USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

Proceedings of the USENIX Annual Technical Conference

Monterey, California, USA, June 6-11, 1999

The UVM Virtual Memory System

Charles D. Cranor and Gurudatta M. Parulkar
Washington University

The UVM Virtual Memory System

Charles D. Cranor Gurudatta M. Parulkar

Department of Computer Science
Washington University
St. Louis, MO 63130
{chuck,guru}@arl.wustl.edu

Abstract

We introduce UVM, a new virtual memory system for the BSD kernel that has an improved design that increases system performance over the old Mach-based 4.4BSD VM system. In this paper we present an overview of both UVM and the BSD VM system. We focus our discussion on the design decisions made when creating UVM and contrast the UVM design with the less efficient BSD VM design. Topics covered include mapping, memory object management, anonymous memory and copy-on-write mechanisms, and pager design. We also present an overview of virtual memory based data movement mechanisms that have been introduced in BSD by UVM. We believe that the lessons we learned from designing and implementing UVM can be applied to other kernels and large software systems. Implemented in the NetBSD operating system, UVM will completely replace BSD VM in NetBSD 1.4.

1 Introduction

In this paper we introduce UVM[1], a new virtual memory system that replaces the 4.4BSD virtual memory system in BSD-based operating systems. UVM is the third generation BSD virtual memory system that improves the performance and reduces the complexity of the BSD kernel. UVM's improved performance is particularly useful to applications that make heavy use of VM features such as memory-mapped files and copy-on-write memory.

Versions of BSD prior to 4.4BSD used the old BSD-VAX virtual memory system that was tightly bound to the VAX architecture and lacked support for memory-mapped files (`mmap`) and fine-grain data structure locking for multiprocessors. To address these issues, the old VAX-based VM system was replaced with a new VM system for 4.4BSD [12]. The 4.4BSD virtual memory system (BSD VM) is a modified version of the virtual memory system that was written for Carnegie Mellon University's Mach operating system [18]. The BSD VM system features a clean separation of machine-dependent functions, support for `mmap`, and fine-grain data structure locking suitable for multiprocessor systems.

1.1 Why Replace BSD VM?

The BSD VM system has four main drawbacks that contributed to our decision to replace it: complex data structures, poor performance, no virtual memory based data movement mechanisms, and poor documentation.

The data structures and functions used by BSD to manage memory are complex and thus difficult to maintain. This is especially true of the structures used to represent copy-on-write mappings of memory objects. As memory objects are copied using the copy-on-write mechanism [2] (e.g., during a fork) they are linked together in lists called *object chains*. If left unchecked, use of the copy-on-write mechanism can cause object chains to grow quite long. Long object chains are a problem for two reasons. First, long object chains slow memory search times by increasing the number of objects that need to be checked to locate a requested page. Second, long object chains are likely to contain inaccessible redundant copies of the same page of data, thus wasting memory. If the BSD VM system allows too many pages of memory to be wasted this way the system's swap area will eventually become filled with redundant data and the system will deadlock. This condition is known as a *swap memory leak* deadlock. To avoid problems associated with long object chains, the BSD VM system attempts to reduce their length by using a complex *collapse* operation. To successfully collapse an object chain, the VM system must search for an object that contains redundant data and is no longer needed in the chain. If a redundant object is found, then it is either bypassed or discarded. Note that even a successfully col-

[1] Note that "UVM" is a name, not an acronym

lapsed object chain can still contain inaccessible redundant pages. The collapse operation can only repair swap memory leaks after they occur, it cannot prevent them from happening.

BSD VM exhibits poor performance due to several factors. First, the overhead of object chain management slows down common operations such as page faults and memory mapping. Second, I/O operations in BSD VM are performed one page at a time rather than in more efficient multi-page clusters, thus slowing paging response time. Third, BSD VM's integration into the BSD kernel has not been optimized. For example, unreferenced memory-mapped file objects are cached at the I/O system (vnode) layer and redundantly at the virtual memory layer as well. Finally, several virtual memory operations are unnecessarily performed multiple times at different layers of the BSD kernel.

Another drawback of the BSD VM system is its lack of virtual memory based data movement mechanisms. While BSD VM does support the copy-on-write mechanism, it is not possible in BSD VM for the virtual memory system to safely share memory it controls with other kernel subsystems such as I/O and IPC without performing a costly data copy. It is also not possible for processes to easily exchange, copy, or share chunks of their virtual address space between themselves.

Finally, the BSD VM system is poorly documented. While some parts of the BSD kernel such as the networking and IPC system are well documented [20], the BSD VM system lacks such detailed documentation and is poorly commented. This has made it difficult for developers of free operating systems projects such as NetBSD [17] to understand and maintain the 4.4BSD VM code.

1.2 The UVM Approach

One way to address the shortcomings of the BSD virtual memory system is to try and evolve the data structures and functions BSD inherited from Mach into a more efficient VM system. This technique has been successfully applied by the FreeBSD project to the BSD VM system. However, the improved VM system in FreeBSD still suffers from the object chaining model it inherited from BSD VM. An alternative approach is to reimplement the virtual memory system, reusing the positive aspects of the BSD VM design, replacing the parts of the design that do not work well, and adding new features on top of the resulting VM system. This is the approach we used for UVM. In UVM we retained the machine-dependent/machine-independent layering and mapping structures of the BSD VM system. We replaced the virtual memory object, fault handling, and pager code. And, we introduced new virtual memory based data movement mechanisms into UVM. When combined with I/O and IPC system changes currently under development, these mechanisms can reduce the kernel's data copying overhead.

UVM's source code is freely available under the standard BSD license. UVM was merged into the NetBSD operating system's master source repository in early 1998 and has proven stable on the architectures supported by NetBSD including the Alpha, I386, M68K, MIPS, Sparc, and VAX. UVM will appear as the official virtual memory system of NetBSD in NetBSD release 1.4. A port of UVM to OpenBSD is also in progress.

In this paper we present the design and implementation of the UVM virtual memory system, highlighting its improvements over the BSD VM design. In Section 2 we present an overview of BSD VM and UVM. In Sections 3, 4, 5, 6, and 7 we present the design of UVM's map, object, anonymous memory, pager, and data movement facilities, respectively. In Section 8 we present the overall performance of UVM. Section 9 contains related work, and Section 10 contains our concluding remarks and directions for future research.

2 VM Overview

Both BSD VM and UVM can be divided into two layers: a small machine-dependent layer, and a larger machine-independent layer. The machine-dependent layer used by both BSD and UVM is called the *pmap* layer. The pmap layer handles the low level details of programming a processor's MMU. This task consists of adding, removing, modifying, and querying the mappings of a virtual address or of a page of physical memory. The pmap layer has no knowledge of higher-level operating system abstractions such as files. Each architecture supported by the BSD kernel must have its own pmap module. Note that UVM was designed to use the same machine-dependent layer that BSD VM and Mach use. This allows pmap modules from those systems to be reused with UVM, thus reducing the overhead of porting a UVM-based kernel to a new architecture.

