Variable-Length Capabilities as a Solution to the Small-Object Problem

Edward F. Gehringer

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Much of this work was performed when the author was at Purdue University, supported by NSF grant GJ-41289

Abstract

A capability system which supports very small objects can achieve flexible and efficient protection. This paper presents a scheme for representing both large and small entities in a computation, down to integers and character strings, as objects. This is achieved by a generalization of tagged memory to encompass extended data types; and by the use of variable-length capabilities, which can be very short if they are close to the object they reference. As developed here, the design assumes a single, systemwide virtual-address space, and a stack architecture; but it could probably be modified for use in other environments.

1. Introduction

Capability systems control access to objects using a ticket-oriented approach. That is, a process is allowed to use only those objects for which it possesses a ticket (called a capability [17]). An *object* is a unit of information to which access may be independently controlled. Let us define a *primitive object* as an object which is not made up of other objects; that is, no capability will refer to merely *part* of the object.

Nearly all capability systems provide a mechanism for *type extension*, which allows the structure of a more complex object to be specified in terms of the primitive objects which make up its representation. This object is then called an *extended-type* object. Capabilities for an extended-type object may be freely distributed, but only a small set of procedures (collectively known as the *implementing domain* of the extended type) are permitted to manipulate its representation.

In deciding how large primitive objects are to be, the system designer is confronted with a dilemma. On the one hand, it is desirable for objects to be very small--on the order of a few bytes or words:

- If each array used by a process is represented as a separate object, run-time subscript checking can be performed by the address-translation hardware without additional overhead.

- To protect a calling procedure from a called procedure, the called procedure must be denied access to the activation record of its caller. This is straightforward if each activation record is a separate object, but not if a process's activation records are all contained within a single object known as the "stack segment".

- In general, programming simplicity is promoted if the representation of an entity is based on its usage, not its size. For example, three-element stacks and three-hundred element stacks should have the same sort of representation; then one set of routines can operate on both.

On the other hand, when objects are small, two deleterious effects may ensue. Since there must be at least one capability for each accessible ("non-garbage") object, the fraction of storage devoted to capabilities grows large as mean object size shrinks toward the size of a capability (the *storage-overhead problem*). In addition, a common implementation is to represent each object as a

separate segment, which may be transferred into or out of main memory independently of other segments. When objects are very small, this is impractical, for serious I/O congestion would result (the *I/O problem*).

Several solutions have recently been proposed for the I/O problem. CAP [18, 15] combines several segments into one swapping unit at compile time; the StarOS operating system for Cm* aggregates small segments into one at linkage time [10]; and other virtual-memory designs [12, 1, 7] do away with segments altogether and allow objects to flow freely across the pages of a paged virtual memory. As this problem is mitigated, the storage-overhead problem begins to dominate. On most systems, capabilities are about two words in length. Since there is at least one capability per object, when mean object length decreases to, say, six words, the fraction of storage occupied by capabilities becomes quite large.

The scheme to be described in this paper minimizes the size of capabilities by employing variable-length capabilities. It allows hardware-implemented entities (such as integers and bit strings) to be represented as objects. Capabilities for the smallest objects can be manipulated in the same ways as capabilities for large objects, enabling hardware-implemented objects and extended-type objects to be handled in very similar ways.

This organization depends on an efficient mapping from capabilities to objects. Because objects are so numerous, we can afford no large table with entries giving the location of each object in the system. Yet it must still be possible to locate an object quickly, given a capability for it. The construction of such a mapping scheme is still a research problem. As one solution, Bishop [1] suggests a paged, single virtual-address space: each object in the system is assigned a unique virtual address, and the unit of I/O transfer is the page, which may hold several objects. A capability contains the virtual address of the object it references, the virtual address serving as the object's unique identifier. The mapping table needs to have only one entry for each page, rather than one entry for each of the much more numerous objects. In this paper, we will assume this sort of paged memory, though any similarly efficient mapping scheme would suffice.

2. Typed Memory

An operating system must guarantee the integrity of capabilities. Programs must not be allowed to modify the unique identifiers (*id*'s) in capabilities; otherwise there would be no protection. Capability systems have traditionally used one of two types of memory organization to safeguard capabilities against modification by users: *tagged memory* or *partitioned memory* [4].

On a system with tagged memory, each value in memory has one or more *tag bits* associated with it. This is usually accomplished by placing a tag field in each word of memory. A particular value of this tag field identifies the word as containing a capability. The tag field is interpreted by the hardware, which will not carry out ordinary data operations on capabilities.

With partitioned memory, all segments are divided into two classes: capability segments and data segments (executable code segments are considered data segments here). Capability segments may contain only capabilities; data segments may not contain capabilities. There is no simple "read" or "write" access right. Instead, there are "read- and write-data" and "read- and write-capability" access rights. These access rights apply only to segments of the corresponding type. No capability for a data segment will have read-capability or write-capability enabled, and vice versa. In this way, the system prevents data from being interpreted as capabilities.

The use of either tagged or partitioned memory forces an inconvenient tradeoff. If the addressable unit is large, one can afford a few tag bits in each word. If the byte or even the bit is the unit of addressing, it is out of the question to associate tag bits with each unit. Partitioned memory avoids this kind of storage overhead, at the cost of exacerbating the I/O problem. Without tagging, the need to separate data and capabilities requires the use of many more segments than in a non-capability system. Worse still, most of these segments are small [15], leading to the I/O congestion described above.

