A capability representation with embedded address and nearly-exact object bounds

Jeremy Brown J.P. Grossman Andrew Huang Thomas F. Knight, Jr.

Abstract

We present a capability format which improves upon prior capability formats by simultaneously providing four key features. First, address and bounds information are embedded directly in the capability representation. Second, a capability may point to an arbitrary word in its segment. Third, internal fragmentation due to segment/object size mismatch is less than 6%; with a simple, high-locality allocation scheme, total fragmentation is less than 12%. Fourth, objects of 32 or fewer words, e.g. most class instances in object-oriented systems, may be allocated with no fragmentation and will thus have precise hardware bounds-checking. These features make it entirely practical to use a capability-guarded segment per allocated object, thus ensuring robust inter- and intra-program memory protection. Additionally, we describe the implementation and application of increment-only pointers which enable precise hardware-only bounds-checking for Java-style objects/arrays. Finally, we also demonstrate how to generate capabilities for sub-segments from which the enclosing segment can be recovered by system routines.

1 Introduction

Fine-grained capability-addressing systems offer significant data-integrity advantages over conventional computer architectures. A capability is a hardware-recognized token which denotes a region of memory – a segment – as well as some permissions controlling the operations that may be performed on that segment. Since capabilities are distinguishable from other data, it is always dynamically detectable when a program accidentally confuses them, e.g. by trying to treat an integer as a pointer. In a fine-grained capability system, each object may reside in its own segment; this dramatically reduces the incidence of undetected intra-program problems such as buffer overruns. At the inter-program level, per-object protection eliminates the need for per-process virtual address spaces; this both eliminates the need for expensive TLB and cache flushes during context-switches, and enables data sharing between programs at the granularity of individual objects.

These advantages have traditionally been offset by the difficulty of efficiently representing capabilities, and in particular efficiently representing object bounds. In many capability systems, a microcoded capability-dereferencing operation performs multiple memory references to verify object bounds; this overhead is unacceptably high in modern RISC architectures. A few capability representations include embedded object, or more precisely, *segment*, bounds in the capability word itself. This approach enables pipelined bounds-checked operations on object references, but the schemes proposed to date suffer either from the requirement that a capability must point to the first word of a segment, or else from a high degree of fragmentation due to segments which may be much larger than the objects they contain.

Our core contribution in this paper is a format for embedded segment bounds which overcomes both of these limitations. We use a floating-point size field to maintain a close relationship between segment and object sizes (within 6%); we use a special *finger* field to enable a capability to point to an arbitrary address within its segment.

In addition to our basic capability format, we introduce increment-only capabilities with which precise bounds-checking may be implemented for languages such as Java. We also present a method for generating capabilities describing sub-segments from which the original segment may be reconstructed; sub-segments are thus compatible with segment-relocation due to, for instance, compacting garbage collection.

The remainder of this paper is structured as follows: in Section 2, we describe some important points in the history of capability formats. In Section 3, we present our capability representation, describe the function of each field, explain how to perform important operations, and discuss hardware implementation issues. In Section 4, we describe increment-only capabilities and their appli-

Project Aries Technical Memo ARIES-TM-005
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA USA

Research performed under
DARPA/AFOSR Contract Number F306029810172

cations. In Section 5, we explain how to generate subsegment capabilities from which the original segment can be reconstructed. Finally, in section 6 we conclude with a brief discussion of applications of, and variations on, our capability format.

2 Prior capability representations

The idea of using *capabilities* for addressing first appeared formally in a paper by Fabry ([Fab74].) In concept, each capability specifies a segment of memory and a set of permissions detailing the operations permitted upon that region. Fabry's scheme represents each capability as a set of permissions and a segment-identifying unique ID. A hash table keyed on UIDs contains segments' base and bounds information; a small associative cache preserves mappings for the most recently accessed segments. Fabry's approach is thus efficient when using small numbers of segments but inappropriate to a system in which every independently-allocated object is placed in its own segment.

A number of early capability systems used centralized tables mapping from capabilities to segment addresses; several are described in [Lev84]. While some of these machines segregate non-capability data from capabilities in distinct segments, the evolutionary trend is clearly toward tagging capabilities in a hardware-recognizable fashion so that they may be freely intermingled with other data.

