Implementation of Multiple Pagesize Support in HP-UX

Indira Subramanian, Cliff Mather, Kurt Peterson, and Balakrishna Raghunath
Hewlett-Packard Company

Implementation of Multiple Pagesize Support in HP-UX

Indira Subramanian
Cliff Mather
Kurt Peterson
Balakrishna Raghunath

Hewlett-Packard Company
Cupertino, CA 95014
indira@cup.hp.com

Abstract

To reduce performance degradation from Translation Lookaside Buffer (TLB) misses without significant increase in TLB size, most modern processors implement TLBs that support multiple pagesizes. For example, Hewlett-Packard's PA-8000 processor allows 8 hardware pagesizes, in multiples of four, ranging from 4 Kbytes to 64 Mbytes.

In implementing multiple pagesize support in HP-UX, we chose to create large pages at page-fault service time. We have a buddy system allocator that provides interfaces for allocating and freeing multiple pagesizes. We maintain the Virtual Memory (VM) data structures such as the pagetable entry, virtual page frame descriptor, and physical page frame descriptor based on the smallest pagesize, and represent a large pagesize as a collection of these base pagesize structures. In our implementation, VM operations on a large pagesize such as 16KB are carried out by looping over the 4KB-based constituent VM data structures. Our system offers significant application performance improvement when using large pagesizes.

1 Introduction

Translation Lookaside Buffer (TLB) misses can degrade the performance of applications with large working set sizes [2, 4, 18, 21, 23]. A TLB is a cache of recently accessed virtual-to-physical page translation information. The *working set* of a process is the memory actively referenced during a certain time interval [6]. A typical TLB that performs translations using small pagesizes such as 4KB, cannot hold all the translations for a large working set. Consequently, TLB misses will result, and each miss must be handled by copying the translation information from a software or a hardware translation table (pagetable) into the TLB. A high TLB miss rate (misses per second) will result in performance degradation. While this degradation is common to all contemporary processors, the penalty can be significant in processor implementations that lack hardware support for TLB miss handling.

To increase the TLB reach, that is, the amount of memory translated by the TLB, most modern processors support multiple pagesizes. Support for a wide range of pagesizes allows for miss reduction without undue increase in working set sizes. For example, in addition to the base 4KB pagesize, the PA-8000 processor which is an implementation of the PA-RISC 2.0 architecture [10], supports 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, and 64MB pagesizes. Other architectures that allow multiple pagesizes include DEC Alpha [3], MIPS R10000 [16], and SPARC [7].

Several issues need to be considered when implementing OS support for multiple pagesizes [11, 17, 21]. How should the VM data structures, which were originally designed to represent a uniform pagesize such as 4KB, be redesigned or adapted to allow coexistence of multiple pagesizes? How will the pagesize be chosen for a given mapping? Should a large mapping be created at fault service time? Or, should it be created at a later time through page promotion [18, 23]? How should the interfaces across VM and filesystems be modified to deal with multiple pagesizes? How should physical memory be managed such that a properly aligned page of any TLB-supported pagesize may be allocated? How should candidates for page replacement be selected?

To support multiple pagesizes in HP-UX, we preserve the underlying 4KB-based VM data structures, and represent a large page by a collection of these 4KB-based structures. In this approach, a large page is effectively a set of contiguous 4KB sized pages. The PA-RISC 2.0 TLB requires that the virtual and physical addresses of a large page be aligned on the pagesize boundary. We use the terms *subpage*, *base page*, and *member page* interchangeably to refer to the 4KB pages that constitute a large page. The VM subsystem components such as the fault path, the interfaces that support page fault handlers, and the physical memory allocator operate on a large page by looping over the associated base page data structures. Our implementation offers significant performance improvement for applications with large working set sizes.

The remainder of this paper is organized as follows. Section 2 discusses related work. In Section 3, we describe the HP-UX VM system primarily focusing on the data structures, and discuss the rationale for our approach to representing large pagesizes. Section 4 describes the pagetable management. Section 5 presents the physical memory allocator. Section 6 discusses how pagesize hints may be specified for an application. The VM subsystem uses this pagesize hint as the starting point, when deciding the pagesize for mapping an address range. In Section 7, we describe support for large page creation at fault-service time. Section 8 discusses page replacement in the presence of multiple pagesizes. Section 9 presents performance data demonstrating performance improvements from TLB miss reduction. In Section 10, we summarize our approach to multiple pagesize support and reaffirm its benefits.

2 Related Work

Several approaches for supporting multiple pagesizes have been reported in the literature. In this section, we describe these approaches and relate them to our work.

2.1 Pagesizes in Partitioned Memory

Large pagesize and adaptive prepaging are among the techniques Kagimasa et al. employ to reduce memory management overhead in a terabyte virtual and gigabyte physical memory system [9]. The goal of their Super Terabyte System (STS) was to reduce overhead from page fault processing, choosing candidates for page replacement, and process swapping. Kagimasa et al. do not discuss performance effects of TLB misses. Nonetheless, their approach for implementing dual pagesize support is relevant to our discussion.

STS supports one small pagesize (4KB) and one large pagesize simultaneously. The large pagesize is one of 16KB, 64KB, 256KB, and 1MB, and is chosen at boot time. The virtual and the physical memory are each partitioned into a small page region and a large page region. The size of the virtual small page region is 2 gigabytes, while that of the physical small page region can be set at boot time. A system available physical page area (SAPA) is maintained for each of the two pagesizes. When setting up a mapping for a large virtual page, in the event the SAPA for large pages is empty, a contiguous set of small physical pages are located if possible, and used. Similarly, when setting up a mapping for a small virtual page, if the SAPA for small pages is empty, a small page is carved out of a large physical page, and the remaining small pages are placed into the local available page area (LAPA) for the process.

The system manages the two pagesizes as follows. The key storage (per physical page data structures) is maintained for each 4KB page. Large pagesize is used for allocation, page fault handling, setting referenced and modified bits, page replacement, and swapping. However, management of secondary storage as well as reading and writing operations involving secondary storage utilize 4KB pagesize.

With regard to support for multiple pagesizes, the STS design offers some flexibility but suffers from several drawbacks. The 4KB-based physical page data structures facilitate the allocation of small and large physical pages interchangeably. This flexibility enables efficient use of available physical memory in the two pagesizes. There is one limitation however – all the small pages carved out from a large page must be mapped to the same process. STS has two major drawbacks. First, only two pagesizes are supported. The large pagesize chosen at boot-time may not be appropriate for a broad range of applications. Second, virtual storage is statically partitioned into small page and large page regions. Such static partitioning may not be best suited for TLB miss reduction for different workloads.