The machine-independent code is shared by all BSD-supported processors and contains functions that perform the high-level operations of the VM system. Such functions include managing a process' file mappings, requesting data from backing store, paging out memory when it becomes scarce, managing the allocation of physical memory, and managing copy-on-write memory. Figure 1 shows the five main abstractions that correspond to data structures in both BSD VM and UVM that the activities of the machine-independent layer are centered around. These abstractions are:

virtual memory space: describes both the machine-dependent and machine-independent parts of a pro-

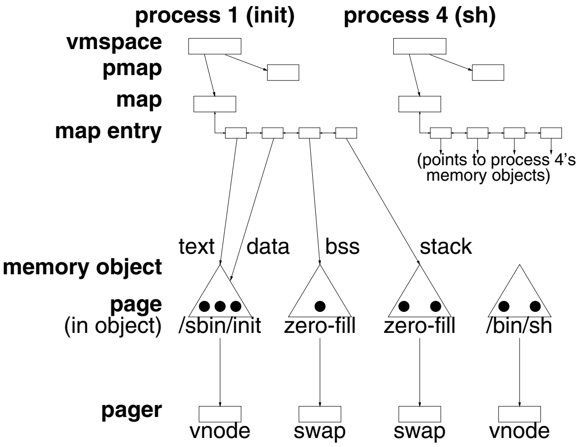

Figure 1: The five main machine-independent abstractions. The triangles represent memory objects, and the solid circles within them represent pages. A memory object can contain any number of pages. Note that the text and data areas of a file are different parts of a single object.

cess' virtual address space. The `vmspace` structure contains pointers to a process' pmap and memory map structures, and contains statistics on the process' memory usage.

memory map: describes the machine-independent part of the virtual address space of a process or the kernel. Both BSD VM and UVM use map (`vm_map`) structures to map memory objects into regions of a virtual address space. Each map structure on the system contains a sorted doubly-linked list of *entry* structures. Each entry structure contains a record of a mapping in the map's virtual address space. This record includes information such as starting and ending virtual address and a pointer to the memory object mapped into that address range. The entry also contains the attributes of the mapping. Attributes include protection, memory usage pattern (advice), and wire count. The kernel and each process on the system have their own map structures to handle the allocations of their virtual address space.

memory object: describes a file, a zero-fill memory area, or a device that can be mapped into a virtual address space. Memory objects contain a list of pages that contain data from that object. In BSD VM, a memory object consists of one or more `vm_object` structures. In UVM, a memory object consists of either a `vm_amap` anonymous memory map structure or a `uvm_object` structure (or both). The details of the handling of memory objects in both BSD VM and UVM are discussed in Section 4 and Section 5.

pager: describes how backing store can be accessed. Each memory object on the system has a pager that points to a list of functions used by the object to fetch and store pages between physical memory and backing store. Pages are read in from backing store when a process faults on them (or in anticipation of a process faulting on them). Pages are written out to backing store at the request of a user (e.g., via the `msync` system call), when physical memory is scarce, or when the object that owns the pages is freed.

page: describes a page of physical memory. When the system is booted a `vm_page` structure is allocated for each page of physical memory that can be used by the VM system[2].

In Figure 1, the system has just been booted single-user so there are only two processes (`init` and `sh`). The `init` process has four entries in its memory map. These entries map the process' text, data, bss, and stack. The entries are sorted by starting virtual address. Each entry describes a mapping of a memory object into `init`'s address space. Note that a single memory object can be mapped into different areas of an address space. For example, the `/sbin/init` file is mapped into `init`'s address space twice, once for the text and once for the data. These mappings must be separate because they have different protections. Each memory object has a list of pages containing its resident data, and a pointer to a pager that can transfer data between an object's pages and backing store. Note that each process' map structure has an associated pmap structure that contains the low-level machine-dependent memory management information (e.g., page tables) for that process' virtual address space.

When a process attempts to access an unmapped area of memory a page fault is generated. The VM system's page fault routine resolves page faults by locating and mapping the faulting page. In order to find which page should be mapped, the VM system must look in the process' map structure for the entry that corresponds to the faulting address. If there is no entry mapping the faulting address an error signal is generated. If an object is mapped at the faulting address, the VM system must determine if the requested data is already resident in a page. If so, that page can be mapped in. If not, then the fault routine must issue a request to the object's pager to make the data resident and resolve the fault.

[2] On a few systems that have small hardware page sizes, such as the VAX, the VM system has a page structure that manages two or more hardware pages. This is all handled in the pmap layer and thus is transparent to the machine-independent VM code.

The following sections examine the design and management of these five abstractions in more detail.

3 Memory Maps

UVM introduces two important improvements to memory maps. First, we have redesigned the memory mapping functions so that they are more efficient and secure. Second, we have greatly reduced map entry fragmentation due to memory wiring.

3.1 Memory Mapping Functions

The `uvm_map` and `uvm_unmap` functions are two of a number of functions that perform operations on maps. The `uvm_map` function is used to establish a new memory mapping with the specified attributes. The `uvm_map` function operates by locking the map, adding the mapping, and then unlocking the map. The BSD VM system does not have an equivalent function to `uvm_map`. Instead, BSD VM provides a function that establishes a mapping with default attributes and a set of functions that change the attributes of a mapping. This is both inefficient and insecure. It is inefficient because it requires extra map locking and lookup steps to establish a mapping with non-default values. For example, the default BSD VM protection is read-write, and thus establishing a read-only mapping under BSD VM is a two step process. First, the mapping must be established with the default protection. Second, the map must be relocked and the desired mapping located again in order to change its protection from read-write to read-only. Note that when establishing a read-only mapping, there is a brief period of time between the first and second step where the mapping has been fully established with a read-write protection. Under a multithreaded kernel two threads sharing the same address space could exploit this window to bypass system security and illegally modify read-only data.

Both BSD VM and UVM have unmap functions with the same API, however the internal structure of these two functions differ. The BSD VM unmap function keeps the target map locked for a longer period of time than necessary, thus blocking other threads from accessing it. In BSD VM, an unmap operation is performed by locking the map, removing the requested map entries, dropping the references to the mapped memory objects, and then unlocking the map. Though the map is locked throughout BSD VM's unmap operation, it really only needs to be locked when removing entries from the map. The target map does not need to be locked to drop references to memory objects (note that dropping the final reference to a memory object can trigger lengthy I/O operations). UVM's unmap function breaks the unmap operation into two phases. In the first phase the target map is locked while the requested map entries are removed. Once this is complete, the map is unlocked and the memory object references can be dropped. The second phase is done with the target map unlocked, thus reducing the total amount of time access to the target map is blocked.

3.2 Wiring and Map Entry Fragmentation

Map entry fragmentation occurs when an area of virtual memory mapped by a single map entry is broken up into two or three adjoining pieces, each with their own map entry. Map entry fragmentation is undesirable for a number of reasons. First, the more entries a map has the longer it takes to perform operations on it, for example, searching the map for the proper entry when resolving a page fault. Second, the process of fragmenting a map entry can add overhead to a mapping operation. To fragment a map entry, new map entries must be allocated and initialized, and additional references to backing objects must be gained. Finally, in the case of the kernel, the total number of available map entries is fixed. If the kernel's pool of map entries is exhausted then the system will fail. While map entry fragmentation is unavoidable in many cases, it is clearly to the kernel's advantage to reduce it as much as possible.

