Open access • Journal Article • DOI:10.1145/850708.850709

A hardware implementation of capability-based addressing — Source link

Glenford J. Myers, B. R. S. Buckingham

Institutions: Systems Research Institute

Published on: 01 Oct 1980 - Operating Systems Review (ACM)

Topics: Applications architecture, Space-based architecture, Capability-based addressing, Architecture and Representation (systemics)

A HARDWARE IMPLEMENTATION OF CAPABILITY-BASED ADDRESSING

G. J. Myers and B. R. S. Buckingham
IBM Systems Research Institute
205 East 42nd Street
New York, N. Y. 10017

Abstract

The SWARD architecture, an experimental higher-level architecture, contains the naming and protection concept of capability-based addressing. After discussing the merits of capability-based addressing, its general representation in the SWARD architecture is discussed. The initial representation of capability-based addressing in the architecture led to a set of problems; these problems are described, as well as their solutions. Finally, the implementation of capabilities by the processor is discussed.

Introduction

The concept of viewing a computing system as a set of objects, and addressing the objects by a protected name known as a capability, has existed for 14 years [1] but has been slow in influencing computer architectures. The idea is often discussed in the context of operating systems [2-5] and has been implemented, in software, in such experimental operating systems as Cal [6] and Hydra [7]. The idea is implemented in the computer architectures of the commercial Plessey 250 [8] and IBM System/38 [9], the experimental Cambridge CAP [10-11] and IBM SWARD [12-14] systems, as well as a variety of other proposals [15-16].

The interest in capabilities is motivated by issues of protection, information sharing, and, in at least the SWARD architecture, program reliability. Considering protection for the moment, the typical commercial computing system of today attempts to achieve protection through a plethora of policies and mechanisms. The mechanisms might include privileged instructions, storage-protection keys, the execution of processes in separate virtual machines or address spaces, sign-on passwords, file passwords, and so on. Such designs lead to at least four problems

1. Difficulty of verification. Where the protection mechanisms exist in both hardware and software and are spread throughout the system, the act of establishing confidence that the system is secure is a difficult task.

2. Locating the weak link. In such systems, the protection mechanisms do not usually reinforce each other; each is there to treat a different aspect of the problem. If one of them can be subverted, the system becomes totally unprotected.

3. Complexity. Because the designers are looking at the problem in a piece-meal fashion (e.g., How do I protect the privileged instructions? How do I protect the operating system from modification? How do I protect

one program from another? How do I protect against unauthorized use of files?), the system design is often complex, the user interface is often complex, and the designers are never sure that they posed all the pertinent questions.

4. Difficulty of sharing. One must also provide the means for controlled resource and information sharing. For instance, process A might wish to share its procedure X, but no other procedures, with process B. Process A might wish to give process C read-only access to array Z, but no access to anything else. Fabry [17] does an able job of showing the difficulty of such sharing with conventional addressing mechanisms, such as relocation, paging, and segmentation.

Capability-based addressing is a uniform mechanism oriented toward both the protection and sharing of information and resources. In most implementations, storage is viewed as a single set of objects, all being intrinsically addressable by all processes. Depending on the design, the objects' construction might be visible to programs (e.g., simply byte spaces) or they might be defined abstractly (i.e., their construction is not visible to programs; the objects are defined only by the operations that can be performed on them). The entities that are represented by objects typically might be procedures, files, dynamically allocated data storage, and message queues or ports.

The concept is further generalized if each object is addressed by a unique name, rather than a linear storage address. The name is unique in that, when an object is created, it will be assigned a name that was never used in the past for a prior object. When an object is destroyed, its name is forgotten by the machine and never reassigned. These names are system wide and are usually just an arbitrary pattern of bits, rather than a symbolic name.