2.2 Subblocked TLB

Talluri et al. recommend subblocked TLB organizations as better alternatives to existing TLB organizations that have been simply extended to support multiple pagesizes [21]. *Subblocking* refers to grouping of mapping information in the TLB for several base pages that are part of a page block. A *page block* is made up of 16 4KB pages that are aligned on a 64KB boundary. Subblocking saves TLB space by sharing the virtual tag across all the subpages of the page block. A *complete-subblock* TLB entry stores protection and other page attributes and a physical pagenumber for each of the subpages in the page block. A *partial-subblock* TLB entry stores a single set of page attributes and a single physical page-number for the entire set of subpages in the page block, and therefore requires less TLB space.

Talluri et al. propose subblock TLB designs as an alternative to multiple pagesizes for improving TLB performance. They argue that invasive changes to the OS that introduce significant overhead are necessary to take advantage of multiple pagesizes. Overheads include increased disk and network traffic for page-ins and page-outs, coalescing smaller pages to create large pagesizes, and existing VM data structures not scaling efficiently for handling large pagesizes. In contrast, the complete-subblocking approach requires no changes and the partial-subblocking approach requires minimal changes to the OS for improving TLB performance. To exploit the benefits from partial-subblocked TLB, as many subpages as possible in a virtual page block must be mapped to the corresponding subpages in a physical page block. For this purpose, a reservation based memory allocation is used, and is discussed in the next section.

Our goal was to improve the TLB performance of processors based on the PA-RISC 2.0 architecture, which supports multiple pagesizes. Our implementation demonstrates that multiple pagesizes can be exploited effectively to improve TLB performance.

2.3 Page Reservation and Promotion

In the case of partial-subblocking (discussed above) and multiple pagesize TLB, Talluri et al. employ a reservation-based allocation of a page block [20, 21]. Reservation refers to setting aside properly aligned 4KB physical pages for possible use with specific virtual subpages of a process. These physical subpages are placed at the end of the freelist. When the process references these virtual pages, the ensuing page faults are serviced using the prereserved physical subpages. If the process did not reference these virtual pages, some of the reserved pages may move to the head of the freelist. These unused reserved pages are then allocated to service other page faults. A *single-page-size framework* is employed to support two page sizes – a 64KB page is represented by 16 4KB page-based data structures.

In addition to the reservation method described above, for supporting two pagesizes, their system implements a threshold-based promotion policy. This policy decides when to combine the 4KB subpages to create a 64KB superpage (large page). After a certain *promotion threshold* such as 50% of 4KB pages have been faulted in, the unreferenced pages in the page block are fetched from secondary storage into corresponding prereserved 4KB subpages. The page block is then promoted to 64KB pagesize. In the case of uninitialized data, promotion involves only zerofilling the prereserved pages. If some of the prereserved pages are no longer available, a *gather* operation may be needed when performing page promotion. In this case, a new page block is allocated, and the original 4KB physical pages are copied into the new page block to create a large page mapping. Talluri did not implement this gather mechanism in his system[20].

Talluri's system poses several limitations. First, it provides simultaneous support for only 2 page sizes, 4KB and 64KB. We use the single-page-size framework as well in our implementation, and we demonstrate that this method is suited for all pagesizes that the PA-8000 processor supports. Second, it is unclear that the reservation approach will scale well for larger pagesizes. A promotion threshold of 50% would involve many faults before the large page is created by promoting the subpages. Furthermore, if some of the prereserved pages were unavailable resulting in random physical subpage allocations, promotion will require copying the source subpages. In contrast, we allocate and map large pages when servicing a page fault, thereby eliminating the additional faults and the promotion overhead.

2.4 Clustered Pagetable

Through simulation, using estimates of pagetable size and access time as metrics, Talluri et al. [22] demonstrate that clustered pagetables work better for superpages (large pages) than the conventional hashed pagetables. A *hashed* pagetable organization [8] uses a hash function that hashes a virtual page number to a specific hash bucket in which the translation information for the virtual page is stored. A *clustered* pagetable is a hashed pagetable enhanced with subblocking. *Subblocking* refers to grouping of mapping information for several pages, and it amortizes the per pagetable entry (PTE) overhead over many potential mappings. The aligned group of consecutive pages is called a page block. Space saving is achieved by using a single virtual tag and a single hash chain pointer for the entire subblock. In a clustered pagetable with a subblocking factor of 16 and a base pagesize of 4KB, a single clustered PTE can support pagesizes up to 64KB. A clustered pagetable provides effective support for a subblocked TLB, which was discussed earlier in Section 2.2.

Talluri et al. present approaches for storing a large page in different types of pagetables. Two solutions that work well are the multiple pagetables method and the replicated PTE method. The multiple pagetable method entails one pagetable for each of the pagesizes used in the system. To locate the mapping that caused a TLB miss, the TLB miss handler must search each of the pagetables, starting from the most likely table. A more promising approach is the *replicated* PTE method, in which a large page is represented by replicating a PTE once for each subpage.

With a clustered pagetable design, representation of pagesizes larger than page block size involves a space/time tradeoff as in conventional tables but are more efficient. Assuming a subblocking factor of 16 and a base pagesize of 4KB, pages larger than a page block (64KB) can be represented by replicating the 64KB clustered PTEs. In contrast, with a conventional pagetable, sixteen times as many 4KB PTEs must be replicated. With the multiple pagetable approach, clustered pagetables require fewer tables. For example, one clustered table can be used for pagesizes 4KB to 64KB, and another for up to 1MB, and so on. With conventional pagetables, we will need as many tables as the number of pagesizes supported.

The clustered pagetable does not avoid the major complexities involved in supporting large pagesizes. Talluri et al. proposed using replicated clustered PTE to represent pagesizes larger than 64KB. Therefore, clustered pagetable implementation entails the complexity along the same lines as ours, namely looping over operations across subblock structures. Indeed it is true that a single lock could be used for each subblock of PTEs in the case of a clustered pagetable. With the conventional hashed table which is used in HP-UX [8], the locks have to be acquired individually for each base page PTE, and therefore, can entail high overhead. However, our implementation performs well despite this overhead.

We preferred using the replicated PTE method when implementing multiple pagesize support in the page directory (pagetable), as discussed in Section 4. Saving space was not compelling enough a reason for moving to clustered structures – we needed to gather performance data from an implementation to justify total redesign of the data structures. Regarding hashed pagetable without clustering, Talluri et al. raise several concerns. One concern is that the hash chains will be longer because a large page will use multiple base page entries. However, in our design we size the hash table based on the size of physical memory, and hence, the hash chains are small and no different from the case when only base page mappings are used. Another concern is that the entries corresponding to the subpages will be in different hash buckets thereby making operations that involve all the subpage entries less efficient. Our implementation does involve updating all the subpage translation entries in such cases as modifying a translation, and our system performs well despite this inefficiency.

2.5 Online Page Promotion

Another approach for reducing TLB misses employs online superpage (large page) promotion [18]. This work describes a technique for monitoring TLB miss traffic to decide when a superpage should be constructed. On each miss, page reference information (a set of counters) is updated to indicate the number of TLB misses that would not have occurred had the set of already assigned superpages been larger. Several online policies that perform suitable superpage promotions based on the counters, the TLB miss cost, and page copying cost are presented. The modifications to the TLB miss handler for the purpose of tracing does slow the miss handling. However, the overhead is absorbed into the significant performance gains made through page promotions.