With an extra control register, data and capabilities can be kept in the same segment. The extra register points into the middle of the segment, segregating capabilities on the one side from data on the other [11]. The I/O problem is mitigated, but the need to separate data and capabilities remains, and occasionally forces an awkward treatment of data structures:

- The usual method of parameter passing is the use of a call/return stack. A partitioned memory system requires two such stacks, one for data and one for capabilities.

- In certain data structures, the separation of data and capabilities is difficult to mask by programming-language constructs. For example, it would be quite awkward to design a deque to hold both data and capabilities.

This tradeoff can be circumvented by a variant of tagged memory first suggested by Iliffe [8] and developed further by Jagannathan [9] and Myers [13, 14]. This method takes advantage of the fact that most memory is occupied by sets of one sort or another--arrays of integers, character strings, arrays of machine instructions, and so forth--and that associating a tag with each array element is very wasteful, since within the array the tag values will all be the same. Instead, the first cell (e. *g.*, byte) within the array can be used as a tag. A capability for the array will point (directly or indirectly) to its first cell. Protection is guaranteed if no machine instruction can ever treat the first location within an object as data. Toward this end, an offset of zero may be treated as out-of-bounds, or more likely, may be automatically interpreted to refer to the second cell within the array.

With this sort of tagging, a minimal amount of memory is required to hold the tags. If we assume a mean object size of six words and a one-byte tag, the storage overhead is 1.25 bits per word, which is less than the three bits per word used on the B6700. This expenditure can be balanced by the saving occasioned by having smaller operation codes (as Iliffe proposed for his Basic Language Machine).

It is easy to generalize this tagging scheme to apply to extended-type objects. Each extended-type object normally carries some designation of its type, such as the *id* of its implementing domain. This designation is usually kept in the capabilities for the object, or in the descriptor of the object.[1] It could instead be stored in the object itself. The type field within the object then serves exactly the same function as the tag field in an elementary data object, so that the two objects can be represented in an analogous way: as a tag/type field followed by a representation. Indeed the analogy between memory tags and data types is so strong that any capability system which does not use tagged memory could be said to make an unnecessary distinction between the representation of hardware-implemented and software-implemented data types.

Seen from this perspective, both tagged and partitioned memory are restricted cases of something more general. In traditional tagged memory, tag bits

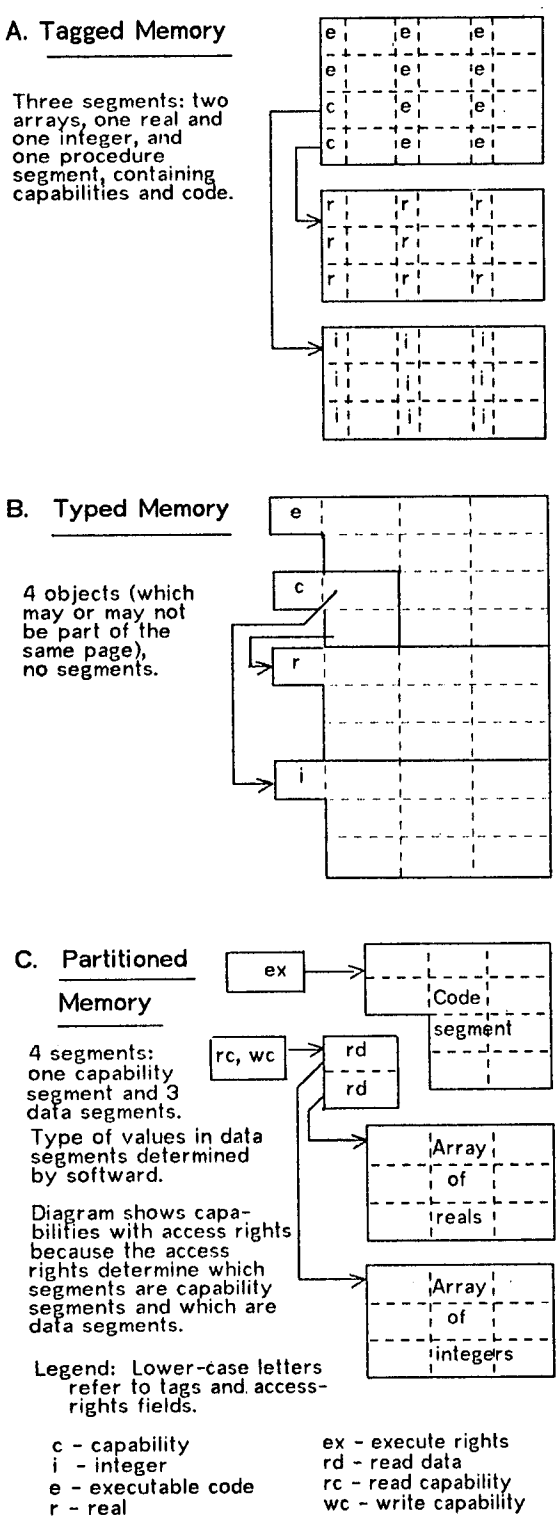


Figure 1: Tagged, Typed, and Partitioned Memory Compared

[1] We distinguish here between capabilities and descriptors. There may be many capabilities for a given object, and these may have different access rights to the object. Certain attributes of the object, such as its type and size, do not depend on which capability is used to access the object. Rather than being duplicated in each capability, this information is thus kept in a single table entry for the object, called the object's *descriptor*. This usage of the term "descriptor" corresponds to that of StarOS [11], but not to that of Multics or the B6700.

are associated with each word (see Figure 1). As we have just noted, the requirement that each word be separately tagged is an unnecessary restriction. If it is removed, memory is occupied by objects, each of which has only one type designation. An object is either a primitive object (which is implemented in hardware), or else it is a set of one or more other objects. This organization, using type designators, one per object, will be known as *typed memory* to distinguish it from traditional tagged memory. Tagged memory is thus a restricted case of typed memory, the restriction being that each tag must apply to exactly one memory cell.