Most modern capability systems have avoided the centralized table by embedding object addresses directly in the capability representation. We will not attempt to list them all here, but rather to provide a small number of examples representing key points in the design spectrum.

The Symbolics 3600 Lisp Machine architecture ([Moo85]) features per-object pointers which directly reference the first word of their target objects. Rather than encoding bounds information in these pointers, however, the object size is encoded in the object's first word; performing bounds-checking on object references requires an extra memory reference. [1]

The M-machine [FKD$^+$95] uses a guarded pointer format described in [CKD94] to encode permissions, an address, and a segment length descriptor L. The upper bits of the address are fixed, while the lower L bits are mutable by pointer arithmetic; the segment is thus of size and alignment 2^L, and the capability may point to any address within it. This scheme requires hardware-recognized (i.e. tagged) capabilities, and a small amount of hardware support in the processor to verify bounds information in parallel with performing pointer arithmetic. Notably, it requires neither indirections through a central table, nor the embedding of size information in an object representation.

Two disadvantages of this scheme both arise from the exponentiated segment sizes. First, for an object only slightly larger than some power of two, nearly half of the segment allocated to hold it will be unused; this internal fragmentation both wastes space[2] and means that accesses which overrun the end of the object will not immediately be detected as they will still fall into the segment.

Second, due to the strict alignment requirement, a simple allocation strategy based on advancing a pointer may wind up wasting nearly half of memory due to external fragmentation. Together, external and internal fragmentation combined could, in the worst case, waste nearly 75% of memory. On the other hand, space-efficient "buddy-system" allocators may place consecutively allocated objects nowhere near one another in memory. This failure of locality is unfortunate: "...several studies have shown that objects should be laid out in the heap in such a way that objects that refer to each other, or are related in some other way, are placed in close proximity in order to reduce the size of the program's working set. There is considerable evidence that allocation order is a good indicator of such a relationship between objects..."([JL96], pages 113-114.)

An alternate set of tradeoffs appear in the ORSLA capability-system described in [Bis77]. In ORSLA, a capability contains (in addition to permissions bits) the address of the first word of an object, and a small size field (5-9 bits.)

The first bit of the size field distinguishes small and large objects. For small objects, the remaining bits simply represent the size in words. For large objects, the remaining bits are divided into exponent and mantissa components; this floating-point number can describe (at coarser granularities) much larger segments than can an integer represented with the same number of bits. Rather than setting the size field's value to be slightly larger than the number of words in the contained object, ORSLA sets it to be slightly smaller; the object's precise size is stored in the object itself. When a reference to the last few words of a large object violates the capability's size field, microcode performs additional memory references to check the precise size stored in the object.

This scheme gives ORSLA precise object bounds at the cost of an extra memory reference for words toward the end of an object. Object allocation is extremely easy in ORSLA – objects may be allocated consecutively by simply advancing a shared allocation pointer. However, com-

[1]Since a Symbolics pointer contains no permissions bits, it is a fairly degenerate form of "capability" – possession of a pointer implies permission to perform any and all operations on its target object.

[2]Note that internal fragmentation only wastes real memory up to the level of a page; any whole page which is unused need not be allocated physical resources. Since our interest is in placing each object in its own capability-guarded segment, we must expect most fragmentation to cost physical memory, since most objects will be small enough that fragmentation will waste physical memory.

pared to guarded pointers, ORSLA capabilities are limited in that the embedded addresses must always point to the first word of their objects.

3 Our capability representation

Our capability representation takes inspiration from both the M-machine's guarded pointers and ORSLA's floating-point-bounded capabilities. Our goal is to define a representation encoding both address and size information such that:

- No memory references are ever required to check segment bounds.
- The segment defined by a capability is not significantly larger than the contained object in order to avoid wasting memory.
- The segment defined by a capability has loose alignment requirements in order to simplify allocation and thereby improve locality of reference.
- A capability may contain an address pointing to any location in the segment it describes.
- ○ None

We propose a 128-bit capability composed of a 64-bit *address* field, a 15-bit *bounds* field, a 1-bit *increment-only* field, a 16-bit *permissions* field, and a 32-bit *miscellaneous* field available to the operating system; a special 129th *capability tag* bit distinguishes capabilities from other data. This section will focus entirely upon the bounds field. Section 4 will discuss the increment-only field. We will not discuss the use of the permissions bits beyond mentioning that we favor the "take-diminish" scheme described in [Sha99]. In Section 5, we will describe the use of some of the the *miscellaneous* bits to enable sub-segmentation.