While the online methods have advantages, the extent of modifications to the hardware independent layer may be significant. The key advantage of online methods is that superpages are created only when necessary, thereby ensuring that the working set size does not increase dramatically. In addition, the online approach appears to be less invasive, since the superpage awareness is confined to the hardware dependent layer. However, Romer et al. do not discuss all the details pertaining to page promotion. First, when their system is about to promote to a superpage whose subpages are not all resident, the nonresident pages must be brought in. Presumably, the fault-path could be repeatedly invoked to bring in the nonresident pages. Second, their system will need an allocator for allocating multiple pagesizes. Furthermore, demoting a superpage to subpages whenever the modified and referenced bits are queried from the hardware independent side, may not be desirable. For instance, paging out an unreferenced large page may be more efficient.

Despite the concern that OS support for multiple pagesizes would be invasive and complex, we concluded that by preserving the underlying 4KB data structures and using simple locking protocols, we had a promising approach that would lead to a successful implementation. We create large pages at fault-service time thereby eliminating the overhead from slower TLB miss handling as well as copying costs that the online promotion method entails. Since our approach will reduce the number of faults, it has the potential to offset some of the overhead from other large page related operations.

3 Data Structures for Large Pages

First we present an overview of PA-RISC and HP-UX VM architecture and data structures. Then we discuss the alternatives we considered with regard to representing large pagesizes and the approach we have taken.

3.1 VM Data Structures Overview

We begin with a description of addressing and access control in PA-RISC. Next we present the key data structures employed by the hardware dependent and the hardware independent components of the HP-UX VM subsystem. The hardware independent component is based on UNIX System V Release 2 and Release 3 [1].

The PA-RISC architecture defines a global virtual address space [14]. A global virtual address is made up of two components: a space identifier (space-ID), and an offset. The offset in turn is partitioned into a virtual page number, and a page offset. The PA-RISC 2.0 allows a 64 bit space-ID and a 64 bit offset, which are combined to generate up to a 96 bit global virtual address [10]. In the 32-bit and 64-bit HP-UX implementations, a process can access up to 4GB and 16TB respectively, using 4 space-IDs.

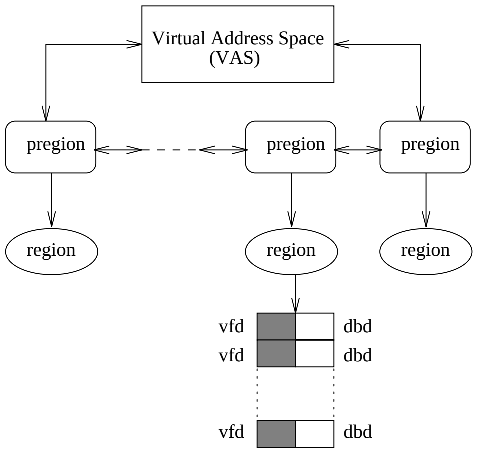

Figure 1: Hardware Independent VM – vas

Since the precision architecture uses global addressing, it employs a set of mechanisms to restrict what parts of this global space a process can access. The privilege level (0 through 3), access rights (read, write, execute), and a protection ID (PID) are used to control the access of a page by a process. The most privileged level of 0 is kernel mode, and the least privileged level of 3 is the user mode. Each process is assigned a set of PIDs, four of which are cached in control registers. To be allowed access to a page, the page's PID must match one of the process' PIDs.

Each process is associated with a *virtual address space* (*vas*) shown in Figure 1, which is made up of a list of *pregions*. Each pregion represents a range of virtual pages. As shown in Figure 1, each pregion points to a system-wide kernel data structure called a *region*. Each page in a region is associated with a *virtual frame descriptor* (*vfd*) that specifies the *page frame number* (*pfn*) and a *disk block descriptor* (*dbd*) that specifies the location of a page on disk. The vfd, dbd pair constitute the hardware independent pagetable entry. The vfds and the dbds are maintained in chunks of 32 pairs. The chunks are organized as a B-tree [5] for efficient access.

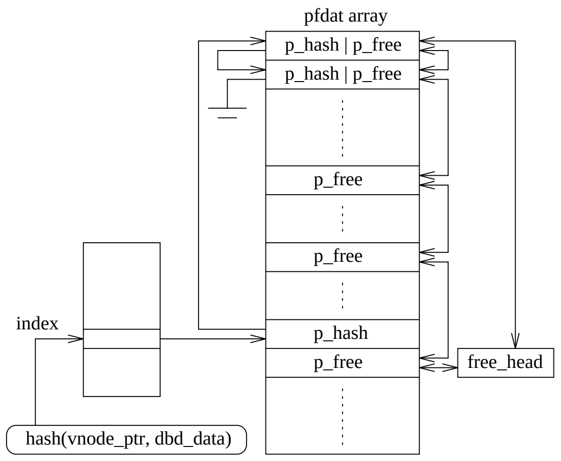

Figure 2: Hardware Independent VM – pfdat

Another central kernel data structure on the hardware-independent side called *pfdat* shown in Figure 2, is used to manage physical memory that can be allocated to processes on demand. Each physical page frame (pfn), which is 4KB in size is represented by a pfdat structure. The pfdat entries of physical pages that are available for subsequent allocation are placed on a doubly linked list referred to as the *freelist*. Page frames associated with space allocated on secondary storage are placed on the hashed *pagecache* list. These pages are caches of data on secondary storage. The pfdat structure does not hold a pointer to the region structure associated with the virtual page frame that is mapped to the pfn.

On the hardware-dependent side, a system-wide hashed page directory (pagetable) [8] referred to as the *pdir* is used to hold the virtual-to-physical address translation information. The pdir shown in Figure 3 contains one entry (*pde*) for every 4KB page of physical memory in the system, plus entries for mapping virtual I/O pages. The pdir layer performs operations

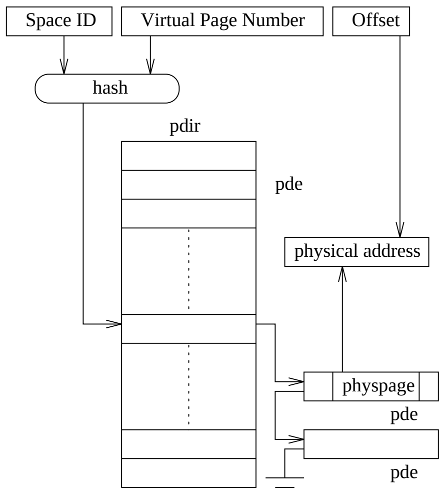

Figure 3: Hardware Dependent VM - pdir

to establish and manage the virtual-to-physical mappings for pages. Additionally, to manage address aliasing, it maintains a table of physical-to-virtual mappings. The hardware dependent layer (HDL) functions provide the interface for requesting such operations as adding and deleting translations.

