Zero-Copy Data Movement Mechanisms for UVM

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 specifically designed to provide the I/O and IPC systems with a range of flexible data movement mechanisms. Implemented in the NetBSD operating system, UVM completely replaces the Mach based 4.4BSD VM system. UVM provides three new virtual memory based data movement mechanisms: page loanout, page transfer, and map entry passing. Page loanout and page transfer allow processes to efficiently lend out and receive pages of memory, thus providing the operating system with data movement mechanisms with page-level granularity. Map entry passing allows processes to exchange chunks of their virtual address space, thus providing the operating system with a data movement mechanism with mapping-level granularity. In addition to featuring flexible data movement mechanisms, UVM also improves virtual memory performance over BSD VM in traditional areas such as forking and pageout.

1 Introduction

New computer applications in areas such as multimedia, imaging, and distributed computing expect the highest performance from an operating system's I/O and IPC systems. Although hardware speeds have improved, the operating system has had a hard time keeping up [11, 14]. Unnecessary data copying in traditional Unix-like operating systems has been one of the barriers to high performance. Bulk copying of large chunks of data is expensive — the processor must spend its time copying the data from the source buffer to the destination one word at a time. The speed of this process is limited by the bandwidth of main memory. Also, as these buffers are often larger than a processor's cache, copying them causes much of the useful information stored in the cache to be flushed out to make room for the data being copied.

Unnecessary data copies are often reduced by using a system's virtual memory hardware to share or remap data buffers. The virtual memory hardware is controlled by the heart of a modern operating system — the virtual memory system. One common approach to reducing data copying is to focus on I/O and IPC system design and either rely on existing mechanisms within the virtual memory system (with possible superficial changes) or to completely bypass the high-level virtual memory system and access the hardware directly [2, 7, 9, 15]. In contrast, our approach is to focus on the internal design of the virtual memory system and integrate highly-flexible virtual memory based data movement into it. We introduce UVM [6], a new virtual memory system specifically designed to provide the I/O and IPC systems with a range of flexible data movement mechanisms.

Implemented in the NetBSD operating system, UVM completely replaces the Mach based 4.4BSD VM system. UVM provides three new virtual memory based data movement mechanisms: page loanout, page transfer, and map entry passing. Page loanout and page transfer allow processes to efficiently lend out and receive pages of memory, thus providing the operating system with data movement mechanisms with page-level granularity. Map entry passing allows processes to exchange chunks of their virtual address space, thus providing the operating system with a data movement mechanism with mapping-level granularity. In addition to featuring flexible data movement mechanisms, UVM also improves virtual memory performance over BSD VM in traditional areas such as forking and pageout.

In this paper we present the design, implementation, and measurement of UVM's data movement mechanisms. In Section 2 we present background and related work. Section 3 contains a high-level overview of UVM. Section 4 contains a more detailed description of UVM's data movement mechanisms. In Section 5 we present performance measurements for these mechanisms. Finally, we conclude in Section 6.

2 Background and Related Work

In this section we present background information on the virtual memory system that UVM replaces: BSD VM. We also examine other related work. Such work falls into two categories: virtual memory system designs, and I/O and IPC system designs.

2.1 BSD VM Overview

The BSD VM system is divided into two layers: a large machine-independent layer, and a smaller machine-dependent layer. The machine-independent code is shared by all BSD-supported processors and contains the code that performs the high-level functions of the VM system. The machine-dependent code is called the "pmap" (for physical map) layer, and it handles the lower-level details of programming a processor's MMU. Each architecture supported by the operating system must have its own pmap module. Each layer of the VM system does its own level of mapping. The machine-independent layer maps "memory objects" (usually files) into the virtual address space of a process or the kernel. The machine-dependent layer does not know about higher-level concepts such as memory objects; it only knows how to map physical pages of memory into a virtual address space.

One important aspect of the VM system is how it handles memory objects that are mapped copy-on-write. In a copy-on-write mapping, changes made to an object's mapped pages are not shared — they are private to the process that made the changes. The BSD VM system manages copy-on-write mappings of VM objects by using "shadow objects." A shadow object is an anonymous memory object that contains the modified pages of the copy-on-write object it is shadowing. The object being shadowed can be a file object or another shadow object. Each time an object is copy-on-write copied, a new shadow object is created for it. A linked list of objects shadowing each other is called a "shadow object chain." The final object on the chain is the object that was originally copy-on-write copied. To determine which page of memory should be mapped into a process, the object chain must be searched.

There are several problems with using shadow object chains for copy-on-write. First, it complicates the page fault routine by requiring it to handle arbitrary length object chains. The fault routine must properly protect the object chain in order to avoid race conditions. Second, if an object chain grows too long it will slow searches and waste both kernel and swap space. In order to avoid long object chains, BSD VM uses a complex "object collapse" strategy that attempts to compress object chains by merging objects together. The object collapse code has limitations that can cause significant performance problems in BSD in certain cases. Third, the partial unmapping of a copy-on-write area of memory does not free any memory resources associated with the inaccessible unmapped area. These resources are held until the copy-on-write area of memory is completely unmapped. This can cause a condition known as a "swap memory leak."

2.2 Related Virtual Memory Systems

The Mach virtual memory system is used for memory management and IPC in the Mach microkernel developed at Carnegie Mellon University [16, 19]. BSD VM is a simplified version of Mach VM. Mach VM avoids data copying by using two mechanisms: copy-on-write and "out-of-line" IPC messages. Mach uses a more complex version of the BSD VM object chaining mechanism for copy-on-write, and thus suffers from problems similar to the ones described in Section 2.1. Mach's "out-of-line" IPC message passing mechanism allows processes to pass chunks of virtual memory between themselves without data copies. This provides mapping-level granularity and is similar to UVM's map entry passing mechanism. This mechanism was found to be insufficient for certain types of IPC and later versions of Mach were modified to allow a list of busy pages to be extracted from a process and sent as a message [2]. Further optimizations along these lines allow pages to be to be pinned in memory rather than being marked busy. While this approach does provide page-level granularity, it is not as flexible as UVM's page loanout mechanism, as all copy-on-write operations must be done through Mach's bulky object-chaining mechanism. Additionally, Mach's IPC operations cause unnecessary map fragmentation when extracting memory for page based out-of-line messages.

The FreeBSD virtual memory system is an improved version of the BSD VM system [8]. Work on FreeBSD VM has focused on a number of areas including simplifying data structure management, data caching, and efficient paging algorithms. FreeBSD VM's data structures are similar to BSD VM's data structures, although some structures have been eliminated. While FreeBSD retains Mach-style shadow object chaining for copy-on-write, the swap memory leaks associated with BSD VM's poor handling of the object collapse problem have been addressed. Many of FreeBSD's improvements in the areas of data caching and paging algorithms are applicable to UVM, thus FreeBSD will be a good reference for future work on UVM in these areas. FreeBSD does not have UVM-style features such as page loanout and page transfer. These features could be added to FreeBSD with some difficulty (due to the use of object chaining). Alternatively object chaining could be eliminated from FreeBSD and UVM features added.