3.1 Representing size

Although we intend to use a 15-bit bounds field, we begin by introducing a 16-bit version; we will reduce it to 15 bits in the following section.

In the 16-bit version, shown in Figure 1, a capability's bounds field is divided into subfields L, B, and F; we shall defer discussion of the 5-bit finger field F until Section 3.5. L and B together specify that the segment consists of $L+1$ blocks of 2^B words, aligned on 2^B-word boundaries.[3] We suggest 5 bits for length L and 6 bits for log-block-size B, in which case the bounds field is 16

[3]The size field may alternately be viewed as a floating point number with mantissa L and exponent B – indeed, we use this terminology in overviews of the representation – but this viewpoint obscures the block-alignment requirement.

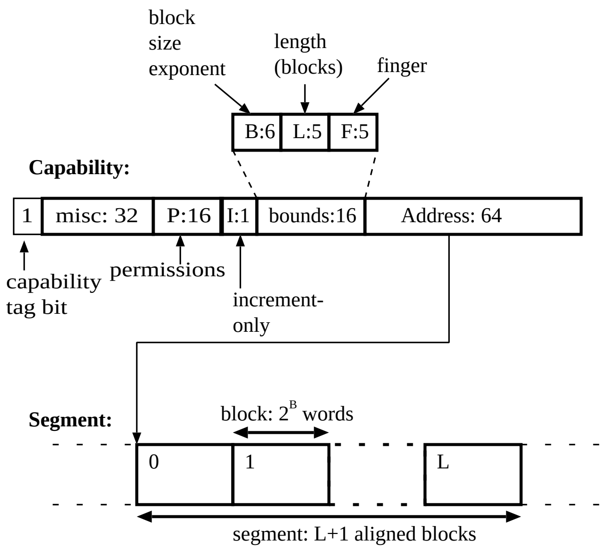

Figure 1: A simple, 129+1 bit version of a capability identifying a memory segment.

bits and a segment may be as large as $2^5 * 2^{2^6-1} = 2^{68}$ words.[4]

3.2 A more efficient size representation

Note that any segment whose size can be represented as L_1, B_1 where $B_1 > 0$ and $L_1 < 16$ can also be represented with finer granularity blocks as $L_2 = L_1 * 2 + 1$, $B_2 = B_1 - 1$. By requiring that sizes always be represented at the finest possible granularity, we can re-encode the size field as L',B', where L' is only 4 bits, thus saving one bit over the L, B representation.

[4]In other words, we can describe a segment which is larger than our entire address space.

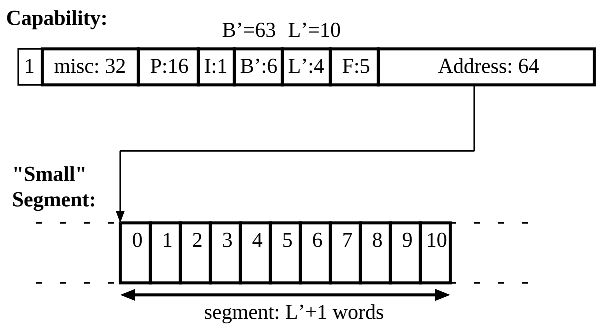

Figure 2: A compressed-bounds format (128+1 bits) capability identifying an 11-word memory segment ($B = 0$, $L = 10$; $B' = 63$, $L' = 10$).

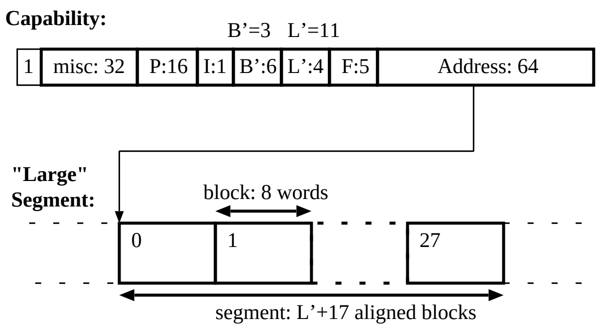

Figure 3: A compressed-bounds format capability identifying a 224-word memory segment ($B = 3$, $L = 27$; $B' = 3$, $L' = 11$).