The PA-RISC 2.0 architecture allows for either a separate or a combined data and instruction TLB. The PA-8x00 implementations use a combined TLB. Each TLB entry contains the virtual page number (tag), physical page number, an encoded 4-bit pagesize, access control information, and single-bit flags such as *dirty*, *break*, and *uncacheable*. A range of pagesizes from 4KB to 64MB (in multiples of four) are supported. If a TLB entry holds the matching virtual page number, based on the pagesize field, the corresponding 38 to 52-bit physical page number is concatenated with the 12 (4KB) to 26-bit (64MB) offset, to generate a 64-bit physical address. In the event of a TLB miss, a software miss handler fetches the translation from the pdir and inserts it into the TLB.

3.2 Representing Large Pagesizes

We considered two approaches for representing large pagesizes in HP-UX. One approach was to employ variable pagesize based structures, that is, to have each of the data structures represent variable pagesizes. Even with this method, we were not looking to redesign the VM system from scratch, and we wanted to reduce the extent of changes such as the pregion and the region algorithms. One implication of this requirement was that when using this method, each pregion/region represent a uniform pagesize. Another approach was to preserve the VM data structures based on 4KB pagesize, and represent a large pagesize as a collection of these base pagesize structures. Talluri et al. refer to this method as the replicate-PTE method [22] and the single-page-size framework [20] in the context of pagetables and the hardware independent VM module respectively. We will refer to it collectively as the replication method. After considering the benefits and the drawbacks of the two alternatives, we chose the replication approach.

While variable pagesized VM structures use less space and are updated efficiently, this method suffers from several drawbacks. First, changes are pervasive in that most of the VM system assumes systemwide single pagesize, and therefore, must be modified to use variable pagesized data structures. Second, since each pregion/region must represent a uniform pagesize, typically the number of pregions/regions that need to be traversed will increase. For example, when servicing a fault, if the large pagesize specified for a region cannot be allocated, the pregion/region must be suitably split so that a smaller pagesize can be used. The alternative of waiting for a large page to become available may lead to an unacceptably long page-fault latency. Also, since a large page must be aligned on the pagesize boundary, multiple pregions/regions may be needed for a virtual address range, to meet the alignment restrictions. Third, swap management will need to handle different sizes on swap space, and must minimize fragmentation, much like the physical memory allocator. Fourth, a large page protected by a single lock will lead to contention, when I/O operations are performed to non-overlapping portions of a memory region such as shared memory.

We chose the replication method for our implementation. This method is attractive for several reasons. Changes needed for multiple pagesize support are not pervasive – modifying attributes of part of a large page does not need splitting operations from pregion and region layers. Chances for contention are low, since many operations need to lock a specific 4KB pfdat even when the pfdat represents a member 4KB page of a large page. For example, examining a translation requires holding only one subpage pfdat lock. On the other hand, modifying a translation requires that locks on all subpage pfdat structures be held. Indeed, the replication method also suffers from some drawbacks. First, it does not take advantage of large pagesizes to reduce space used by the VM data structures. Second, locking, access, and update of data structures are inefficient – need to loop over the base pages of a large page. However, this looping overhead is no worse than when all the mapping are 4KB pagesize. We chose the replication method, because its benefits outweigh its shortcomings.

4 Hashed pdir Management

A large page is represented by multiple pde (page directory entry) structures, one each for a 4KB subpage. Each pde includes pagesize information. When a TLB miss occurs on a large page, the miss handler has no knowledge of the pagesize of the page on which the miss occurred. The miss handler locates the pde corresponding to the faulting address, and inserts it into the TLB. The replication method entails no additional complexity in the TLB miss handler, and the miss penalty remains unchanged in general, from the 4KB mapping case.

There is one caveat to the claim above that the TLB miss handler does not entail additional complexity – the handler for shared memory multiprocessors (MP) had to be modified to avoid disabling interrupts for too long. When a property of a translation for a large page is to be changed, the pdir management module must invalidate all the associated pdes, purge the TLBs, update the pdes, and then revalidate the pdes. If a TLB miss occurs on another processor in the meantime, it would be spin waiting in the TLB miss handler with interrupts disabled, until the subpage pde becomes valid again. To avoid holding off interrupts for too long, the MP TLB handler has been modified to enable interrupts, handle any pending interrupts, and then try accessing the pde again. This process is continued until the pde becomes valid.

Operations pertaining to a large page translation could involve access to a specific subpage or all the subpages. For example, the TLB miss handler will update information such as the *referenced* and the *modified* bits for the subpage pde on which the miss or the trap occurred. However, these bits indicate the status of the entire large page (if any) associated with this subpage. Therefore, the HDL functions must return the values accordingly, in response to queries from the hardware independent layer. To add a translation for a large page, pdes are allocated and added for each subpage. All operations that update a translation must update the pde for each subpage, and consequently, are more expensive. This overhead has not been a problem on the several benchmarks that we used for our performance measurements.

5 The Physical Memory Allocator

Support for multiple pagesizes places several new requirements on the physical memory allocator. First, the allocator must be able to allocate any of the pagesizes supported by the architecture. The page allocated must be aligned at a starting physical address that is a multiple of the pagesize. Second, the allocator must maintain the free and cached pages in such a way that fragmentation is minimized, and large pages can be found easily. The allocator must be efficient – its performance should not be much worse than the original 4KB page frame allocator.

We have implemented a binary buddy system allocator [13]. The allocator maintains the available memory pool as two subpools, the *uncached* subpool and the *cached* subpool. Pages in the cached subpool are linked to the pagecache list as discussed in Section 3.1. Each subpool has one *freelist* per pagesize. Only the first pfdat of a large page is linked to the appropriate freelist. The allocator maintains a pagesize field in the pfdat structure. Given a member pfdat of a large page, it is possible to find the first pfdat of the large page. The total count of pages allocated is maintained for each pagesize. Count of free pages in each of the cached and the uncached subpools is also maintained for each pagesize. These counters are updated as appropriate when pages are allocated, freed, or demoted. *Demotion* refers to breaking a large page mapping such as 64KB into smaller pagesize mappings such as 16KB. For instance, a process may request that the protection attributes of a memory range be modified, and this range may be a part of a large page. In this case, the VM subsystem must demote the large page, and then update the protection attributes for the demoted pages that lie in the requested range.

The buddy system reduces fragmentation, and increases the chance of finding a large page. In response to an allocation request, the allocator first searches the list for the pagesize requested, and if that list is empty, it searches the lists for bigger pagesizes. The uncached subpool is searched before the cached subpool, with the goal of preserving the cached pages as much as possible to facilitate reuse. In response to a request to free a page, the allocator attempts to locate the pfdat structure for the buddy page. If the buddy is on the same list, the allocator can coalesce.

