The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

General Purpose Operating System Support for Multiple Page Sizes

Narayanan Ganapathy and Curt Schimmel
Silicon Graphics Computer Systems, Inc.

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

General Purpose Operating System Support for Multiple Page Sizes

Narayanan Ganapathy
Curt Schimmel

Silicon Graphics Computer Systems, Inc.
Mountain View, CA
{nar, curt}@engr.sgi.com

Abstract

Many commercial microprocessor architectures support translation lookaside buffer (TLB) entries with multiple page sizes. This support can be used to substantially reduce the overhead introduced by TLB misses incurred when the processor runs an application with a large working set. Applications are currently not able to take advantage of this hardware feature because most commercial operating systems support only one page size. In this paper we present a design that provides general purpose operating system support that allows applications to use multiple page sizes. The paper describes why providing this support is not as simple as it first seems. If not designed carefully, adding this support will require significant modifications and add a lot of overhead to the operating system. The paper shows how our approach simplifies the design without sacrificing functionality and performance. In addition, applications need not be modified to make use of this feature. The design has been implemented on IRIX 6.4 operating system running on the SGI Origin platform. We include some performance results at the end to show how our design allows applications to take advantage of large pages to gain performance improvements.

1 Introduction

Most modern microprocessor architectures support demand paged virtual memory management. In such architectures, a process address space is mapped to physical memory in terms of fixed size *pages* and an address translation is performed by the processor to convert the virtual memory address to a physical memory address. The translation is done by traversing a *page table*. The processor performs the virtual to physical address translation on every memory access and hence, minimizing the translation time is of great importance. This is especially true for processors with physically tagged caches that reside on the chip because the address translation has to be completed before data can be retrieved from the cache. (A thorough treatment of caches on uniprocessor and multiprocessor systems can be found in [Schim94]). To achieve this goal, most microprocessors store their recently accessed translations in a buffer on the chip. This buffer is called the *translation lookaside buffer* (TLB). If a translation is not found in the TLB, the processor takes a *TLB miss*. Some processors like the i860 have built-in hardware to walk the page tables to find the missing translation, while others like the MIPS R10000 generate an exception and a TLB miss handler provided by the operating system loads the TLB entries.

The number of entries in a TLB multiplied by the page size is defined as ***TLB reach***. The TLB reach is critical to the performance of an application. If the TLB reach is not enough to cover the *working set* [Denn70] of a process, the process may spend a significant portion of its time satisfying TLB misses. The working set size varies from application to application. For example, a well tuned scientific application can have a small working set of a few kilobytes, while a large database application can have a huge working set running into several gigabytes. While the working set size can be reduced to a certain extent by tuning the application, this is not always possible, practical or portable. Hence, if a system has to perform well running a variety of applications, it has to have good TLB reach. Modern processor caches are getting large (> 4MB) and few microprocessors have a TLB reach larger than the secondary cache when using conventional page sizes. If the TLB reach is smaller than the cache, the processor will get TLB misses while accessing data from the cache, which increases the latency to memory.

One way to increase the TLB reach is to increase the page size. Many microprocessors like the MIPS R10000 [Mips94], Ultrasparc II [Sparc97] and PA8000 [PA-RISC] now use this approach. By supporting selectable page sizes per TLB entry, the TLB reach can be increased based on the application's working set. For example, the MIPS R10000 processor TLB supports 4K, 16K, 64K, 256K, 1M, 4M and 16M page sizes per

TLB entry, which means the TLB reach ranges from 512K if all entries use 4K pages to 2G if all entries use 16M pages. As a different page size can be chosen for each TLB entry, a single process can selectively map pages of different sizes into its address space. The operating system can choose pages of larger sizes for an application based on its working set.

While many processors support multiple page sizes, few operating systems make full use of this feature because providing such a feature has its challenges. Current operating systems usually use a fixed page size. The virtual memory subsystem is highly dependent on the fact that the page size is constant. For convenience, let us define this page size as the *base page size* and define *base pages* to be pages of base page size. Let us also define pages larger than the base page size to be *large pages*. The functions of the VM subsystem like allocating virtual addresses, memory read/write access protections, file I/O, the physical memory manager that allocates memory, the paging process, the page tables, etc., assume that there is only one page size which is the base page size. Additionally, many VM and file system policies like the page replacement policy, read ahead policy, etc., also assume a fixed page size. The file system manages all its in-core data in pages of the base page size. As we shall see later, modifying the file system and the I/O subsystem to support multiple page sizes is a formidable task. The design is further complicated by the fact that many processes typically share physical memory when they map files using the UNIX `mmap()` system call or when they use shared libraries. This means that two processes might share the same physical memory but may need to map them with different page sizes. To complicate matters further, the processor also adds several restrictions to the use of this feature.

Due to the invasive nature of the operating system modifications needed to support them, large pages have been put to limited use. They have only been used in special applications to map frame buffers, database shared memory segments, and so on.