We split this compressed-bounds encoding into two cases: small segments, and large segments. A small segment is between 1 and 16 words, and is represented by setting $B' = 63$ and taking $L' + 1$ to be the size of the segment in words (see Figure 2.) A large segment consists of exactly $L' + 17$ blocks of $2^{B'}$ words, where $0 \leq L \leq 15$ and $0 \leq B' \leq 62$. (See Figure 3.) With this encoding the largest segment representable is 2^{67} words.

The hardware to translate from compressed size to regular size is quite simple. If the ith bit of B' is b'_i, let $s = (B' \neq 63) = \overline{b'_5 \cdot b'_4 \cdot b'_3 \cdot b'_2 \cdot b'_1 \cdot b'_0}$; we can compute s with a 6-input NAND gate.

Given s, B is easily calculated with six AND gates: $b_i = b'_i \cdot s$. Finally, no additional gates are necessary to compute L, which is represented simply with $l_5 = s$, and $l_i = l'_i$ for $0 \leq i \leq 4$.

In the remainder of this paper we will generally speak in terms of B and L because they have simpler semantics than B' and L' – we will, however, always assume that segment sizes are represented at their finest possible granularities.

3.3 Segment/object size mismatches and wasted memory

Although our "floating point" size representation clearly gives a better fit between segment and object sizes than a purely exponential representation, for objects of more than 32 words there is still the possibility that the segment will be larger than the object it contains – i.e. one of the blocks of the segment will not be entirely covered by the object the segment contains. Large objects always have at least 17 blocks, and less than one block is wasted, so the worst-case memory loss due to this internal fragmentation is less than $1/17$ (less than 5.9%). By adding bits to the mantissa L we could further reduce the degree of waste; for instance, one additional bit would reduce internal fragmentation to less than $1/33$ (about 3%.)

3.4 The alignment requirement, allocation, and more wasted memory

We have specified that the blocks of a segment must be block-aligned – i.e. for a block of size 2^B, the lower B bits of the address of the first word of the block must all be zero. We justify this requirement in the next section, but we shall explore its implications for allocation in this one.

In particular, we would like to be able to allocate objects by simply advancing a counter on demand; this style of allocation is extremely simple and provides desirable spatial locality between consecutively allocated objects. Unfortunately, when a large object is allocated immediately following a small object, the last word of the small object may occupy a slot that would otherwise be the first word of a block for the large object. This forces the large object to be allocated at the next large block alignment boundary, wasting most of the large block preceding it. Since large segments consist of at least 17 blocks, at worst less than one block in 18 (less than 5.6%) is lost to this external fragmentation.

Combined with internal fragmentation due to object/segment size mismatches, the systemic worst-case space wastage is less than 2 blocks in 18, or less than 11.2% total wastage. Of course, like internal fragmentation, external fragmentation only wastes real memory up to the level of a page; again, though, our goal is to place each object in its own capability-guarded segment, and so we must still expect fragmentation to waste physical memory, since most objects will be too small to generate page-sized, page-aligned fragments.[5]

3.5 The finger field

With floating-point size representations, it is not possible to compute the base of a segment from an arbitrary address pointing into it; as a result, systems like ORSLA can only store capabilities which point to segment bases.

We overcome this limitation in our capability representation with the *finger* field F. The address in a capability may point to an arbitrary word in the target segment; as shown in Figure 4, the finger field records the fact that the pointed-to word is contained in the Fth block in the segment. Using our 5-bit size-field mantissa L (4-bit L'), an address can point into any of up to 32 blocks; F must therefore be 5 bits long.

[5]Specifically, an object must be more than 32 pages in length in order to potentially generate page-level fragmentation. The derivation of the number 32 is left as an exercise for the reader.