Map entry fragmentation occurs when modifications are made to only part of an area of virtual memory mapped by an entry. Since all pages of virtual memory mapped by a single map entry must have the same attributes, the entry must be fragmented. For example, the adjoining text and data segments of the `init` process shown in Figure 1 must be mapped by separate map entries because they have different protections. Once a map entry has been fragmented neither BSD VM nor UVM will attempt to reassemble it in the unlikely event that the attributes are changed to be compatible.

One of the most frequent causes of map entry fragmentation is the wiring and unwiring of virtual memory. Wired memory is memory that must remain resident in physical memory, and thus cannot be paged out. In BSD, there are five ways for memory to be wired. Unlike BSD VM, UVM avoids map entry fragmentation and the penalties associated with it in four out of five of these cases by taking advantage of the fact that the wired state of a page is often stored in other areas of memory in addition to the entry mapping it, and thus there is no need to disturb the map structure. Memory is wired by the BSD kernel in the following cases:

- The memory used to store the kernel's code segment, data segment, and dynamically allocated data structures is wired to prevent the kernel from taking an unexpected page fault in critical code. Since this memory is always wired, there is no need for UVM to note this fact in the kernel's map structure.

- Each process on the system has a user structure containing its kernel stack, signal information, and process control block. A process' user structure must be wired as long as the process is runnable. When a process is swapped out its user structure is unwired until the process is swapped back in. In this case the wired state of the user structure is stored in the process' proc structure and there is no need for UVM to store it in the kernel's map as well.

- The sysctl system call is used to query the kernel about its status. The sysctl call temporarily wires the user's buffer before copying the results of the call to it. This is done to minimize the chances of a page fault causing inconsistent data to be returned. Rather than store the wired state of the user's buffer in the process' map, UVM stores this information on the process' kernel stack since the buffer is only wired while the sysctl operation is in progress.

- The kernel function physio is used to perform raw I/O between a device and a user process' memory. Like sysctl, physio temporarily wires the user's buffer while the I/O is in progress to keep it resident in physical memory. And like sysctl, UVM stores the wired state of the users' buffer on the process' kernel stack.

- The kernel provides the mlock system call to allow processes to wire their memory to avoid page faults in time-critical code. In this case the wired state of memory must be stored in the user process' map because there is no other place to store it.

In addition to these five cases, under the i386 architecture the machine-dependent code uses wired memory to allocate hardware page tables. Under UVM the wired state of page table memory is stored only in the i386's pmap structure rather than in both the pmap structure and the user process' map.

By reducing the amount of map entry fragmentation due to wired memory, we significantly lowered map entry demand under UVM. For example, consider the statically linked program cat and the dynamically linked program od. On the i386 platform, BSD VM requires 11 map entries for cat and 21 for od, while UVM requires only six map entries for cat and 12 for od. The difference between BSD VM and UVM is due to the user structure allocation, the sysctl system call, and the i386's pmap page table allocation routine. We found that calls to mlock and physio seldom occur under normal system operation. Table 1 shows a comparison of the number of allocated map entries for several common operations. While the effect of this reduction in the number of allocated map entries on overall system performance is minimal, it should be noted that the total number of map entries available for the kernel is fixed and if this pool is exhausted the system will panic. This could become a problem under BSD VM since each process requires two kernel map entries.

Operation	**Number of Map Entries**	
	BSD	**UVM**
cat (static link)	11	6
od (dynamic link)	21	12
single-user boot	50	26
multi-user boot (no logins)	400	242
starting X11 (9 processes)	275	186

Table 1: Comparison of the number of allocated map entries on the i386 for some common operations. On an i386 a map entry is fifty-six bytes.

4 Memory Objects

UVM manages memory objects significantly differently than BSD VM. In BSD VM, the memory object structure is considered a stand-alone abstraction under the control of the VM system. BSD VM controls when objects are allocated, when they can be referenced, and how they can be used. In contrast, in UVM the memory object structure is considered a secondary structure designed to be embedded within some larger structure in order to provide UVM with a handle for memory mapping. The structure in which UVM's memory object is embedded is typically part of a structure managed externally to the VM system by some other kernel subsystem. For example, UVM's object structure for file data is embedded within the I/O system's vnode structure. The vnode system handles the allocation of UVM's memory object structure along with the allocation of the vnode structures. All access to the memory object's data and state is routed through the object's pager functions. These functions act as bridge between UVM and the external kernel subsystem that provides UVM with its data (see Section 6).

UVM's style of management of memory objects is preferable to BSD VM's style for several reasons. First, UVM's management of memory objects is more efficient than BSD VM's. In UVM, memory objects are allocated and managed in cooperation with their data source (typically vnodes). In BSD VM, memory objects and their data sources must be allocated and managed separately. This causes the BSD VM system to duplicate work that the data source subsystem has already performed. BSD VM must allocate more structures and have more object management code than UVM to perform the same operations.

Second, UVM's memory object structure is more flexible than BSD VM's structure. By making the memory object an embeddable data structure, it is easy to make any kernel abstraction memory mappable. Additionally, UVM's routing of object requests through its pager operations gives the external kernel subsystem that generates the memory object's data a finer grain of control over how UVM uses it.

Finally, UVM's memory object management structure creates less conflict between the VM system and external kernel subsystems such as the vnode subsystem. BSD's vnode subsystem caches unreferenced vnodes in physical memory in hopes that they will be accessed again. If vnodes become scarce, then the kernel recycles the least recently used unreferenced vnode. In the same way, the BSD VM system caches unreferenced memory objects. While vnode structures are allocated when a file is opened, read, written, or memory mapped, BSD VM vnode-based memory objects are allocated only when a file is memory mapped. When an unreferenced memory object is persisting in BSD VM's object cache, the VM system gains a reference to the object's backing vnode to prevent it from being recycled out from under it. Unfortunately, this also means that there are times when the most optimal unreferenced vnode to recycle is in BSD VM's object cache, resulting in the vnode system choosing a non-optimal vnode to recycle. Another problem with the BSD VM object cache is that it is limited to one hundred unreferenced objects in order to prevent the VM system from holding too many active references to vnode structures (preventing recycling). If the BSD VM system wants to add an unreferenced object to a full cache, then the least recently used object is discarded. This is less than optimal because the object's vnode data may still be persisting in the vnode system's cache and it would be more efficient to allow the memory object to persist as long as its vnode does.