A capability is generally considered to be a data type containing a protected, system-wide name of an object. In addition, a capability contains a definition of the access rights possessed to the object (e.g., read, write, destroy). The only way to make a reference from one object to another is by the possession of a capability to the latter object. The possession of a capability is the sole determinant of access rights; hence one controls access to any object by controlling the distribution of capabilities to the object. Also, objects have no "owners" in the traditional sense; the operations that one can perform on an object are determined solely by the access rights in the capability possessed to the object. These rights can be transferred among objects or processes by simply transferring the capabilities.

Physical Analogy

When analyzing protection mechanisms, it is often helpful to use physical analogies. For capability-based addressing, the physical analogy is that each object is contained within a box having several springed doors, and capabilities

are keys to these doors.[1] The object name in a capability is analogous to a pattern of notches on the key, and the access rights are a set of auxiliary "bumps" that permit access to particular doors of the box, for instance, to a door exposing a transparent shield (read-only access) or a door exposing a plunger (destroy access). When a box (object) is created, the machine gives the creating person (process) a key (capability) that will open all doors in the new box. What transpires after this is up to the person with the initial key.

To make this world of locked boxes and keys a secure and useful one, one arrives at the following considerations

1. It must be impossible for a person to fabricate a key. Also, given a key, it must be impossible to alter the notches (e.g., file a new notch with the hope that the altered key will open a different box).
2. Given a key, it should be possible to make a copy of the key, either for one's self or someone else.
3. Given a key, it should be possible to remove (but not add) one or more of the auxiliary bumps to remove some of the key's access rights.
4. For generality, one should be able to move and store keys in the same way as any other entity. For instance, one should be able to store keys, or a combination of keys and other treasures, in the boxes.

The SWARD Architecture

The SWARD architecture was developed with an unusual motivation - that of enhancing the reliability of programs executing upon the machine. The architecture approaches this from several angles, for instance, by containing functions to aid software testing and debugging tools, by containing functions to facilitate the efficiency of highly modular and structured programs, and by installing barriers to minimize the effects of software errors, but the primary emphasis is on semantic checking during program execution. The current definition of the architecture has not been published, although an early version has been [13].

One basic attribute of the architecture is tagged or typed storage, although it is carried further than most implementations. The tags are variable in size, the size being dependent on the amount of information needed to express the attributes of particular data types, and the machine recognizes and processes such data types as strings, records, arrays, arrays of records, and so on. All data types have a specific values defined as the "undefined value," allowing the machine to detect what our studies show to be the most common programming error - using a variable that has never been given a value.

The architecture can be termed a domain architecture [18], since every procedure is encapsulated within a private memory space. One of the object types in the architecture is the module. A module contains both a set of machine instructions and a private address space, consisting of a series of tagged storage

---

[1]Since there may be confusion over whether the term "capability" refers to the value or the storage cell holding the value, one might more correctly say that the access-rights, object-name couple is the key and that the placeholder or data type is a single-key key chain.

cells (occurrences of data types). A machine instruction can refer to cells within only the private address space of the module in which the instruction resides. Any and all references to other objects must be done indirectly through capabilities stored in the private address space.

Furthermore, the architecture contains a high-level generic instruction set, a send/receive mechanism for interprocess communication, synchronizing instructions oriented toward the design concept of monitors, automatic subroutine management through the use of activation records in activation stacks, a hierarchical error detection and handling mechanism, and the concept of a single-level storage.

The Initial Capability Mechanism

In this section, the initial definition of objects, capabilities, and addressing in the architecture is discussed. Development of these concepts led to the discovery of a set of problems, most of which were solved by extensions to the mechanism. These problems and solutions are discussed in the next section.

The SWARD architecture contains five types of objects: module, data-storage object, port, process machine, and activation record. The module was mentioned above. The data-storage object is an occurrence of one or more typed storage cells (including arrays and structures) that is dynamically allocated space under program control. A data-storage object might represent a file, an array dynamically allocated by a program, and so on. A port is an abstract device through which processes send and receive collections of cell values. A process machine is the only active object; it is a virtual processor which executes modules concurrently with other process machines. The activation record is an object that is implicitly created by the machine when a module is called (activated); its primary purpose is to hold that part of the module's address space that is to be allocated space whenever the module is called. It should be noted that the five objects are abstractions; their physical representation is unavailable to programs.

