. Any improvement in the efficiency of storage management that results from their use becomes visible as an effective increase in the available processing rate.

The requirement for storage use declarations provokes conflicting technical opinion. From the standpoint of responsibility for implementing the machine's storage management implementation, a simple storage resource control based on capacity alone, coupled with an implementation-defined demand paging and Process affinity structure may provide the best overall system performance. From the standpoint of responsibility for the programming system (especially for applications such as sorting), experience tends to indicate that such a structure may be inadequate. Thus any knowledge programs have of their storage usage patterns should be passed on to the storage system to improve performance. The question of whether simplicity of implementation or knowledge of usage patterns produces better performance has no clear answer. The architecture provides what is considered to be a balanced approach to the problem.

Processing Rate Resource

The rate at which the machine can execute instructions in a Process is called processing rate, denoted \( \text{V} \) for Velocity. Many attributes of the machine implementation contribute to the total available processing rate, including: the number and speed of the central processing units; the speed, capacity, and structure of the storage hierarchy; the number and speed of the I/O channels; the structure of the underlying scheduling and dispatching microcode, etc. Processing rate is expressed in terms of Normalized Instructions Per Second (NIPS) where a normalized instruction is a statistical measure of the average instruction executable by the machine. Similarly, execution of each particular machine instruction is then quoted in terms of being statistically equivalent to the use of a certain amount of processing as measured in normalized instructions.

Activities that compete for processing rate are called scheduled objects. The machine contains three types of scheduled objects:

1. Processes
2. Process creation requests
3. Event Specifications for certain kinds of actions

Processes consume processing rate to execute their instructions and to raise actions and signal events that are synchronized with their execution (e.g., signal that the Process is attempting to violate the pointer protection structure). The enqueuing of a Message consumes processing rate associated with the Process performing the ENQUEUE. However, if that Queue is attached to a Process Model, any resulting asynchronous creation of a Process consumes processing rate associated with the resource allocation for the Process Model. As soon as the Process is created, processing rate comes from the resource allocation to the Process. Event signaling for actions not synchronized with a given Process (e.g., signaling that a timer action occurred) requires the use of the processing rate resource associated with the Event Specification.
How the resources are allocated to scheduled objects and how machine scheduling occurs are discussed under "Resource Partitioning" and "Scheduling".

Resource Partitioning

The storage and processing rate resources are partitioned between sets of objects through the use of systems objects called Resource Pools. Like all other objects, Resource Pools can be created and destroyed. Facilities exist for the transfer of resource from one Pool to another, subject to a law of constant total resource among Pools of a given type.

Resource Pools are connected to Contexts within the ownership tree, subject to the constraint that only one Resource Pool of a given type can be connected to a given Context. A given object draws its resources from the first Resource Pool it encounters as one "walks up the Context tree". All objects drawing resource from a Pool are said to be in the Pool's domain. Changing the connection of a Pool to a Context or destroying the Pool requires reevaluating the domain and the use of resources in the Pool. Because this is a somewhat costly operation, restructuring the ownership structure and/or the Pool connections should occur very infrequently, i.e., only at points of major system reorganization.

A storage Resource Pool represents an amount of storage measured in kilobytes. When storage pools are created or destroyed, resource must be transferred into the Pool or out of it to maintain the law of constant total storage. A storage Pool also contains a measure of how much storage is currently distributed to the objects drawing from the Pool. Storage is distributed on a first-come, first-served basis as objects are created and destroyed within the Pool's domain. When the Pool's capacity is about to be exceeded, an action can be raised and an event signaled to the programming system's resource allocation function.

A processing rate Resource Pool represents specified amounts of processing rate measured in kilo normalized instructions per second (KNIPS). A key architectural premise is that processing rate is too valuable a resource to remain unused only because the scheduled objects in some domain are not using their allotment. Thus, rather than use a scalar control of this resource, as in storage, processing rate Pools contain a vector of 64 different rate levels. In a given Pool, the processing rate values increase monotonically as the rate level i increases from 0 to 63. At rate level i, the total processing rate in all Pools in the machine must sum to:

\[ 2^{** ([i+25] /4)} \text{KNIPS} \]

to satisfy the law of constant processing rate. This particular formulation was selected to provide a reasonable logarithmic spread in total available rate from 91 KNIPS for level 0 (the lowest rate expected on any machine implementation) to 4,987,896 KNIPS for level 63 (a rate considered to be high enough to cover any conceivable commitment of processing rate).