Rather than having two layers of unreferenced object caching, UVM has only one. Instead of maintaining its own cache, UVM relies on external kernel subsystems such as the vnode system to manage the unreferenced object cache. This reduces redundant code and allows the least recently used caching mechanism to be fairly applied to both vnodes and memory objects. When recycling a vnode, UVM provides the vnode subsystem with a hook to terminate the memory object associated with it. This change can have a significant effect on performance. For example, consider a web server such as Apache that transmits files by memory mapping them and writing them out to the network. If the number of files in the server's working set is below the one-hundred-file limit, then both BSD VM and UVM can keep all the file data resident in memory. However, if the working set grows beyond one hundred files, then BSD VM flushes older inactive objects out of the object cache (even if memory is available). This results in BSD VM being slowed by disk access. Figure 2 shows this effect measured on a 333MHz Pentium-II. To produce the plot we wrote a program that accesses files in the same way as Apache and timed how long it took to memory map and access each byte of an increasing number of files.

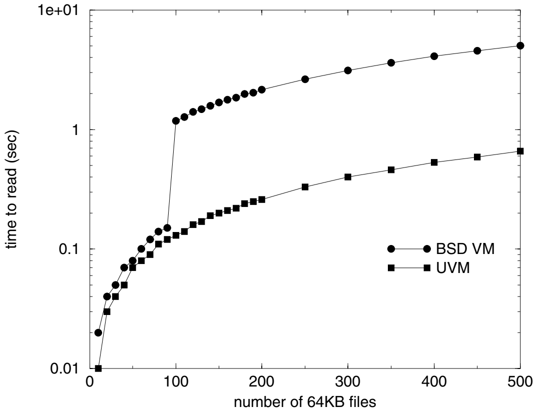

Figure 2: BSD VM object cache effect on file access

5 Anonymous Memory

Anonymous memory is memory that is freed as soon as it is no longer referenced. This memory is referred to as *anonymous* because it is not associated with a file and thus does not have a file name. Anonymous memory is paged out to the swap area when memory is scarce. Anonymous memory is used for a number of purposes in a Unix-like operating system including for zero-fill mappings (e.g., bss and stack), for System V shared memory, for pageable areas of kernel memory, and to store changed pages of a copy-on-write mapping. A significant part of the code used to manage anonymous memory is dedicated to controlling copy-on-write memory. In this section we first present a brief overview of the management of anonymous memory in both BSD VM and UVM. We then describe the improvements introduced in UVM which result in the elimination of swap memory leaks, a more efficient copy-on-write mechanism, and less complex code.

5.1 BSD VM Anonymous Memory

Creating an anonymous zero-fill mapping under BSD VM is a straight forward process. BSD VM simply allocates an anonymous memory object of the specified size and inserts a map entry pointing to that object into a map. On the other hand, the management of copy-on-write memory under BSD is more complex.

The BSD VM system manages copy-on-write mappings of memory objects by using *shadow objects*. A shadow object is an anonymous memory object that contains the modified pages of a copy-on-write mapped memory object. The map entry mapping a copy-on-write area of memory points to the shadow object allocated for it. Shadow objects point to the object they are shadowing. When searching for pages in a copy-on-write mapping, the shadow object pointed to by the map entry is searched first. If the desired page is not present in the shadow object, then the underlying object is searched. The underlying object may either be a file object or another shadow object. The search continues until the desired page is found, or there are no more underlying objects. The list of objects that connect a copy-on-write map entry to the bottom-most object is called a *shadow object chain*.

The upper row of Figure 3 shows how shadow object chains are formed in BSD VM. In the figure, a three-page file object is copy-on-write memory mapped into a process' address space. The first column of the figure shows how copy-on-write mappings are established. To establish a copy-on-write mapping the BSD VM system allocates a new map entry, sets the entry's needs-copy and copy-on-write flags, points the map entry at the underlying object (usually a file object), and inserts it into the target map. The needs-copy flag is used to defer allocating a new memory object until the first write fault on the mapping occurs. Once a write fault occurs, a new memory object is created and that object tracks all the pages that have been copied and modified due to write faults. Under BSD VM, needs-copy indicates that the mapping requires a shadow object the next time the mapped memory is modified. Read access to the mapping will cause the underlying object's pages to be mapped read-only into the target map.

The second column in Figure 3 shows what happens when the process writes to the middle page of the object. Since the middle page is either unmapped or mapped read-only, writing to it triggers a page fault. The VM system's page fault routine must catch and resolve this fault so that process execution can continue. The fault routine looks up the appropriate map entry and notes that it is a needs-copy copy-on-write mapping. It first clears needs-copy by allocating a shadow object and inserting it between the map entry and the underlying file. Then it copies the data from the middle page of the backing object into a new page that is inserted into the shadow object. The shadow object's page can then be mapped read-write into the faulting process' address space. Note that the shadow object only contains the middle page. Other pages will be copied only if they are modified.

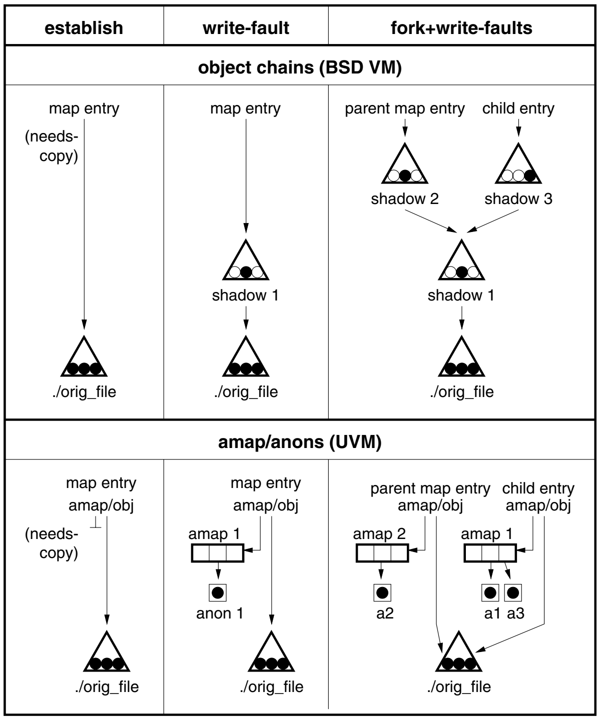

Figure 3: The copy-on-write mechanisms of BSD VM and UVM. In the figures a process establishes a copy-on-write mapping to a three page file (the solid black circles represent pages). When the mapping is established the *needs-copy* flag is set. After the first write fault the needs-copy flag is cleared by allocating a shadow object or an amap. If the process forks and more write faults occur additional shadow objects and amaps are allocated.

The third column in Figure 3 shows the BSD VM data structures after the process with the copy-on-write mapping forks a child, the parent writes to the middle page, and the child writes to the right-hand page. When the parent forks, the child receives a copy-on-write copy of the parent's mapping. This is done by write protecting the parent's mappings and setting needs-copy in both processes. When the parent faults on the middle page, a second shadow object is allocated for it (clearing needs-copy) and inserted on top of the first shadow object. When the child faults on the right-hand page the same thing happens, resulting in the allocation of a third shadow object.