The SunOS4 virtual memory system (also used in Solaris) is a modern VM system that was designed to replace the 4.3BSD VM system that appeared in SunOS3 [10, 13]. SunOS VM avoids data copying through a copy-on-write mechanism based on individual pages of anonymous memory (anons) grouped into anonymous memory maps (amaps). UVM uses a similar scheme for copy-on-write, however there are some key differences. First, in UVM the anon is a general purpose virtual memory abstraction used in both copy-on-write and data passing. In SunOS the anon is not a generalized abstraction and is limited to certain types of mappings that require copy-on-write or zero-fill memory. It is not used for data passing. Second, UVM allows processes to share amap-based copy-on-write regions with other processes or the kernel. This is used in map entry passing and to support Mach-style memory inheritance. SunOS lacks this feature. Third, UVM's amap data structure contains additional indexes to speed up operations on sparsely populated amaps. Finally, as part of its page loanout mechanism, UVM allows a page to belong to both a file object and an anon at the same time. SunOS does not support this type of memory sharing. SunOS VM currently does not provide UVM-like features such as page loanout, page transfer, and map entry passing. However, SunOS's anon-style anonymous memory system and modular design would ease the implementation of these UVM-style features. Recent experimental work under Solaris introduces two new virtual memory features: page soft locking and page flipping [5]. These features were implemented in the machine-dependent layer of the VM (the "HAT" layer), thus bypassing the machine-independent part of the VM system. Most of the UVM work was done in the machine-independent layer.

Linux[1] is a popular free Unix-like operating system written by Linus Torvalds [18]. Linux VM avoids data copying through its copy-on-write mechanism. This mechanism uses a copy-on-write mapping flag and a per-page reference counter to determine if a shared page should be copied. Like SunOS, Linux does not allow processes to share copy-on-write regions. Additionally, since Linux stores copy-on-write state in its page tables, mapping-level granularity VM operations on copy-on-write memory require page-level operations. Linux also does not support write-protecting all mappings of a page of physical memory. This makes it impossible to transition a shared page into a copy-on-write state, and thus prevents Linux from supporting VM features such as UVM's page loanout. Linux has some support for remapping memory, but only within a process' address space. The Linux-specific `mremap` system call is used by some versions of Linux's `malloc` to resize its heap.

Although not a Unix-like operating system, Microsoft's Windows-NT operating system's VM system supports many of the same features as Unix-like operating systems [17]. NT avoids data copies through copy-on-write and local procedure calls (LPCs). NT's general structure and copy-on-write mechanism are similar to those found in Linux, and thus the same limitations that apply to Linux apply to NT. LPCs are an internal-only IPC facility that uses virtual memory features in some cases to transfer data. For LPC messages of less than 256 bytes data copying is used. For LPC messages larger than that, a shared object is allocated and used to pass data. For very large data that will not fit in a shared section, NT's LPC mechanism allows the server to directly read or write data from the client's address space.

[1] We examined the virtual memory system that appears in the most recently available version of Linux — 2.1.106.

2.3 Related I/O and IPC Subsystems

In this section we examine research on I/O and IPC subsystems that can take advantage of services offered by virtual memory systems to reduce data movement overheads. While this research shares some of the same goals as UVM, our approach is different. In UVM we are interested in creating a virtual memory system whose internal structure provides efficient and flexible support for data movement. In I/O and IPC research, the focus is on using features that the virtual memory is assumed to already provide. Thus, I/O and IPC research addresses some problems that UVM does not address such as buffering schemes and API design.

Brustoloni and Steenkiste from Carnegie Mellon University have analyzed the effects of data passing semantics on kernel I/O and IPC and provide optimizations that reduce data transfer overhead while maintaining a traditional Unix-like I/O API [3]. Implemented under Genie (a prototype I/O system), two optimizations of interest are temporary copy-on-write (TCOW) and input alignment. TCOW, like UVM's page loanout, allows a process to lend wired pages to the kernel without having to worry about the page being modified or freed. TCOW differs from page loanout in three key areas. First, while UVM allows pages to be loaned out as either wired kernel memory, or a pageable anonymous memory, TCOW can only be used with wired kernel memory. Second, while UVM's page loanout mechanism was implemented on top of UVM's new anon-based anonymous memory system that provides page granularity with low overhead, TCOW was implemented as an add-on to the old BSD VM system, and thus has to contend with all the drawbacks of BSD VM's object chaining mechanism. Third, while we have demonstrated how page loanout can be integrated with BSD's mbuf-based IPC system, TCOW is a prototype, and it has only be demonstrated with Genie (completely bypassing the traditional BSD IPC system). Input alignment is a technique that can be used to preserve an API with copy semantics on data input while using page remapping to actually move the data. In order to do input alignment either the input request must be issued *before* the data arrives so Genie can analyze the alignment requirements of the input buffer, or the client must query Genie about the alignment of buffers that are currently waiting to be transfered. Genie also has several techniques for avoiding data fragmentation when the input data contains packet headers from network interfaces.

Druschel and Peterson's fast buffers (fbufs) kernel subsystem is an operating system facility for IPC buffer management developed at the University of Arizona [7]. Fbufs, an add-on to the Mach microkernel, provide fast data transfer across protection domain boundaries. The fbuf system is based on the assumption that IPC buffers are immutable. The kernel and each process on the system share a fixed sized area of virtual memory set aside as the "fbuf region" where all fbufs must be mapped. The fbuf facility has two useful optimizations: fbuf caching and volatile fbufs. In fbuf caching, the sending process specifies the path the fbuf will take through the system at fbuf allocation time. This allows the fbuf system to reuse the same fbuf for multiple IPC operations. Volatile fbufs are fbufs that are not write protected in the sending process, saving some VM operations. The fbuf facility has several limitations. First, the only way to get data from the filesystem into an fbuf is to copy it. Second, if an application has data that it wants to send using an fbuf but that data is not in the fbuf region, then it must first be copied there. Third, it is not possible to use copy-on-write with an fbuf. It should be possible to combine UVM features such as page loanout and page transfer with the fbuf concept to lift these limitations.

Pasquale, Anderson, and Muller's Container Shipping I/O system allows a process to transfer data without having direct access to it [15]. Developed at University of California (San Diego), the container shipping API allows a process to allocate a container, and the fill it with pointers to data buffers. Once full, a container can be shipped to another process or the kernel. Rather than receiving the data directly, the receiving party receives a handle to the container. The receiver can choose to map some of the container's buffers into its address space for modification, and then it can unmap them. The container can be passed on to another process. By using these operations processes can pass data between themselves without mapping it into their address space. A container shipping I/O system could easily take advantage of UVM data movement mechanisms when loading, unloading, and transferring containers.

3 UVM Overview

Our primary objective in creating UVM is to produce a virtual memory system for a Unix-like operating system that has flexible, general, and efficient VM-based data movement facilities integrated into its design. The kernel's I/O and IPC systems can take advantage of these facilities to reduce data movement overhead. Unlike many other prototype-based VM research projects, our work has been implemented as part of an operating system that is in widespread use. Thus, we have designed our new virtual memory features so that their presence in the kernel does not disrupt other kernel subsystems. This allows experimental changes to be introduced into the I/O and IPC subsystem gradually, thus easing the adoption of these features. Our work centers around several major goals:

Allow 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 mechanism used to do this should allow copied memory to come from a memory mapped file, anonymous memory, or a combination of the two. It should provide copied memory either as wired pages for the kernel's I/O or IPC subsystems, or as pageable anonymous memory for transfer to another process. It should gracefully preserve copy-on-write in the presence of page faults, pageouts, and memory flushes. Finally, it should operate 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. Section 4.1 describes how UVM meets this goal through the page loanout mechanism.