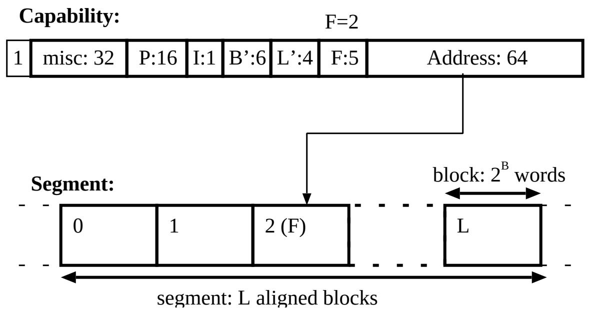

Figure 4: The finger field F identifies the block of a segment into which a capability points.

Another way to view F is as the high bits of the offset of the capability's address A from the segment base S; i.e. if we decompose the 64-bit address A as

$$A = U{:}Z \tag{1}$$

where $\mathrm{size}(Z) = B$, we can exploit the fact that the segment is block-aligned to express A as

$$A = S + F{:}Z \tag{2}$$

where the term $F{:}Z$ is padded with leading 0's to match A's 64-bit length.

3.5.1 Pointer arithmetic

Using the finger field, we can add an arbitrary integer offset X, where

$$X = U_x{:}Z_x \tag{3}$$

to a capability C_1 while simultaneously verifying that the address in the resulting capability C_2 still points within the segment bounds; note that X may be negative. The complete datapath for this operation is shown in Figure 5; the explanation of the datapath follows.

We compute the new address A_2 in straightforward fashion using a 64-bit adder:

$$A_2 = A_1 + X \tag{4}$$

We can compute F_2 more efficiently than is immediately obvious. Using Equations 2 and 3, we can rewrite Equation 4 as

$$A_2 = S + F_1{:}Z_1 + U_x{:}Z_x \tag{5}$$

If we subtract S from both sides of Equation 5, and right-shift the result by B bits, we discover that

$$F_2 = F_1 + U_x + (Z_1 + Z_x) >> B \tag{6}$$

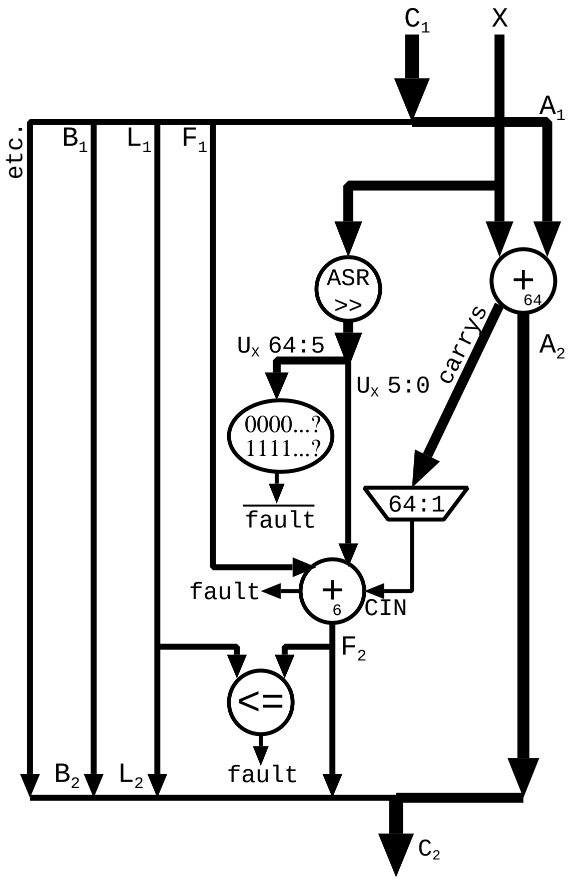

Figure 5: The datapath for adding signed constant offset X to capability C_1 to produce capability C_2. Note that this diagram uses the simple B, L size encoding.

The last term of Equation 6 is simply the carry-bit resulting from adding the first B bits of A_1 and X. Thus, we can compute F_2 using a 64:1 multiplexor to steal the correct carry bit, a 64-bit *arithmetic* right shift unit to extract U_x while preserving its sign, and a 64-bit adder to sum U_x and F_2.