The allocator employs different coalescing policies for the cached and the uncached subpools. The allocator coalesces the uncached pages as soon as they are freed. Coalescing can bubble up to the larger pagesize freelists. In contrast, the allocator uses a lazy approach to coalescing pages in the cached subpool. The freelist for each pagesize in the cached subpool is allowed to grow to a certain fraction of the total count allocated in that pagesize, before coalescing is performed. This fraction at this time is 25%. Experience with more workloads will be needed to determine suitable values for this watermark. The lazy approach reduces the amount of time spent in coalescing, given that the cached pages may be reused again.

The current implementation does not coalesce across the cached and uncached subpools. Once again, this choice was made to allow for reuse of cached pages.

Fragmentation is an issue despite the use of a buddy system allocator. Pervasive or transient heavy workloads can lead to fragmentation of the available memory pool. The allocator may not be able to find a large page, because one or more of the subpages may be in use. We are aware of some environments where memory fragmentation is not likely to occur, and others, where fragmentation could lead to low availability of large pages. Support for reducing fragmentation would involve paging out a certain 4KB page or copying it to a different page and freeing the source page, to create contiguous physical base pages.

6 Pagesize Hints

A pagesize hint is available to the fault-path from the region data structure. A region's pagesize hint is determined using one of two methods – neither of these methods require recompilation of the application. In the *chatr* method, a user specifies pagesizes for an executable's text and data regions to the *chatr* (change attributes) program. The chatr program places these pagesize hints in the executable header. When the executable is *exec*'ed, the region creation routine copies the hints from the executable header into the respective region structures. In the *transparent* method, the region creation routine computes the pagesize hint for a region based on the region size (number of 4KB pages), and saves it in the region structure. Hints computed using the transparent method are bounded by a minimum pagesize and a maximum pagesize, which are system tunables. The actual pagesize selected by the fault-path could be smaller than a region's pagesize hint, for reasons discussed in Section 7.1 Furthermore, if a large page allocation fails, the fault-path reverts to using a 4KB pagesize mapping.

Pregions that grow dynamically such as the heap, need additional support for exploiting benefits from large page mappings. Recall that in our implementation, we create large pages at fault-service time only – we do not perform online promotion of subpages into a large page. Since many existing applications tend to grow their heap in small increments such as 4KB, these pregions will be subsequently faulted in as small pages. To overcome this problem, we track pregions/regions that grow to a large size in small increments, and increase their growth rate so that large page mappings can be created at fault-service time. In this approach, a process that makes a break request of size such as 4KB through a *sbrk*() call could receive a larger break size such as 16KB. The scaled-up break value is determined by the prior history of break requests, and the data pagesize hint for the process. Subsequent requests from the process involving a break value that is smaller than what the kernel returned previously require no action from the kernel.

7 The Fault-path

The fault-path uses any of the hardware supported pagesizes when servicing validation or protection faults on various parts of a process address space including text, initialized data, uninitialized data, heap, stack, shared memory, and shared libraries. Faults on uninitialized data, heap, and stack segments are serviced using zerofilled memory. Faults on text and initialized data pages, if not found in the pagecache, require secondary storage access by the page-in handler of the associated filesystem. Faults resulting from copy-on-write or copy-on-reference sharing are resolved by copying the source page. The fault-path can create a large page mapping, when a process faults on a page that was previously paged out. In this case, the swap page-in handler may use a pagesize that is different from the one that was used prior to page-out.

In the next several sections, we describe our implementation, focusing on the representative aspects of multiple pagesize support. First, we present the infrastructure used in implementing zero-fill, filesystem page-in, and swap page-in. Then, we describe multiple pagesize support for filesystem page-in, followed by copy-on-write.

7.1 The Infrastructure

The fault-path uses three key interfaces. The pagesize selection interface determines the virtual pagesize to be used for servicing a fault. The vfd-fill interface allocates a large page, and fills the vfds with the corresponding pfns. The vfd-set interface updates status information in the vfds associated with a range of subpages.

The pagesize selection interface selects the pagesize that can be used, given a faulting 4KB page virtual address. It extracts the pagesize hint from the region structure, and lowers it if the available physical memory is less than 4 times the pagesize. Starting from this adjusted pagesize hint, this interface determines the pagesize that encompasses the faulting 4KB page. As outlined in Figure 4, a large page must meet alignment restrictions, and other conditions. The pagesize selected could therefore be smaller than the region's pagesize hint.

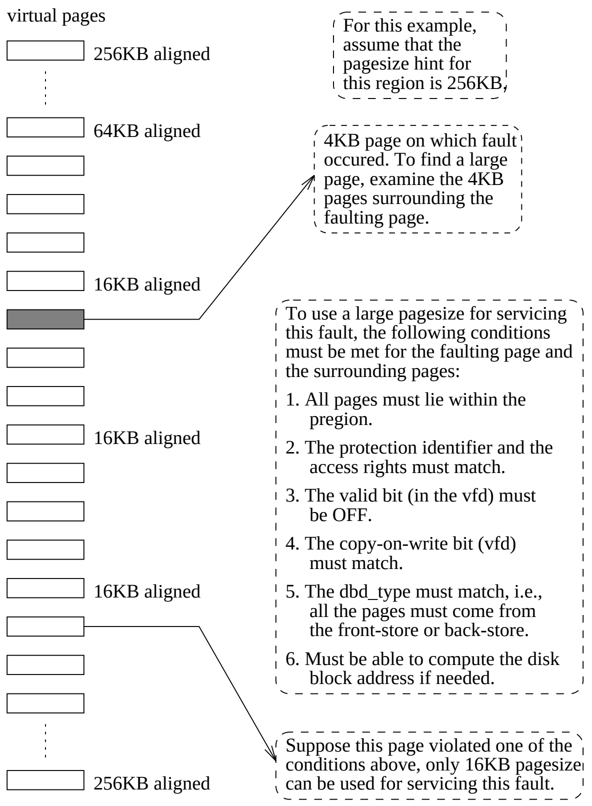
virtual pages
256KB aligned
64KB aligned
16KB aligned
16KB aligned
16KB aligned
256KB aligned
For this example, assume that the pagesize hint for this region is 256KB.
4KB page on which fault occured. To find a large page, examine the 4KB pages surrounding the faulting page.
To use a large pagesize for servicing this fault, the following conditions must be met for the faulting page and the surrounding pages:
1. All pages must lie within the pregion.
2. The protection identifier and the access rights must match.
3. The valid bit (in the vfd) must be OFF.
4. The copy-on-write bit (vfd) must match.
5. The dbd_type must match, i.e., all the pages must come from the front-store or back-store.
6. Must be able to compute the disk block address if needed.
Suppose this page violated one of the conditions above, only 16KB pagesize can be used for servicing this fault.

