[12] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell, "Pilot: an operating system for a personal computer," *Communications of the ACM,* February 1980, pp. 81-92.

[13] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A structural view of the Cedar programming environment," *ACM Transactions on Programming Languages and Systems,* October 1986, pp. 419- 444.

[14] A. Silberschatz, J. L. Peterson, and P. B. Galvin, *Operating System Concepts,* Addison-Wesley Publishing Company, 1990.

References

[1] R. C. Daley, and J. B. Dennis "Virtual memory, processes, and sharing in Multics," *Communication of the ACM,* May 1968, pp. 306-312.

[2] D. M. Ritchie, and K. Thompson, "The UNIX time-sharing system," *Communication of the ACM,* July 1974, pp. 365-375.

[3] A. Tevanian, Jr., "Architecture-independent virtual memory management for parallel and distributed environments: the Mach approach," Department of Computer Science Technical Report, Carnegie Mellon University, Pittsburgh, PA, CMU-CS-88-106, December 1987.

[4] R. B. Lee, "Precision Architecture," *IEEE Computer,* January 1989, pp. 78-91,

[5] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R400 Microprocessor User's Manual,* 1991.

[6] Digital Equipment Corporation, Maynard, MA, *Alpha Architecture Handbook,* 1992.

[7] B. Ozden, and A. Silberschatz, "A taxonomy of shared address space systems," Department of Computer Sciences, The University of Texas at Austin, TR-92-33, July 1992.

[8] J. S. Chase, H. M. Levy, M. B-Harvey, and E. D. Lazowska, "How to use a 64-bit virtual address space," Department of Computer Science and Engineering, University of Washington, Seattle, TR 92-03-02.

[9] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architectural support for single address space operating systems." *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992, pp. 175-186.

[10] A. Chang, and M. F. Mergen, "801 storage: architecture and programming," *ACM Transactions on Computer Systems,* February 1988, pp. 28-50.

[11] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, "Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors," *IBM Journal of Research and Development,* January 1990, pp. 105-109.

SAS paradigm can be classified into various models, which differ in the way the address space and protection domain are managed.

Existing shared address space systems [12, 13, 8] are based on models that suffer from one or more of the following drawbacks. They require reclamation of virtual addresses, cause fragmentation of the address space, yield high space overhead or provide inadequate interprocess protection. These drawbacks can potentially decrease the performance or restrict the computation domains of the existing systems. In this paper, we proposed an alternative SAS model—the SVAS model, which eliminates these problems. We compared our model to the other SAS models and identified the hardware support for the SVAS model. We also compared our proposed architecture to the existing architectures with similar features.

from the calling procedure. This is because the content of the segment registers are changed by the called procedure, and the calling procedure cannot reach its private data sections to restore its state. The solution is to associate a *stub segment* wirh each shared procedure in a process. It is the responsibility of the stub segment to save the segment registers of the calling procedure, transfer to the called procedure during a procedure call, restore the segment registers and transfer back to the calling procedure during a return. The parameters can be passed between the procedures by sharing segments. The stub segment contains both data and code. Note that this scheme does not eliminate the conventional procedure calls.

When the operating system loads a shared procedure into a protection domain, it also adds the stub segment into the domain, and returns the address of the segment to the caller. In order to provide the same interface for both procedure calls and RPC's, the operating system loads the stub segment with the appropriate access rights. If the calling process has the right to access the procedure in its own domain, the stub segment is loaded into the protection domain of the calling process with an execute access right. Otherwise, the operating system loads the stub segment with a permission, which causes a trap to the operating system, so that the operating system can invoke a new process for the called procedure. In order to generate a trap to the operating system, either the stub segment is loaded with no execution access right into the protection domain of the calling process and the fault handler for the memory violation is altered, or the stub segment is made invalid in the address space and the page fault handler is altered, or a new permission bit is introduced which causes a trap to a separate RPC handler.

