An Implementation of Guarded Pointers with Tight Bounds on Segment Size

J.P. Grossman, Jeremy Brown, Andrew Huang, Tom Knight

1 General Purpose

Since the advent of computers capable of running several programs concurrently, data security has been an important issue in system design. When different programs share the same hardware resources it is essential to ensure that they are not able to access or alter each other's data unless sharing is explicitly allowed by the programmer. Data security within a single program is also desirable to the developer, as it eliminates potential sources of program errors.

Traditionally, inter-process security has been addressed by maintaining a per-process set of page tables providing a translation from virtual address to physical location. This gives each process a separate address space and allows the operating system to ensure that it is impossible for a process' data to be inspected or corrupted by other applications. While such an approach is functional, there are three main objections to it. First, a process dependent address translation mechanism dramatically increases the amount of processor state associated with a given process. This makes context switching correspondingly slower, thus increasing system overhead and reducing efficiency. Second, data can only be shared between processes at the page granularity. Finally, this mechanism does not provide security within a single context; a program is free to create and use invalid pointers.

An alternate approach which addresses these problems is the use of **guarded pointers** [Carter94]. A guarded pointer is an unforgeable capability [Fabry74] with all relevant permission and segment size bits contained in the pointer itself (we will use the term **segment** to denote an allocated block of memory and **object** to denote data within a segment). Since user programs are not permitted to create capabilities, this allows the use of a single shared virtual address space within the system. Furthermore, the inclusion of permission and segment size bits within the guarded pointer obviates the need to perform expensive table lookup operations for every memory reference and every pointer arithmetic operation. Finally, the elimination of segment tables allows the use of an essentially unbounded number of segments without a prohibitive overhead; in particular object-based protection schemes become practical.

One of the challenges of designing a guarded pointer scheme is to represent segment size in such a way that, given a pointer, a small number of bits are used to determine the base and size of the segment into which the pointer points. In [Carter94] this is accomplished using only six bits by imposing the restrictions that all segment sizes should be powers of two, and all segments should be aligned to even multiples of their length. The six bits are then used to store the base 2 logarithm of the segment size, allowing for segments as small as one byte or as large as the entire address space.

This scheme is effective, but since the size of most objects is not a power of two, the restriction on segment size causes a large amount of internal fragmentation within the segments. This reduces the likelihood of detecting pointer errors in programs since pointers can be incremented past the end of objects while remaining within the allocated segment. Furthermore, fragmentation causes the apparent amount of allocated memory to exceed the amount of in-use memory by as much as a factor of two. This can seriously impact the performance of system memory management strategies such as garbage collection.

This technology disclosure describes a modification to the guarded pointer format which allows for more flexibility in the size and alignment of segments. 15 bits are used to derive segment size and base from a pointer. The representation guarantees an internal fragmentation of less than 6%, and allows objects to be aligned at a granularity of less than 7% of their size. Furthermore, additional bits may be used in the representation to make the internal fragmentation and alignment granularity arbitrarily small. Finally, a single additional "increment-only" bit provides efficient hardware support for high-level constructs such as exact object bounds and sub-object data security.

Project Aries Technical Memo ARIES-TM-02
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA

2 Technical Description

The following description assumes a 64 bit word size and a 64 bit address space, however the concepts are readily generalized. Figure 1 shows the proposed pointer format, which is an extension of the guarded pointer scheme presented in [Carter94]. The pointer is stored in a 128 bit double word; one extra tag bit is used to indicate whether or not the double word represents a pointer. The low order 64 bit word contains the actual virtual address of the pointer, while the high order 64 bit word contains the segment descriptor and unused bits which are available to the operating system. The figure shows 16 permission bits and 32 unused bits for purposes of illustration, but these fields are not important to the current discussion and no further mention of them will be made. The remaining fields will be described in the following sections.

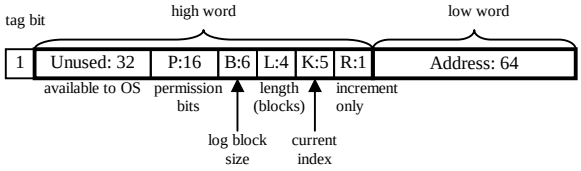

Figure 1: Pointer Format

2.1 Segment Size

The representation of segment size resembles the floating point representation of real numbers. Each segment is divided into **blocks** of size 2^B bytes where $0 \le B \le 62$. Blocks are aligned on even multiples of 2^B bytes. A segment consists of exactly (L+17) of these blocks where $0 \le L \le 15$ (there is never any need to use 16 or fewer blocks as in these cases we can use twice as many blocks of half the size). With this representation the minimum segment size that can be represented is 17 bytes and the maximum size is 267 bytes. To represent objects which are between 1 and 16 bytes in size we set B = 63 and take L+1 to be the size of the object in bytes; in this case the block size is 1 byte.