As it turns out, we can actually replace the 64-bit adder with smaller quantities of hardware. We first observe that if U_x is positive, it must be smaller than 32 or it is guaranteed to cause F_2 to be larger than L; if U_x obeys this requirement, bits u_i for $i \geq 5$ will all be 0. Similarly, if U_x is negative, its magnitude can be at most 31 since otherwise F_2 is certain to be negative; if U_x obeys this requirement, bits u_i for $i \geq 5$ will all be 1.

Based on these observations, we can perform simple checks on the high bits of U_x to ensure that they meet one or the other of these criteria; if they fail, a bounds-check interrupt is raised. To compute the valid-positive-value check requires a 64-input NOR; to compute the valid-negative-value check requires a 64-input AND.

If the checks on the high bits of U_x pass, we can compute Equation 6 using only the 6 lowest bits (5 value bits and one sign bit) of U_x. Hence, we use a 64-bit AND, a 64-bit NOR, and a 6-bit adder, all instead of a 64-bit adder. Of course, if the result of the 6-bit addition generates a carry, we signal a bounds violation.

Having finally computed F_2, we now compare it to L with a 5-bit comparator. If $F_2 < 0$ or $F_2 \geq L$, there is a bounds violation and an interrupt must be signaled; if not, A_2 and F_2 are valid and may be composed, along with B_1 and L_1, to form the new capability C_2.

3.5.2 Computing a segment's base address

An important operation for garbage collection is quickly discovering an object's base address. If we subtract $F{:}Z$ from both sides of Equation 2 to produce

$$S = A - F{:}Z \tag{7}$$

we see that it is simple to compute a segment's base address using the pointer-math hardware of the previous section; we just subtract $F{:}Z$ from A. Assembling $F{:}Z$ in hardware requires a mask operation to extract Z from A; a shift operation to move F to the appropriate bitwise position; and finally an OR operation to merge the two values.

4 Increment-only capabilities and front-padded allocation

Although our bounds on wasted space are fairly tight, segments will still sometimes be larger than the objects they contain. Because the bounds-checking hardware checks segment bounds rather than object bounds, languages such as Java which must precisely bounds-check array accesses are stuck performing software checks. In this section we offer a solution to this problem: increment-only capabilities combined with front-padded allocation.

A single bit I is set to mark a capability as increment-only; if I is set, only positive offsets may be added to the capability. For example, a routine which receives an increment-only capability pointing to the middle of an array can only access the second half of that array; it cannot add negative offsets to the capability to access the first half.

Front-padded allocation simply means that when we allocate an object of size N into a segment of size M where $M > N$, the first word of the object is located at word $M - N$ of the segment, causing the last word of the object to coincide with the last word of the segment. In other words, all of the wasted space (padding) is at the front of the segment, rather than at the end.

We can get precise bounds-checking on objects by using an allocator which returns increment-only capabilities to the first words of front-padded objects. This approach works perfectly for languages such as Java which only store pointers to the first words of objects and arrays. Obviously we can not use the increment-only bit with languages such as C that allow arbitrary pointer math; however, front-padded allocation still has the potential to be helpful since most loops move from the front to the end of an object (array), rather than vice-versa, and thus the danger of accidental bounds-overrun is greatest at the end of an object.

The increment-only check is easily performed in hardware simply by examining the high bit of any (signed) constant being added to a capability, and throwing an interrupt if it is 1.

Increment-only pointers may be useful in other applications. For instance, we can protect an object's "private" fields from broken or malevolent code by placing them at its head. When passing a capability to untrusted code, we pass an increment-only capability which points just after the private fields, thus giving access only to the public fields in the latter part of the object.