5.2 UVM Anonymous Memory

UVM manages anonymous memory using an extended version of the *anon* and *amap* abstractions first introduced in the SunOS VM system [4, 9, 13]. An anon is a data structure that describes a single page of anonymous memory, and an amap (also known as an "anonymous map") is a data structure that contains pointers to a set of anons that are mapped together in virtual memory. UVM's amap-based anonymous memory system differs from SunOS' system in four ways. First, UVM's anonymous memory system introduces support for Mach-style memory inheritance and deferred creation of amaps (via the needs-copy flag). Second, in SunOS the anonymous memory system resides below the vnode pager interface and was not designed to be visible to generic VM code. In UVM, we expose the anonymous memory system to the pager-independent code, thus allowing it to be centrally managed and used by all pagers and the IPC and I/O systems. Third, SunOS' pager structure requires that each pager handle its own faults. UVM, on the other hand, has a general purpose page fault handler that includes code to handle anonymous memory faults. Finally, in UVM we separate the implementation of amaps from the amap interface in order to easily allow the amap implementation to change.

In BSD VM, a copy-on-write map entry points to a chain of shadow objects. There is no limit on the number of objects that can reside in a single shadow object chain. UVM, on the other hand, uses a simple two-level mapping scheme consisting of an upper amap anonymous memory layer and a lower backing object layer. In UVM, a copy-on-write map entry has pointers to the amap and underlying object mapped by that entry. Either pointer can be null. For example, a shared mapping usually has a null amap pointer and a zero-fill mapping has a null object pointer.

UVM's anon structure contains a reference counter and the current location of the data (i.e., in memory or on backing store). An anon with a single reference is considered writable, while anons referenced by more than one amap are copy-on-write. To resolve a copy-on-write fault on an anon, the data is copied to a newly allocated anon and the reference to the original anon is dropped. The lower row of Figure 3 shows how UVM handles copy-on-write mappings using the same example used for BSD VM. In UVM a copy-on-write mapping is established by inserting a needs-copy copy-on-write map entry pointing to the underlying object in the target map. When the process with the copy-on-write mapping writes to the middle page the UVM fault routine resolves the fault by first allocating a new amap to clear needs-copy and then copying the data from the backing object into a newly allocated anon. The anon is inserted into the middle slot of the mapping's amap.

The third column in the UVM row of Figure 3 shows the UVM data structures after the process with the copy-on-write mapping forks a child process, the parent process writes to the middle page, and the child process writes to the right-hand page. When the parent process forks, the child receives a copy-on-write copy of the parent's mapping. This is done by write protecting the parent's mappings and setting needs-copy in both the parent and child. When the parent process faults on the middle page, a second amap is allocated for it (clearing needs-copy and incrementing the reference count of anon 1) and the data is copied from the first anon (still in the original amap) to a newly allocated anon that gets installed in the new amap. When the child process faults on the right-hand page the fault routine clears needs-copy without allocating a new amap because the child process holds the only reference to the original amap. The fault routine resolves the child's fault by allocating a third anon and installing it in the child's amap.

5.3 Anonymous Memory Comparison

Both BSD VM and UVM use needs-copy to defer the allocation of anonymous memory structures until the first copy-on-write fault. Thus, in a typical fork operation where the child process immediately executes another program most amap copying and shadow object creation is avoided[3]. In both systems there is a per-page overhead involved in write protecting the parent process' mappings to trigger the appropriate copy-on-write faults. To clear needs-copy under UVM a new amap must be allocated and initialized with anon pointers (adding a reference to each anon's reference counter). To clear needs-copy under BSD VM a new shadow object must be allocated and inserted in the object chain. Future write faults require BSD VM to search underlying objects in the chain for data and promote that data to the top-level shadow object. Also, in addition to normal write-fault processing, BSD VM attempts an object collapse operation each time a copy-on-write fault occurs.

BSD VM's kernel data structure space requirements for copy-on-write consist of a fixed-size shadow object and the pager data structures associated with it. The number of pager data structures varies with the number of virtual pages the object maps. Pages are clustered together into swap blocks that can be anywhere from 32KB to 128KB depending on object size. Each allocated swap block structure contains a pointer to a location on backing store. UVM's kernel data structure space requirements for copy-on-write consist of an amap data structure and the anons associated with it. An amap's size is dictated by the amap implementation being used. UVM currently uses an array-based implementation whose space cost varies with the number of virtual pages covered by the amap. This is expensive for

[3] And even this could be avoided with the vfork system call.

larger sparsely allocated amaps, but the cost could easily be reduced by using a hybrid amap implementation that uses both hash tables and arrays. UVM stores swap location information on a per-page basis in anon structures. UVM must store this information on a per-page basis rather than using BSD VM-like swap blocks because UVM supports the dynamic reassignment of swap location at page-level granularity for fast clustered page out (described in Section 6).

There are a number of design problems and shortcomings in BSD VM's anonymous memory system that contributed to our decision to completely replace it with UVM's amap-based anonymous memory system. BSD VM's copy-on-write mechanism can leak memory by allowing pages of memory that are no longer accessible to remain allocated within an object chain. For example, consider the final BSD VM diagram in Figure 3. If the child process exits, then the third shadow object will be freed. The remaining shadow object chain contains three copies of the middle page. Of these three copies only two are accessible — the page in the first shadow object is no longer accessible and should be freed. Likewise, if the child process writes to the middle page rather than exits, then the page in the first shadow object also becomes inaccessible. If such leaks were left unchecked, the system would exhaust its swap space.

Clearly the longer a shadow object chain is, the greater the chance for swap space to be wasted. Although BSD VM cannot prevent shadow object chains from forming, it attempts to reduce the length of a chain after it has formed by collapsing it. BSD VM attempts to collapse a shadow object chain when ever a write fault occurs on a shadow object, a shadow object reference is dropped, a shadow object is copied, or a shadow object pages out to swap for the first time. This work is done in addition to normal VM processing.

Searching for objects that can be collapsed is a complex process that adds extra overhead to BSD VM. To contrast, no collapsing is necessary with UVM because the amap and anon reference counters keep track of when pages should be freed. This allows new features of UVM such as copy-on-write based data movement mechanisms to be implemented more efficiently than under BSD VM.

Another problem with BSD VM's copy-on-write mechanism is that it is inefficient. For example, consider what happens if the child process in Figure 3 writes to the middle page. Under BSD VM, the data in the middle page of shadow object 1 is copied into a new page of shadow object 3 to resolve the fault. This page allocation and data copy are unnecessary. Ideally, rather than copying the data from shadow object 1 to shadow object 3 the middle page from shadow object 1 would simply be reassigned to shadow object 3. Unfortunately this is not possible under BSD VM because the data structure do not indicate if shadow object 1 still needs its page or not. In UVM, writing to the middle page is handled by allowing the child process to directly write to the page in anon 1 (this is allowable because anon 1's reference count is one), thus avoiding the expensive and unnecessary page allocation and data copy.