These two cases can be easily unified in hardware. Define the bit b to be 0 if B = 63 and 1 otherwise. Let B' = B & 63b and L' = L | 16b; then in all cases the block size is $2^{B'}$ and the segment size is $(L' + 1)2^{B'}$. The hardware required for this transformation is a 6 input NAND, 64 AND gates, and wires. All further references to B and L will assume that this transformation has already been applied, unless otherwise stated.

When a segment is allocated for an object, the size specified by B and L is the size of the segment; the actual size of the object may be smaller, in which case some memory is wasted due to internal fragmentation. To compute the amount of memory that is wasted, observe that the actual size of the object must be between L+16 and L+17 blocks (it can be no larger that its size in memory, and if it were smaller than L+16 blocks in size then a smaller value for L would be chosen). Thus, less than one block is wasted out of a total of L+17 blocks, so the fraction of wasted memory is less than $1/(L+17) \le 1/17 < 5.9\%$. As noted in [Carter94], this is the maximum amount of virtual memory which is wasted; the amount of physical memory wasted will in general be smaller.

2.2 Computing the Base Address

A valid pointer can point anywhere within a segment. In particular, it can point to any of up to 32 blocks. Five bits are therefore required to recover the segment's base address from a pointer's value. Define a **group** of blocks to be a set of 32 consecutive blocks aligned to integer multiples of 32x2B. Any 64 bit address can be broken up as follows:

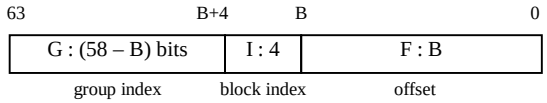

Figure 2: Breakdown of address bits

The high (58-B) bits form the **group index** and indicate which group of blocks the pointer is pointing to. The next four bits form the **block index** which specify one of the blocks in the group. Finally, the lower B bits specify a byte offset into this block.

Set K to be the index of the block within the segment which is currently being pointed to, where the first block in the object has index 0. Then $0 \le K \le 31$. Since the offset F of a segment's base address is always zero, the base address of a segment can be computed given B, K and a 64 bit address G:I:F as follows:

base address = G:I:0 - 0:K:0

Finding the base address of an object given an arbitrary pointer to the object is an important memory management operation which is essential for garbage collection. To implement this operation in hardware, let A be the 64 bit address and let A' = (A >> B) - K. Then the base address is A_b = A' << B. The hardware required for this operation is therefore a 64 bit right

shifter, a 64 bit integer adder (where one of the summands is only 4 bits), and a 64 bit left shifter.

2.3 Pointer Arithmetic

It is allowable to add/subtract an integer value to/from a pointer so long as the modified pointer remains within the bounds of the segment in memory. The hardware must perform bounds checking for every pointer arithmetic operation to ensure that it is impossible to create an invalid pointer. When adding an integer C to an address A, the bounds check can be performed in parallel with this addition in two steps:

1. Compute the offset of A from the base address of the object
2. Add C to this offset and verify that it does not go below zero or above the size of the object

If A = G:I:F then the offset of A from the base address of the segment is 0:K:F. Thus, if C = G':I':F', we must compute (0:K:F) + (G':I':F') and verify that it is at least zero but less than 0:(L+1):0 (the segment is L+1 blocks in size).

To perform this computation, begin by observing that the sum F+F' is already formed when A and C are added, so only (0:K) + (G':I') needs to be computed using as a carry-in the carry into the Bth bit in the sum A+C.

Next, note that G' must be either 0 or -1; any other value for G' will be guaranteed to violate the object bounds. If G' is 0 or -1, then the group index of the sum (0:K:F) + (G':I':F') will be one of {-1, 0, 1}. If it is -1 or 1 then the object bounds have certainly been violated. It follows that in performing the computation only one bit needs to be used for the group index (since there is no need to distinguish between -1 and 1 in the group index of the sum).