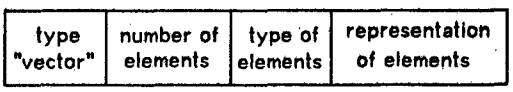
2.1. Vectors and Records

As we have seen, typed memory associates a type field with each object. If the object is a single elementary object, like a floating-point number or a boolean value, its format looks like this.

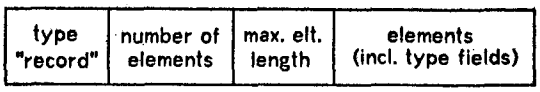
Objects can be aggregated in *sets* of two kinds. A *vector* is a set of objects of the same type. A single type field gives the type of all objects in the vector. A *record* is a set of objects of mixed types; thus each element within a record has its own type field. Examples of records are procedure activation records (in an Algol-like language they contain local variables of various types and capabilities for arrays) and sets of parameters passed to procedures. Records are also among the constituents of most extended-type objects.

The vector and the record are themselves data types. As such, they have their own type fields which prefix their representations. The exact format of their representation is a design decision; we shall now proceed to outline one possible approach.

Immediately following the type field is a field which indicates the number of elements in the vector or record; and, in the case of a record, another field specifying the maximum element length, in bytes. A vector, then, is represented in this fashion:



A record has this representation.



The vector and the record should be thought of as data types whose sole function is to affect the size of the object whose representation follows immediately.[2] Strictly speaking, the vector data type is represented by a type field followed by a size field, which indicates the number of elements in the vector. The record data type is represented by a type field, followed by a number-of-elements field, followed by an element-length field.

To determine the location of individual elements of a vector (in this implementation), the hardware uses the length of a single vector element. Thus if objects of a certain type are of variable length, it is impossible to have a vector of that type of object. If the maximum length of that object type is known, it is still possible to have a *record* composed entirely of objects of that type. To provide for the case where the maximum length of an object of a certain type is not known, or where it would be too wasteful to reserve the maximum amount of storage for each object within a record, we could provide another type of record: a record which contains a list of simple pointers to its elements instead of a maximum-length field. Such a solution was rejected in favor of an alternative: a record of capabilities pointing to each of the elements (see Figure 2). The variable-length capabilities which will be introduced in the next section are hardly larger than simple pointers would be, and using them avoids the need for introducing a new mechanism.

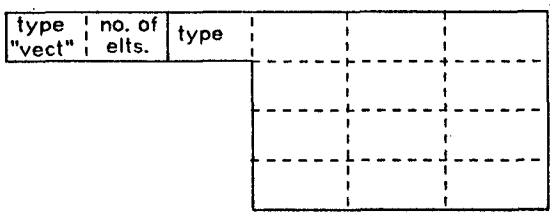
2.2. Extended-Type Objects

An extended-type object is created from its representation; that is, from the objects which form its constituent parts. A process which holds a capability for the object may call the implementing domain to operate on the object, but it may not access the representation in any other way. Only the implementing domain may manipulate its representation.

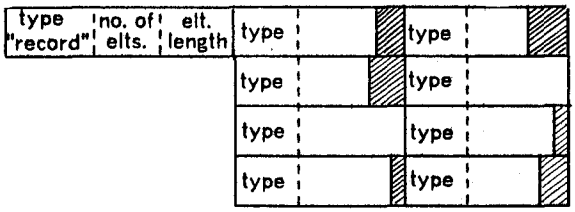
---

[2]The vector data type provides automatic bounds checking and index calculation for one-dimensional arrays. The architecture could easily be extended by adding a more general array data type in hardware [5, 14, 20], to allow automatic bounds checking and index calculations for multi-dimensional arrays. The choice is a tradeoff between programming convenience and error detection on the one hand, and hardware complexity and the space required to store bounds or stride information on the other.

1. For elements of the same type and of fixed length.



2. For objects whose maximum length is known, and where not too much space is wasted by reserving the maximum length for each object.



3. For objects where it is impractical to reserve the maximum length for each element.

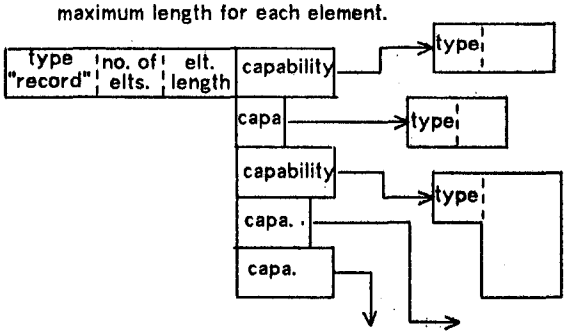


Figure 2: Three Ways of Representing a Set of Objects

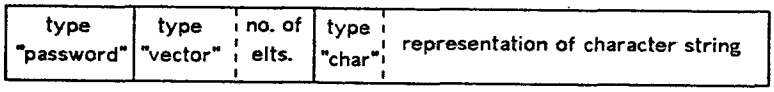
Typed memory differs from ordinary capability systems in the way that it safeguards the representation of extended-type objects. In conventional systems, the type field in the capability or the descriptor is changed from the type of the representation to the type of the extended-type object. This operation is called "sealing" the capability.[3] In typed memory, it can be performed simply by prefixing to the representation of the object a type field which designates the extended type.