Finally, the code used to manage anonymous memory under BSD VM is more complex than UVM's amap-based code. BSD VM must be prepared to loop through a multi-level object chain to find needed data. Each object in the chain has its own set of I/O operations, its own lock, its own shadow object, and its own pool of physical memory and swap space. BSD VM must carefully manage all aspects of each object in the chain so that memory remains in a consistent state. At the same time, it needs to aggressively collapse and bypass shadow objects to prevent memory leaks and keep the object chains from becoming too long, thus slowing memory searches. In contrast, UVM can perform the same function using its simple two-level lookup mechanism. Rather than looping through a chain of objects to find data, UVM need only check the amap and then the object layer to find data. Rather than using lists of objects, UVM uses reference counters in amaps and anons to track access to anonymous memory. UVM's new anonymous memory management system has contributed to a noticeable improvement in overall system performance (see Section 8).

5.4 Amap Adaptation Issues

UVM's amap-based anonymous memory system is modeled on the SunOS VM system vnode segment driver anonymous memory system [9, 13]. (Segment drivers in SunOS perform a similar role to pagers in UVM.) While this system is sufficient for SunOS, it required a number of adaptations and extensions in order to function in a BSD environment and to support UVM's new data movement features (described in Section 7). First, SunOS's anonymous memory mechanism is not a general purpose VM abstraction. Instead, it is implemented as a part of the SunOS vnode segment driver. This is adequate for SunOS because copy-on-write and zero-fill memory can be isolated in the vnode layer. However, in UVM parts of the general purpose VM system such as the fault routine and data movement mechanisms require access to amaps. Thus in UVM we have repositioned the amap system as a general purpose machine-independent virtual memory abstraction. This allows any type of mapping to have an anonymous layer.

Second, the BSD kernel uses several mechanisms that are not present in SunOS. In order for UVM to replace BSD VM without loss of function the design of UVM's amap system must account for these mechanisms. For example, BSD supports the `minherit` system call. This system call allows a process to control its children's access to its virtual memory. In traditional Unix-like systems (including SunOS) child processes get shared access to a parent's shared mappings and copy-on-write access to the rest of the mappings. In BSD the traditional behavior is the default, however the `minherit` system call can be used to change this. The `minherit` system call allows a process to designate the inheritance of its memory as "none," "shared," or "copy." This creates cases such as a child process sharing a copy-on-write mapping with its parent, or a child process receiving a copy-on-write copy of a parent's shared mapping. In addition to `minherit`, BSD also uses a mapping's needs-copy flag to defer the allocation of anonymous memory structures until they are needed. SunOS does not have a needs-copy flag. Thus UVM, unlike SunOS, must be prepared to delay the allocation of amaps using needs-copy until they are actually needed. In order to maintain consistent memory for all processes while supporting both `minherit` and needs-copy, UVM's amap code must carefully control when amaps are created and track when they are shared.

A third area where the adaptation of an amap-based anonymous memory system affected the design of UVM is in the design of UVM's page fault routine. In SunOS, other than the map entry lookup, all of the work of resolving a page fault is left to the segment driver. On the other hand, BSD VM has a general purpose page fault routine that handles all aspects of resolving a page fault other than I/O, including memory allocation, and walking and managing object chains. In fact, the majority of the BSD VM fault routine's code is related to object chain management. Neither of these two styles of fault routine is appropriate for UVM. A SunOS style fault routine forces too much pager-independent work into the pager layer, and as UVM does not use object chaining the BSD VM fault routine is not applicable. Thus, a new fault routine had to be written for UVM from scratch. The UVM fault routine first looks up the faulting address in the faulting map. It then searches the mapping's amap layer to determine if the required data is in there. If not, it then checks the backing object layer for the data. If the data is not there, then an error code is returned. In addition to resolving the current page fault, the UVM fault routine also looks for resident pages that are close to the faulting address and maps them in. The number of pages looked for is controlled by the `madvise` system (the default is to look four pages ahead of the faulting address and three pages behind). This can reduce the number of future page faults. Table 2 shows the results of this change on an i386 for several sample commands. Note that this mechanism only works for resident pages and thus has a minimal effect on execution time for these non-fault intensive applications. As part of our future work, we plan to modify UVM to asynchronously page in non-resident pages that appear to be useful.

Command	**BSD VM**	**UVM**
`ls /`	59	33
`finger chuck`	128	74
`cc`	1086	590
`man csh`	114	64
`newaliases`	229	127

Table 2: Page fault counts on an i386 obtained through `csh`'s "time" command. The `cc` command was run on a "hello world" program.

6 Pagers

UVM introduces three important improvements to pagers. The allocation of pager-related data structures has been made more efficient, the pager API has been made more flexible giving the pager more control over the pages it owns, and aggressive clustering has been introduced into the anonymous memory system.

There is a significant difference between the way the pager-related data structures are organized in BSD VM and UVM. In BSD VM the pager requires several separately allocated data structures. The left side of Figure 4 shows these structures for the vnode pager. In BSD VM a memory object points to a `vm_pager` structure. This structure contains pointers to a set of pager operations and a pointer to a pager-specific private data structure (`vn_pager`). In turn, this structure points to the vnode being mapped. In addition to these structures, BSD VM also maintains a hash table that maps a pager structure to the object it backs (note that there is no pointer from the `vm_pager` to the `vm_object`). To contrast, the right side of Figure 4 shows the UVM pager data structures for a vnode. All VM related vnode data is embedded within the vnode structure rather than allocated separately. The pager data structure has been eliminated—UVM's memory object points directly to the pager operations. So, in order to set up the initial mappings of a file the BSD VM system must allocate three data structures (`vm_object`, `vm_pager`, and `vn_pager`), and enter the pager in the pager hash table. On the other hand, UVM does not have to access a hash table or allocate any data structures. All the data structures UVM needs are embedded within the vnode structure.

Another difference between the BSD VM pager interface and the UVM pager interface is in the API used to fetch data from backing store. To get a page of an

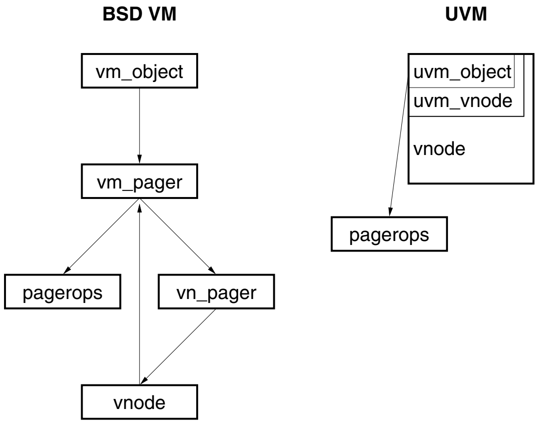

Figure 4: Pager data structures

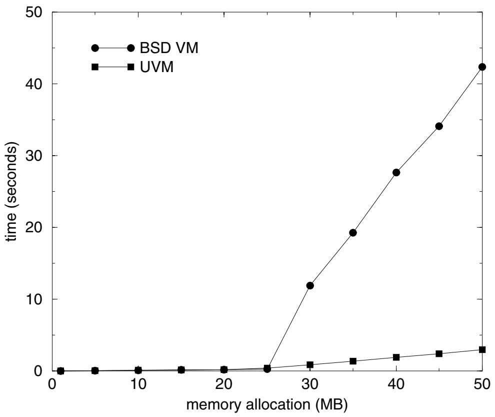

Figure 5: Anonymous memory allocation time under BSD VM and UVM

object's data from backing store in BSD VM, the VM system must allocate a new page, add it to the object, and then request that the pager fill it with data. In UVM, the process fetching the data does not allocate anything, this is left to the pager. If a new page is needed the pager will allocate it itself. This API change allows the pager to have full control over when pages get added to an object. This can be useful in cases where the pager wants to specifically choose which page to put the data in. For example, consider a pager that wants to allow a process to map in code directly from pages in a ROM.