6 Conclusions

The availability of 64-bit processors [4, 5, 6], indicates a trend towards processors in which the size of the virtual address space exceeds the possible physical storage (main memory and secondary storage) by several orders of magnitude. Current technology also indicates that this gap will remain since processor densities are increasing at a rate at least as fast as memory and storage densities. 64-bit processors can safely support the SAS paradigm where all processes execute in the same address space. This is in contrast to most existing operating systems that use the PAS paradigm where each process views the entire space as dedicated to itself. The SAS paradigm simplifies the use of virtually addressed caches, and provides methods for efficient implementations of sharing code and data, interprocess communication primitives and memory management algorithms. The type of access list approach, in which a unique access identifier is associated with each object. A limited version of this scheme is implemented in the Precision Architecture, in which only four registers are provided to maintain the access identifiers. In order to allow simultaneous reads and writes to a shared object, the access identifiers given to processes are extended with a write disable bit.

5 Code Sharing

The simplest form of code sharing is to allow several processes to execute the same program. In order to facilitate this abstraction in the SAS paradigm, the private data sections of a shared program cannot be referenced with direct virtual addresses or with offsets relative to a base pointer that points to the beginning of the program (e.g., program counter). Private data must be accessed indirectly (e.g., segment register indirect, or base register with index or displacement addressing modes, depending on the memory management). Hence, our proposed dynamic address binding mechanism can support this type of code sharing.

The SVAS model can also support sharing of procedures between programs (e.g., libraries). Shared libraries can be implemented either by static linking or dynamic linking. Furthermore, the SVAS model can effectively support a more general type of code sharing; namely, any program can call any other program with a procedure call without recompilation or relocation in the same protection domain or with a remote procedure call (RPC) in a separate protection domain. A procedure call can be used when both the caller and the called programs trust each other. Otherwise an RPC is used. A procedure call has less overhead than an RPC, since it does not involve a context switch. In most PAS systems, the first case is not possible due to possible address conflicts in the calling program and the called program. A shared address space model automatically removes the problem of address conflict.

Furthermore, the system can provide a transparent interface for procedure calls and RPC's, so that a program can be invoked both with a procedure calls and RPC. This permits the system to change the access rights of the clients to the server program dynamically without recompiling the client programs. Below, we elaborate how the SVAS model can implement these abstractions with our proposed hardware. Since separately compiled procedures may use the same segment registers, segment registers have to be saved and restored at each procedure call. If the state of the caller is saved in its private data section (e.g., stack), its state cannot be restored at the return

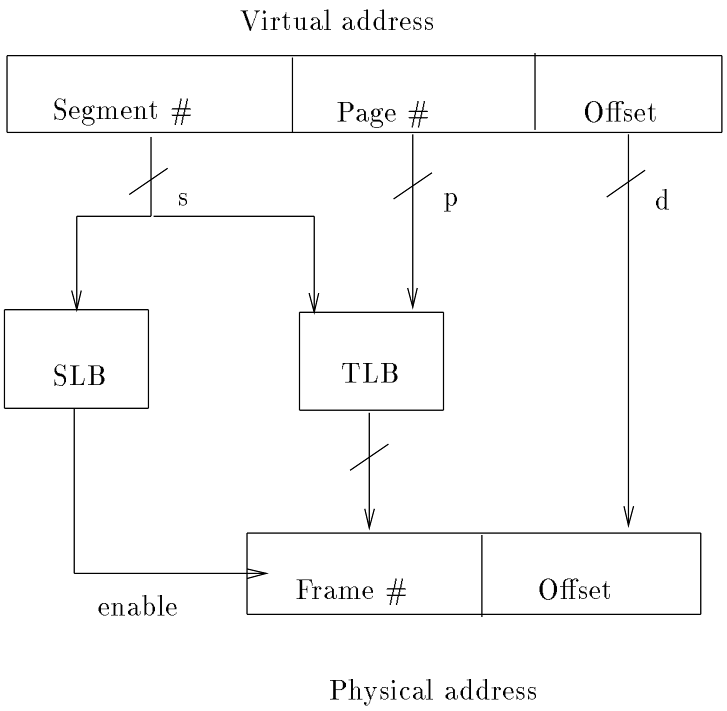

Figure 4: Address translation.