Figure 4: Finding a Large Virtual Page

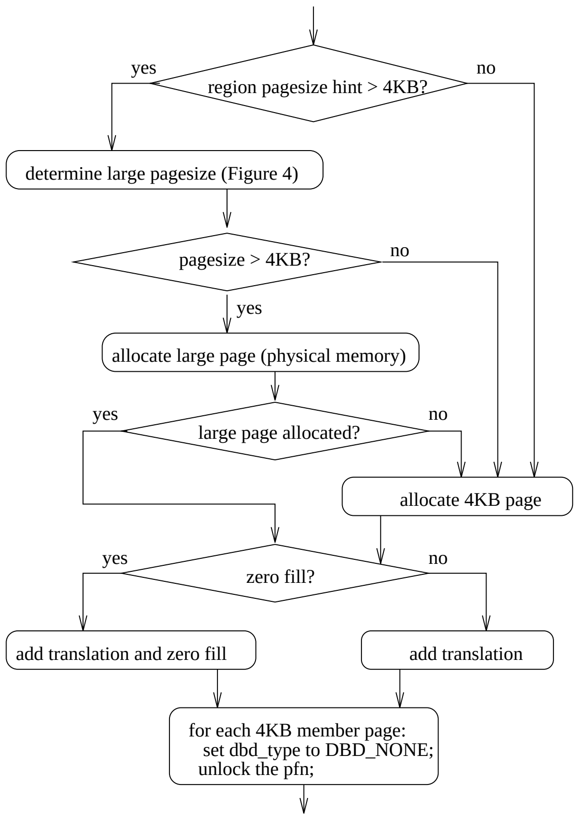
region pagesize hint > 4KB?
yes
no
determine large pagesize (Figure 4)
pagesize > 4KB?
no
yes
allocate large page (physical memory)
large page allocated?
yes
no
allocate 4KB page
zero fill?
yes
no
add translation and zero fill
add translation
for each 4KB member page:
set dbd_type to DBD_NONE;
unlock the pfn;

Figure 5: Creating a Zero-filled Large Page

The primary purpose of pagesize hint adjustment is to avoid depleting available memory by allocating large pages too aggressively. We would rather allocate a 1MB page and not a 4MB page, when 5MB of memory is available. This policy ensures that more medium-size large pages will get used in the system, instead of a few very large pages. Pagesize adjustment also increases the chance of a large page allocation request succeeding, because the allocator is more likely to find a 1MB page than a 4MB page, when 5MB of memory is available. It should be noted that since the number of pages available are low when in near memory-pressure conditions, that is, most of the memory is used by existing processes, the pagesize hint will likely get adjusted to 16K.

The vfd-fill interface makes an allocation request for a large page with the no-wait option, and if the allocation succeeds, fills the 4KB based virtual frame descriptors (vfd) with the corresponding 4KB page frame numbers (pfn). It should be noted that when a pagesize larger than 4KB is selected, the fault-path does not sleep and wait for that size to be allocated. Instead, it requests allocation with the no-wait option – the allocator will return failure if a page of that size is not readily available.

The vfd-set interface is used for setting or clearing the valid bit and the copy-on-write bit, in a range of vfds pertaining to a large page. Other looping operations involving large pages are implemented as macros, or as in-line code segments. The use of some of these interfaces is demonstrated in Figure 5.

7.2 Filesystem Page-in

Each filesystem has a page-in handler to service the fault pertaining to the respective filesystem's vnodes [12]. HP-UX 11.0 supports multiple pagesizes for UFS (UNIX File System), NFS (Network File System), and VxFS (Veritas journaling File System). Using the VFS (Virtual File System) VM initialization interface, all page-in handlers communicate whether or not they have been adapted for multiple I/O. Two of the significant tasks relevant to multiple pagesize support are finding a large page in the pagecache, and performing multiple I/O operations to bring in a large page.

The pagecache is examined after selecting the pagesize as discussed in Section 7.1. Prior to multiple pagesize support, it was necessary to look for only the faulting 4KB page in the pagecache. If the page is not found in the pagecache, then the page-in handler must bring the page in from secondary storage. In the case of a large page, it is necessary to find all the subpages in the pagecache, to have a "pagecache hit". The 4KB pages that are found must be contiguous, and the first page must be aligned on the large pagesize boundary. If the large page found is bigger than the selected pagesize, it is demoted, and then used.

The UFS, NFS, and VxFS page-in handlers have been modified to expand their I/O, taking large pagesize into consideration. The original UFS and NFS page-in handlers make the assumption that the pagesize is smaller than the filesystem blocksize, and perform a single I/O. However, this assumption is no longer valid with large pagesizes. Support has been added to the UFS and NFS page-in handlers to perform multiple I/O operations to bring in a large page. The original VxFS page-in handler makes no assumptions about the relative sizes of pages and filesystem blocks; it already handles multiple I/O when bringing in clustered-4KB pages. Therefore, the only modifications to the VxFS page-in handler involve calls to VFS VM interfaces, for the purpose outlined in the next paragraph.

It is preferable to hide the notion of large pages within the VFS VM routines. To meet this goal, existing VFS VM interfaces have been modified, and also new interfaces have been added. The page-in handlers themselves deal only with I/O expansion; they are oblivious to the pagesize being used. They make calls to the VFS VM interfaces, which determine if the I/O expansion meets the pagesize-related restrictions. If the restrictions are not met, the VFS VM routines return a result to indicate that the page-in handler must retry for a clustered-4KB page-in.

7.3 Copy-on-write

The original implementation of the module that performs copy-on-write operations, handled one 4KB page only. Here the source (physical) page is located, and the fault is resolved with or without copying the page. In the no-copy case, the source page itself is used to resolve the fault. Extending the no-copy case to handle large pages is straightforward, requiring only looping of relevant operations over the 4KB subpages.

The complexity arises in the copy case because of certain assumptions that were made in the original implementation. Operations such as allocating the physical memory for the destination page, and allocating kernel virtual space to map the source page for copying, are always expected to succeed. Some of the copy-on-write related data structure and translation manipulations are interspersed with these resource allocation operations. But in the case of a large page, the allocation request with no-wait option could fail. In addition, allocating a large page from the kernel virtual space could also fail. Either of these failures would necessitate that certain operations be backed out, which is not easily done if we simply extend the original implementation to handle large pages.

To handle the copy case for large pages, the copy-on-write module was modified to perform all resource allocations upfront. With this implementation, in the event of any resource allocation failures, the source page is demoted to 4KB pages, and copy-on-write is performed on the faulting 4KB page only. If large page resource allocations do succeed, other operations pertaining to the copy case are carried out by looping over 4KB subpages.

8 Page Replacement

The pageout daemon process uses a two-hand clock algorithm [15, 24] and a *not-recently-used* policy for page replacement. The algorithm uses two clock hands walking memory, with the first hand (*age-hand*) clearing the referenced bits of pages, and after a certain delay, the second hand (*steal-hand*) sampling the referenced bits. HP-UX bases its scanning on pregions/regions rather than physical frames, for reasons that include locking, and the need for the ability to avoid paging out from high priority processes. Also, UNIX System V Release 2 [1], on which the HP-UX VM system is based, takes this approach. As the age-hand scans pregions in the list, it ages 1/16 part of each pregion at a time.