If the group index bit of the sum is 0 (which immediately implies that the sum is ≥ 0), it must be verified that the sum is less than (L'+1):0, which is true if and only if the block index of the sum is less than or equal to L'. This can be checked with a single 5 bit comparison.

The hardware required for this computation is (in addition to the 64 bit adder used the generate the new address A') a 64 bit MUX to select the appropriate carry, a 64 bit right shifter to shift C to the right by B bits, two 59 input gates which inspect the high bits of the shifted value to ensure that G' is 0 or -1, a 6 bit adder to compute (0:K) + (G':I'), and a 5 bit comparator. This is illustrated in Figure 3. Note that the 6 bit adder will produce a 5 bit result which replaces K in the modified pointer.

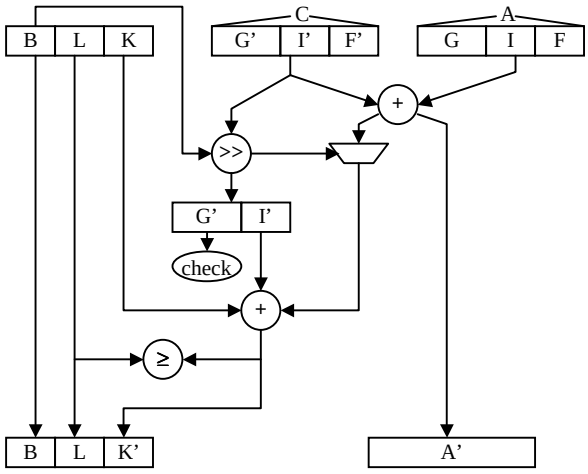

Figure 3: Hardware required for pointer arithmetic

2.4 Increment-Only

A single bit R can be used to mark the pointer as **increment-only**. It is an error to add a negative integer (or subtract a positive integer) from an increment-only pointer; in all other respects an increment-only pointer behaves identically to a normal pointer. The hardware implementation is trivial; all that is required is the addition of a two input gate to figure 3 to disallow the case G' = -1 for increment-only pointers. Increment-only pointers provide a simple mechanism for implementing exact object bounds and sub-object security.

2.4.1 Exact Object Bounds

As previously mentioned, it is desirable from a programming standpoint to minimize internal fragmentation within a segment, as this helps eliminate sources of program error. By using increment-only pointers it is possible to provide *exact* object bounds: when a segment of size *k* is allocated for an object of size *m* ($k \geq m$), the object is placed at an offset of *k-m* within the segment, and an increment-only pointer to the object is created. This method works for arbitrarily-sized objects, with the result that it is impossible for a user program to access the unused portion of the segment. Thus, for example, it is possible to allocate an array and have the hardware perform exact bounds checking.

2.4.2 Sub-object Security

Object oriented languages provide a model of sub-object security by allowing object members to be declared as public or private. These declarations are

typically used at compile time only, and on hardware that supports pointer arithmetic it is still possible for untrusted code to access private object members at run time. Increment-only pointers allow sub-object security to be strictly enforced at run time by placing all private information at the start of the object and sharing the object via increment-only pointers to the start of the public data. This mechanism is also advantageous to programmers as it again eliminates a potential source of program error.

3 Advantages over Existing Methods

The use of guarded pointers improves both system efficiency and system programmability. The inclusion of segment information within pointers obviates the need for expensive segment tables and lookaside buffers. There is no cost associated with a large number of segments beyond the creation of a large number of pointers, so it becomes feasible to implement object-based security by creating a new segment for every object. Since guarded pointers are unforgeable capabilities, there is no need to maintain a separate virtual address space for every process. This reduces system overhead provides a natural and efficient mechanism for sharing data between processes. Finally, unforgeable capabilities improve system programmability by eliminating the possibility of creating invalid pointers, a common source of errors in programs.

Guarded pointers were originally described by [Carter94]. Their format includes a six bit segment length field which specifies the size of a pointer's valid range as a power of two. This results in a worst case memory wastage due to internal fragmentation of 50%. The current scheme allows segments to be allocated in sizes which are closer to actual object sizes, resulting in a worst case memory wastage of less than 6%. Moreover, this percentage can be made as small as desired by adding more bits to the L and K fields. For example, if seven bits are used for L and eight for K, then the worst case memory wastage is less than 0.8%.