Figure 3 illustrates a password whose representation consists of a character string (that is, a vector of characters). Its representation is immediately preceded by a type field which specifies "password" as the type of the object. If a capability for the password is passed to another domain, that domain will not be able to read the password. If the other domain attempts to use a hardware instruction to operate on the password, it will fail by type conflict, since hardware operations are defined only on elementary data types such as integers, reals, and characters. The domain cannot remove the "password" type field to obtain a character string (which it could operate on) except by calling the implementing domain and passing a capability with the necessary access rights. The representation of the password is thus protected from manipulation by unauthorized domains.

A less trivial example is provided by a "stack" data type. It consists of an array which holds the stack elements, and an integer which is used as a stack pointer. The array and the pointer form a record, which is the representation of the stack. Since the array is much longer than the pointer, it would not be worthwhile to represent the stack as a record of fixed-length elements. Instead, we use a record of two capabilities pointing at the array and the stack pointer (see Figure 3). At first glance it may appear

[3] A more complete description is given by Redell [16].

A password, consisting of a character string.



A stack, consisting of an array of reals and a stack pointer.

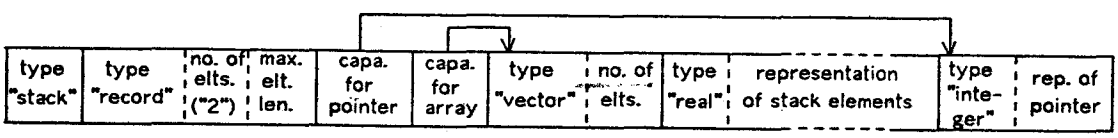


Figure 3: The Representation of Extended-Type Objects

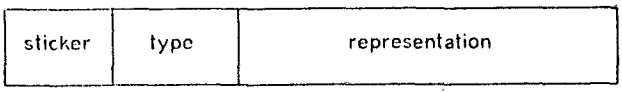
that this is an inefficient way to represent a stack since so many fields of control information are needed. As we shall discover in the next section, only one of those fields--the "stack" type field, is likely to be more than 16 bits long.

3. Variable-Size Capabilities and Type Fields

The design of capabilities and objects follows two guidelines: (1) the first few bits of an object (*i.* e., the type field) must indicate whether it is a capability or some other type of object, and (2) type fields and capabilities should be made as short as possible by using variable-length coding.

Thus at the beginning of an object, we assign a very short field (3 bits) to distinguish between capabilities and other types of objects. Let us call this the *sticker* field. (Just as tags and stickers are used in the physical world to identify objects by type and price, we use them to classify objects in memory. The function of the sticker field is slightly different from a tag field in ordinary tagged memory; hence the need for a new term.) A sticker of 0 to 5 indicates a capability, 6 means the object is another hardware-implemented object, and 7 means it is an extended-type object. If the object is a capability, then, its sticker completely determines its type. If the object is not a capability, another field is necessary to establish its type. So at this point, a capability consists of a sticker followed by a representation, and other objects consist of a sticker followed by a type field followed by the representation (see Figure 4).

A capability (simplified diagram)



An object which is not a capability (simplified diagram)

Figure 4: Simplified Object Diagrams

Conceptually, the type field we referred to in earlier sections has now been split into two parts: the sticker, and the new "type" field (which is not present, however, in capabilities). The reason for doing this, of course, is to minimize the size of capabilities, which are expected to be the most numerous objects in the system, since they are used to manipulate all other objects.

The largest field in a capability is typically the *id* (or address) field. The key to reducing its size is to appeal to the principle of locality. Since most capabilities within objects tend to reference nearby objects, if we arrange it so that capabilities for nearby objects are short, then most capabilities will be short. One idea which suggests itself is to have the address field specify the signed distance between the capability and the object it references. This is certainly not a totally new idea, since relative addresses have been used for years in the destination field of jump instructions on several computers, such as the PDP-11 [19].

The approach we select is slightly different. In a single address-space system, a byte address is several dozen bits long. A forty-eight bit address seems reasonable for a single virtual-address system. That is the address size employed on the IBM System 38 [7], and is only slightly larger than the 34-to-44 bit address suggested by Bishop, who postulated a word-addressable memory. We desire some way to name an address without specifying all of its bits. This can be done by taking most of the bits from the memory-address register as the capability is fetched. If a capability is near the object it references, then the address of the capability itself and the address of the object will differ only in their last few bits. The capability need contain only the least-significant bits of the address.

This is the reason that sticker values from 0 to 5 all mean "capability". The six possible values correspond to address (*id*) fields of 0, 8, 16, 24, 32, and 48 bits in length.[4] Note that the unique *id* specified by the capability can always be expanded to a unique 48-bit virtual address. Hence two capabilities with different lengths may nevertheless refer to the same object.

This scheme has two advantages over relative addresses. (1) It requires less complex hardware, since a virtual address is formed by concatenation of two bit strings rather than by addition of two binary integers, which simplifies address calculation. (2) Capability systems typically use garbage collection to remove inaccessible objects. This is especially necessary in a single virtual-address space system, where address space cannot be reused until it is

[4] A zero-bit address field means that the address of the object is the same as the address of the capability; in other words, that the representation of the object immediately follows the capability for it. This feature will be exploited in Section 4 to obtain more compact data representation and greater run-time efficiency.

assured that no capabilities for old objects point into it. In a large computer system, garbage collection must be done in small increments, a portion of on-line storage at a time, so that it is not necessary to suspend processing in the rest of the system. Garbage collection assigns new addresses to objects. The capabilities throughout the system for these objects will then have to be updated. Garbage collection of one area of the address space must not cause capabilities outside the area to expand (see Figure 5); otherwise those external capabilities could not be updated in place. Properly designed concatenated addresses can satisfy this requirement; relative addresses cannot.