One of the data or cell types provided is the pointer. A pointer is the type of cell in which a capability resides. A pointer cell occupies 88 bits of storage, consisting of a 4-bit tag (1001) and an 84-bit value. A pointer's value (a capability) consists of a 4-bit access or authority code and an 80-bit "logical address." The access code limits the types of operations that can be performed on the entity to which the capability refers. Initially, the access codes specified read authority, write authority, destroy authority, and undefined (i.e., the pointer has no defined value). Rather than defining a large number of access types, their interpretation varies slightly depending on the entity addressed by the pointer. For instance, read authority to a port permits one to use the RECEIVE instruction to obtain a set of cell values from the port.

At this point, a few major differences from other capability systems should be noted. For reasons of programming generality, pointers can be used in a manner similar to addresses in conventional systems. Hence, pointers are data types and are not constrained to be stored in separate "capability lists" (a scheme used to protect capabilities in machines without tagged storage). Pointers can be stored as variables in programs, can exist within user data structures, can

be passed as arguments, sent through ports, and so on.

Another difference is that capabilities are not restricted to referring to only entire objects. For instance, if one possesses a capability to a data-storage object containing, perhaps, an array, one can ask the machine (via the COMPUTE-CAPABILITY instruction) for a capability to any addressable entity within the object, such as a particular array element. This gives one a selective and finer granularity of protection. For instance, this second capability can then be passed on to someone else (e.g., a process or a module called as a subroutine). The recepient now has access to only this element, and not the entire array. In fact, the recepient has no knowledge that this represents an array element, or that it resides in a data-storage object.

Returning to the definition of the capability, its principal part is the 80-bit logical address. The architecture purposely does not define the representation of a logical address; it is simply defined as information, meaningful to only the machine, that refers to a particular object or entity within a particular object. Part of the logical address is the unique name of the object, called a SON (system object name).

Capabilities, then, can refer to the following:

Data-storage object
Cell within a data-storage object[1]
Module
Cell within a module ("own" or "static" variable)[1]
Entry point within a module
Cell within an activation record[1]
Port
External

(The latter refers to an input/output device.)

The major instructions dealing with pointers are:

ALLOCATE.....given a description of the collection of cells to be allocated, creates a data-storage object and returns a capability to it (All instructions that create objects return a capability with full authority to the object.)
CREATE MODULE.....creates a module object.
CREATE PORT.....creates a port object.
CREATE PROCESS MACHINE..creates a process machine object.
DESTROY.....given a capability, destroys the object (module, data-storage object, port, process machine) to which the capability refers.
COMPUTE CAPABILITY.....given an addressable operand, computes a capability to it.
COMPUTE ENTRY CAP.....computes a capability to a specified instruction in a

[1]As well as components and elements within cells (e.g., array elements, record components).

module object.

CHANGE ACCESS.....further restricts the access code to a specified level in a capability.

LINK.....used to bind modules together (e.g., initializing a pointer cell in module A to refer to an entry point in module B).

In general, all of the remaining machine instructions can refer to pointers to refer to operands indirectly. One does this by associating one or more "relocatable cells" with a pointer, and referring to the relocatable cell as an operand. Such a reference causes the machine to resolve the logical address and, if the type information in the relocatable cell matches the type of that to which the logical address refers, to use the latter storage as the actual operand.

Other instructions refer to pointers directly as operands. For instance, the CALL instruction, in addition to referring to a set of cells as arguments, refers to a pointer cell which, in turn, refers to an entry point in a module.

Returning to the "locked box" analogy, one sees that the mechanism above achieves the four considerations mentioned. It is impossible to fabricate or manipulate a capability; this is enforced by the use of tagged storage. The only instructions that cause a pointer cell's value to change are (1) the MOVE instruction, and this allows one to move, into a pointer cell, only another pointer value, (2) the three instructions that create objects, (3) CHANGE-ACCESS, and (4) the two instructions that compute a pointer value.[1] The CHANGE-ACCESS instruction satisfies consideration 3 mentioned earlier, and treating the pointer as a data type achieves consideration 4. In addition, the COMPUTE-CAPABILITY instruction goes beyond these considerations by allowing one to dynamically create additional doors in the box to a subset of the treasures in the box.