Allow 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 should become anonymous memory. Such anonymous memory should be indistinguishable from anonymous memory allocated by traditional means. The mechanism used to do this should be able to handle pages that have been copied from another process' address space using the previous mechanism (page loanout). Also, if the operating system is allowed to choose the virtual address where the inserted pages are placed, then it should be able to insert them without fragmenting or disrupting the VM system's higher-level memory mapping data structures. Section 4.2 describes how UVM meets this goal through the page transfer mechanism.

Allow processes and the kernel to exchange large chunks of their virtual address spaces using the VM system's higher-level memory mapping data structures. Such a mechanism should be able to 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. The per-page cost for this mechanism should be minimized. Section 4.3 describes how UVM meets this goals through the map entry passing mechanism.

3.1 UVM Structure

When designing UVM, we focused our efforts on the aspects of the VM system that directly effect data movement. For VM design elements not directly related to data movement we reused useful parts of the BSD VM design to avoid reinventing the wheel and to ease UVM's integration into the NetBSD source tree. For example, we reused BSD VM's VM map structure and machine-dependent pmap layer. Reusing the pmap layer allows us to take advantage of existing machine-dependent work in both BSD VM and Mach.

UVM's machine-independent layer handles all high-level VM functions. These functions are centered around eight major machine-independent data structures, shown in Figure 1.

All processes have a `vmspace` structure that contains pointers to the machine-dependent and machine-independent data structures that define the mappings in a process' address space. Low-level machine-dependent mappings are contained in the pmap structure. The machine-independent mappings are stored in the `vm_map` structure. All processes and the kernel have their own

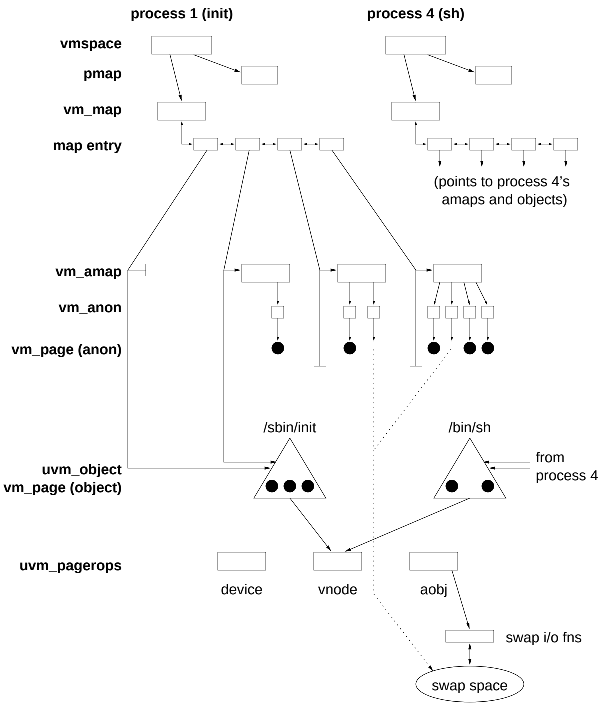

Figure 1: UVM data structures at a glance. Note that there is one vm_aref data structure within each map entry structure. The page queues and object-offset hash table are not shown.

vm_map structure that contains a list of map entries that define mapped areas in their virtual address spaces. Each map entry maps an area of virtual address space to memory objects. In BSD VM, the map entry points to an object chain, as described in Section 2.1. UVM, on the other hand, has a two-level memory mapping scheme. Each map entry points to an anonymous memory map (amap) and a single, non-chained, memory object. The two-level mapping scheme greatly simplifies the implementation of memory lookup, object management, and UVM's page granular data movement mechanisms. Note that some mappings may only make use of one of the two layers.

For the upper-level amap layer, a map entry contains a vm_aref structure — a small structure that points to an offset in an anonymous memory map (vm_amap). An amap describes an area of anonymous memory. The area may have "holes" in it that allow references to fall through to the underlying object layer. The amap structure contains a list of vm_anon structures that are currently mapped into it. Each anon represents a single virtual page of anonymous memory. An anon's data may reside in a resident vm_page of memory, or it may be paged out to to the swap area. All UVM copy-on-write operations are handled through the anon structure. While the design of UVM's amap layer was inspired by SunOS's anonymous memory layer, there are a few key differences (see Section 2.2).

For the lower-level object layer, a map entry contains a pointer to a uvm_object structure. A UVM object represents a file, a zero-fill memory area, or a device that can be mapped into a virtual address space. Each UVM object contains a list of vm_page structures that belong to it and a pointer to a uvm_pagerops structure. The UVM pagerops are used to access backing store.

Finally, a vm_page structure describes a page of physical memory. These structures are allocated when the system is booted. In addition to being referenced by UVM objects and anons, pages are also referenced by the page queues and the object-offset hash table. The page queues are used by the pagedaemon to identify pages whose data can be paged out so that the page can be used elsewhere. The object-offset hash table allows UVM to quickly lookup a page in an object.

3.2 UVM Data Structure Locking

Data structure locking is an important aspect of a virtual memory system since many processes can access VM data concurrently and we want to avoid data structure corruption and deadlock. As we have significantly changed the data structures in UVM, we had to replace BSD VM's locking strategy with a new one. UVM data structures must be locked in the following order: map, amap, object, anon, and finally the page queues. All the data structure locks are spin locks with the exception of the map lock. Functions that need to lock data structure in the wrong order must use non-blocking lock attempts and give up (or start over) if the non-blocking lock attempt fails.

3.3 UVM Copy-On-Write

Copy-on-write is achieved in UVM using the amap layer. Copy-on-write data is originally mapped in read-only from a backing uvm_object. When copy-on-write data is first written, the page fault routine allocates a new anon with a new page, copies the data from the uvm_object's page into the new page, and then installs the anon in the amap for that mapping. When UVM's fault routine copies copy-on-write data from a lower-layer uvm_object into an upper-layer anon it is called a "promotion." Once copy-on-write data has been promoted to the amap layer, it becomes anonymous memory. Anonymous memory also can be subject to the copy-on-write operation. An anon can be copy-on-write copied by write-protecting its data and adding a reference to the anon. When the page fault routine detects a write to an anon with a reference count greater than one, it will copy-on-write the anon.

4 UVM Data Movement

This section describes the design and function of UVM's page loan out, page transfer, and map entry passing interfaces. All three interfaces allow a process to move data using virtual memory operations rather than costly data copies. These interfaces provide flexible and efficient building blocks on which advanced I/O and IPC subsystems can be built.