It is observed in [Carter94] that the amount of physical memory wasted is in general less than the amount of virtual memory wasted. However, internal fragmentation of the virtual memory space is still a concern for reasons of memory management efficiency and error detection. The memory system works with segments rather than objects, so each time a segment is relocated in the virtual or physical address space, and each time a segment is moved to or from disk, internal fragmentation causes unnecessary extra work to be performed. Software errors are often detected when they manifest themselves as a segment bounds violation, for example when a pointer is incremented past the end of an array. Internal fragmentation reduces the effectiveness of this error detection, for when objects are much smaller than the segments in which they reside it becomes increasingly probable that an incorrect pointer will remain within its segment and thus pass unnoticed. Thus, system efficiency and programmability are further improved by the reduction in internal fragmentation provided by the proposed guarded pointer format.

The increment-only extension to the guarded pointer format has negligible architectural overhead and provides efficient hardware support for exact object bounds and sub-object security. Each of these high-level constructs plays an important role in modern object oriented languages. The described mechanism allows such language features to be faithfully implemented without software overhead.

4 Commercial Applications

The proposed pointer format provides fine-grained data security, enhanced programmability and greater overall system efficiency. Since these are all crucial considerations in the design of any architecture, the technology described herein is broadly applicable to any commercial computer system. For the purposes of illustration we outline several domains for which this technology is particularly well suited.

1. **Object Oriented Computing**. The proposed format is naturally suited to systems which make use of an object model for data. Each object can be placed in a unique segment with only a small amount of internal fragmentation. The use of a single virtual address space allows objects to be shared between processes simply by copying pointers. Increment-only pointers provide a natural protection scheme for private object data.

2. **Strict Bounds Semantics**. Many programming languages such as Java [Gosling96] specify that array bounds are strictly checked at run time. With increment-only pointers, these checks can be implicitly performed in hardware with no software overhead. Commercial systems are judged in part by the programming environments which they support; the proposed format enables these important environments to be supported efficiently.

3. **Garbage Collected Memory Systems**. Two operations fundamental to garbage collection are the extraction of a segment's base address and the copying of data in memory. The proposed format provides efficient support for both of these operations. A segment's base address can be quickly extracted from a pointer to anywhere within the segment. The tight bounds on segment size minimize internal fragmentation of segments and thus avoid wasted effort when a segment is copied.

4. **Secure Computing**. In a secure computing environment, extreme emphasis is placed on data security. Any commercial system which is intended for secure computing should be able to enforce confinement [Karger88]; the proposed format provides unforgeable capabilities with enough permission bits to do so. It is also important to provide flexible mechanisms for sharing data without compromising security. The described format provides two such mechanisms: increment-only pointers allow a segment to be partly shared and partly hidden, and the flexible nature of a segment's base and length make it practical to support a "restriction" operation, whereby a new capability is created which provides access to a restricted window within an object.

Of significance is the fact that Java, which is the language of choice for web programming and is becoming increasingly important as a standalone development environment, lies in the intersection of these four domains. Java is a garbage-collected object-oriented language which, as previously mentioned, specifies strict bounds semantics for arrays. Since Java is designed to execute on client machines, security is a critical issue. A natural application of the described pointer format is therefore the implementation and/or improvement of systems which are specifically designed to run Java efficiently (such systems are already under development, e.g. [Tremblay99]).

References

[Bishop77] Peter B. Bishop, "Computer Systems with a Very Large Address Space and Garbage Collection", Ph.D. Thesis, Dept. of EECS, M.I.T., May 1977.

[Carter94] Nicholas P. Carter, Stephen W. Keckler, William J. Dally, "Hardware Support for Fast Capability-based Addressing", Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[Fabry74] R. Farby, "Capability-based addressing", Communications of the ACM, 17,7, July 1974, pp. 403-412.

[Gosling96] James Gosling, Bill Joy, Guy L. Steele Jr., The Java Language Specification, Addison-Wesley Publication Co., Sept. 1996, 825pp.

[Karger88] Paul Karger, "Improving Security and Performance for Capability Systems", Technical Report No. 149, University of Cambridge Computer Laboratory, October 1988 (Ph. D. thesis).

[Tremblay99] Marc Tremblay, "An Architecture for the New Millenium", Proc. Hot Chips XI, Aug. 15-17, 1999.