Problems

After development of the system began with the concepts above, a number of problems were encountered. Some of the problems surfaced during the design of an operating system for the machine; others were uncovered simply by further thinking. Where feasible, the problems are discussed below in terms of the locked-box analogy. In fact, had we considered this analogy earlier, the problems would have been solved earlier, since the analogy makes them obvious.

1. The "new lock" problem.

An obvious act in the locked-box world is the replacement of a lock on a box, thus invalidating all existing keys. One might take this action, for instance, if (1) the box were sold to someone else, (2) one suspected that one or more of the current key holders were using the contents improperly, or (3) one became uneasy about not knowing how the keys had been distributed over time, and wishes

---

[1]Although the pointer can be used as an address, no address arithmetic is permitted (or needed), since the architecture directly supports the concepts of array elements and string processing, the areas normally requiring address arithmetic.

to start anew with a new lock.

The original architecture provided no mechanism to change locks, other than the unwieldy and time-consuming process of creating a new copy of an object and destroying the old copy. The problem was solved by the addition of the CHANGE-LOGICAL-ADDRESS instruction. Given a pointer cell with a capability to an object as an operand, and providing that the capability contains destroy authority, the machine forgets the current logical address, assigns the object a new unique logical address, and stores it in the pointer cell. Any storage reference using a capability with the forgotten logical address will result in an error.

2. The "do not copy" problem.

In the locked-box world, person A might wish to give person B a key to a door in a box, but might want to preclude B from copying the key (e.g., to give it to a third party). Hence, one needs a mechanism to stamp "do not copy" on a key.

This was achieved by adding a fourth authority type to the access code of capabilities - "copy" authority. If a capability has no-copy access, the machine refuses to allow a process to copy the capability into another pointer cell (e.g., via the MOVE instruction, by sending the capability through a port). The four authorities - read, write, destroy, and copy - are independent and can exist in any combination.

3. The "retraction" problem.

One of the major problems with locks and keys (and capabilities) is the ability to withdraw authority after it has been given. For instance, one might have given 10 people keys to a door, and then later decide that person D should no longer have a key.

Although the physical analogy breaks down a bit here, a possible solution is indirection, that is, rather than handing out keys to the box itself, one might hand out keys to a second box that contains a key to the first box. Hence one can withdraw the authority of a particular person or class of people by destroying, or changing the lock of, one of the secondary, "key-holding," boxes.

This problem, and others discussed later, was solved by the addition of an "indirect capability." An indirect capability is not an additional data type; it is the value of pointer cell whose value was created by a new instruction (COMPUTE-INDIRECT-CAPABILITY). The instruction has two operands, both of which must be capabilities. The logical address of the second operand is computed and stored in the first operand, marking it as an indirect capability.

An indirect capability cannot be distinguished, by a program, from a normal capability; thus a program is oblivious to whether it is using a normal (direct) or indirect capability. An indirect capability physically points to another capability, but logically points to where the latter capability points. Any reference through an indirect capability has the same effect as if the direct capability were used. Operations that can be performed on capabilities (e.g., copying them into other pointer cells, restricted their access) can be performed

on indirect capabilities.

The indirect capability has many uses. One is security, or a solution to the retraction problem. Suppose A wishes to give process B access to some data, but wishes to retain the ability to withdraw this access at any time. By giving B an indirect capability to a capability (in A's space) to the data, A, at any time, can modify the latter pointer to withdraw B's access.

Another problem solved by the indirect capability is

4. The "Is he done?" problem.

This problem arose in implementing a directory service in the operating system. The service, which is optional and need not be used by applications with unusually strict security concerns, allows users to associate symbolic names with capabilities. The service provides an object-control function to allow a process to ask for a capability under such qualifications as "I'm willing to share access to the object with any other processes that wish to use it in shared mode" and "I require exclusive access to the object."