4.1 Page Loanout

Under a traditional kernel, sending data from one process to another process or a device is a two-step procedure. First the data is copied from the process' address space to a kernel buffer. Then the kernel buffer is handed off to its destination (either a device driver or the IPC layer). The copy of the data from the process' address space to the kernel buffer adds overhead to the I/O operation. It would be more efficient if I/O could be performed directly from the process' address space, but in a traditional kernel it is necessary to copy the data for three reasons: the data may not be resident in main memory, a process may modify its data while the I/O operation is in progress, or the data may be flushed or paged out by another process or the page daemon.

Page loanout is a feature of UVM that addresses these issues by allowing a process to safely loan out its pages to another process or the kernel. This allows the process to send data directly from its memory to a device or another process without having to copy the data to an intermediate kernel buffer. This reduces the overhead of I/O and allows the kernel to spend more time doing useful work and less time copying data.

At an abstract level, loaning a page of memory out is not overly complex. To loan out a page, UVM must make the page read-only and increment the page's "loan count." Then the page can safely be mapped (read-only). The complexity of page loanout arises when handling the special cases where some other process needs to make use of a page currently loaned out. In that case the loan may have to be broken.

4.1.1 Loan Types and Attributes

A page of memory loaned out by a process must come from one of UVM's two mapping layers. As shown in Figure 2, a process can loan its memory out to either the kernel or to another process. To loan memory to the kernel, UVM creates an array of pointers to wired `vm_page` structures that can be mapped into the kernel's address space. To loan memory to other processes, UVM creates an array of pointers to anon structures that can be entered into an amap that belongs to the target process. The target process treats the loaned memory like normal anonymous memory. There are four possible types of loans: object to kernel, anon to kernel, object to anon, and anon to anon. Note that it is possible for a page from a `uvm_object` to be loaned to an anon and then later loaned from that anon to the kernel.

Before a page can be loaned out, it must be made read-only (immutable). This involves write-protecting in hardware all writable mappings of the page. However there are many cases where pages are already read-only, for example a video server may transmits data from a file that is mapped read-only.

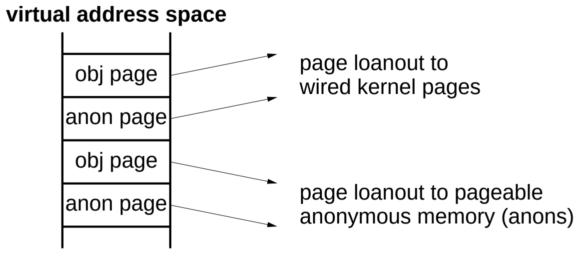

Figure 2: Page loanout

In order to modify data in a loaned out page, the kernel must terminate or "break" the loan. Breaking a loan involves allocating a new page off the free list, copying the data from the loaned page to the new page, and then replacing the old page with the new one. Note that the old page remains allocated and immutable because it is still being used for loanout. When replacing an old page with a new one it is possible for the old page to become "ownerless" because it is no longer associated with a memory object. Ownerless pages are freed when the final loan on them is dropped.

Memory pages loaned to the kernel have three important properties. First, pages loaned to the kernel must be resident. If the data is on backing store then it is fetched before the loanout proceeds. Second, each page loaned to the kernel is "wired" to prevent the pagedaemon from attempting to pageout a page loaned to the kernel while the kernel is still using it. This is important because if the pagedaemon was allowed to pageout the page then the next time the kernel accessed the loaned page an unresolvable page fault would be generated (and the system would crash). Third, pages loaned to the kernel are entered into the kernel's pmap with a special function that prevents page-based pmap operations from affecting the kernel mapping. This allows UVM to perform normal VM operations on the loaned-out page without having to worry about disrupting the kernel's loaned mapping and causing an unresolvable page fault.

Memory pages loaned from an object to an anon have two important properties. First, a `uvm_object`'s page can only be loaned to one anon at a time. Future loans to an anon need only gain a reference to the anon the page is already loaned to (rather than allocating a new anon to loan to). Second, while most pages are referenced by either a `uvm_object` or an anon, pages loaned from an object to an anon are referenced by both. Although both object and anon reference the page, the object is always considered to be the owner of the page. Thus, the object must be locked in order to access the page's fields. All data structure locking must follow the locking protocol presented in Section 3.2.

An anon that has a page on loan from an object does not hold a reference to that object. Thus, when locking data structures, it is possible for the object to terminate between the time the page's object field is read and the time the lock on the object is attempted. In order to prevent this, the page queues must be locked before trying to lock the object. Locking the page queues prevents the page's loan count from changing and thus ensures that the object cannot be terminated until the page queues are unlocked.

4.1.2 Handling Loaned Out Pages

There are several places in UVM where loaned out pages have to be treated specially. First, when locking an anon that points to a page that is ownerless, the anon should take over as owner of the page. This allows objects to donate data to anons without the need for a data copy. Second, when freeing a page with a non-zero loan count, the page becomes ownerless and should not be freed until all loans to it are terminated. Third, when there is a write fault on a loaned page care must be taken to map in pages with a non-zero loan count read-only. If the write fault is on a shared mapping or an anon with a reference count of one, then the loan must be broken before the fault can be resolved. Note that page loanout to anons makes use of the copy-on-write feature of UVM's anonymous memory layer.

4.1.3 Using Page Loanout

Page loanout can be used in a number of ways. For example, to quickly transfer a chunk of data from one process to another the data can be loaned out to anons, and those anons can then be inserted into an amap belonging to the target process. Page loanout to the kernel can also be used to improve both network and device I/O. For example, when a process wants to send data out over the network the kernel would normally copy the data from the process' memory into kernel network buffers (mbufs). However, instead of copying the process' pages, the pages could be loaned out to the kernel and then associated with mbufs, thus avoiding a costly data copy. Page loanout can also be used by devices. For example, an audio device could directly access pages of audio data loaned out from a user process. This would eliminate the need to copy the data to a private kernel buffer.

4.2 Page Transfer

Under a traditional BSD kernel, when the kernel wants to move some of its data from a kernel buffer to a user process' virtual address space it uses the `copyout` function. This function copies the data word by word to a user buffer. If the user's buffer is not resident, then a page fault is triggered and the the buffer is paged in before the copy is started. This may result in the paged-in data being completely replaced by the copied data.

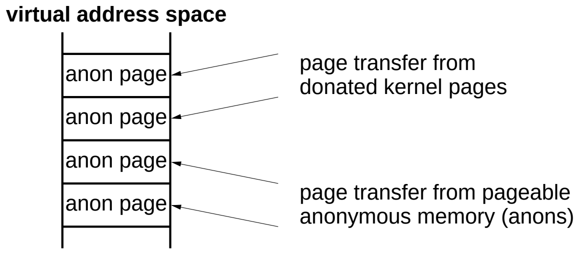

Figure 3: Page transfer

For large chunks of data all this copying can get expensive. Page transfer provides the kernel a way to work around this data copy by allowing the kernel to inject pages of memory into a process' virtual address space. Page transfer takes kernel pages or anons and installs them at either a virtual address specified by the user or in an area of virtual memory reserved for page transfer. UVM keeps a pool of amaps, preallocated for reuse by the page transfer mechanism, in the page transfer area of memory. Once a page has been transfered into a process' address space it is treated like any other page of anonymous memory.

