Appears in the Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI).

Hardware Support for Fast Capability-based Addressing

Nicholas P. Carter npcarter@ai.mit.edu
Stephen W. Keckler skeckler@ai.mit.edu
William J. Dally billd@ai.mit.edu

Artificial Intelligence Laboratory
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

Abstract

Traditional methods of providing protection in memory systems do so at the cost of increased context switch time and/or increased storage to record access permissions for processes. With the advent of computers that support cycle-by-cycle multithreading, protection schemes that increase the time to perform a context switch are unacceptable, but protecting unrelated processes from each other is still necessary if such machines are to be used in non-trusting environments.

This paper examines **guarded pointers**, *a hardware technique which uses tagged 64-bit pointer objects to implement capability-based addressing. Guarded pointers encode a segment descriptor into the upper bits of every pointer, eliminating the indirection and related performance penalties associated with traditional implementations of capabilities. All processes share a single 54-bit virtual address space, and access is limited to the data that can be referenced through the pointers that a process has been issued. Only one level of address translation is required to perform a memory reference. Sharing data between processes is efficient, and protection states are defined to allow fast protected subsystem calls and create unforgeable data keys.*

1 Introduction

Memory system designers must provide security without sacrificing efficiency and flexibility. Objects must be protected from modification by unauthorized processes, and user programs must not be allowed to affect the execution of trusted system programs. It must be possible to share data between processes in a safe and efficient manner; merely providing private data spaces or globally accessible data spaces is insufficient. An efficient mechanism must also be provided to change protection domains (the set of objects that can be referenced) when entering a subsystem.

The current trend towards the use of multithreading as a method of increasing the utilization of execution units makes traditional security schemes undesirable, particularly if context switches may occur on a cycle-by-cycle basis. Traditional security systems have a non-zero context switch time as loading the protection domain for the new context may require installing new address translations or protection table entries.

A number of multithreaded systems such as Alewife [2], and Tera [3] have avoided this problem by requiring that all threads which are simultaneously loaded share the same address space and protection domain. This is sufficient for simultaneous execution of threads from a single user program, but precludes interleaving threads from different protection domains, eliminating a potential source of concurrency.

This paper presents guarded pointers, a mechanism that provides efficient protection and sharing of data. Guarded pointers are an implementation of capabilities [12] that encode permission and segmentation information within tagged pointer objects. A guarded pointer may reside in a general purpose register or in memory, eliminating the need for special storage for capabilities. Because memory may be accessed directly using a guarded pointer, higher performance may be achieved than with traditional implementations of capabilities, as table lookups to translate capabilities to virtual addresses are not required.

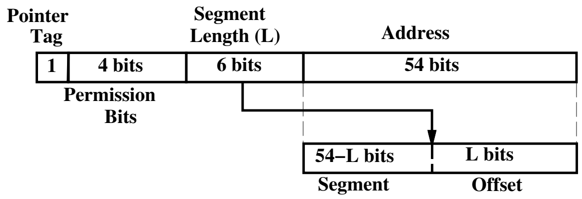

Figure 1: Format of a guarded pointer. A guarded pointer identifies a byte in the virtual address space, the segment containing that byte, and the set of operations permitted on the segment. The permission field determines what operations may be performed using the pointer, and the segment length field separates the address into a fixed segment field and a variable offset field by specifying the base-2 logarithm of the length of the segment containing the address.

Figure 1 shows the format of a guarded pointer. A single *pointer* bit is added to each 64-bit data word. Fifty-four bits contain an address, while the remaining ten bits specify the set of operations that may be performed using the pointer (4 bits) and the length of

*The research described in this paper was supported by the Advanced Research Projects Agency and monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045.

the segment containing the pointer (6 bits). Segments are required to be a power of two bytes long, and to be aligned on their length. Thus, a guarded pointer specifies an address, the operations that can be performed using that address, and the segment containing the address. No segment or capability tables are required. Since protection information is encoded in pointers, it is possible for all processes to share the same virtual address space safely, eliminating the need to change the translation scheme on context switches and facilitating the use of virtually-addressed caches.

Memory must be accessed using a guarded pointer with a valid permission field. User level programs may not forge a guarded pointer by setting the pointer bit on a word, although they may manipulate pointers with instructions that maintain the protection scheme. This prevents users from creating arbitrary pointers, while allowing address arithmetic within the segments that have been allocated to a user program. Privileged programs may set the pointer bit of a word and thus create any pointer.

Section 2 of this paper examines guarded pointers in more detail, and shows how they may be used to implement a memory system. The M-Machine, a multicomputer architecture that shows how guarded pointers satisfy the requirements of an aggressively multithreaded system, is described in Section 3. Section 4 discusses the costs and benefits of guarded pointers. Section 5 compares guarded pointers to other related protection schemes. Our conclusions are presented in Section 6.

2 Guarded Pointers

Memory systems that use guarded pointers provide a single virtual address space, which is shared by all processes [5]. A guarded pointer identifies a byte in the virtual address space, the segment containing that byte, and the set of operations permitted on the segment. As shown in Figure 1, a guarded pointer is tagged with a pointer bit to prevent user processes from forging it. A four-bit permission field identifies the set of operations permitted on the segment. The length field of the pointer holds the base-2 logarithm of the segment length, which allows segments to range from a single byte to the entire 2^{54} byte address space in power of two increments. The length field separates the address into a fixed segment portion and a variable offset portion. Because of the logarithmic encoding, segments are required to be aligned on their length. This allows the base of a segment to be determined by setting all of the offset bits to zero.