The problem was that the directory service could not guarantee that a process would not continue to use an object after it had reported back that it was "done" with the object. The directory service now accomplishes this by passing out indirect capabilities. When a process reports that it is done with an object, the capability to which the indirect capability points is destroyed, thus blocking all future use of the indirect capability.

The indirect capability has another use for which there is no straightforward physical analogy. It allows one to dynamically replace objects without having to rebind programs, or even stop them in some cases. As an illustration, consider the case of module M where its callers have been given indirect capabilities to capability P to the entry point of M. If one wishes to replace M dynamically, one simply creates the new version and moves its entry capability into P. All calls to M now enter the new version. The old version can be destroyed immediately, since the DESTROY instruction, when applied to a module, causes the module's SON to become invalid but does not destroy the module itself if it is active. (The machine remembers the pending destroy operation and frees the space when the activation count drops to zero.)

5. The "lost key" problem.

Another obvious problem in the world of the analogy is the loss of all keys to a box. In terms of capabilities, the problem is the loss of all capabilities to an object (e.g., one creates an object and then mistakenly stores into the only pointer to the object). A related problem is the loss of necessary authority to an object (e.g., none of the capabilities to object X have destroy authority, meaning that X will occupy system space forever).

No solution was found that did not violate the security of the architecture, so the problem remains unsolved. Actually, security concerns are not the only impediment. The only thread tieing together the machine's knowledge of objects and the programs' understanding of objects is the SON in capabilities. If the capa-

bilities to an object disappear, the programs and machine have lost their only form of communication.

Several steps were taken to lessen the severity of the problem. First, the instructions that create objects allow one to specify whether the object should be automatically destroyed by the machine when the process terminates. This largely handles the problem of having a large number of supposedly temporary objects linger in the system because a process terminated abnormally. Second, programs that create permanent objects are encouraged (but not required) to deposit their capabilities in the operating system's directories. Third, the prototype implementation contains a mechanism to allow service personnel to search for, and recover or destroy, lost objects.

6. The "what is this key" problem.

The closest physical analogy to this problem is having no information about the properties of a key on a key chain, or perhaps a key discovered on the ground. It represents a set of situations where the machine has useful information that is not available to programs. Some of these situations were

- Although the capability has an architected access-code field, there was no way for a program to reference this information (e.g., to test a particular capability to determine if it possesses write access).
- Given a capability, there was no way to determine what type of entity it references (e.g., module, entry point in module, cell in data-storage object, port, and so on).
- The machine possesses certain state information about objects that was unavailable to programs. Examples of needed information were - Is this object designated to be automatically destroyed upon process termination? Is this module active? Are there any sets of cell values enqueued in this port?

The problem was solved by the addition of a DESCRIBE-CAPABILITY instruction. Given a capability and an array as operands, the instruction returns information in the array describing the capability (e.g., the authority it possesses and the class of entity to which it refers) and, if the capability refers to an entire object, state information about the object. Some of the state information is independent of the type of object, while some is dependent on the type of object. The instruction, however, does not return any information about the contents of an object (e.g., in the case of a data-storage object, it does not describe anything about the cell types or values in the object).

The Second Level of Protection

A further aspect of the architecture worth mentioning is the way that capabilities and tagged storage reinforce each other to enhance protection. Access to an object actually requires two things: its capability and exact knowledge of the nature of the object. If one does not possess the latter (and there is no way of obtaining it from the machine), the capability is quite useless.

To obtain or store information in a data-storage object, one needs a capability to it with the appropriate authority, and a set of relocatable cells that define the object. For instance, suppose one has created a data-storage object

that represents a record of four components: a character string of size 4, a floating-point cell of precision 17, a two-dimensional array of integers, and a character string of size 8. To reference this data-storage object from a module, the module must contain a set of relocatable cells that state "a record of four components, the first of which...." Any discrepancy between the module's notion of the object and the tagged storage within the object itself will result in a machine-detected error. Hence, if one is interested in further protection, one can "hide" the sensitive information in a data-storage object of sufficient complexity.