4.2.1 Page Transfer Types

There are two types of page transfer operations: kernel page transfer and anonymous page transfer. These operations are shown in Figure 3. In kernel page transfer the kernel donates pages to a process from one of its page pools or from its address space. These pages become part of that process' anonymous memory. To perform kernel page transfer the kernel first locates the `vm_page` structures of the buffer being transfered. Next, if any of the pages are mapped into the kernel's address space those mappings are removed. The pages are then removed from any kernel-related data structures that they are associated with. At this point a new anon is allocated and attached to each page being transfered. Finally, the anons are inserted into the receiving process' address space in one of two ways. If the receiving process specified a target virtual address range, then UVM first ensures that that range of memory has amaps associated with it (allocating new amaps if necessary) and then inserts the anons into the appropriate amaps, replacing any previously allocated anons in that area. On the other hand, if the receiving process did not specify a target virtual address, then UVM locates free space in the page transfer area for the anons and installs them there. Once this is done the transfer is complete. The pages are now part of the receiving process' normal anonymous memory pool.

In anonymous page transfer the data being transfered is already in anon structures. Typically these anons are the result of a page loanout operation. In this type of page transfer, UVM does not have to remove the pages from the kernel or other processes. Instead, it only has to insert the anons into the receiving process' address space in the same way as described above.

4.2.2 Disposing of Transfered Data

Processes receiving data via page transfer need to release these pages when they are no longer needed in order to avoid accumulating too much memory. This can be done in three ways. First, a process may unmap transferred data using the standard `munmap` system call. A problem with this is that `munmap` will free both the transfered anons and the amap containing them. This forces UVM to allocate a new amap the next time pages are transfered to to the same virtual space. A second way to dispose of transfered data is to use the new `anflush` system call. This system call removes anons from a specified virtual address range without freeing the amap allocated to it. This allows the amap to be reused for future transfers. The final way to dispose of transfered data is to push the anon pages down into the object mapping layer. This can be done either by donating the ownership of the anons' pages to an object (establishing a loanout relationship between the anon and the object), or by freeing the anons and inserting the pages into an object.

4.2.3 Using Page Transfer

Kernel and anonymous page transfer can be used in several ways. For example, kernel page transfer can be used with device drivers or the IPC subsystem. For devices such as audio cards, when recording the driver can allocate pages from UVM to DMA into. These pages can then be transfered into the application that is reading from the audio device. For the IPC system, mbuf data can be read with page transfer. There are two sizes of mbufs: large and small. If large mbufs are made page sized, then they may be eligible for kernel page transfer. There are two possible types of large mbufs. Large mbufs that have normal page-sized data areas allocated out of the kernel are called "cluster" mbufs. Large mbufs that refer to memory managed by other subsystems are called "external" mbufs. Pages from cluster mbufs that have a reference count of one can easily be donated to a process through page transfer. External mbufs can refer to many types of memory including a network interface card's memory or pages loaned out from a process. Pages that belong to networking hardware cannot be transfered, but external mbufs that refer to loaned out pages can be transfered by adding an additional loan to an anon and then using that anon for anonymous page transfer. This is an example of page loanout and page transfer being used together to avoid data copies. Note that a new API is required in order for a process to receive data with page transfer.

Anonymous page transfer can be used with page loanout to anons as part of an I/O or IPC mechanism. For example, anonymous page transfer could be used for aligned page-sized `read` operations on a file. Rather than copying the data to anonymous memory with `copyout`, an object's pages can be loaned to anons and those anons can be transfered into the reading process' address space. A similar effect could be achieved using `mmap`, but mmap could fragment the high-level memory mapping structures. Page transfer does not fragment high-level memory mappings because it provides page-level granularity.

4.3 Map Entry Passing

Under a traditional system, processes often exchange data either through pipes or through shared memory. In BSD, pipes are implemented as a connected pair of sockets. When the sending process writes data into a pipe, the data is copied from the sending process' address space into a kernel buffer (an mbuf). The mbuf is then placed on a queue for the receiving process. When the receiving process reads from the pipe, the data is copied from the queued mbuf into the receiving process' buffer. Thus, data transmitted over a pipe is copied twice.

Shared memory can be established in three ways. First, the System V shared memory interface allows a process to create a shared segment of anonymous memory. Each segment of shared memory has a size and a protection. Once created, any process with the appropriate permissions can attach the shared memory segment into their address space. The second way shared memory can be established is through the `mmap` system call. Changes made to files that have been mapped `MAP_SHARED` will be seen by all processes accessing the file. The final way shared memory can be established occurs is when a forking process arranges for all or part of its memory to be shared with the child process.

UVM provides a new way for processes to exchange data: map entry passing (shown in Figure 4). For large data sizes, this is more efficient than pipes because data is moved without being copied. Map entry passing also has several advantages over traditional memory sharing mechanisms. Unlike System V shared memory, map entry passing is not restricted to anonymous memory. Map entry passing allows a range of virtual addresses to be shared, copied, or moved. This range can include multiple memory mappings and unmapped areas of virtual memory. Map entry passing also does not require the services of the filesystem layer, unlike `mmap`. While page loanout and page transfer have page-level granularity, map entry passing has mapping-level granularity.

4.3.1 Map Entry Passing Implementation

The map entry passing mechanism operates as follows. Virtual space is first reserved in the target map for the passed

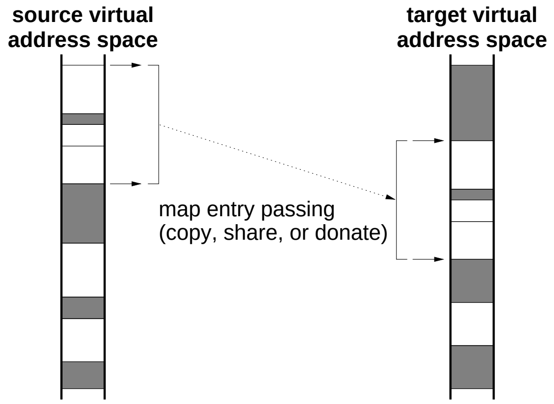

Figure 4: Map entry passing

memory. Then the source map's entry list is searched for the first map entry that contains the starting virtual address. There are four possible ways for a receiving process to get passed memory. The memory can be shared, copied (with copy-on-write), donated, or donated with zero-fill. Donating the memory causes it to be removed from the sending process' virtual address space and placed in the receiving process' space. After the memory has been passed the virtual space in the sending process will be unmapped. Donate with zero-fill is the same as donate, except that rather than leaving the exported range of virtual space unmapped after it has been passed it is reset to be zero-fill memory.

Each source map entry that is in the virtual range is copied into a list of map entries that will be inserted in the destination map (if the memory is being donated rather than copied or shared then the source map entry is removed from the source map). When the end of the virtual range is reached, the new map entries are inserted in the destination map at the virtual address previously reserved.