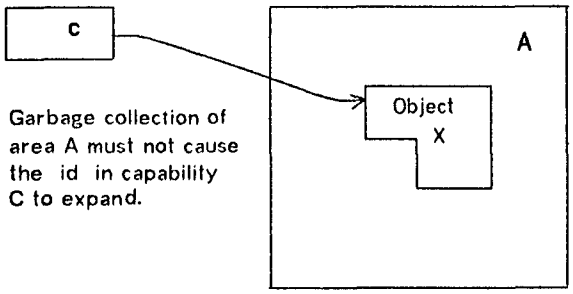


Figure 5: Garbage Collection and the Length of Capabilities

To see why concatenated addresses will not expand during garbage collection, consider that the various sizes of address fields in capabilities define *neighborhoods* in the address space: we say that two addresses are in the same *m-bit neighborhood* if they differ only in their *m* least significant bits. Thus $003B00_{16}$ and $003C00_{16}$ would be in the same 16-bit neighborhood but not in the same 8-bit neighborhood.

We further define the *order* of a neighborhood as follows: the neighborhood defined by the smallest possible address field (0 bits) is of order 0; the neighborhood defined by the next smallest address field (8 bits) is of order 1, and so forth. Thus in our example, a 48-bit neighborhood (which includes all of the address space) is of order 5.

An area is *safe* if garbage collection of it cannot cause expansion of addresses external to it. It turns out that a safe area *A* consists of the union of one or more neighborhoods of order *k*, all of which are contained in a single neighborhood *N* of order *k*+1 (see Figure 6). This is true because all capabilities external to *N* which point to objects in *A* point to objects in a different neighborhood of order *k*+1. Now

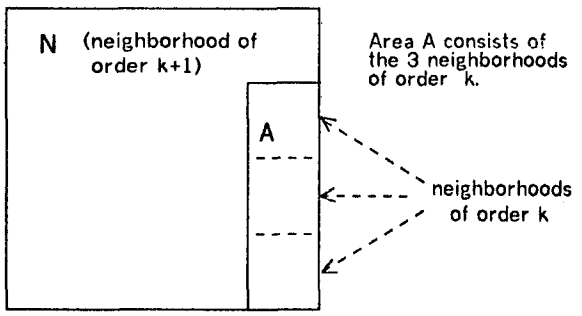


Figure 6: A Safe Area

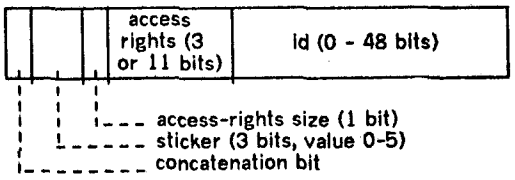
assume *A* is compacted by garbage collection. Capabilities outside of *N* still point to objects in the same neighborhood of order *k*+1 as before; hence the length of their address fields need not expand. Before garbage collection, capabilities in *N-A* which point to objects in *A* point to objects which are in a different neighborhood of order *k* but in the same neighborhood of order *k*+1. After garbage collection compacts *A*, these capabilities still point into the same neighborhood of order *k*+1, but into a different neighborhood of order *k*. Thus addresses in these capabilities do not expand either, and so no addresses external to *A* expand as a result of garbage collection. (For a more complete treatment of this matter, see [6].)

Relative addresses do not provide safe areas. Imagine that an object *x* is referenced by two capabilities outside its area, one of which contains the largest relative address which will fit in its address field, and the other which contains the largest-magnitude negative relative address which will fit. If the compaction phase of garbage collection moves *x*, the magnitude of the address in one of the two capabilities will expand, and thus one of the two address fields will have to be expanded.

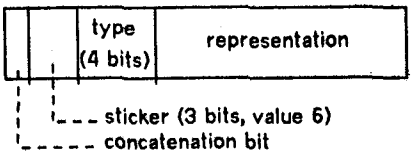
Concatenated addresses allow short capabilities for most user-defined objects. In some systems, the most commonly used system routines may be found at low virtual addresses. Concatenated addresses would not save much in this case, since the addresses of system routines would be far away from most user objects which used them. To optimize for this situation, to each capability we can add a bit (called the *concatenation bit*) which specifies whether the *id* is to be interpreted as a concatenated address or an absolute address. If it is an absolute address, then it will be extended on the left with zeros as it is expanded to 48 bits. This allows short capabilities for

system routines as well.

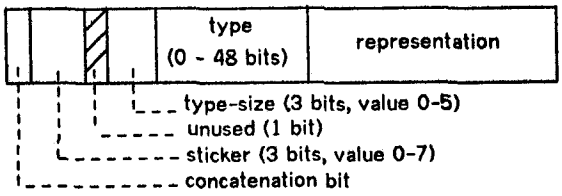
We have now described all fields of a capability, except for the access-rights field. Small hardware-implemented objects, such as integers, have only two operations (read and write) defined on them. Extended-type objects may require much larger access-rights fields. Consequently, the access-rights field also has variable size. One *access-rights size* bit specifies whether the access-rights field is three or 11 bits long. Figure 7a shows the complete structure of a capability.



a. The structure of a capability



b. The structure of a hardware-implemented object



c. The structure of an extended-type object

Figure 7: The Structure of Objects

For objects other than capabilities, the sticker along with another type field determines the object type. For hardware-implemented (sticker-6) objects, this type field is four bits long and immediately follows the sticker. Table 1 suggests 16 elementary data types.[5] Thus the type of a hardware-implemented object is completely specified by 7 bits (see Figure 7b). The concatenation bit is unused; or alternatively it might be incorporated into the type field to give a maximum of 32 hardware-implemented data types.