2.1 Permission Types

The permission field of a pointer indicates how a process may access the data within the segment. Pointer permissions may specify data access, code access, protected entry points, and unforgeable identifiers (keys). The following is a representative set of permissions:

- A **Read-Only** pointer may only be used to load data from memory.
- A **Read/Write** pointer may be used to either load or store data to memory.
- **Execute** pointers are *read-only* pointers that may be used as targets for jump instructions. An execute pointer to a code segment enables a program to jump to any location within the segment and to read the segment. Execute pointers may be either **execute-user** or **execute-privileged**, which encodes the supervisor mode bit explicitly within the instruction pointer. Privileged instructions may only by executed with an execute-privileged instruction pointer.

A *read-only*, *read/write*, or *execute* pointer's address field may be altered as long as it remains within its segment bounds.

- **Enter** pointers are an efficient mechanism for implementing gateways, as they enable a program to enter a code segment only at particular locations. Jumping to an enter pointer converts it to an execute pointer which is then loaded into the instruction pointer. Enter pointers may not be modified or used to load or store to memory. The two types of enter pointers are **enter-user** and **enter-privileged**, which are converted to the corresponding type of execute pointer by a jump.
- A **Key** pointer may not be modified or referenced in any way. It may be used as an unforgeable, unalterable identifier.

2.2 Pointer Operations

Implementing guarded pointers requires adding a small number of pointer manipulation instructions to the architecture of a conventional machine as well as some hardware to verify that each instruction operates only on legal pointer types and that address calculations remain within pointer bounds.

Load/Store: Every load or store operation requires a guarded pointer of an appropriate type as its address argument. Protection violations are detected by checking the permission field of the pointer. If the address is modified by an indexed or displacement addressing mode, bounds violations are checked by examining the length field as described below. The protection provided by guarded pointers does not slow load or store operations. All checks are made before the operation is issued, without reference to any permission tables. Once these initial checks are performed, the access is guaranteed not to cause a protection violation, although events in the memory system, such as TLB misses, may still occur.

Pointer Arithmetic: An `LEA` (load effective address) instruction may be used to calculate new pointers from existing pointers. This instruction adds an integer offset to a data or execute pointer to produce a new pointer. An exception is raised if the new pointer would lie outside the segment defined by the original pointer. For efficiency, an `LEAB` operation, which adds an offset to the base of the segment contained in a pointer, may be implemented as well. If a guarded pointer is used as an input to a non-pointer operation, the pointer bit of the guarded pointer is cleared, which converts the pointer into an integer with the same bit fields as the original pointer.

Figure 2 details the validation required on a pointer calculation. The permission field of the pointer is checked to verify that it is a *read-only*, *read/write*, or *execute* pointer. An integer offset is added

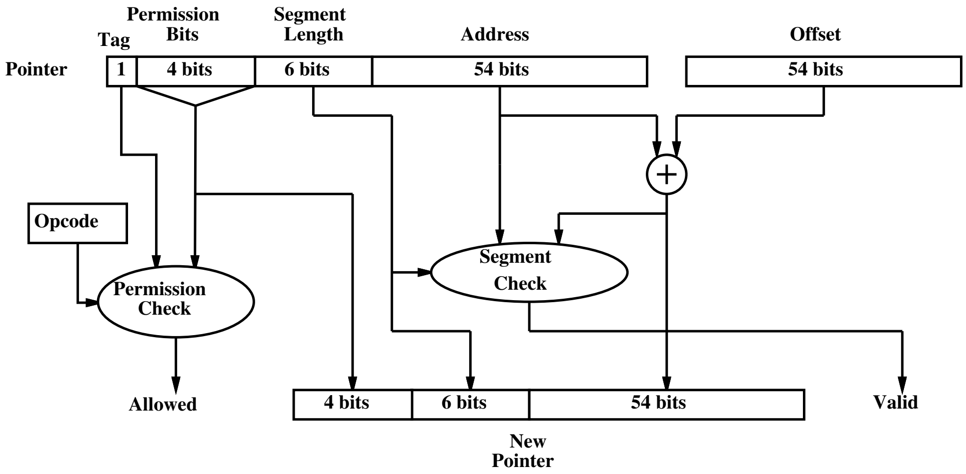

Figure 2: Pointer Derivation: a new pointer may be created using an LEA operation on an existing pointer and an offset. The permission field must be checked and the new pointer must not lie outside of the old pointer's segment.

to the address field of the pointer. An exception is raised if the result of this add over- or underflows into the fixed segment portion of the address, which would create a pointer outside the original segment. This error may be detected by comparing the fixed portion of the address before and after the addition occurs.