Both the age and the steal hands must recognize large page boundaries. If the daemon encounters a large page, it completes the scan or pageout of the entire large page. The pageout daemon does not need to look for and free pages of specific sizes, because the fault-path never sleeps waiting for specific pagesizes to be allocated. This design choice is in tune with our goal of avoiding unnecessary complexity unless there is demonstrable performance gain.

In choosing a candidate for pageout, "fairness" regarding large pages and small pages is a concern. A large page has a better chance of being in referenced state, when compared to a relatively smaller page. In only one case, the pageout clearing-scan breaks a large page into smaller pages. This demotion is performed, if a portion of this large page has been wired down via a memory locking interface such as *mlock*(2). Otherwise, the daemon will scan and push pages at whatever pagesize they are. This solution involves the risk that larger pages may be kept around at the expense of pushing smaller pages, especially 4KB pages to disk.

Whether this situation is frequent and problematic will become clear from the feedback from users of real world applications. The alternate solution of breaking a referenced large page during clearing-scan into pages of the next smaller size may not be the best choice. Without support for page promotion, the benefits of a large page would be lost, leading to performance degradation from TLB misses, once memory pressure eases.

9 Performance Evaluation

We measured the performance of the benchmarks and one commercial application (Verilog-XL from Cadence Design Systems, Inc.) described in Table 1. The benchmarks and Verilog-XL were chosen, because their performance improves significantly when using large pagesize mappings. We focus on the reduction in TLB miss overhead due to large pagesizes. We do not discuss other effects such as reduced validation and protection faults, and large I/O. While these additional factors may be interesting and worthy of analysis, they are outside the scope of this paper.

All the benchmarks were run on a HP 9000 Series 800 machine with a 180 MHz PA-8000 processor. The Verilog-XL application was run on an earlier version of the machine with a 160 MHz PA-8000 processor. The processor includes 96 combined data and instruction TLB entries. The results that we report here are under no-paging conditions, that is, the working sets always fit within available physical memory. We made the performance measurements on the 32-bit version of the HP-UX 11.0 operating system. Table 2 describes the performance metrics.

We used the chatr program discussed in Section 6, to set the data and text pagesize hints in an executable header. While it is possible to specify distinct pagesize hints, we used identical hints for both data and text. Once an executable header has the hints set, the kernel will not attempt the transparent pagesize-hint selection for the process. To prevent large pages from being used for other processes that may not have been chatr'ed, we set the minimum and maximum pagesize kernel-tunables to 4KB, thereby disabling the transparent pagesize-hint selection altogether. This setup ensures that performance benefits from TLB miss reduction come from using large pages in our benchmarks alone.

The impact of TLB misses, and the benefits from using large pagesizes for the SPEC95 benchmarks apsi, compress, and vortex are shown in Table 3. For the benchmark apsi, the predominant component of the TLB misses is due to initialized data references. With 16KB pagesize, TLB misses are reduced significantly. Compared to 4KB pagesize, the memory usage increased only by 5%. Beyond the 16KB chatr pagesize, large pages do get allocated, and the memory usage increases considerably. However, there is no further

Benchmarks	Description
apsi	determines temperature, distribution of pollutants; part of the CFP95 group from the SPEC95 suite [19];
compress	compresses and uncompresses data in memory; part of the CINT95 group from the SPEC95 suite;
vortex	database program; part of the CINT95 group from the SPEC95 suite;
VMbench	3 benchmarks Nastran, ProE, and Verilog simulate VM behavior of commercial applications NASTRAN (mechanical analysis, Computerized Structural Analysis & Research Corporation), Pro/ENGINEER (mechanical design, Parametric Technology Corporation), and Verilog-XL (electronic simulation, Cadence Design Systems, Inc.);
Verilog-XL	commercial engineering application (electronic simulation) from Cadence Design Systems, Inc;

Table 1: Applications

Metrics	Description
TLB Misses (1000's)	Number of TLB misses in 1000's. Data collected using Hewlett Packard's *cyclemeter* tool.
TLB Time (m:s)	Estimated TLB miss overhead in minutes:seconds. Calculated using the average miss handling overhead of 70 cycles for a PA-8000.
Total Time (m:s)	Benchmark's total execution time in minutes:seconds. Measured using the *time* command.
User Time (m:s)	Time spent in user mode by the benchmark. Measured using the *time* command.
Mem Usage (4KB)	Total physical memory in 4KB pages mapped to the benchmark's address space, as determined at the end of the run.
Pagesize Distribution	Count of each of the pagesizes mapped to the benchmark's address space, as determined at the end of the run.

Table 2: Performance Metrics

apsi					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	132747	0:52	2:47	2:46	600
16KB	392	***	2:05	2:05	630
64KB	51	***	2:05	2:05	649
256KB	37	***	2:05	2:05	713
1MB	36	***	2:05	2:05	905
4MB	35	***	2:05	2:05	905

compress					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	53378	0:21	2:24	2:22	8947
16KB	115	***	2:03	2:03	8957
64KB	48	***	2:03	2:02	8985
256KB	33	***	2:03	2:02	9129
1MB	29	***	2:04	2:03	9257
4MB	28	***	2:03	2:02	9769

vortex					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	156828	1:01	3:25	3:24	14571
16KB	81192	0:32	2:57	2:56	15083
64KB	41401	0:16	2:41	2:40	15427
256KB	14668	0:06	2:30	2:29	15603
1MB	242	***	2:23	2:22	15795
4MB	50	***	2:23	2:22	16563

Table 3: Results for apsi, compress, and vortex

VMbench Nastran					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	739179	4:38	9:56	9:53	11188
16KB	299314	1:56	6:40	6:37	12142
64KB	83808	0:33	4:56	4:54	13378
256KB	969	***	4:18	4:17	13730
1MB	112	***	4:18	4:16	13922
4MB	101	***	4:18	4:16	14434

VMbench ProE					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	230495	1:30	3:06	3:05	20557
16KB	129450	0:50	2:16	2:15	21543
64KB	65804	0:26	1:46	1:45	22563
256KB	16628	0:06	1:21	1:20	22627
1MB	95	***	1:13	1:12	22819
4MB	31	***	1:13	1:12	23587

VMbench Verilog					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	408487	2:39	5:22	5:16	53127
16KB	178919	1:10	3:42	3:39	53802
64KB	46225	0:18	2:43	2:41	53810
256KB	1407	***	2:21	2:19	53858
1MB	105	***	2:20	2:19	54050
4MB	88	***	2:20	2:19	54306

Table 4: Results for Nastran, ProE, and Verilog

performance improvement, because TLB misses are no longer a significant overhead. For the benchmark compress, with 4KB pagesize, the predominant component of TLB misses are due to dynamically-allocated (dynamic) data references. TLB misses are reduced with 16KB chatr pagesize. The increase in memory usage is negligible. With 16KB and larger chatr pagesizes, TLB miss overhead becomes insignificant. The benchmark vortex also benefits from dynamic data being mapped using large pagesizes. The TLB miss overhead is reduced gradually as the chatr pagesize is increased from 4KB to 1MB. The reduction in total time closely follows the reduction in the estimated TLB miss overhead.