Table 1: Hardware-Implemented Data Types

Type Field Value	Data Type	Type Field Value	Data Type
0	integer	8	sparse vector of
1	short integer	9	single-linked list of
2	real	A	double-linked list of
3	double-precis. real	B	static link
4	character	C	descriptor
5	Boolean (bit)	D	procedure
6	vector of	E	executable code
7	record	F	semaphore

We wish to allow an essentially unlimited number of software-implemented object types. These are identified by a sticker of 7. Conceptually we provide a 48-bit type field; in practice, its size can usually be much shorter if we adopt the same concatenation convention used for capabilities. A three-bit *type-size* field tells how long the type field is (see Figure 7c), and the concatenation bit indicates whether the most significant bits are taken from the memory-address register or set to zero.

The value of the type field for an extended-type object could be the virtual address of the implementing domain. This allows short type fields for software types defined at low levels of the operating system. The concatenation bit will be set to "absolute" and the type field will reference a low virtual address. Similarly, an object of a user-defined type can have a short type field if it is close to its implementing domain. This might happen, for example, when a program defines a new type of data structure for its own private use.

The prefix information (concatenation bit, sticker, type-size field, and type field) for a software-implemented object can be as short as 16 bits (if the type field is 8 bits long). The prefix information will never require more than 56 bits. The smallest possible capability is 8 bits long. It has a zero-bit ID field and a three-bit access-rights field. The longest possible capability is 64 bits long, with an 11-bit access-rights field and a 48-bit ID.

[5] A similar table is found in Feustel [5].

4. Activation Records

Whenever the set of objects accessible to a process changes, a *domain switch* is said to occur. If we desire each procedure call to induce a domain switch, then each activation record must be a separate object. As its name implies, the activation record for a procedure is implemented as a record. It is created when the procedure is called, and is destroyed at the return. Contained within it are capabilities for all of the local-workspace objects (such as arrays, local variables, and compiler temporaries) of the procedure activation.

To be general, an activation record should be able to hold a capability for any object. This means it has to be able to hold the largest of the variable-length capabilities; in other words, its maximum element length must be at least 64 bits. As we shall see, 64 bits is sufficient to hold a capability for a small object such as an integer or a real number plus its representation. If the object is larger, a capability for it can be placed in the activation record and the object can be located elsewhere, so the maximum element length need be no longer than 64 bits. The number of elements in an activation record is initially the number of parameters to the procedure, plus the number of objects declared locally. If a stack architecture is employed, it will grow as further elements are pushed onto the stack. A capability for the activation record of the executing procedure is always held in a processor register known as the W-register (because it points to the procedure's *working storage*.) In addition to the capability, the W-register contains a length field which gives the number of elements in the activation record. This field serves as the descriptor[6] for the W-record. The activation record, which is pointed to by the W-register, is also known as the W-record.

As in conventional implementations, the activation records associated with a process form a stack. In this capability-oriented implementation, each activation record is a separate object. When a procedure call is about to take place, a capability for the current W-record is deposited on the stack. Then the activation record is temporarily extended as parameters are pushed onto the stack. The call instruction adjusts the W-register to point to a new activation record which begins at the first parameter, thus denying the called procedure access to the activation record of its caller. The return instruction restores the capability for the caller's W-record, extending that record by 1 element if the called procedure has returned a result.

As items are pushed onto the top of a record, the size of the record is increased accordingly. When an item is popped from a record on the stack, the size of the record is decreased by one element. If an attempt is made to pop an item from an empty record, a trap occurs.

Conceptually, an activation record contains capabilities for the objects created upon procedure entry. To permit the efficient manipulation of small objects such as integers, real numbers, and short character strings, they can be stored immediately adjacent to the capabilities for them, as shown in Figure 8.

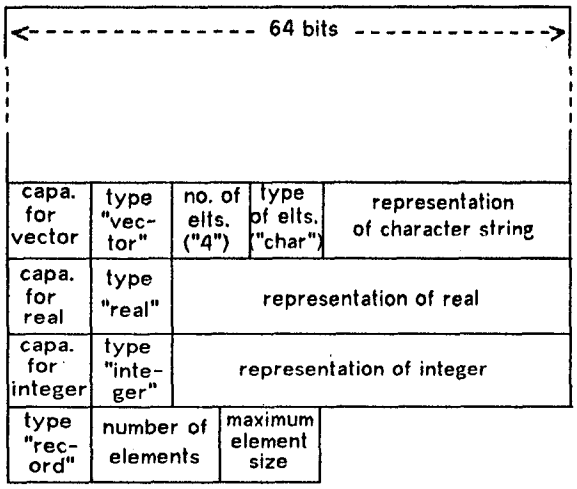


Figure 8: An Activation Record, Showing Capabilities and Objects in Packed Format

This is known as storing an object and a capability in *packed format*.

When a capability is stored in packed format it has an ID field of zero bits. This is interpreted as meaning that the object immediately follows the capability for it.[7] Unless an element has more than 3 access modes, a packed-format capability for it will consist of 8 bits. The sticker and type field for an elementary

[6] Recall that, in general, a descriptor holds information about an object which need not appear in every capability for the object (see footnote 1 in Section 2).