The function that extracts the map entries from the source map has a special mode of operation that is used for short-term shared map entry passing of a small number of pages. This mode is called "quick reference mode." Normally, using map entry passing to pass small areas of memory would be inefficient because it would cause the source map's larger entries to be fragmented into smaller mappings in order to satisfy the map entry passing requests. Quick reference mode takes advantage of the fact that the passed memory is going to be briefly used and then unmapped. This allows UVM to relax the management of references to object and amap structures and avoid fragmenting the source map. Quick references are currently used by the `ptrace` system call to read and write the memory of a process that is being debugged.

4.3.2 Using Map Entry Passing

Map entry passing can be used to exchange data between processes and the kernel. For example, we built a memory `export/import` IPC facility on top of UVM's map entry passing mechanism that allows a process to export ranges of its virtual address space to other processes. A process with the appropriate permission can import one of these virtual address ranges into its address space. This IPC facility is similar to System V shared memory, however it is more generalized.

5 Measurements

In this section we describe five sets of tests designed to measure the performance of UVM's new features. The results of these measurements show that page loanout, page transfer, and map entry passing can significantly improve I/O and IPC performance over traditional systems.

All our tests were performed on 200 MHz Pentium Pro processors running NetBSD 1.3A with either UVM or BSD VM. Our Pentium Pros have a 16K level-one cache, a 256K level-two cache, and 32MB of physical memory. The main memory bandwidth of our Pentium Pros is 225 MB/sec for reads, 82 MB/sec for writes, and 50 MB/sec for copies, as reported by the `lmbench` memory bandwidth benchmarks (version 1.2) [12].

5.1 Page Loanout Performance

A convenient place to measure the effect of page loanout on I/O is in the transfer of data from a user process to a device. In a traditional system, this would involve copying the data from the user's address space to a kernel buffer, and then performing I/O on that buffer. With page loanout, such an I/O operation could be done by loaning the pages from the user's address space directly to the device, thus avoiding the data copy. Networking devices are often chosen for such measurements, so we measured the effect of page loanout on the speed with which data can be pumped through the BSD networking subsystem.

One problem with measuring the performance of page loanout with the networking subsystem is that current processors can easily generate data faster than current generation network interfaces can transmit it[2]. For example, the Pentium Pro processor on our test machine can easily swamp the network interfaces we have available (100Mbps fast Ethernet and 155Mbps ATM cards). This results in either the transmitting process being stalled until the network interface layer can catch up, or in data being discarded at the network interface layer due to a full network queue. If one of these network interface cards was used in measuring page loanout, any positive results achieved by page loanout would be seen as a reduction in processor load rather than increased bandwidth. In order to avoid being limited by network interface hardware a new "null" protocol layer was introduced into the BSD kernel. This protocol discards data

[2] Next-generation network interfaces such as Gigabit Ethernet and higher-speed ATM interfaces can transmit data at higher speeds.

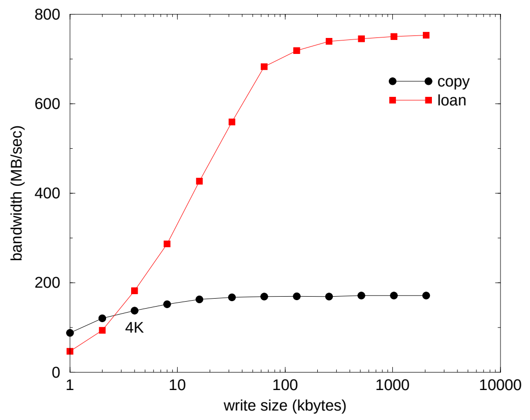

Figure 5: Comparison of copy and loan procedures for writing to a null socket

rather than passing it to a network interface. This allows the data transfer overhead of copying verses page loanout to be directly measured.

To measure the effect of page loanout we wrote a test program that uses the null protocol to transmit large chunks of data through the socket layer. The actions performed by the test program are similar to programs such as ftp, web, and video servers. Such programs operate by opening data files and transmitting the file's content over the network. The test program transmits data using either the normal `write` system call to copy data from user space into kernel buffers, or using a modified `write` system call that moves user data into the networking system through page loanout.

The test program was run using a two megabyte buffer that was transmitted 1024 times. Thus, two gigabytes of data was transmitted for each run of the test program. Each run of the program was timed so that the bandwidth of the null socket could be determined. The write size was varied from 1K to 2048K. The results of the test using copying and page loanout are shown in Figure 5.

For write sizes that are less than the hardware page size (4K), copying the data produces a higher bandwidth than using page loanout. This is due to the page loanout mechanism being used to loan out pages that are only partially full of valid data. For example, when the write size is 1K for each page loaned out there is 3K of data in the page that is not being used. Once the write size reaches the page size, page loanout's bandwidth overtakes data copying. As the write size is increased to allow multiple pages to be transmitted in a single `write` call, the data copying bandwidth increases until the write size reaches 32K. The data copy bandwidth never exceeds 172 MB/sec. Note that the data copy benefits from the fact that a streaming socket was used. This allows the socket layer to break up large writes into smaller ones that fit into the processor's cache. If a datagram socket is used, then the data copy bandwidth drops to 50 MB/sec, the bandwidth limit obtained from lmbench. Meanwhile, the loanout bandwidth rises sharply as the write size is increased. When the write size is 32K the loanout bandwidth is 560 MB/sec. The loanout bandwidth levels off at 750 MB/sec when the write size is 512K. Clearly, page loanout improved the I/O performance of the test program by reducing data copy overhead. Thus, applications that transmit data without touching it, such as the multimedia video server described in [4], could benefit from page loanout.

5.2 Page Transfer Performance

Page transfer moves data in the opposite direction than page loanout does. In page loanout, a process loans its data out to some other process or the kernel. In page transfer, a process receives pages from other processes or the kernel. The pages received may either be loaned from another process or donated from the kernel. The benefit of using page transfer is that the transferred data is moved into the receiving process' virtual address space without the overhead of a data copy.

The effect of page transfer on the overhead of I/O operations can be measured using the kernel's networking subsystem in a way that is similar to the procedure used to measure the effect of page loanout. In a traditional system, when data is read from a socket it is copied from a kernel mbuf into the receiving process' virtual address space. With page transfer, the network read operation on larger chunks of data can be done by using page transfer to transfer a large mbuf's data area from the kernel directly into the receiving process' virtual address pace, bypassing the copy. For this to be feasible, the kernel should be configured so that the buffer size of a cluster mbuf is equal to the system page size. To measure the effect of page transfer a new "null" socket read call was introduced into the BSD kernel. When the socket layer detects a null socket read, it allocates an mbuf chain of the requested size (containing random data) and uses it to satisfy the socket read request. This allows us to directly measure the effect of page transfer on I/O overhead by observing the bandwidth of null socket reads. Note that unused portions of a cluster mbuf's data area are zeroed to ensure data security (this is a concern when we are transferring real data rather than null-socket data).

We wrote a test program that uses null socket reads to receive large chunks of data through the socket layer to measure page transfer performance. Thus, the test program is similar to programs that receive large chunks of data from the network or from other processes. The test program was used to transfer 1GB of data first using data copying and then using page transfer. The read size was varied from 4K (one page) to 2048K, and the bandwidth was measured. The results of the test are shown in Figure 6.