4.3 Comparison to Existing Architectures

The IBM RISC System/6000 processor [11] is a 52-bit address architecture. A short address (32-bit) is expanded to a global virtual addresses (to 52 bits) via a segment register. Although the segment register indirect addressing scheme we proposed is similar to the virtual address generation in the RISC System/6000, there are two major differences between the two architectures. First, applications in the RISC System/6000 can only specify short addresses but not virtual addresses. Second, the segment registers cannot be loaded and stored in the user (unprivileged) mode, since the contents of segment registers determines part of the protection domain of a process. These differences rule out passing virtual address pointers among processes as well as static linking.

The HP Precision Architecture is a 64-bit address architecture. The effective address calculation is similar to the proposed hardware, The user can specify both short addresses that are extended to 64-bit addresses through the segment registers (space registers) or long global virtual addresses. Hence, processes can pass virtual address pointers in user mode. However, the generation of long addresses takes four instruction cycles due to the 32-bit data path. The Precision Architecture differs from our proposed hardware in its interprocess protection scheme. Protection is based on

Virtual Segment #	Virtual Page #	Physical Frame #	Reference

Figure 2: An entry of the TLB.

Virtual Segment #	Access Rights

Figure 3: An entry of the SLB.

entries of the page table for the virtual address space. In the SVAS model, the TLB stores only the shared information (see Figure 2). A TLB entry contains the virtual address (segment and page number) as the tag and the physical address (frame number) and dirty bit as data. The SVAS model requires another associative cache to store the most recently used entries of the segment table. We refer to this cache as a *segment-lookaside buffer* (SLB). An SLB entry holds the virtual segment number as the tag and the protection bits (access rights) as data (see Figure 4).

Figure 3 illustrates the necessary address translation and access rights verification hardware for the proposed scheme. At address translation time, the segment number of the virtual address is searched in the SLB. The virtual segment and the virtual page number are searched in the TLB. These searches can be done in parallel. If the address is found in both the SLB and TLB, the frame number is available in one memory cycle. If the segment number is not in the SLB, at least one additional memory access is necessary to locate the segment number in the segment table. When the segment number is obtained, it is placed into the SLB. Similarly, if a TLB miss occurs, the page table for the shared address space is searched. The TLB and SLB misses can be handled either by hardware or with a software interrupt. When a context-switch takes place, the SLB is flushed out, but the content of the TLB remains unchanged. If the SLB entries can be tagged with process identifiers, flushing out the SLB can be eliminated.

An SLB can be smaller than a TLB, since the number of segments that a process accesses will be typically less than the number of pages accessed. The increase in the processor densities makes the addition of another associative cache as an SLB feasible. The hit rate of the SLB will be potentially higher than the TLB since the locality of references in a segment is expected to be higher than the locality in a page.