[7] A strict interpretation of the rule for determining the value of the ID field would mean that the capability referred to an object stored at the same location as the capability, or, in other words, that the capability referred to itself. A zero-bit ID field is thus interpreted in a special way, so that it refers to an object whose address is one or two greater than that of the capability (depending on the setting of the access-rights size field).

object occupy another 8 bits, which leaves 48 bits for the representation of an object. (Recall that an element in an activation record has 64 bits.) This is sufficient to represent integers and real numbers, and character strings of length 3 or 4 (allowing for the vector-length field and the vector and character type fields).

This call/return design provides automatic domain-switching whenever a procedure call occurs. The called procedure has no capability for the activation record of the caller, so it can manipulate only those objects which have been passed to it as parameters. At the same time, activation records are kept in a stack, which allows them to be created and deleted without undue overhead. Further, capabilities are kept in the same stack with other objects, which simplifies addressing and storage management.

5. Descriptors

When resident in memory, a capability may be anywhere from eight to 64 bits in length. In order to simplify the logic of the instructions which deal with capabilities, we specify that when a capability is loaded into a processor register or placed onto the stack, it is expanded to its 64-bit form. The address is extended on the left with zeros or from the memory-address register, depending on the setting of the concatenation bit; and the access-rights field (if short) is extended on the right with zeros (denying the 4th through 11th modes of access). This is called the *evaluated* form of a capability.

To avoid the need for a special descriptor table giving the attributes of each object, we have located type and size information with objects themselves. In effect, this means that an object's descriptor immediately precedes its representation. The descriptor must be consulted when an object like an array or a record is referenced, to prevent accessing outside the bounds of the object. It is also needed when an object is stored, to prevent storing a longer object in the space formerly occupied by a shorter object, which might destroy the representation of an adjacent object and cause capabilities for it to point into the middle of the new object.

It would be possible but not very efficient to access the descriptor for an object at each use of a capability for the object. The need to access main memory could be circumvented in either of two ways: by caching descriptors, or by providing special instructions. The system might possess a general-purpose cache for recently accessed data, or it might furnish a special cache for descriptors, possibly with hidden registers.

The other approach would provide an instruction for loading a descriptor--either into a register or onto the stack, depending on the architecture.[8] Just as in the case of a capability, the load operation would convert the descriptor to a standard form. It would have a special type field identifying it as a descriptor; and in the case of a compound object such as a vector or a record, it would somehow indicate the length of each element, so that the hardware could determine the address of any particular element. For example, the descriptor for a record might look like this:

type "descrip-tor"	type "record"	number of elements	length of elements

It is also possible to manipulate a descriptor and a capability simultaneously. When an object and a capability are in packed format (see Figure 8), these are found in close proximity. A special instruction could load both the capability and descriptor in a single memory reference. Other instructions could also be designed to take advantage of the fact that a small object and a capability for it can be read or written in one memory access.

Even when loaded into a fast register, an object such as an integer retains its type field. One should keep in mind the distinction between this type field and the descriptor which is required to modify the object in main memory. The type field is associated with the representation of the integer and the descriptor is associated with the virtual address from which the integer came. It is possible to use a type-conversion operator to convert the integer to a double-precision real with the same value, but it is not then possible to use the capability and descriptor to store the double-precision real back into the same location from which the integer came. It would not fit; the hardware would detect this by comparing the type field of the object with the descriptor, and would abort the request.

This design achieves a symmetry between hardware- and software-implemented objects. Capabilities for both are created only when the respective objects are created. A hardware object is created by a machine instruction (such as an arithmetic operation), which also creates a capability in packed format at the same time. A software object is created by a "seal" operation applied to its representation. The seal operation returns a capability for the new object.

[8] This method, like a cache, allows the copying of a descriptor. But such a descriptor copy could exist only in fast registers; it could not be copied back into main memory. Special provisions would have to be made to guarantee integrity in the case of a multiprocessor.

Once created, capabilities can be passed anywhere in the system, which allows for a very fine-grained protection. For example, the ability to read or write an arbitrary word of memory can easily be controlled. When a request for any operation is issued, type-checking is performed by the implementing domain. In the case of hardware-implemented objects, the implementing domain is hardware or firmware, which carries out type checking in the same fashion as a traditional tagged machine. For software-implemented objects, type-checking is done by the software which defines the object just as in other capability systems.

6. Performance

The proposed design has many similarities to the large-scale Burroughs machines (5500, 6700, and their successors). Both designs manipulate 48-bit arithmetic quantities; however on the B6700 an element on the stack is 51 bits (including three bits for the tag, and not counting the parity bit), while a stack implementation of the new design would use 64 bits. (A register implementation of it would demand correspondingly longer registers.) This represents a memory overhead of about 25% in each stack frame for the capability design. Stack frames are typically quite small though; a study of Algol programs [2] found them to average 13 words in length, so the impact of the proposed design on stack size would probably be unimportant. The same holds for arrays: the prefix information for an array occupies 8 + 24 + 8 = 40 bits (assuming a 24-bit field to specify the number of elements in an array). The median array size in the Brundage study was 30 elements, and the mean much greater than that, so the prefix information represents an overhead of no more than about one bit per element, which is even less than the three tag bits used on the Burroughs machines.