When copying data, smaller reads produce a greater bandwidth. As the read size increases, the null socket read bandwidth decreases from 315 MB/sec to 50 MB/sec. This

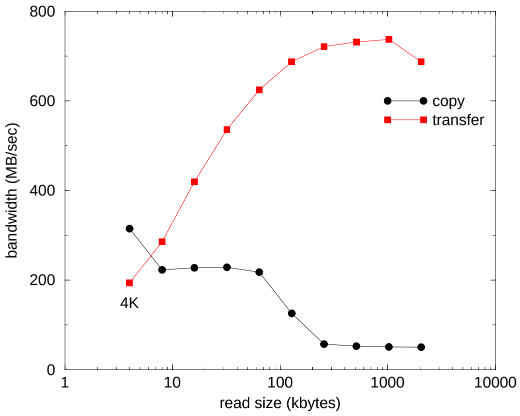

Figure 6: Comparison of copy and transfer procedures for reading from a null socket

decrease is due to data caching on the Pentium Pro. Memory accesses to the cache are significantly faster than memory accesses to main memory. If an application's buffer entirely fits within a cache then memory references to it will be faster than the bandwidth of main memory. For example, when the test program uses 4K buffers both the source and destination buffers of the data copy fit within the level-one cache producing a bandwidth of 315 MB/sec. When the size of the test program's buffers are increased to 8K they no longer fit within the level-one cache, but they do fit within the level-two cache. This results in a bandwidth drop to 225 MB/sec. The bandwidth remains nearly constant for read sizes from 8K to 64K. However, between 128K and 256K the bandwidth drops to 50 MB/sec as the size of the data buffers becomes larger than the level-two cache. Finally, the bandwidth eventually levels out at 50 MB/sec (the same value we got from `lmbench`'s memory copy benchmark). The 4K read size gets the full benefit of the high bandwidth of the Pentium Pro's level-one cache since it was the only process running on the system when the measurements were taken (so its data was always fresh in the cache).

On the other hand, the cache does not play as great a role in page transfer. The bandwidth starts around 190 MB/sec for page sized transfers, and then increases up to 730 MB/sec for a 1MB page transfer. The final dip in the page transfer curve is due to the cache. As the transfer size gets larger, the page transfer code touches more kernel data structures, and at some point this causes touched data structures to exceed the size of the level one cache.

5.3 Using UVM to Transfer Data Between Processes

In the previous two sections we discussed the effect of page loanout and page transfer on the movement of data between a process and a network interface. In this section, we examine using UVM to move data between two processes using a pipe. In a traditional system, a pipe is implemented as a pair of connected sockets. Each byte of data written to the pipe is copied twice. UVM's page loanout and page transfer mechanisms can be used to reduce kernel overhead by eliminating both these copies.

The kernel's socket layer was further modified in order to use UVM to eliminate data copies when using pipes. While the page loanout modifications described earlier was sufficient for use with a pipe, the page transfer modifications for the null socket read operation were not. In order to support page transfer for pipes, the socket receive function was modified to use page transfer on large mbufs when requested. The modifications allows the socket layer to use page transfer for both cluster mbufs and external mbufs whose buffers are pages loaned out from some other process.

To measure the effect of both page loanout and page transfer on the transfer of data between two processes over a pipe, we wrote a test program that operates as follows. The parameters are parsed and a pipe is created. A child process is then forked off. The child process writes a specified amount of data into the pipe, using page loanout if requested. Once the data is written the child process closes the pipe and exits. The parent process reads data out of the pipe, using page transfer if requested. After a read operation, the data is released using the `anflush` system call. Once all data is read, the parent process exits. The test program takes as parameters the amount of data to transfer over the pipe and the read and write sizes.

We used the test program to produce four sets of data. For each set of data we transfered 1GB of data varying the read and write sizes from 4K to 2048K. The four data sets are:

copy: Data was copied on both the sending and receiving ends of the pipe.

transfer: Data was copied on the sending side and moved with page transfer on the receiving side of the pipe.

loan: Data was loaned out on the sending side and copied on the receiving side of the pipe.

both: Data was loaned out on the sending side and moved with page transfer on the receiving side of the pipe.

The results of this test are shown in Figure 7.

For page sized data transfers (4K) copying produces a bandwidth of 40 MB/sec, while page transfer and page loanout produce bandwidths of 33 MB/sec and 45 MB/sec, respectively. Using both page transfer and page loanout at the same time with page sized transfers produces a bandwidth of 70 MB/sec.

As the transfer size is increased, the bandwidth of data copying drops and levels off at 26 MB/sec due to the effects of the cache. Meanwhile, the bandwidth of both transfer

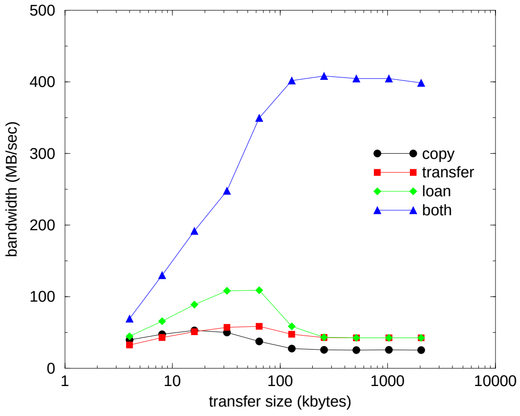

Figure 7: Comparison of copy, transfer, loan, and both procedures for moving data over a pipe

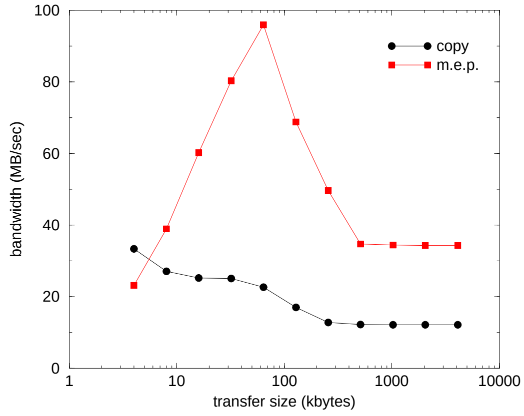

Figure 8: Comparison of copy and map entry passing (m.e.p.) transferring memory between two processes

and loanout levels off at 43 MB/sec as the transfer size is increased. Since both these tests still have data copying at one end of the pipe, they are effected by caching in the same way as data copying, however their bandwidth is almost double that of copying because they are only touching the data on one end. The page loanout curve rises higher than the page transfer curve in mid-range transfer sizes because page loanout has less overhead than page transfer. Page transfer has the added complication of having to replace the data page removed from cluster mbufs with another page. The bandwidth of the test in which both loanout and transfer are used rises sharply and levels off at 400 MB/sec for large transfer sizes. In this case the data is not touched or copied at all, instead a copy-on-write mapping to the sending process' page is added to the receiving process.