Guarded pointers expose to the compiler address calculations that are performed implicitly by hardware in conventional implementations of segmentation or capabilities. With the conventional approach, the segmentation hardware performs many redundant adds to relocate a series of related addresses. Consider, for example, the following loop:

```
for(i=0;i<N;i++) s = s + a[i];
```

In a conventional system, each reference to array `a` would require the segmentation hardware to automatically add the segment offset for each `a[i]` to the segment base. With guarded pointers, the add can be performed once in software, and then the resulting pointer can be incrementally stepped through the array, avoiding the additional level of indirection.

Languages that permit arbitrary pointer arithmetic or type casts between pointers and integers, such as C, are handled by defining code sequences to convert between pointer and integer types. The pointer-to-integer cast operation takes a guarded pointer as its input and returns an integer containing the offset field of the guarded pointer. This can be performed by subtracting the segment base, determined using the LEAB instruction, from the pointer:

```
LEAB    Ptr, 0, Base
SUB     Ptr, Base, Int
```

The integer-to-pointer cast operation uses the LEAB instruction to take an integer and create a pointer into the data segment of the process with the integer as its offset, as long as the integer fits into the offset field of the data segment. Note that neither of these cast operations requires any privileged operations, which allows them to be inlined into user code and exposed to the compiler for optimization.

Pointer Creation: A process executing in privileged mode has the ability to create pointers and hence access the entire address space. Privileged mode is entered by jumping to an enter-privileged pointer. It is exited by jumping to a user pointer (enter or execute). While in privileged mode, a process may execute the SETPTR instruction to convert an integer into a pointer by setting the guarded pointer bit. Thus, a privileged process may amplify pointer permissions and increase segment lengths while a user process can only restrict access. No other operations need be privileged, as guarded pointers can be used to control access to protected objects such as system tables and I/O devices.

Restricting Access: A process may derive pointers with restricted permissions from those pointers that it holds. This allows a process to share part of its address space with another process or to grant another process *read-only* access to a segment to which it holds *read/write* permission.

A RESTRICT instruction takes a pointer, P, and an integer permission type, T, and creates a new pointer by substituting T for the protection field of P. The substitution is performed only if T represents a strict subset of the permissions of P. Otherwise, an exception is raised.

Similarly the SUBSEG instruction takes an integer length, L, and a pointer, P, and substitutes L into P if L is less than the original length field of P. The RESTRICT and SUBSEG instructions allow a user process to control access to its memory space efficiently, without system software interaction.

The RESTRICT and SUBSEG instructions are not completely necessary, as they can be emulated by providing user processes with enter-privileged pointers to routines that use the SETPTR instruction to create new pointers that have restricted access rights or segment boundaries. The M-Machine, which will be described in the next section, takes this approach.

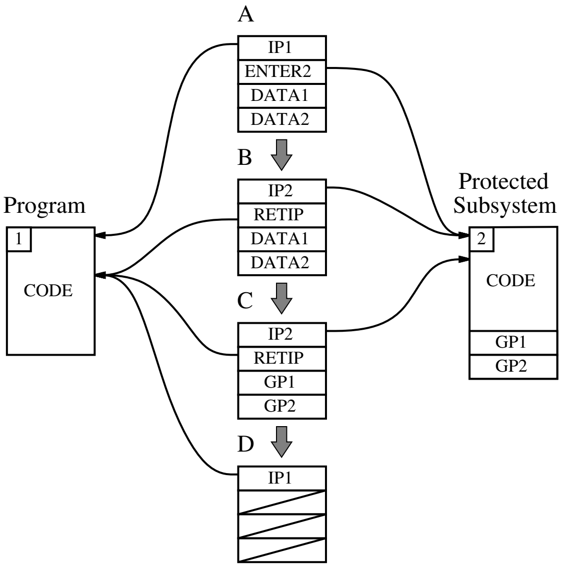

Figure 3: A program enters a protected subsystem by jumping to an enter pointer. After entry the subsystem code loads pointers to its data structures from the code segment. `A` represents the register state of the machine before the protected subsystem call, `B` the register state just after the call, `C` the register state during the execution of the protected subsystem, and `D` the register state immediately after the return to the caller.

Pointer Identification: The `ISPOINTER` instruction is provided to determine whether a given word is a guarded pointer. This instruction checks the pointer bit and returns its state as an integer. Quick pointer determination is useful for programming systems that provide automatic storage reclamation, such as LISP, which need to find pointers in order to garbage collect physical space [21].

2.3 Protected Subsystems

Enter pointers facilitate the implementation of protected subsystems without kernel intervention. A protected subsystem can be entered only at specific places and may execute in a different protection domain than its caller. Entry into a protected subsystem, such as a file system manager, is illustrated in Figure 3. Before the call (Figure 3A), the calling program (segment 1) holds an enter pointer to the subsystem's code segment (segment 2) which contains the subsystem code as well as pointers to the subsystem's data segments, such as the file system tables. To enter the subsystem, the caller jumps to `ENTER2`, causing the hardware to transfer control to the entry point and and convert the enter pointer to the execute pointer `IP2` (Figure 3B). The return instruction pointer (`RETIP`) is passed as an argument to the subsystem. The subsystem then uses the execute pointer to load `GP1` and `GP2`, the pointers to its data structures (Figure 3C). The subsystem returns to the calling program by overwriting any registers containing private pointers and jumping to `RETIP` (Figure 3D).