The necessity for frequent handling of capabilities does create new demands on the hardware and firmware. Memory bandwidth must be at least 64 bits, but this is becoming more practical with advancing technology. Several more fields have to be extracted and tested in the course of each memory access; this complicates the microcode but does not necessarily slow it down, since the opportunity for parallelism exists (access rights can be tested as a memory access is initiated, and the access aborted if found to be unauthorized, for example).

How much space would be saved by variable-length addresses in capabilities? The list structure of Lisp approximates an extreme case of a small object system, if one considers the atoms as objects and the pointers as capabilities. In a study of Lisp programs, Clark [3] found that 78 to 84 per cent of list pointers pointed to a location within their own 512-word DEC-10 page. The results are not directly applicable because DEC-10 memory is addressable in 36-bit words while we postulate a byte-addressable memory, and because we do not expect most objects to be quite as small as Lisp atoms. Nonetheless, Clark's data indicates that 16-bit addresses may be large enough for a clear majority of capabilities. Furthermore, depending on the program, from 71 to 84 per cent of list pointers were found to point to locations no more than 31 words away (that is, within a 63-word, or approximately 300-byte, "window" about the pointer). This suggests that an 8-bit address field, capable of accommodating values up to 255, may often suffice.

This discussion is necessarily speculative. The hardware design itself can be expected to influence the size of arrays and procedures, for example, so projections based on existing program samples may be invalid. The design inherits most of the complexities of Bishop's single large virtual-address space, unless another similarly efficient means of mapping capabilities to objects can be found.

7. Conclusion

This paper has advanced a new organization for capability systems, based upon a form of tagged memory and variable-length capabilities. It achieves a symmetry between hardware- and software-implemented objects, and is amenable to fine-grained protection and rapid domain switching. The presentation has assumed a single virtual-address space, but the basic principles could be applied to different designs, provided there is an efficient way to map capabilities to objects.

Acknowledgment

Robert S. Fabry provided detailed comments which helped improve the clarity of the presentation.

References

[1] Bishop, Peter B.
Computer systems with a very large address space and garbage collection.
PhD thesis, Massachusetts Institute of Technology, May, 1977.
Available as MIT LCS TR-178.

[2] Brundage, Robert E.
A study of process behavior in virtual computer systems.
PhD thesis, University of Virginia, May , 1974.

[3] Clark, Douglas W.
List Structure: Measurements, Algorithms, and Encodings.
PhD thesis, Carnegie-Mellon University, August, 1976.

[4] Fabry, Robert S.
Capability-based addressing.
Communications of the ACM 17(7):403-12 July, 1974.

[5] Feustel, Edward A.
On the advantages of tagged architecture.
IEEE Transactions on Computers C-22(7):644-656, July, 1973.

[6] Gehringer, Edward F.
Functionality and performance in capability-based operating systems.
PhD thesis, Purdue University, May, 1979.

[7] Houdek, Merle E. and Mitchell, Glen R.
Translating a Large Virtual Address.
In *IBM System/38 Technical Developments*, pages 22-24. IBM General Systems Division, 1978.
0-933186-00-2.

Iliffe, John K.
Basic machine principles.
American Elsevier, Inc., New York, 1968.

[9] Jagannathan, Anand.
An implementation model for a multiprocessor operating system on a descriptor oriented architecture.
Master's thesis, Rice University, July, 1976.

[10] Jones, Anita K. and Schwans, Karsten.
TASK Forces: Distributed Software for Solving Problems of Substantial Size.
In *Proceedings of the Fourth International Conference on Software Engineering*, pages 315-330. IEEE Computer Society, Munich, Germany, September 17-19, 1979.

[11] Jones, Anita K.; Chansler, Robert J., Jr.; Durham, Ivor; Mohan, Joseph; Schwans, Karsten; and Vegdahl, Steven.
StarOS, a Multiprocessor Operating System.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM/SIGOPS, Pacific Grove, California, December 10-12, 1979.

[12] Lomet, D. B.
Scheme for invalidating references to freed storage.
IBM Journal of Research and Development 19(1):26-35, January, 1975.

[13] Myers, Glenford J.
The design of computer architectures to enhance software reliability.
PhD thesis, Polytechnic Institute of New York, December, 1977.

[14] Myers, Glenford J.
Storage concepts in a software-reliability-directed computer architecture.
In *Proceedings of the Fifth Ann. Symposium on Computer Architecture*, pages 107-113. ACM/SIGARCH, April 3-5, 1978.
ACM Computer Architecture News, 6:7.

[15] Needham, Roger M.
The CAP project: an interim evaluation.
In *Proceedings of the Sixth Symposium on Operating Systems Principles*, pages 17-22. ACM/SIGOPS, Purdue University, November 16-18, 1977.
ACM Operating Systems Review, 11:5.

[16] Redell, David D.
Naming and protection in extendible operating systems.
PhD thesis, University of California, Berkeley, September, 1974.
Reprinted as Project MAC TR-140, Massachusetts Institute of Technology.

[17] Saltzer, Jerome H. and Schroeder, Michael D.
The protection of information in computer systems.
Proceedings of the IEEE 63(9):1278-1308, September, 1975.

[18] Slinn, Christopher J.
Aspects of a capability-based operating system.
PhD thesis, University of Cambridge, February, 1977.

[19] Strecker, W. D.
VAX-11/780--a virtual address extension to the DEC PDP-11 family.
In *National Computer Conference, Proceedings, Vol. 47*, pages 967-80. AFIPS, 1978.

[20] Tanenbaum, Andrew S.
Implications of structured programming for machine architecture.
Communications of the ACM 21(3):237-246, March, 1978.