5.4 Map Entry Passing Performance

Data passing between programs through pipes or sockets is quite common on Unix-like systems. If an application passes large amounts of data over a pipe, it might benefit from map entry passing. To measure the effect of map entry passing, we wrote two test programs that pass a fixed-sized chunk of data between two processes a specified number of times. One test program passes data between processes by sending it over a local socket. This causes the kernel to copy the data from the sending process into an mbuf chain, and then to copy it from the mbuf chain to the receiving process. The other test program passes data between processes by using map entry passing. The test program can optionally "touch" the data after it arrives to simulate data processing.

Each test program was run using transfer sizes ranging from 4K to 4096K. The number of times the data was transferred from parent to child process and back was adjusted so that the test program ran for at least thirty seconds. Each run of the program was timed so that the bandwidth could be determined.

As shown in Figure 8, the bandwidth curves for the test program that uses data copying via the socket pair start off near 35 MB/sec for a one page transfer size. The bandwidth decreases as the transfer size increases due to caching effects. The bandwidth of the data-copying tests level off at 12 MB/sec. On the other hand, the bandwidth curve for the map entry passing test program peaks around 96 MB/sec for a transfer size of 64K and then caching effects on kernel VM data structures cause it to fall and level off at 34 MB/sec.

If the application does not touch the data after it is transfered then the bandwidth for data copying levels off at 18 MB/sec rather than 12 MB/sec. On the other hand, if the map entry passing test program does not touch the data, then the bandwidth rises rapidly, reaching 18500 MB/sec for a transfer size of 4096K. This high bandwidth occurs because the data being transfered is accessed neither by the application program nor the kernel, so the data is never loaded into the processor from main memory.

Another test program we wrote combines the use of map entry passing with page loanout. The test program consists of three processes. The first process allocates and initializes a data buffer of a specified size and then sends it to the second process. The second process modifies the data and then sends it to the third process. The third process receives the data and writes it to a null protocol socket. Then the first process can generate another buffers worth of data. Such a data pipeline application is similar to the multimedia "I/O-pipeline model" described in [15].

In the test program the data can be exchanged between the processes either with data copying or with map entry passing. The final process can write the data out with either data copying or page loanout. We ran the test program with transfer sizes from 4K (one page) to 4096K. The number of transfers was adjusted so that the test program ran for at least thirty seconds. The results of the the tests are shown

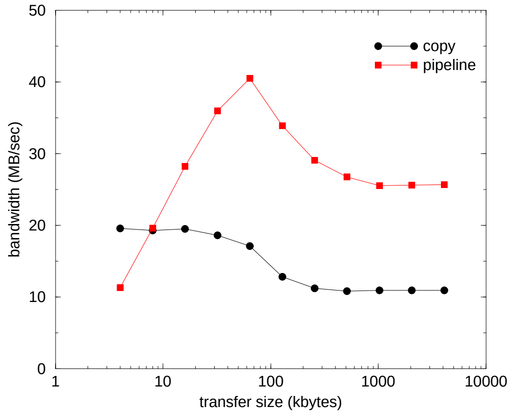

Figure 9: Comparison of copy and map entry passing/page loanout pipeline

in Figure 9.

When using data copying, the bandwidth started at 20 MB/sec and then dropped and leveled off at 11 MB/sec once the cache size was exceeded. When using map entry passing and page loanout, the bandwidth starts at 11 MB/sec for a page-sized transfer, and increases to a peak of 40 MB/sec for a 64K transfer size. It then levels off at 26 MB/sec as the kernel data structures exceed the size of the cache.

5.5 UVM Compared to Related Work

Comparing the performance of UVM data movement mechanisms with related work is difficult due to the wide variety of hardware platforms and different testing procedures used across projects. However, we can make some rough estimates of relative performance improvements based on our results and results published in the literature. For example, the Solaris zero-copy TCP mechanism with checksum in hardware [5] produces a factor of 1.4 improvement over data copying. UVM's page loanout facility (without the overhead of protocol processing) produces a factor of 2.6 improvement for the same transfer size. In the Container Shipping project [1] the performance of an IPC pipe using the container shipping mechanism to send, receive, and both send and receive was compared to the performance of using data copying. The reported improvement factors are 1.2, 1.6, and 8.3, respectively. A comparable experiment under UVM using page loanout, page transfer, and both mechanisms at the same time produced improvement factors of 2.1, 1.7, and 14. Other projects report results that appear to be comparable. Our rough comparisons show that UVM produces improvements on the same order of magnitude as the related work.

6 Conclusions

UVM reduces or eliminates the need to copy data thus reducing the time spent within the kernel and freeing up cycles for application processing. Unlike the approaches that focus exclusively on the networking subsystem, our approach provides a general solution that can improve efficiency of the entire I/O subsystem.

UVM provides the BSD kernel with three new mechanisms that allow processes to exchange and share page-sized and mapping-sized chunks of memory without data copies: page loanout, page transfer, and map entry passing. While it would be foolish to use these mechanisms on small data chunks, our measurements clearly show that for larger chunks of data they provide a significant reduction in kernel overhead as compared to data copying. For example: UVM's page loanout mechanism provides processes with the ability to avoid costly data copies by loaning a copy-on-write copy of their memory out to the kernel or other processes. UVM's page transfer mechanism allows a process to receive anonymous pages of memory from other processes or the kernel without having to copy the data. UVM's map entry passing mechanism allows processes to easily copy, share, and exchange chunks of virtual memory from their address space. In addition to being used separately, we have shown that these mechanisms can be used together.

UVM is now part of the standard NetBSD distribution and is scheduled to replace the BSD VM system for NetBSD release 1.4. *(Note, some of the data movement related code is currently being merged in, all should be merged in before the 1.4 release and before OSDI.)* UVM already runs on almost all of NetBSD's platforms and is expected to run on every NetBSD platform soon. A port to OpenBSD is also expected.

UVM can improve the performance of current applications through the page loanout mechanism and secondary improvements to the VM. Further work to introduce APIs for page transfer and map entry passing will allow applications to take advantage of these mechanisms as well.

References

[1] E. Anderson. *Container Shipping: A Uniform Interface for Fast, Efficient, High-Bandwidth I/O*. PhD thesis, University of California, San Diego, 1995.

[2] J. Barrera. A fast Mach network IPC implementation. In *Proceedings of the USENIX Mach Symposium*, pages 1–11, November 1991.

[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] M. Buddhikot, X. Chen, D. Wu, and G. Parulkar. Enhancements to 4.4 BSD UNIX for networked multimedia in project MARS. In *Proceedings of IEEE Multimedia Systems'98*, June 1998.

[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*. PhD thesis, 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] J. Dyson, D. Greenman, et al. The FreeBSD VM system. See `http://www.freebsd.org` for more information.

[9] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *Proceedings of USENIX Winter Conference*, pages 327–333. USENIX, January 1993.

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

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2nd edition, 1996.

[12] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of USENIX Conference*, pages 279–294. USENIX, 1996.

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

[14] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer Conference*, pages 247–256. USENIX, 1990.

[15] J. Pasquale, E. Anderson, and P. Muller. Container Shipping: Operating system support for I/O intensive applications. *Computer*, 27(3), March 1994.

[16] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computing*, 37(8), August 1988.

[17] D. Solomon. *Inside Windows NT*. Microsoft Press, 2nd edition, 1998. Based on the first edition by Helen Custer.

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

[19] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.