At any given instant of time, all Pools operate at the same rate level, and their entry value for that level specifies the processing rate available for scheduled objects in the Pool domain. As the rate level increases, the constant total that represents the amount of processing
rate that may be distributed increases, as can be determined from the preceding formula. It makes no difference whether more rate is inherently available in the machine because of an increase in machine capability or apparently available because of the inability of scheduled objects in some domains to use the rate allocated to them. The machine merely makes the decision, and increases the rate level to make more available to those who can use it. If at a later time machine capability decreases or the scheduled objects begin to fall behind because they are not getting their allocation, the rate level is automatically reduced by the machine.

Basically, at the lower rate levels, Pools for critical tasks have positive entries; others have zero entries. At higher rate levels, the noncritical tasks begin to have positive processing rate entries in their Pools. Thus, if the available rate is too low, only the critical tasks run. If more rate becomes available, either because it exists in the hardware or the critical tasks are not using it, the rate level shifts upward and some processing rate now becomes available to the noncritical tasks.

How the rate in a given Pool is distributed to each scheduled object is the subject of the scheduling architecture.

Scheduling

Machine scheduling is based on the dual concepts of processing rate, as discussed in the previous sections, and granularity.

Granularity defines the response time characteristics of a scheduled object by specifying, in seconds, the minimum time period during which the processing rate distributed to a scheduled object can be observed at the machine interface. For each scheduled object, a requested processing rate, $V_r$, and a requested granularity, $g_r$, are specified. The machine attempts to give the scheduled object an amount of processing, $P$, equal to $V_r g_r$ normalized instructions within the time period $g_r$. Scheduled objects within a given Pool domain compete with each other for the Pool's processing rate based on their respective $V_r$, $g_r$ values. Because the machine normally cannot exactly match the requests for each scheduled object, it reports back that it gave the scheduled object its requested $P = V_r g_r$ normalized instructions in a distributed granularity period $g_d$. The difference, $g_d - g_r$, is a measure of how far the scheduled object is behind schedule or ahead of it. The architecture specifies that the machine must resolve contention within a Pool domain so as to keep all scheduled objects the same amount behind, or ahead of, schedule.

Competition criteria for the scheduled objects in a Pool domain are specified through a system object called a Scheduling Table, which is connected to the processing rate Resource Pool. A Scheduling Table consists of as many as 32,768 independent scheduling levels. A scheduled object in the Pool domain is always associated with a specified level in the table. The key elements in a Scheduling Table entry level are:
1. A token pointer used to authorize use of this level.
2. \( V_r \) value.
3. \( q_r \) value.

For each "architected" change criterion, there is a corresponding specification of the new Scheduling Table level to be used if the criterion is satisfied. An example of change criterion is a specification of a threshold on \( q_d - q_r \) for the Pool. If this threshold is exceeded, the scheduled objects are falling too far behind, and their scheduling level is changed to readjust the load.

A Process can have its effective scheduling criteria changed in one way. If Process A holds a Gate, and Process B is in wait on the Gate, Process A can be promoted to take on Process B's scheduling criteria if it is better than Process A's; i.e., higher \( V \) and smaller \( q \). This feature was added to the architecture to overcome the dilemma of a high response Process waiting for long periods of time, while a low response Process holding the Gate awaited scheduling. It could be argued that this should not happen in a well-structured system, but examples exist where the simplest system design has interactive and batch processes contending for the same data locks.

Two major concerns with the scheduling architecture are:

1. The overhead required to keep current the measures of being behind or ahead and the adjustment of rate levels and scheduling levels.
2. The ability of machines covering a broad spectrum of performance ranges to implement independently the scheduling architecture with common pragmatics. It is this latter point that provokes conflicting opinions. To have a single operating system serve a spectrum of implementations, common pragmatics are essential. This implies some commonality of design within the implementations. On the other hand, machine implementers tend to feel that meeting performance objectives requires designs that are uniquely tailored to the individual hardware configurations.

It may also be that the scheduling architecture suffers from excessive functional flexibility. It is conjectured that when the system has been in operation for a period of time, the uses of rate Pools and scheduling tables will fall into a relatively small set of highly stylized categories. If this turns out to be the case, the excess function could be removed from the architecture, or the implementations could be optimized to reflect the dominant usage patterns.
CONCLUSION

Clearly, a major issue permeating specific debates about this machine architecture is whether it is feasible to furnish high-level, general-purpose functions for a wide spectrum of application environments. Where the function matches the application requirements, defining it to be within the machine provides an efficient implementation. But where the function is not needed, or poorly matched to the requirements, often even the most careful attempt at providing degenerate options leaves a residue overhead which, in total, becomes substantial.

Coupled to this question is the fact that while the cost (in time, memory space, and hardware) for a function (such as protected addressing) is relatively easy to measure, the benefits (in overall system performance, reliability, changeability, ease-of-use) are much more subjective, difficult to quantify, and dependent upon how they are used.
REFERENCES