General purpose support for multiple page sizes would go a long way in improving the performance of a substantial number of applications that have large working sets. In this paper we present a design that provides general purpose operating system support for multiple page sizes. An important characteristic of the design is that it adds no overhead to common operations and there is no performance penalty for applications when not using large pages. Another important feature of this design is that the knowledge of various page sizes is restricted to a small part of the VM subsystem. The file system, the I/O system and other subsystems are not aware of multiple page sizes. More importantly, large pages are transparent to applications and they need not be rewritten to take advantage of large pages.

The design has been implemented in SGI IRIX 6.4 running on the SGI Origin 2000 platform which uses the MIPS R10000 microprocessor. The following sections concentrate on the details of MIPS R10000 and IRIX 6.4, but the techniques presented here can be applied to most commercial processors and operating systems. Unless specified explicitly, the base page size is assumed to be 4K bytes.

Section 2 describes some related work in the area. Section 3 explains the microprocessor support for multiple page sizes, specifically describing the MIPS R10000 TLB. Section 4 describes the current IRIX VM architecture. Section 5 describes our design goals. Section 6 describes in detail our design and section 7 shows performance results for some sample benchmarks.

2 Related Work

The paper by Khalidi et. al [Khal93] studies the issues involved with providing multiple page size support in commercial operating systems. Although the study finds that such a support is non-trivial to implement, it does not propose any methods or algorithms to help provide this feature. The paper by Talluri et. al [Tall94] describes a new hardware TLB architecture that considerably reduces the operating system support. It proposes a new feature called *sub-blocking* and goes on to show how a sub-blocking TLB simplifies the operating system work involved while providing performance benefits similar to those provided by large pages. Sub-blocking is not yet provided by popular commercial microprocessors. The paper by Romer et. al [Romer95] describes different ***on-line page size promotion*** policies and how effective they are in reducing the TLB miss cost. For the policies to be effective, the operating system should maintain TLB miss data per large page. The cost of collecting the data in our implementation is significantly higher than what is mentioned in the paper. In addition, one of our important goals is to not penalize applications that do not use large pages until we fully understand the practical benefits of using large pages on commercial applications. Collecting the TLB miss data to do page promotion will affect the performance penalty of all processes and does not meet our goal. We are planning to do on-line page promotion in one of our future releases.

3 Microprocessor Support

The MIPS R10000 processor [Mips94] TLB supports 4K, 16K, 64K, 256K, 1M, 4M and 16M page sizes. Figure 1 shows a sample TLB entry. The MIPS R10000 TLB has 64 entries. Each TLB entry has two *sub-entries*. The subentries are related to one another in that they map adjacent virtual pages. Both pages must be of the same size. Thus, each TLB entry maps a virtual address range that spans twice the chosen page size.

The TLB entry also carries a *TLB PID* that identifies the process to which the TLB entry belongs. On a TLB miss, the processor generates an exception. The exception invokes the software TLB miss handler which inserts the appropriate TLB entry. The software TLB miss handler is part of the operating system.

Like most microprocessors, the R10000 TLB restricts the alignment of the virtual addresses and physical addresses that can be mapped by a large page to the large page size boundary. For example, a 64KB page can be mapped only to a virtual address aligned to a 64KB boundary. The physical address of the large page must also be aligned to a 64KB boundary. The virtual address that maps the first of the two sub-entries in the TLB, called the even sub-entry, must be aligned to a *2*pagesize* boundary. The virtual address range covered by the large page must have the same protections and cache attributes if the entire range is to be mapped by one TLB entry.

4 Basic VM Structure

Almost all commercial operating systems provide virtual memory support [Cox94]. Most use pages of only one page size and all major subsystems manage their memory in pages of this size. These include the memory management subsystem that maps virtual to physical memory for a given process, the file system, the I/O subsystem and the machine dependent portion of the operating system kernel. The IRIX VM subsystem has two main components:

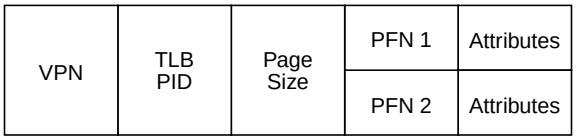


Figure 1: MIPS R10000 TLB Entry Format.

A *physical memory manager* that manages memory in terms of page frames of a fixed size. Each physical page frame in the system is represented by a kernel data structure, called the page frame data structure or the *pfdat*. The physical memory manager provides methods for other systems to allocate and free pages.

A *virtual memory manager* manages the process's address space. An address space consists of several segments called *regions*. Each region maps a virtual address range within the address space. The regions are of different types. For example, the *text* region maps a process's text and the *stack* region maps the process stack. Each region is associated with a particular memory object. For example the text region's memory object is the file containing the process text. The data for the text region is read from the file into physical memory pages and they are mapped into the process address space. Each memory object has a cache of its data in physical memory. The cache is a list of pfdats that represent the physical pages containing that object's data. For this reason the cache is called the *page cache*. Thus the physical pages containing a file's data are kept in the file's page cache. A memory object can be shared across multiple processes. The file system also performs its read and write requests in terms of pages. The device drivers do their I/O in terms of pages.