The sequence described above provides one-way protection, protecting the subsystem's data structures from the caller. To provide two-way protection, the caller (segment 1) encapsulates its protection domain in a *return segment* (segment 3) as shown in Figure 4. Before the call (Figure 4A), the caller holds both enter and read/write pointers to a return segment. The caller writes all the live pointers in its registers into the return segment to protect them from the subsystem (segment 2). It then overwrites all of the pointers in its register file except the enter pointer to the return segment (`ENTER3`), the subsystem enter pointer (`ENTER2`), and any arguments for the call (Figure 4B). The subsystem call then proceeds as described above. After entry, the subsystem holds only an enter pointer to the return segment and thus cannot directly access any of the data segments in the caller's protection domain (Figure 4C). The subsystem returns by jumping to the return segment (Figure 4D), which reloads the caller's saved pointers and returns to the calling program.

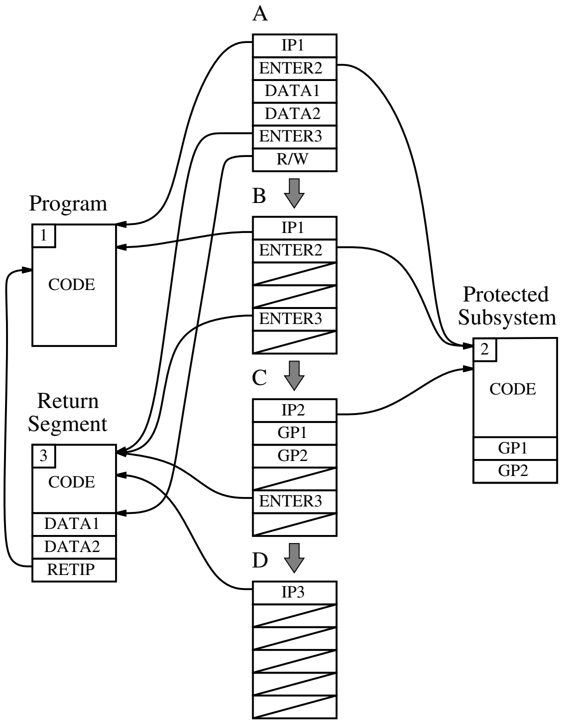

Figure 4: Two-way protection is provided by creating a return segment that encapsulates the protection domain of the calling program. `A` represents the register state of the machine before the protected subsystem call, `B` the register state just after the call, `C` the register state during the execution of the protected subsystem, and `D` the register state immediately after the jump to the return segment.

Enter pointers allow efficient realization of protected system services and modular user programs that enforce access methods to data structures. Modules of an operating system, e.g., the file-system, can be implemented as unprivileged protected subsystems that contain pointers to appropriate data structures. Since these data structures cannot be accessed from outside the protected subsystem, the file-system's data structures are protected from unauthorized use. Even an I/O driver can be implemented as an unprivileged protected subsystem by protecting access to the *read/write* pointer

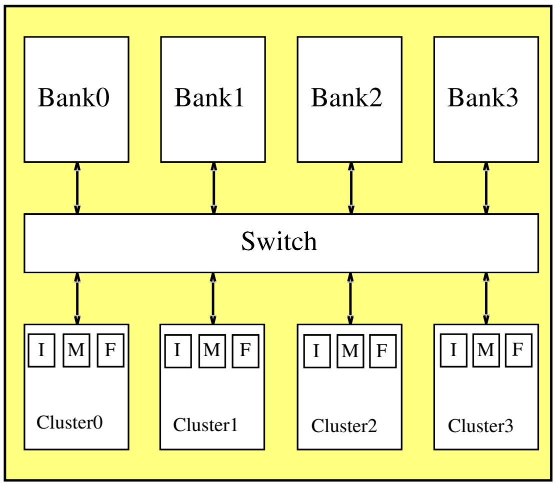

Figure 5: Block diagram of a MAP chip. The cache is interleaved into 4 banks accessed across a switch. Each cluster contains an integer unit (I), a memory access unit (M), and a floating point unit (F).

of a memory-mapped I/O device. With protected entry to user-level subsystems, very few services actually need to be privileged. This can bring higher efficiency to modern microkernel operating systems such as Mach [1].

3 The M-Machine

The M-Machine memory system provides an example of how guarded pointers may be used. The M-Machine is a multicomputer with a 3-dimensional mesh interconnect and multithreaded processing nodes [9, 16]. One of the major research goals of the M-Machine is to explore the best use of the increasing number of transistors that can be placed on a single chip.

The processing nodes of the M-Machine (known as multi-alu processors, or MAPs) operate on 64-bit integer and floating-point data types and use 64-bit guarded pointers (plus a tag bit) to access a 54-bit, byte-addressable, global address space, which is shared by all processes and nodes of the machine. Figure 5 shows a block diagram of a MAP chip. Each MAP chip contains twelve execution units: four integer, four floating-point, and four memory units. These execution units are grouped into four clusters, each containing one execution unit of each type.