The same applies to other objects. For a port, one would need not only a capability, but also knowledge of the exact number and types of cell values to be sent or received by the process(es) on the other side of the port. To successfully call a module for which one surreptitiously obtained a pointer, one would have to know the number of parameters expected by the module and their exact data types.

Note that the DESCRIBE-CAPABILITY instruction mentioned above does not aid one in attempting to break through this second level of protection.

The Implementation of the Capability Mechanism

Capability-based addressing, when embodied in the machine architecture, also raises many interesting implementation questions. Although the traditional distinctions of processor power and cost have become blurred, the prototype implementation of the SWARD architecture could be categorized as a "fast minicomputer" implemented in TTL logic and relying heavily on bit-slice logic and programmable logic arrays. Because of a desire to use existing technology, a desire to avoid (initially) excessive hardware complexity, and because of the changing nature of the architecture, the architecture was embodied largely in a microprogram, rather than fixed logic, although the organization of the processor is oriented toward the architecture. The microinstruction design is horizontal, with the architecture implemented in approximately 6000 52-bit control-storage words.

The implementation of capabilities is quite straightforward. One needs a unique-name generator and a fast mechanism for the translation of object names (SONs) to physical storage locations. An obvious name generator is a counter. Since one must provide uniqueness across system outages, the counter must be implemented in nonvolatile storage or be battery powered.

The obvious mechanism for translating names is an associative storage array. However, to lessen costs, the prototype uses a hash-addressed table in a RAM. To avoid the overhead associated with hashing (key transformations and collisions), the following is done. Recognizing that the creation of SONs is far less frequent than the translation of SONs, the strategy is to pay whatever penalty must be paid during creation rather than translation. When a unique SON is needed, the microprogram obtains the next value from the counter, but, rather than deciding immediately to use this value, looks at the corresponding table entry (SON modulo the table size). If this entry is in use (for another SON), the counter is incremented and the next value is tried, since the actual value of a SON is meaningless. This continues until an empty table slot is located.

To translate a SON, the microprogram simply references the corresponding table slot (again, the SON modulo the table size). If the SON matches the one in the slot, translation continues. If not, the program is using a pointer to a destroyed object, and an error is signalled. As a result, the translation time of a SON is constant and approximately two microseconds, but the creation time is variable and somewhat longer.

As mentioned in an earlier section, the architecture purposely does not define the representation of the 80-bit logical address. In the prototype implementation, the logical address consists of a 32-bit unique SON, a bit to distinguish between a normal and indirect pointer, and two 23-bit values indicating the displacement of the addressed cell's tag and content within the object. (These two displacements are needed because logical addresses can refer to entities within objects, as well as entire objects.)

This representation of a logical address seemingly provides a sufficiently large addressing range (2^{32} objects, each containing up to 2^{23} tokens or half-bytes), but, since the processor must assign unique SONs, the supply of SONs can be depleted rather quickly. Since the object type most frequently created is the activation record, and since activation records, as a whole, are never addressed or referenced by programs, as a matter of course the processor does not assign a SON when an activation record is created. An activation record needs a SON only if and when one wants to compute the address of a cell that resides in the activation record (in PL/I parlance, when one executes P=ADDR(X), where X is in the AUTOMATIC storage class). Execution of the COMPUTE-CAPABILITY and COMPUTE-INDIRECT-CAPABILITY instructions, if specified with an operand that falls in an activation record and if the activation record does not already have a SON, causes the processor to assign a SON to the activation record.

Hence, objects that require SONs are modules, ports, data-storage objects, process machines, and an occasional activation record. Given a typical load, the system should run for about 10 years before it has to begin assigning "second-hand" SONs to new objects. Given the second level of protection discussed earlier, it is extremely unlikely, even if a program secreted a pointer to a destroyed object and tried to use it 10 years later, that the use would be successful.