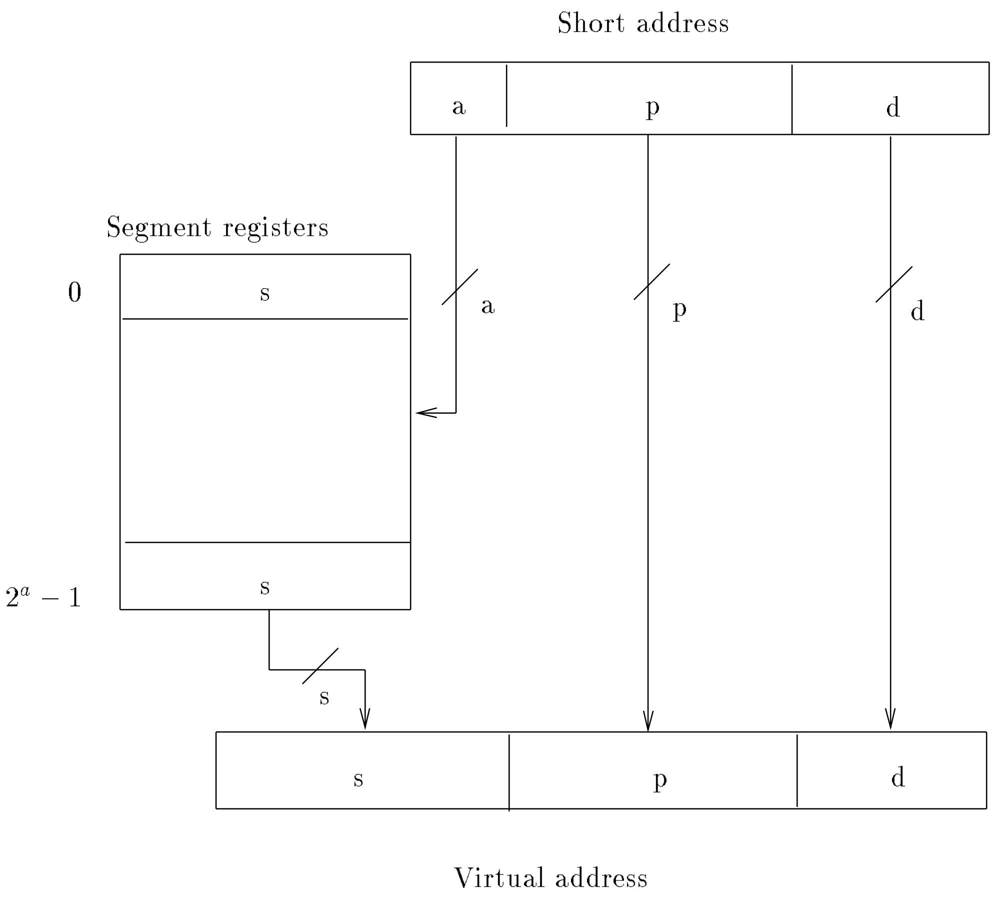

Figure 1: Virtual address generation in segment register indirect mode.

4.2 Hardware Support for Protection Domain Management

In this section, we present the hardware support for address translation and access right verification. For simplicity of presentation, we consider one privilege level. This hardware can be easily extended to provide different access rights to different privilege levels (see Section 3.3).

The operating system maintains a table for the shared address space and a table per protection domain (the capability list). The table for the shared address space contains the virtual and physical addresses of the active objects. The table for a protection domain contains an entry for each object that the process is allowed to access. Each entry contains both the segment number of the object and the access rights to the object. We refer to this structure as the segment table. At access time, both tables need to be searched to determine the address translation and the access rights.

The standard solution to speed up memory accesses is to include a translation-lookaside buffer (TLB) to the processor. The TLB is an associative cache which stores the most recently used address register or memory indirect and base register with index or displacement, are not covered here.

The processor contains 2^a segment registers. There are two sizes of address—long and short. A long address is n bits long with the fields as defined above. A short address consists of m bits, where $m < n$. The high order a bits correspond to a segment register number, the next p bits correspond to a page number within a segment, and the remaining d bits correspond to the offset within a page. The short addresses provide a contiguous local address space for the application. If a short address is specified, a bits of the address index a segment register. The segment number in the register is added as the prefix to the lower bits of the address. This concatenated address is the virtual address which is sent to the address translation logic. We refer to this mode of addressing as the segment register indirect.

The instruction specifies the addressing mode. The instruction set allows only the generation of short addresses in segment register indirect addressing mode. Figure 1 outlines the address generation in this mode. If segment register indirect addressing is disabled, the long address (virtual address) is directly sent to the address translation logic. This mode can be used when objects are bound to virtual addresses statically. The address translation and access right verification scheme is explained in Section 4.2.

With this scheme, code references can be linked to short addresses at compile time and then dynamically linked to virtual addresses at run time. Segment registers can be loaded and stored in unprivileged (user) mode. This ability enables pointer passing between processes. The sender process stores the content of a segment register into a shared memory location and the receiver process either loads a segment register from that location or references the object indirectly.