To increase the utilization of these hardware resources when executing programs that have insufficient instruction-level parallelism, the M-Machine implements multithreading. Four user threads share the processing resources of each cluster, for a total of sixteen user threads in execution at any time. Each cycle, the hardware on each cluster examines the executing threads and selects one thread to execute on the hardware resources. The three execution units in a cluster are allocated and statically scheduled as a long instruction word processor.

Each M-Machine node contains 16KWords (128KBytes) of on-chip cache, which is divided into 4 banks, and 1MWord (8MBytes) of off-chip memory. The cache is virtually addressed and addresses are interleaved across the banks. This allows the memory system to accept up to four memory requests during each cycle, matching the peak rate at which the processor clusters can generate requests. Requests that miss in the cache arbitrate for the external memory interface, which can only handle one request at a time.

The M-Machine presents two challenges to a protection system. The first is cycle-by-cycle interleaving of instructions and memory references from different protection domains, while still allowing efficient sharing among them. Because guarded pointers provide memory protection without requiring each thread to have its own virtual to physical translations, memory references from different threads may be in flight simultaneously without compromising security. This enables zero cost context switching, as no work is required to switch between protection domains.

The other challenge for both the protection and translation systems is the interleaved cache of the M-Machine, which may service up to four references simultaneously. The single address space implemented with guarded pointers allows the cache to be virtually addressed and tagged so that translations need only to be performed on cache misses. In addition, encoding all protection information in a guarded pointer eliminates any need for table lookup prior to or during cache access. These two features eliminate the need to replicate or quad-port the TLB or other protection tables.

4 Critique

4.1 Hardware Costs

Guarded pointers have two hardware costs: a small increase in the amount of memory required by a system, and some additional hardware to perform permission checking. To prevent unauthorized creation or alteration of a guarded pointer, a single tag bit is required on all memory words, which results in a 1.5% increase in the amount of memory required by the system.

The hardware required to perform permission checking on memory access and segment bounds checking on pointer manipulation is minimal. One decoder for the permission field of the pointer, one decoder for the opcode of the instruction being executed, and a small amount of random logic are needed to determine if the operation is allowed. The pointer bit of an operand can be checked at the same time, to determine if it is a legal pointer. To check for segment bounds violations when altering a pointer, a masked comparator is needed. It compares the address before and after alteration and signals a fault if any of the segment bits of the address field change.

Memory systems based on guarded pointers do not require any segmentation tables or protection lookaside buffers in hardware, nor is it necessary to annotate cached virtual-physical translations with a process or address space identifier. As with other single address space systems, the cache may be virtually addressed, requiring translation only on cache misses.

4.2 Address Space

Since 10 bits are required to encode the permission and segment length fields of the guarded pointers described in this paper, the virtual address space is reduced, from 64 to 54 bits. A 54-bit address space allows 1.8×10^{16} bytes to be addressed, which should be sufficient for the immediate future. Several current processors support 64-bit virtual addresses, but only use some of the available address bits. For example, the DEC Alpha 21064 only translates 43 bits of each 64-bit address [11].

There is an opportunity cost associated with reducing the virtual address space, however. Some system designers take advantage of large virtual address spaces to provide a level of security through sparse placement of objects. For example, the Amoeba distributed operating system [22] protects objects using a software capability scheme. These capabilities are kept secret by embedding them in a huge virtual address space, a strategy which becomes less attractive if the virtual address space shrinks by a factor of 1000. Of course, this particular use of a sparse virtual address space can be replaced by the capability mechanism provided by guarded pointers.

Virtual address space fragmentation is another potential problem with guarded pointers, as segments must be powers of two words in length and aligned. Internal fragmentation may result when the space needed by an object must be rounded up to the next power of two words. However, this fragmentation does not result in much wasted physical memory, since physical space is allocated on a page-by-page basis, independent of segmentation. External fragmentation of the virtual address space may occur when recycled segments cannot be coerced into contiguous sections of usable sizes. A *buddy system* memory allocation scheme, which combines adjacent free segments into larger segments, can be used to reduce this fragmentation problem.

4.3 Limitations of guarded pointers

While guarded pointers enable efficient implementation of many desirable operating system features, some shortcomings inherent in single-address-space and capability-based architectures are not addressed. This section examines some of these problems, and suggests ways in which the software system designer can use guarded pointers to solve them.

Protected Indirection: The efficiency of guarded pointers is largely due to eliminating indirection through protected segment tables. With guarded pointers there is no need to store these tables or to access them on each memory reference. Without protected indirection, however, modifying a capability requires scanning the entire virtual address space to update all copies of it. This is needed, for example, when relocating a segment within the virtual address space or revoking access rights to a segment. In some cases this expensive operation can be avoided by exploiting the paging translation, user-level indirection, or protected subsystems, as described below.

All guarded pointers to a segment can be simultaneously invalidated by unmapping the segment's address space in the page table. All subsequent accesses using pointers to this segment will raise exceptions. Segments can be relocated by updating the pointer causing the exception on each reference to the relocated segment. One limitation of this approach is that it operates on a page granularity while segments may be any size, down to a single byte in length. Thus relocating or revoking access to a segment may affect the performance of references to several unrelated segments that happen to reside on the same physical page.