Another difference between the BSD VM pager interface and UVM's pager interface is how UVM handles paging out anonymous memory. One unique property of anonymous memory is that it is completely under the control of the VM system and it has no permanent home on backing store. UVM takes advantage of this property to more aggressively cluster anonymous memory than is possible with the scheme used by BSD VM. The key to this aggressive clustering is that UVM's pagedaemon can reassign an anonymous page's pageout location on backing store. This allows UVM's pagedaemon to collect enough dirty anonymous pages to form a large cluster for pageout. Each page's location on swap is assigned (or reassigned) so that the cluster occupies a contiguous chunk of swap and can be paged out in a single large I/O operation. So, for example if UVM's pagedaemon detects dirty pages at page offsets three, five, and seven in an anonymous object it can still group these pages in a single cluster, while the BSD VM would end up performing three separate I/O operations to pageout the same pages. As a result UVM can recover from page shortages quicker and more efficiently than BSD VM. Figure 5 compares the time it take to allocate anonymous memory under BSD VM and UVM on a 333MHz Pentium-II with thirty-two megabytes of RAM. As the allocation size becomes larger than physical memory, the system must start paging in order to satisfy the request. UVM can clearly page the data much faster than BSD VM.

7 Data Movement

UVM includes three new virtual memory based data movement mechanisms that are more efficient than bulk data copies when transferring large chunks of data [6]. Page loanout allows pages from a process' address space to be borrowed by other processes. Page transfer allows for pages from the kernel or other processes to be inserted into a process' address space easily. Map entry passing allows processes to copy, share, or move chunks of their virtual address space between themselves. We are currently in the process of modifying the kernel's I/O and IPC systems to take advantage of these facilities to reduce data movement overhead.

Page loanout allows a process to safely let a shared copy-on-write copy of its memory be used either by other processes, the I/O system, or the IPC system. The loaned page of memory can come from a memory-mapped file, anonymous memory, or a combination of the two. Pages can be loaned into wired pages for the kernel's I/O system, or they can be loaned as pageable anonymous memory for transfer to another process. Page loanout gracefully preserves copy-on-write in the presence of page faults, pageouts, and memory flushes. It also operates in such a way that it provides access to memory at page-level granularity without fragmenting or disrupting the VM system's higher-level memory mapping data structures. An example of where page loanout can be used is when data is transmitted over a socket. Rather than bulk copy the data from the user's memory to the kernel's memory, the user's pages can be directly shared with the socket layer.

Page transfer allows pages of memory from the I/O system, the IPC system, or from other processes to be inserted easily into a process' address space. Once the pages are inserted into the process they become anonymous memory. Such anonymous memory is indistinguishable from anonymous memory allocated by traditional means. Page transfer is able to handle pages that have been copied from another process' address space using page loanout. Also, if the page transfer mechanism is allowed to choose the virtual address where the inserted pages are placed, then it can usually insert them without fragmenting or disrupting the VM system's higher-level memory mapping data structures. Page transfer can be used by the kernel to place pages from other processes, I/O devices, or the kernel directly into the receiving process' address space without a data copy.

Map entry passing allows processes and the kernel to exchange large chunks of their virtual address spaces using the VM system's higher-level memory mapping data structures. This mechanism can copy, move, or share any range of a virtual address space. This can be a problem for some VM systems because it introduces the possibility of allowing a copy-on-write area of memory to become shared with another process. Because map entry passing operates on high-level mapping structures, the per-page cost of map entry passing is less than page loanout or page transfer, however it can increase map entry fragmentation if used on a small number of pages and it cannot be used to share memory with other kernel subsystems that may access pages with DMA. Map entry passing can be used as a replacement for pipes when transferring large-sized data.

The preliminary measurements of UVM's three data movement mechanisms show that VM-based data movement mechanisms improve performance over data copying when the size of the data being transfered is larger than a page. For example, in our tests, single-page loanouts to the networking subsystem took 26% less time than copying data. Tests involving multi-page loanouts show that page loaning can reduce the processing time further, for example a 256 page loanout took 78% less time than copying data. We are currently in the process of applying these mechanisms to real-life applications to determine their effectiveness.

8 Overall UVM Performance

Replacing the old BSD VM system with UVM has improved both the overall efficiency and overall performance of the BSD kernel. For example, Figure 6 shows the total time it takes for a process with a given amount of dynamically allocated anonymous memory to fork a child process and then wait for that child process to exit under both BSD VM and UVM. Thus, the plot measures critical VM-related tasks such as creating a new address space, copying the parent's mappings into the child process, copy-on-write faulting, and disposing of the child's address space. In the upper two plots, the child process writes to its dynamically allocated memory once and then exits (thus triggering a copy-on-write fault). In the lower plots the child exits without accessing the data. In both cases UVM clearly out performs BSD VM.

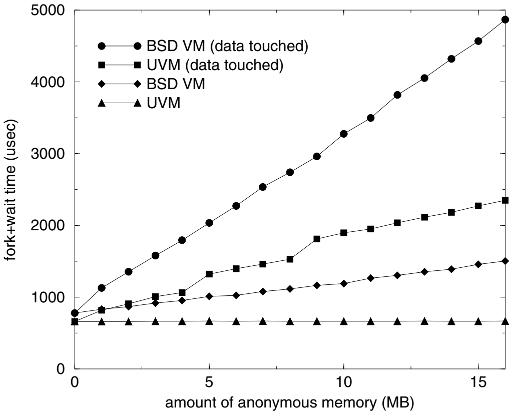

Figure 6: Process fork-and-wait overhead under BSD VM and UVM measured on a 333MHz Pentium-II averaged over 10,000 fork-and-wait cycles

Fault/mapping	**BSD VM** (usec)	**UVM** (usec)
read/shared file	24	21
read/private file	48	22
write/shared file	113	100
write/private file	80	67
read/zero fill	60	49
write/zero fill	60	48

Table 3: Single page map-fault-unmap time on a 333 MHz Pentium II.

Another example of UVM's performance gain is shown in Table 3. The table shows the time (averaged over 1 million cycles) it takes to memory map a page of memory, fault it in, and then unmap the page. UVM outperforms BSD VM in all cases. Note that read faults on a private mapping under BSD VM are more expensive that shared read faults because BSD VM allocates a shadow object for the mapping (even though it is not necessary).