5 Sub-segmentation with original segment recovery

In some cases, one might wish to allocate a large object, and then create a capability whose base and bounds information denote a sub-segment of that object; for instance, one might allocate an array of objects, and wish to generate a capability for exactly one of the objects in the

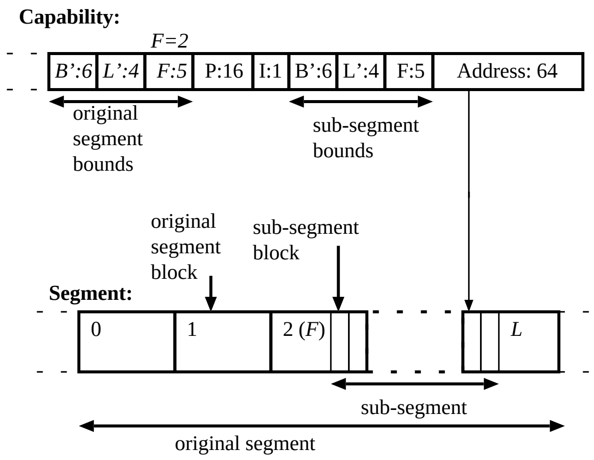

Figure 6: Sub-segmentation: the original segment's bounds are preserved; the original finger F identifies the block in which the sub-segment begins.

array. Generating such a capability is trivial, subject to alignment requirements: in the new capability, the address points into the sub-segment, and the bounds information denotes the sub-segment's bounds.

Uncontrolled sub-segmentation can generate problems in the presence of segment relocations which may happen, for instance, due to compacting garbage collection. A garbage collector faced with a variety of capabilities which overlap to varying degrees would have a great deal of difficulty ensuring that each segment was copied exactly once with no duplication of sub-segments.

To solve this problem we adopt a simple strategy: when a sub-segment capability is generated, the bounds information for the original segment are stored into some of the previously unused *miscellaneous* bits in the capability as shown in Figure 6. In particular, the original segment's size fields B' and L' are preserved; the finger F which is preserved identifies the original block in which the sub-segment begins. Thus, given a sub-segment capability, privileged routines such as the garbage collector can generate a capability for the original segment by first finding the base of the sub-segment, and then using the stored bounds information to recover the base and size of the original segment.

Either a tag bit must be dedicated to distinguish sub-segment capabilities from normal capabilities, or else normal capabilities must include copies of their size information in the "original bounds" fields so that garbage collection routines may treat all capabilities uniformly. Regardless, no special hardware is required to employ sub-segmentation; the operating system, allocation, and garbage-collection routines must simply agree upon the convention.

6 Conclusions

In this paper we have presented a capability format which improves upon prior formats by simultaneously providing four key features:

- Address and bounds information are embedded directly in the capability representation.
- A capability may point to an arbitrary word in its segment.
- Internal fragmentation due to segment/object size mismatch is less than 6%; with a simple, high-locality allocation scheme, total fragmentation is less than 12%.
- Objects of 32 or fewer words, e.g. most class instances in object-oriented systems, may be allocated with no fragmentation and will thus have precise hardware bounds-checking.
- ○ None

These features make it entirely practical to use a capability-guarded segment per allocated object, thus ensuring robust inter- and intra-program memory protection.

In addition to our basic capability format, we have described the implementation and application of increment-only pointers which enable precise hardware-only bounds-checking for Java-style objects/arrays. We have also demonstrated how to generate capabilities for sub-segments from which the enclosing segment can be recovered by system routines.

We should note that our capabilities may even be used with non-object-oriented code to improve software robustness. For instance, most programs written in C could run on our architecture; the *malloc* routine would return capabilities, and many classes of pointer-manipulation error traditionally undetected at the point of error would be caught by hardware bounds-checking.

Finally, for some applications our 128+1 bit capability format is unnecessarily large. A 64-bit encoding comprised of a 15 bounds-field, an increment-only bit, and a 48-bit address would provide several of the most important features of our format. Such an encoding might be particularly appealing for insuring intra-program data integrity in an environment in which inter-program integrity is provided by conventional disjoint virtual address spaces.

References

[Bis77] Peter B. Bishop. *Computer Systems With A Very Large Address Space And Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, May 1977.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 319–27, October 1994.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–12, July 1974.

[FKD$^+$95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The m-machine multicomputer. In *Proc. 28th Annual International Symposium on Microarchitecture*, pages 146–156, 1995.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Dynamic Memory Management*. John Wiley & Sons, 1996.

[Lev84] Henry M. Levy. *Capability-based computer systems*. Digital Press, 1984.

[Moo85] David A. Moon. Architecture of the symbolic 3600. In *12th Annual International Symposium on Computer Architecture Conference Proceedings*, pages 76–83, 1985.

[Sha99] Jonathan Strauss Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.