Indirection can be performed explicitly in software where it is required. If a segment's location is unknown or is expected to move frequently, a program can make all segment references to offsets from a single segment base pointer. Only this single pointer needs to be updated when the segment is moved. With explicit indirection, overhead is incurred only when indirection is needed, and then it is exposed to the compiler for optimization. Since no hardware prevents user code from copying the segment base pointer, relocation or revocation through explicit indirection requires adherence to software conventions.

It is impossible in any capability-based system to directly revoke a single process' rights to access a segment without potentially affecting other processes. Since possession of a capability confers access rights, the only way to remove access rights from a single process is to remove all capabilities containing those access rights from the memory addressable by the process. This can be accomplished by sweeping the memory that the process can address, and overwriting the correct capabilities, so long as none of the memory containing those capabilities is shared. If the pointers that need to be overwritten are contained within a shared segment, all processes which rely on the pointer will lose access privileges.

Finally, protected indirection can be implemented by requiring that all accesses to an object be made through a protected subsystem. In addition to restricting the access methods for the object, the subsystem can relocate the object at will and can implement arbitrary protection mechanisms, such as per-process access control lists. Revoking a single process' access rights can be performed by updating the access control list. Accessing an object through a protected subsystem is advisable if the object must be relocated or have its access rights changed frequently and if the object is referenced infrequently or only via the subsystem access methods.

Address Garbage Collection: Without enforced indirection, address space is allocated "for all time," requiring the system software to periodically garbage collect the virtual address space, so that addresses no longer in service can be reused. This is simplified with guarded pointers, as pointers are self identifying via the tag bit. Thus, the live segments can be found by recursively scanning the reachable segments from all live processes and persistent objects.

5 Related Work

5.1 Page-Based Protection

Separate Address Spaces: The main objection to using a traditional paged memory system on a multithreaded processor is the time required to change protection domains. Page-based protection systems provide security by assigning each process its own set of virtual to physical address translations. On each change of protection domain, the old mapping becomes invalid and a new one must become available. This is typically accomplished by writing the address of the page table that describes the new mapping into a hardware register. Without address space identifiers (or process identifiers) the old translations must be flushed from the TLB and the cache must be purged on each change of protection domain.

Using address space identifiers to identify the process associated with a virtual address alleviates the stale translation problem and allows implementation of a virtually-addressed cache. A virtually-addressed cache is extremely useful for a machine that expects to make multiple cached memory references in each cycle, as it allows translation to be deferred until external memory must be referenced, thus reducing the number of ports needed on the translation lookaside buffers. However, as address space identifiers create synonyms or aliases for shared data, no data can be shared in a virtually addressed cache using this system.

In addition to in-cache sharing problems, page-based protection schemes have difficulties sharing data through main memory. Even with address space identifiers, different processes will have different translations and names for the same object. The page table for each process must be altered so that it contains a translation for the shared physical page. All processes that share a group of pages must have a page table entry for each page in the group, resulting in $n \times m$ page table entries for n physical pages shared among m processes. Finally, basing protection on paging limits the smallest object that can be protected to be the size of a page.

Domain-Page Protection: The Domain-Page system [17] separates protection from translation in a single address space by sharing the page table among all processes and using an independent protection table for each process. Only page-sized objects can be protected and a Protection Lookaside Buffer (PLB) is used to cache recently used protection table entries. When a memory access is performed, the PLB is probed in parallel with the virtually addressed cache to detect protection violations. The TLB is shared by all processes and is only accessed on a cache miss.

Domain-Page protection is a viable alternative to guarded pointers for multithreaded computers, in that it supports fast changes of protection domain. An advantage that guarded pointers have over Domain-Page protection is that guarded pointers do not require the additional lookaside buffer that Domain-Page schemes require to operate efficiently. This is particularly significant for machines that need to support multiple cache accesses/cycle, as the PLB would have to be replicated or multi-ported.

HP PA-RISC: The HP PA-RISC protection architecture [18] performs access control at the page level. Each TLB entry contains a physical page number, permission information, and a page group identifier. If a TLB hit occurs but the page group identifier does not match that provided by the process, an access violation fault occurs. Four special registers are provided to allow a process to quickly access four separate page groups. Each of these must be compared with the page group number provided by the TLB on every memory reference. Thus, two processes in different protection domains may share data by having access to the same page group. Context switching is relatively inexpensive as the TLB and cache need not be flushed.

Page-group protection is essentially an inexpensive implementation of segmentation, with the four special registers acting as segment registers. With page-group protection, access control is on a large grain since the smallest unit of sharing is a physical page. Furthermore, only five page groups (four via special registers plus one global) may be quickly accessed. Guarded pointers eliminate the need for special registers and provide protection at more flexible granularities. In addition, page-group protection requires a TLB lookup and comparisons of the page-group number to four registers on every memory access. This is prohibitively expensive for a multi-banked cache implementation.

5.2 Segmentation

Segmentation-based memory systems provide protection on arbitrary-sized regions of memory through the use of segment descriptors. Any operation involving an address is checked against the segment descriptors to ensure that it uses the address in an appropriate manner, and that segment bounds are not crossed. Segment descriptors often encode the types of accesses that may be performed on the segment, allowing processes to be granted limited access to regions of memory. Since managing variable-length transfers of data between different levels of a memory hierarchy is difficult, segmentation is often implemented on top of a paging system which is responsible for transferring fixed size pages.