As shown in Table 4, all three VMbench benchmarks benefit from using large pagesize mappings for dynamic data. All three benchmarks are trace driven, and a command-line parameter specifies the granularity of dynamic allocation (*malloc* size). The malloc size of 16MB is used in the Nastran benchmark, and hence large pagesizes can be exploited for the heap. The benchmark ProE on the other hand, makes malloc requests in small varying increments, and yet, benefits from large pages because of heap growth-rate adjustment discussed in Section 6. The benchmark Verilog uses a malloc size of 16KB. Nonetheless, it benefits from larger chatr pagesizes, also because of heap growth-rate adjustment.

Results from Cadence Design's Verilog-XL, a real world application, are presented in Table 5. Verilog-XL is a digital simulator that allows an engineer to test the logic of a design. Verilog-XL suffers from TLB misses primarily due to dynamic data references. The text and the initialized data sizes constitute less than 5% of the total memory usage.

Some interesting observations can be made from

chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB	Pagesize Distribution						
						4KB	16KB	64KB	256KB	1MB	4MB	16MB
4KB	601116	4:23	38:32	38:19	33818	33818						
16KB	317135	2:19	35:59	35:47	33924	1124	8200					
64KB	104463	0:46	33:33	33:21	33996	1044	14	2056				
256KB	23840	0:10	32:36	32:25	34193	1113	14	8	514			
1MB	6403	0:03	32:27	32:16	34362	1114	16	10	12	126		
4MB	3523	0:02	32:23	32:12	34641	1113	14	8	13	3	31	
16MB	2844	0:01	32:20	32:10	36666	1114	16	10	12	3	1	8

Table 5: Performance Results for Verilog-XL

Table 5. First, as the chatr pagesize is increased from 4KB to 256KB, the performance gain realized is higher than the gain from TLB miss reduction. One possible reason for the additional gain could be reduced number of virtual faults. Second, Verilog-XL does not show as dramatic a performance improvement as VMbench Verilog, the synthetic version. This lack of correlation is due to VMbench Verilog simulating only the memory reference behavior, and not the computation performed by the real application. Third, both Verilog-XL and VMbench Verilog realize most of the performance gain when 256KB pagesize is used. Fourth, even with large chatr pagesizes, over 1000 4KB mappings remain. Some of these mappings belong to shared libraries that were not chatr'ed to use large pagesizes.

Table 5 demonstrates the benefits of heap growth-rate adjustment in a real application. This technique facilitates the use of large pagesizes, in spite of the application making requests to grow the heap by small increments. Note that with heap growth-rate adjustment, the wasted heap allocation can be at most the chatr pagesize for program data. The 256KB pagesize offers most of the performance improvement for Verilog-XL. This pagesize improves performance by over 15% with only a 1% increase in memory usage.

10 Summary and Conclusion

We have implemented multiple pagesize support in HP-UX, using the existing 4KB pagesize based hardware dependent and hardware independent VM data structures. By using this "replication" based representation, we are not taking advantage of large pagesize based structures that are more efficient in space, and possibly more efficient in time. On the other hand, our approach entails no more overhead than when all mappings are 4KB pagesize; our primary goal was to reduce the TLB miss overhead. We use a buddy system allocator for allocating and freeing multiple pagesizes. Our implementation creates large pages at fault-service time for zero-filled memory, page-ins from secondary storage for UFS, NFS, and VxFS, page-ins from swap space, and copy-on-write. We have a page replacement module that handles multiple pagesizes.

Our performance measurements show that despite the seemingly high overhead of operating on 4KB based data structures, our approach to multiple pagesize support offers significant performance improvement for a variety of benchmarks and a real world application.

Acknowledgments

Thanks to Orran Krieger, the shepherd for our paper, for his detailed and insightful comments that helped improve the presentation. We are grateful to the anonymous referees, Anurag Acharya, and Joe Barrera for their comments on the paper. We appreciate the feedback on an early version of the paper, from Eric Hamilton, Duncan Missimer, and Hal Prince. Thanks to Venkatesh Radhakrishnan for his comments on a near-final version. Jun Su helped us with measuring the performance of Verilog-XL.

Bill Taylor was the project manager. Eeman Wong wrote the functional and reliability tests. Jyothy Reddy contributed to the initial version of the pageout path to handle multiple pagesizes. Cliff Mather implemented kernel support for pagesize hints, changes to HDL functions, and added hooks for collecting large page statistics. Kurt Peterson performed design reviews and code reviews for the project, and implemented the final version of the pageout path. Balakrishna Raghunath implemented pdir support, and the allocator. Indira Subramanian investigated alternatives for multiple pagesize support, implemented the fault-path, and wrote this paper.

References

[1] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., 1986.

[2] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 243–253, November 1994.

[3] P. Bannon and J. Keller. Internal Architecture of Alpha 21164 Microprocessor. In *Compcon Digest of Papers*, pages 79–87, March 1995.

[4] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 114–123, May 1992.

[5] D. Comer. The Ubiquitous Btree. *ACM Computing Surveys*, pages 121–137, June 1979.

[6] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, pages 323–333, May 1968.

[7] D. Greenley et al. UltraSPARC: The Next Generation Superscalar 64-bit SPARC. In *Compcon Digest of Papers*, pages 442–451, March 1995.

[8] J. Huck and J. Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 39–50, May 1993.

[9] T. Kagimasa, K. Takahashi, and T. Mori. Adaptive Storage Management for Very Large Virtual/Real Storage Systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, pages 372–379, May 1991.

[10] G. Kane. *PA-RISC 2.0 Architecture.* Prentice-Hall, Inc., 1996.

[11] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. Virtual Memory Support for Multiple Page Sizes. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS)*, pages 104–109, October 1993.

[12] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–247, June 1986.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, third edition, 1997.

[14] R. B. Lee. Precision Architecture. *IEEE Computer*, pages 78–91, January 1989.

[15] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[16] *MIPS R10000 Microprocessor User's Manual, Version 2.0.* MIPS Technologies, Inc., 1996.

[17] J. C. Mogul. Big Memories on Desktop. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS)*, pages 110–115, October 1993.

[18] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 176–187, June 1995.

[19] SPEC. SPEC Newsletter, September 1995.

[20] M. Talluri. Use of Superpages and Subblocking in the Address Translation Hierarchy. Ph.D. Thesis, University of Wisconsin-Madison Computer Sciences, August 1995.

[21] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 171–182, October 1994.

[22] M. Talluri, M. D. Hill, and Y. A. Khalidi. A New Page Table for 64-bit Address Spaces. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 184–200, December 1995.

[23] M. Talluri, S. Kong, M. D. Hill, and D. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 415–424, May 1992.

[24] U. Vahalia. *UNIX Internals – The New Frontiers.* Prentice-Hall, Inc., 1996.