Although the number of available registers is limited, the programmer or the compiler can store the content of the registers and reassigns the registers in order to increase the number of segments that can be linked dynamically. When the operating system loads an object into the protection domain of a process, the virtual address of the object is returned to the process. The process stores the virtual address in either a segment register or a memory location for a later use in segment register indirect addressing mode (or in memory indirect addressing mode). The contents of the segment registers do not define the protection domain of a process. The addresses generated by the application are checked for the access verification as will be explained in the next section. Therefore, the application can load and store the segment registers freely without violating access rights.

correspond to a segment number, the next p bits correspond to a page number within a segment and the remaining d bits correspond to the offset within a page.

4.1 Hardware Support for Dynamic Address Binding

Our goal is to outline an architecture that can support the following properties— efficient dynamic address binding, context-independent addressing and a contiguous local address space. The first two are necessary to support the SVAS model, while the last one is a desirable property from the perspective of compilers and programmers.

Under the dynamic address binding scheme, persistent objects do no contain and are not assigned virtual addresses. This implies that an object code of a program cannot contain any virtual address. Thus, a program should be able to execute with data and code mapped into arbitrary virtual addresses during different executions of the program.

A contiguous address space gives programmers freedom to choose addresses. The private address space paradigm gives this view to the application level. On the other hand, the PAS paradigm prohibits passing virtual addresses (pointers) between processes and causes possible address conflicts when a shared object is loaded into a domain. The Multics operating system, which is a based on the PAS paradigm, solves the second problem by using dynamic address binding via linkage segments [1]. With this method, each reference from a code segment to another segment is done indirectly through a linkage segment. This method requires two memory accesses when a segment references another segment. Multics, however, does not solve the first problem. To solve this, the linkage segment scheme in Multics can be used by choosing globally unique virtual addresses for objects. Such a scheme, however, does not provide a contiguous space for the application level.

The key to meet all these properties is to provide a contiguous local address space for programs which is separate than the virtual address space and the protection domain. In this case, the virtual addresses are still accessible by the application programs without violating the protection. Hardware support for this approach requires either segment registers or a cache. In this paper, we only consider the segment register scheme, since we have not as yet worked out the details and the cost of the latter scheme.

We consider a processor architecture, in which dynamically linked objects can be accessed through segment registers, so that a reference from one segment to another takes only one memory access. We only outline the aspects of the hardware related to the dynamic address binding. Other details, such as data and address registers, and addressing modes such as register, direct (absolute), rights to any object, and permits the access rights of any domain to be changed without affecting the access rights of any other domain.

In a SAS model which is based on paging for the memory management, page numbers can be used as the access identifiers. This approach is referred as the *domain-page* method [8]. It results in long capability lists for protection domains, which will potentially increase memory access time. To reduce the size of the capability list, a unique number for each object can be used as the access identifier instead of page numbers. Under this scheme, the access identifier must also be maintained with the object in addition to the virtual address. Hence, this method increases the size of the data structures maintained for the shared address space. The SVAS model is based on paged segmentation for memory management. A segment number naturally establishes the access identifier. In this case, the size of the list per protection domain can be kept short without increasing the size of the data structures for the shared address space, since there is no need to maintain additional access identifier (segment number is part of the virtual address).

The combination of access list and capability list approaches can be used to manage multiple privilege levels. In this case, the capability list approach is used for the user mode. The access list approach can be used for the privileged modes that have the same access rights to objects in all processes that execute in these modes. The SVAS model relies on this scheme for handling different privilege levels.

The complexity of loading and removing a segment into and from a protection domain is comparable in access and capability list approaches with the exception of the domain-page method. Loading and removing a segment involve addition or deletion of one entry for each page of a segment under the domain-page method. Under the other methods, loading and removing require addition and deletion of only one entry. In any of the methods, if a segment is to be removed from all domains, the segment can be simply invalidated in the shared address space.

4 Architectural Support for the SVAS Model

In this section, we outline the necessary hardware support for the SVAS model that is based on dynamic address binding, paged segmentation, and capability list approach for protection, and is also capable to support static binding and access list approach. We consider an architecture with a virtual address consists of n bits (e.g., $n = 64$) with the following fields. The high order s bits objects are larger than the segment size will not create a major problem under dynamic address binding scheme, since objects are assigned to virtual addresses only temporarily. Paged segmentation reduces the cost of protection domain management significantly as compared to paging. Access rights are maintained only for segments. This will result in less space overhead and, in turn, shorter search time for address translation and access rights verification, which implies shorter memory access time.