Segmentation has been used in systems such as the B5000 [20] and Multics [4, 6, 10] to provide separate address spaces with controlled sharing in a multiprogrammed system. Other recent systems, such as Monads [23], also employ a traditional segmentation scheme to support protection and relocation. Methods exist to reduce the redundant translation overhead required by performing segmentation in addition to paging [7]. A major disadvantage of segmentation is the fixed division between the segment identifier and offset fields of an address which limits both the number of segments and the size of the largest segment that can be represented. For example, in Multics, a segment is limited to 2^{18} words and in the 8086 [14], a segment is limited to 2^{16} bytes. While the 80386 [15] extends the maximum segment size to 2^{32} bytes, using these segments is unwieldy as it requires handling 48-bit pointers and one process can address at most 2^{16} segments. Guarded pointers borrow from [8] the use of a floating-point address in which the boundary between the segment identifier and the offset field may vary depending on segment length. With guarded pointers, one may address 2^{54} one-byte segments, a single 2^{54} byte segment, or any power of 2 division in between.

Several features of segmentation make it unacceptable for multithreaded processors. First is the cost of swapping segment descriptors for every thread switch. Each process has its own table of segment descriptors that describe the memory that the process can access. The cost of changing the table on a process switch is similar to that required to change the translation function in a paged protection scheme, as described above.

Second, two levels of translation are typically required: one to translate segments and offsets to virtual addresses and one to translate virtual addresses to physical addresses. Determining a virtual address from a segment and offset must be performed before accessing the cache, thus slowing down all memory references. Finally, as in a paging-only system, sharing data requires operating system intervention and replication of protection information. Every process must have its own segment descriptor for each shared segment and only the operating system can make these available.

5.3 Capabilities

Capabilities [12, 19] provide an efficient means of security for a machine that needs to change protection domains frequently. By encoding access rights into the handle that a process uses to reference an object, a capability-based system allows the operating system to restrict access to objects by providing each process with only those capabilities that it needs. Processes can then be interleaved without security violations, as a process can only access those objects for which it has a capability. Sharing data between processes is accomplished by giving each process a capability to the shared object. Most capability-based systems also provide a variety of permissions (such as read, read/write, execute) so that different processes may have different access rights to the same object.

Previous hardware implementations of capabilities, including the IBM System/38 [13] and the Intel 432 [24], have required two levels of translation: one to translate capabilities to virtual addresses, and the second to translate virtual addresses to physical addresses. The additional latency to access memory imposed by two-level translation has prevented traditional capabilities from becoming a widely-used protection method.

Because of their size and the need to prevent them from being altered by user code, traditional capabilities often require special registers or storage. By encoding both the capability and the descriptor into the standard data word (64 bits) and tagging it, a guarded pointer requires no special storage and may be used to access memory directly.

5.4 Software Techniques

Software methods may alternatively be used to prevent programs from accessing memory in unauthorized ways. If the compiler and linker can guarantee safe execution, the hardware need only provide a single flat address space and paging.

Wahbe, *et. al.* [25] suggest several methods for making software safe. A technique called sandboxing places user programs inside isolated fault domains and prevents writes or jumps to locations outside the fault domain. A post pass over the object file can insert extra instructions to explicitly check for out of bounds accesses or branches. Alternatively, the check code can set the high bits of all addresses to be the fault domain identifier, restricting all accesses to be within the domain. As an optimization, the code segment can be placed in a sparsely populated region of virtual address space, surrounded by unmapped virtual addresses. The unmapped regions are large enough so that the immediate offsets specified in instructions can not be used to reach any virtual addresses that are mapped to physical locations.

Software methods suffer from several disadvantages. First, additional instructions are required before every memory reference that can not be statically determined to be safe. Even if a given memory reference is usually safe, the overhead will be paid for every reference. If the check code is inserted directly into the object file, the check code will not be subject to compiler optimization, which could be extremely useful for memory references within loops. In addition, a few registers must be reserved for the check code and not be used by the application program, in order to avoid saving and restoring the contents of those registers on every user program memory reference. Second, the protection scheme depends on the use of software programming tools that enforce the protection. Any program that is written without using these tools is able to violate the protection scheme at will. Because of this weakness, it would be difficult to provide security through software in an environment where users may be malicious, as they could hand-code programs that violate the security scheme.

6 Conclusion

In this paper we have introduced guarded pointers as a hardware mechanism to implement capability-based protection and allow fast multithreading among threads from different protection domains, including concurrent execution of user programs and the operating system. We have described the M-Machine as an example of an architecture which implements guarded pointers.

A guarded pointer is an unforgeable handle to a segment of memory. Each pointer is comprised of segment permission, length, base, and offset fields. The advent of 64-bit machines allows this information to be encoded directly in a single word, without unduly limiting the memory address space. An additional tag bit is required to prevent a user from illicitly creating a guarded pointer. Guarded pointers are an efficient implementation of capabilities without capability tables or mandatory indirection on memory access.