NetBSD users have also reported that UVM's improvements have had a positive effect on their applications. This is most noticeable when physical memory becomes scarce and the VM system must page out data to free up memory. Under BSD VM this type of paging causes the system to become highly unresponsive, while under UVM the system slows while paging but does not become unresponsive. This situation can occur when running large virtual memory intensive applications like a lisp interpreter, or when running a large compile job concurrently with an X server on a system with a small amount of physical memory. In addition to improved responsiveness during paging, users of older architectures supported by NetBSD have noticed that applications run quicker. For example, the running time of `/etc/rc` was reduced by ten percent (ten seconds) on the VAX architecture.

9 Related Work

In UVM, we have focused our efforts on exploring key data structures and mechanisms used for memory management. There has been little recent work in this area, but there has been a lot of work on extensible operating system structure. With UVM, we have created a VM system that is tightly coupled and contains global optimizations that produce a positive impact on system performance. On the other hand, a goal of extensible operating systems is to allow an operating system's functions to be partitioned and extended in user-specified ways. This can be achieved in a number of ways including providing a hardware-like interface to applications (Exokernel [11]), allowing code written in a type safe language to be linked directly into the kernel (SPIN [1]), and allowing software modules to be connected in vertical slices (Scout [14, 19]). While the data structures and mechanisms used by UVM are orthogonal to operating system structure, the effect of extensibility on the tightly coupled global optimizations provided by UVM is unclear. It may be possible to load UVM-like memory management into these systems, for example recent work on the L4 microkernel [10] has shown that a port of Linux to L4 can run with a minimal performance penalty. However, interactions with other extensions may have an adverse effect.

The two virtual memory systems most closely related to UVM are the Mach VM system [18] and the SunOS VM system [4, 9, 13]. Since BSD VM is based on Mach VM, most of the discussion of BSD VM in this paper applies to both VM systems (and to a lesser extent the FreeBSD VM system). As described in Section 5 UVM incorporates and extends parts of SunOS VM's anonymous memory management mechanism. Dyson and Greenman took a different approach to improving the BSD VM data structures in FreeBSD by keeping the same basic structure but eliminating the unnecessary parts of the Mach VM system that BSD inherited [16]. The Linux VM system [21] provides a generic three-level page table based interface to underlying memory management hardware rather than a function-based API like Mach's pmap. All anonymous memory functions are managed through the page table. This is limiting because it does not provide a high-level abstraction for an anonymous page of memory, and it prevents page tables from being recycled when physical memory is scarce. Recent work on virtual memory support for multiple page sizes [8] allows better clustering of I/O operations similar to UVM's aggressive clustering of anonymous memory for page out. However, with large pages data must be copied into a physically contiguous block of memory before it can be paged out. UVM can dynamically reassign anonymous memory's swap location using normal sized pages without copying the data.

Other recent work has focused on zero-copy data movement mechanisms. IO-Lite [15] is a unified buffering system based on Fbufs [7]. IO-Lite achieves zero-copy by forcing all buffers to be immutable once initialized and forcing all I/O operations to be in terms of buffer aggregates. IO-Lite does not interact well with memory-mapped files and is not integrated with a VM system. Solaris zero-copy TCP [5] uses a new low-level pmap API and ATM hardware support to provide zero-copy TCP without effecting higher-level VM code. The L4 microkernel [10] provides granting (remap), mapping, and unmapping primitives to threads to allow for fast VM-based data movement, but it leaves issues such as copy-on-write for higher-level software such as its Linux server. Finally, the Genie I/O subsystem [3] includes mechanisms that allow an operating system to emulate a copy-based API with VM-based mechanisms. Genie's mechanisms could be applied to UVM if such support is desired.

10 Conclusions and Future Work

In this paper we introduced UVM, a new virtual memory system for the BSD kernel. Key aspects of UVM's design include:

- The reuse of BSD VM's machine-dependent layer. This allowed us to focus on the machine-independent aspects of UVM without getting bogged down in machine-specific details. This reuse made porting UVM to architectures supported by BSD VM easy.

- The repartitioning of VM functions in more efficient ways. For example, we combined BSD VM's two-step mapping process into a more efficient and secure single step mapping function, and we broke BSD VM's unmapping function up into smaller functions that hold map locks for a shorter period of time.

- The reduction of duplicate copies of the same state information. This was used in UVM to reduce map entry fragmentation due to page wiring.

- The elimination of the contention between the VM system and other kernel subsystems. For example, we redesigned and improved the management of the inactive memory object cache to work with the vnode system (rather than in parallel to it).

- The elimination of complex data structures such as BSD VM's object chains that make accounting for a program's memory usage difficult and expensive.

- The grouping or clustering of the allocation and use of systems resources to improve system efficiency and performance. For example, we grouped UVM's allocation of pager data structures and we aggressively clustered UVM's anonymous pageout.

- Easy resource sharing with other kernel subsystems. For example, UVM's data movement mechanisms allow it to share its pages with the I/O and IPC subsystem without costly data copies.

Our future plans for UVM include unifying the VM cache with the BSD buffer cache, adding more asynchronous I/O support to UVM (for both pagein and pageout), and adapting the BSD kernel to take advantage of UVM's new data movement mechanisms to improve application performance.

Acknowledgments

We would like to thank Orran Krieger, Lorrie Faith Cranor, and the anonymous reviewers for their helpful comments on drafts of this paper.

References

[1] B. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, 1996.

[2] D. Bobrow et al. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM*, 15(3), March 1972.

[3] J. Brustoloni and P. Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *Proceedings of IEEE INFOCOM 1997*, pages 1124–1132, April 1997.

[4] H. Chartock and P. Snyder. Virtual swap space in SunOS. In *Proceedings of the Autumn 1991 European UNIX Users Group Conference*, September 1991.

[5] H. Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX Conference*, pages 253–264. USENIX, 1996.

[6] C. Cranor. *Design and Implementation of the UVM Virtual Memory System*. Doctoral dissertation, Washington University, August 1998.

[7] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

[8] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the USENIX Conference*. USENIX, 1998.

[9] R. Gingell, J. Moran, and W. Shannon. Virtual memory architecture in SunOS. In *Proceedings of USENIX Summer Conference*, pages 81–94. USENIX, June 1987.

[10] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1997.

[11] M. Kaashoek et al. Application performance and flexibility on Exokernel systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, October 1997.

[12] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[13] J. Moran. SunOS virtual memory implementation. In *Proceedings of the Spring 1988 European UNIX Users Group Conference*, April 1988.

[14] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Operating Systems Design and Implementation (OSDI)*, pages 153–168, 1996.

[15] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. In *Operating Systems Design and Implementation (OSDI)*, pages 15–28. USENIX, 1999.

[16] The FreeBSD Project. The FreeBSD operating system. See `http://www.freebsd.org` for more information.

[17] The NetBSD Project. The NetBSD operating system. See `http://www.netbsd.org` for more information.

[18] R. Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computing*, 37(8), August 1988.

[19] O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *Operating Systems Design and Implementation (OSDI)*, pages 59–72. USENIX, 1999.

[20] R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[21] L. Torvalds et al. The Linux operating system. See `http://www.linux.org` for more information.