The system has an auxiliary maintenance processor and CRT console for system instrumentation and debugging. In deference to the lost-key problem, one can invoke a diagnostic microprogram which, by scanning the SON memory and then examining cells in all objects, can locate objects for which no capabilities exist in other objects. The microprogram can either destroy all such objects or display a description of them (which is possible because of the use of tagged storage). Although the mechanism is unwieldy and requires physical security of the maintenance console, it is possible for someone to describe a lost object (e.g., "I lost a data-storage object containing a one-dimensional array of 50, 16-character, strings") and have the service personnel manually find, and create a capability to, the lost object.

Closing

Capability-based addressing was not embodied originally in the SWARD architecture for reasons of protection. The motivations were enhancement of the software-development process in the following ways:

1. Detection of the "dangling reference" programming error, where an address is used to refer to data after that data has disappeared.
2. Incorporation of the one-level storage concept to simplify and unify the way programs address data. A prerequisite to this concept is a large addressing range.
3. Isolation of each procedure within a private address space to lessen the effects of errors. This led to an object-oriented architecture.

However, the capability mechanism in the architecture, particularly when coupled with the enhancements added to solve the problems discussed, has been found to achieve a high degree of data security.

References

1. J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computation," CACM, 9(3), 143-155 (1966).
2. P. J. Denning, "Fault-Tolerant Operating Systems," Computing Surveys, 8(4), 359-390 (1976).
3. T. A. Linden, "Operating System Structures to Support Security and Reliable Software," Computing Surveys, 8(4), 409-445 (1976).
4. B. W. Lampson, "Dynamic Protection Structures," Proc. 1969 FJCC. Montvale, N. J.: AFIPS Press, 1969, pp. 27-38.
5. B. W. Lampson, "Protection," Proc. Fifth Princeton Symp. on Information Sciences and Systems. Princeton, N.J.: Princeton University, 1971, pp. 437-443, reprinted in Operating System Review, 8(1), 18-24 (1974).
6. B. W. Lampson and H. E. Sturgis, "Reflections on an Operating System Design," CACM, 19(5), 251-265 (1976).
7. E. Cohen and D. Jefferson, "Protection in the Hydra Operating System," Proc. Fifth Symp. on Operating System Principles. New York: ACM, 1975, pp. 141-160.
8. D. M. England, "Architectural Features of System 250," Infotech State of the Art Report 14: Operating Systems. Berkshire, England: Infotech, 1972, pp. 395-428.
9. IBM System/38 Technical Developments. Atlanta: IBM, 1978.
10. R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its Protection System," Proc. Sixth Symp. on Operating System Principles. New York: ACM, 1977, pp. 1-10.
11. A. J. Herbert, "A New Protection Architecture for the Cambridge Capability Computer," Operating System Review, 12(1), 24-28 (1978).
12. G. J. Myers, "Storage Concepts in a Software-Reliability-Directed Computer Architecture," Proc. Fifth Annual Symp. on Computer Architecture. New York: ACM, 1978, pp. 107-113.
13. G. J. Myers, Advances in Computer Architecture. New York: Wiley-Interscience, 1978.
14. G. J. Myers, "SWARD - A Software-Oriented Architecture," Proc. International Workshop on High-Level Language Computer Architecture. University

of Maryland, 1980, pp. 163-168.
15. G. J. Battarel and R. J. Chevance, "Design of a High Level Language Machine," Computer Architecture News, 6(9), 5-17 (1978).
16. H. J. Saal and I. Gat, "A Hardware Architecture for Controlling Information Flow," Proc. Fifth Annual Symp. on Computer Architecture. New York: ACM, 1978, pp. 73-77.
17. R. S. Fabry, "Capability-Based Addressing," CACM, 17(7), 403-412 (1974).
18. M. J. Spier, T. N. Hastings, and D. N. Cutler, "An Experimental Implementation of the Kernal/Domain Architecture," Proc. Fourth Symp. on Operating System Principles. New York: ACM, 1973, pp. 8-21.