3.3 Protection Domain Management

Although the address space is shared in the SAS paradigm, processes must execute in separate protection domains. The SAS models can be categorized into two groups with respect to their protection domain management— *access list* and *capability list* approaches [7]. The Pilot [12] and Cedar [13] operating systems can be classified as single protection domain systems. These systems rely on language-based protection. The Opal [8] operating system is based on capability list approach. The SVAS model is based on capability list approach for managing the user mode, and is based on the access list approach for managing privileged modes.

In the access list approach, a list is maintained for each object in the shared address space. The list contains the access rights and access identifiers of the processes that are allowed to access the object. If process-ID's are used as the access identifiers the process ID is searched in the corresponding list at access time. This scheme results in variable size access list when objects are shared. It is difficult to maintain and to provide hardware support for variable size lists to accelerate memory references [7]. A solution to this problem is to assign a unique access identifier to each object. Each process that is allowed to access the object uses the same identifier. A list of access identifiers are maintained for each process. At access time, the list of access identifiers that the process holds is searched. Under this scheme, the processes which share an object cannot have different access rights to the object. We refer to this scheme as the *access group* approach.

The capability list approach separates the maintenance of the shared address space and protection domains. A capability list is maintained for each protection domain. The capability list contains both the access identifier and the access rights of the objects that the process is allowed to access. The access identifier and access rights are stored in this list when an object is loaded into the protection domain of the process. At access time, this list is searched to determine the access rights. The capability list approach allows each protection domain to have its own set of access even an impossible task in a fragmented space. Compaction methods cannot be used to alleviate external fragmentation, since virtual addresses are statically bound to objects.

Under dynamic binding scheme, the impact of reclamation and fragmentation of the address space is not severe, since objects are assigned to virtual addresses only temporarily. On the other hand, it requires indirection to access an object and dynamic linking. With adequate hardware support, indirection has no additional run-time overhead. We outline a possible hardware support for dynamic address binding in Section 4. The overhead of dynamic linking in the SVAS model is small compared to the overhead of dynamic loading. Since paged segmentation is used as the memory management algorithm, references in programs can be offsets from a segment number. Hence, the only additional cost of dynamic linking as compared to dynamic loading is to allocate a virtual segment number at load time.

3.2 Memory Management

In general, there are three methods for managing memory—*paging, pure segmentation and paged segmentation* [14]. We do not consider here pure segmentation, because it causes external fragmentation of the physical memory and as a result, yields high overhead for the memory management. The Opal [8] operating system is based on paging, whereas the SVAS model is based on paged segmentation.

In the case of paging, the operating system needs to maintain logical segments of different sizes which are built from contiguous pages. Paging complicates address space allocation, since for all objects that are larger than a page contiguous virtual pages have to be found. On the other hand, in the case of static binding, paging is more suitable than the paged segmentation. This is because paged segmentation can cause internal fragmentation of the address space if it is used with static address binding scheme and with current technology, there may not be enough segments to bind all objects in a system statically. However, paging with static binding can cause external fragmentation of the shared address space and therefore reduce the performance of address space management. Paging also yields high overhead to manage protection domains since access rights must be maintained for each page of a process.

In the case of paged segmentation, addresses of segments are predefined. Thus, paged segmentation simplifies both the address space and memory management. The operating system can select any available segment to bind an object. Paged segmentation is more amenable to be used with dynamic address binding. Internal fragmentation and possible external fragmentation when address binding time and memory management algorithm. Although the address space is shared in the SAS paradigm, processes execute in separate protection domains. The protection domain management methods differ in the way the access rights are maintained. Some of these models are already used in the existing shared address space systems [12, 13, 8]. Instead of comparing the SVAS model to each SAS model used in existing systems, we compare different address space and protection domains management methods and argue that the methods which form the SVAS model are preferable to the other methods.