Guarded pointers can be used to implement a variety of software systems. Threads in different protection domains can share data merely by owning copies of a pointer into that segment. A thread can grant another thread access to private data by passing a guarded pointer to it. Protected entry points and cross-domain calls can be efficiently implemented using an entry type guarded pointer.

The costs of implementing guarded pointers are minimal. An additional tag bit is required to identify a pointer, and the virtual address space is reduced by the number of bits required to encode segment permissions and lengths. In a 64 bit machine, 54 virtual address bits are left, which is ample space for the immediate future. A small amount of hardware is also required to perform permission checking on memory operations.

Like all single global virtual address space systems, guarded pointers permit processes from different protection domains to share the cache and paging systems without compromising security. Also like these systems, guarded pointers eliminate multiple translations and permit processes to access an interleaved virtual cache without requiring multiple TLBs. However, guarded pointers also share some of the deficiencies of single address space memory systems (garbage collecting virtual address space), and capability systems (relocating and revoking access to segments).

By encoding a segment descriptor in the pointer itself and checking access permissions in the execution unit, guarded pointers obviate the need to check protection data in the cache bank. This permits in-cache sharing, which is not possible with methods that append a process or address space identifier to the cache tag, without the expense of providing protection tables in hardware. In addition, guarded pointers concentrate process state in general purpose registers instead of auxiliary or special memory, reducing process state, and facilitating fast context switching.

7 Acknowledgements

We would like to thank the other members of the M-Machine team for their contributions to this work: Andrew Chang, Whay Sing Lee, and Marco Fillo. In addition, we thank Frans Kaashoek, Brian Bershad, David Chaiken, Stuart Fiske, and the anonymous referees for their comments on various versions of this paper.

References

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for UNIX development. In *Summer 1986 Usenix Conference* (July 1986), pp. 93–112.

[2] AGARWAL, A., ET AL. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[3] ALVERSON, R., ET AL. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing* (Sept. 1990), ACM SIGPLAN Computer Architecture News, pp. 1–6.

[4] BENSOUSSAN, A., CLINGEN, C., AND DALEY, R. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM 15*, 5 (May 1972), 308–318.

[5] CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. Sharing and protection in a single address space operating system. Tech. Rep. 93-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 1993.

[6] DALEY, R. C., AND DENNIS, J. B. Virtual Memory, Processes and Sharing in MULTICS. *Communications of the ACM 11*, 5 (May 1968), 306–312.

[7] DALLY, W. J. A fast translation method for paging on top of segmentation. *IEEE Transactions on Computers 41*, 2 (1992), 247–249.

[8] DALLY, W. J., AND KAJIYA, J. T. An object oriented architecture. In *Proceedings of the 12th International Symposium on Computer Architecture* (Boston, MA, June 1985), pp. 154–161.

[9] DALLY, W. J., KECKLER, S. W., CARTER, N., CHANG, A., FILLO, M., AND LEE, W. S. M-Machine architecture v1.0. Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1994.

[10] DENNIS, J. B. Segmentation and the Design of Multiprogrammed Computer System. *JACM 12*, 4 (October 1965), 589–602.

[11] DIGITAL EQUIPMENT CORPORATION. *Alpha Architecture Handbook*. Maynard, MA, 1992.

[12] FABRY, R. Capability-based addressing. *Communications of the ACM 17*, 7 (July 1974), 403–412.

[13] HOUDEK, M. E., SOLTIS, F. G., AND HOFFMAN, R. L. IBM system/38 support for capability-based addressing. In *Proceedings of the 8th International Symposium on Computer Architecture* (May 1981), pp. 341–348.

[14] INTEL CORPORATION. *The 8086 Family User's Manual*. Santa Clara, CA, Oct. 1979.

[15] INTEL CORPORATION. *80386 Programmer's Reference Manual*. Santa Clara, CA, 1988.

[16] KECKLER, S. W., AND DALLY, W. J. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture* (Queensland, Australia, May 1992), ACM, pp. 202–213.

[17] KOLDINGER, E. J., CHASE, J. S., AND EGGERS, S. J. Architectural support for single address space operating systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)* (October 1992), ACM, pp. 175–186.

[18] LEE, R. B. Precision architecture. *IEEE Computer 22*, 1 (January 1989), 78–91.

[19] LEVY, H. M. *Capability-Based Computer Systems*. Digital Press, 1984.

[20] LONERGAN, W., AND KING, P. Design of the B5000 system. *Datamation 7*, 5 (May 1961), 28–32.

[21] MOON, D. A. Symbolics Architecture. *IEEE Computer* (1987), 43–52.

[22] MULLENDER, S. J., VAN ROSSUM, G., TANENBAUM, A. S., VAN RENESSE, R., AND VAN STAVEREN, H. Amoeba: A distributed operating system for the 1990s. *IEEE Computer 23* (May 1990), 44–53.

[23] ROSENBERG, J., AND ABRAMSON, D. MONADS-PC - a capability-based workstation to support software engineering. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, 1985* (Clayton, Australia, 1985), Department of Computer Science, Monash University, pp. 222–231.

[24] TYNER, P. *iAXP 432 General Data Processor Architecture Reference Manual*. Intel Corporation, Aloha, OR, 1981.

[25] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Symposium on Operating System Principles* (December 1993), pp. 203–216.