We also note that there are operating systems that can be classified as a combination of the SAS and PAS paradigms [10, 11]. In these systems, each process has an independent address space which also defines its protection domain. At run time, private addresses are translated to globally unique virtual addresses. These systems benefit from some advantages of the SAS paradigm and the main advantage of the PAS paradigm, which is providing a contiguous address space to the programmer [7]. These systems, however, suffer from the following drawbacks. First, virtual addresses (pointers) cannot be passed between processes. Second, (private) address conflicts are possible when a shared object is loaded into a domain.

3.1 Address Binding Time

The SAS models can be categorized into two groups with respect to their address binding time—*dynamic* and *static address binding.* [7]. The Pilot [12], Cedar [13] and Opal [8] operating systems are based on static address binding, whereas the SVAS model is primarily based on dynamic binding, but it also supports static binding for system-wide resources.

The major drawback of static address binding scheme is that the virtual addresses need to be reclaimed once an object is deleted. Since there can be other objects that reference the deleted object with its virtual address, the address cannot be reused unless it is reclaimed. If addresses are not reused the address space can be filled up due to misuse. Garbage collection to reclaim addresses incurs high overhead on the address space management [7]. Methods to decrease the overhead of reclamation, such as reference count and lock-key schemes are prone to errors and intentional misuse [7].

Static binding can cause severe fragmentation problems. Fragmentation complicates address space allocation. For all objects that are larger than a page or segment (depending on the memory management scheme), contiguous addresses have to be found which is a complicated and sometimes the virtual address space into predetermined segments, and each segment is in turn broken into fixed-size pages. The size of a segment can alter depending on the size of the object. The size of the segment is a multiple of the page size; hence, the size of the segment is equal to or larger than the size of the object. Any of the segments can be selected to bind an object, and any of the available frames can be used to map a page of a segment.

A *process* is an environment in which a program is executed. The type of operation that a process is allowed to execute on a segment is called the *access right.* The access rights to a segment are defined by the access rights to the corresponding object. Although the address space is shared, the SVAS model provides *interprocess protection.* Each process executes within a *protection domain* (or execution domain). A protection domain consists of the set of the segments that the process is allowed to access, and the access rights to these segments. Some architectures support process execution in multiple privilege levels (e.g., kernel, operating system supervisor, user). For such architectures, we will consider a protection domain per privilege level.

Segments are loaded into a protection domain when the process is generated or when the process first accesses the segment. The protection domain management for the user mode is based on the *capability list approach.* A list is maintained for each process, which contains both the segment number and the access rights of the objects that the process is allowed to access. The protection domain management for privileged modes, such as kernel, can be accomplished by the *access list approach.* In this case, access rights for the privilege level are maintained in a list for each object in the address space.

Processes that share a protection domain are called *lightweight processes* (or threads). A process is represented in the operating system by a process control block, (PCB). The protection domain of the process is represented with a field in the PCB. If two processes share a protection domain, it means that they share the operating system data structure which implements the protection domain, namely the corresponding field of the PCB. In contrast, heavyweight processes in the SVAS model do not share a protection domain and the address space is not an attribute of a process (not a field in the PCB).

3 Comparison of the SVAS Model to other SAS Models

The SAS paradigm can be classified into various models, which differ in the way the address space and protection domains are managed [7]. The address space management methods differ in virtual

Some of the models do not allow each protection domain to have its own set of access rights to any object.

In this paper, we propose an alternative model which alleviates these problems. We refer to our model as the *shared virtual address space* (SVAS) model. The focus of this paper is the comparison of the SVAS model with the other SAS models and the identification of the hardware support for this model.

The remainder of the paper is organized as follows. In the next section, the SVAS model is defined. Section 3 compares the SVAS model to other SAS models. Section 4 identifies dynamic address binding and interprocess protection hardware support for the SVAS model, and compares the proposed hardware with the existing processor architectures, while Section 5 discusses how the SVAS model can support code sharing. Finally, Section 6 presents our conclusions.

2 The SVAS Model

The SVAS model consists of two components: the shared address space and the set of all protection domains. The *shared address space* is the collection of all segments, where a *segment* is a range of virtual addresses. An *object* is a collection of logically related code or data, such as a file or a program. An object is assigned to a segment and loaded into the shared address space when a process accesses it (when the object becomes active). At this point, the object is bound to a virtual address, namely, to the address of the segment. We refer to this approach as *dynamic address binding.* The segment is released when all the processes that access the object terminate. Hence, an object can obtain more than one virtual address during its lifetime. An object can be either temporary or persistent. There is a storage hierarchy in which the persistent objects are stored permanently and a naming context, which identifies the persistent objects uniquely, such as directory structure, in which each object has a symbolic name. The SVAS model does not rule out static address binding. Static address binding refers to allocation of virtual addresses to objects when they are first created. In this case, an object is bound to the same virtual addresses during its lifetime, and these address cannot be used to name any other object during this period. The choice is left as a policy. For example, system administration may choose to statically bind some system wide services (e.g., shared libraries) to absolute virtual addresses.

The implementation of segments is determined by the memory management algorithm of the operating system. The SVAS model is based on *paged segmentation.* Paged segmentation divides

1 Introduction

Most operating systems on computers with 32-bit or smaller address spaces are based on the *private address space* (PAS) paradigm where each process has a separate address space [1, 2, 3]. This paradigm has emerged as a result of the size of the address space in 32-bit or smaller architectures, in which the number of possible virtual addresses is relatively small. In a small address space system, there can be more objects that need to be accessed by processes than the number of available virtual addresses. Thus, in the PAS paradigm, each process views the entire space as dedicated to itself, so that each process is provided with sufficient number of addresses to name the objects that it accesses. However, with the recent emergence of the 64-bit processors [4, 5, 6], the private address space paradigm can be replaced with the *shared address space* (SAS) paradigm in which all processes execute concurrently in a shared global address space. Since the number of addresses is sufficient to name the objects that all processes access, the 64-bit processors can safely support the SAS paradigm.

The unifying property of the SAS paradigm is *context-independent addressing*. We define context-independent addressing as follows. Two concurrent processes have a common virtual address, if and only if they share an object. This property simplifies the use of virtually addressed caches, and provides methods for efficient implementations of sharing code and data, interprocess communication primitives and memory management algorithms [7, 8, 9].

There are two key issues in the design of a shared address space system— address space management and protection domain management. The SAS paradigm can be categorized into different models with respect to the address space management and protection domain management [7]. In general, the address space management methods differ in virtual address binding time and memory management. Although the address space is shared in the SAS paradigm, processes execute in separate protection domains. The protection domain management methods differ in the way the access rights are maintained. The existing shared address space systems [12, 13, 8] are based on models which suffer from one or more of the following drawbacks. Some of the models require reclamation of virtual addresses. Garbage collection methods to reclaim addresses incur high performance overhead on the address space management. Some of the models cause fragmentation of the address space, which incurs overhead on the address space management, since the address allocation in a fragmented space is complicated and sometimes impossible. Some of the models yield high space overhead, and as a result, potentially high run time overhead for protection domain management.

The Shared Virtual Address Space Model[1]

Banu Özden
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

Avi Silberschatz
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

64-bit processors can safely support the *shared address space* (SAS) paradigm where all processes execute in the same address space. This is in contrast to most existing operating systems that use the *private address space* (PAS) paradigm where each process views the entire space as dedicated to itself. The SAS paradigm simplifies the use of virtually addressed caches, and provides methods for efficient implementations of sharing code and data, interprocess communication primitives and memory management algorithms. The SAS paradigm can be classified into various models, which differ in the way the address space and protection domain are managed. Existing shared address space systems are based on SAS models that suffer from a number of drawbacks that can potentially either decrease the performance or restrict the computation domains. In this paper, we define a new SAS model – the *shared virtual address space* (SVAS) model, argue that this model is preferable over the other SAS models. We also identify the necessary hardware support for the SVAS model.

[1]This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant Nos. IRI-9003341 and IRI-9106450, and grants from the IBM and Hewlett-Packard corporations.

THE SHARED VIRTUAL ADDRESS SPACE MODEL

Banu Özden
Avi Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-92-37 November 1992