The virtual address space is mapped to physical page frames via page tables. There are numerous choices for the page table formats and several have been discussed in the literature [Tall95]. The page table format for IRIX is a simple forward mapped page table with three levels. Each page table entry (PTE) maps a virtual page to a physical page. The PTE contains the page frame number (PFN) of the physical page that maps the virtual page number. It also contains bits that specify attributes for the page including the caching policy, access protections and of course, the page size.

The virtual memory manager supports *demand paging*. The procedure of allocating the page and initializing it with data from a file or the swap device is called *page faulting*.

5 Design Goals

The overall goal is to provide general purpose multiple page size support. This includes dynamic allocation of large pages, faulting them in to a process address space, paging them out, and upgrading and downgrading page sizes. Our emphasis is to make the design simple and practical without compromising functionality and performance.

Applications should be able to use large pages without any significant modifications to their code. System calls which depend on the page size should behave the same way when using large pages as they would when they use base pages.

Application performance should degrade gracefully under heavy memory load. They should at least perform as well as they would when not using large pages.

As indicated in the previous sections, blindly making the extensions to all the OS subsystems that use the knowledge of page size is an enormous task. It is not only difficult to implement but also introduces performance penalties in many key parts of the operating system and slows down applications that do not use different page sizes. It is important to understand that large pages only benefit applications with large working sets and poor locality of reference. Not all applications will need large pages. One of the fundamental goals of the design is to not penalize the performance of applications that do not use large pages. Another goal is to restrict the extensions needed to as small a set of OS subsystems as possible.

6 Our Design

This section describes our approach in detail. We decided to retain the original format of pfdat and PTE data structures. As we see later, these two choices are crucial to the success of our design. The remaining sections describe the TLB miss handler, the large page allocator, the page fault handler, the page replacement algorithm, the algorithms to upgrade and downgrade page sizes and the policies we use with large pages.

6.1. Pfdat Structure

One of the first design decisions to be made is how to handle the pfdat structures for large pages. A simple-minded approach might be to make a pfdat represent a large page frame by adding a page size field to the pfdat. But as we will soon see, this approach is fraught with problems.

The pfdat structure is a very basic data structure and is used widely by the VM, file system and the I/O subsystem. It is the representation of a physical page frame and tracks the state changes that happen to a page. Changes to the pfdat structure will mean changes to all the subsystems that use the data structure. Traditionally pfdats are an array of small structures. Hence the conversion functions from physical page frame numbers to pfdats and vice versa are simple and fast. If pfdats represent pages of different sizes the conversion functions become more complex.

There is an impact on the page cache data structure as well. The data structure is usually a hash table and the pfdats are hashed into the table by a hash function that takes the memory object address and the offset into the memory object. The hash function works well if the pfdats are of fixed size since the pfdats will be distributed evenly across the hash buckets. If pfdats represent pages of different sizes, the simple hash functions do not work very well. A more complicated hash function will slow down the search algorithms that are crucial to system performance.

Another problem is that processes sharing a set of physical pages would have to be able to map these pages with different page sizes. For example, suppose one pfdat represents a large page and it is shared by a set of processes. If one of the processes decides to unmap part of its virtual address space mapped by the large page, the large page would have to be split into smaller pages. In addition, the page table entries in all the processes that map the large page would have to be downgraded as well. If the large page belongs to a heavily shared memory object like a library page, the downgrading operation would incur a significant performance penalty.

From the preceding discussion, it is clear that the disadvantages of having pfdats represent page frames of different sizes are too great. Our design therefore chooses to retain the original structure for pfdats i.e., pfdats represent pages of a fixed size (the base page size). Large pages are treated as a set of base pages. Thus if the base page size is 4K and we use a 64K page to map a segment of a file, there will be sixteen pfdats in the page cache to map that segment.

This essentially means that most subsystems that interact with the pfdats do not have to know about large pages. It enables us to avoid rewriting many parts of the operating system kernel that deal with lists of pfdats. These include the file system, the buffer cache, the I/O subsystem and the various device drivers, considerably simplifying the complexity of providing multiple page size support. For example, if the contents of a 64K large page has to be written to the disk, the device driver is passed a buffer that contains 16 4K pfdats corresponding to the large page. The device driver does not know it's a large page and handles the buffer just like it would handle any other buffer that has a list of pages. Of

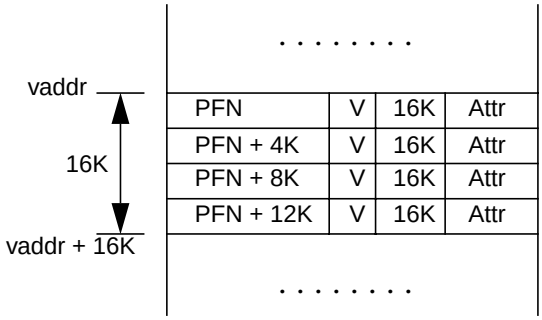


Figure 2: Multiple Page Table Entry Format.

course for the large page case, the device driver can be smart enough to recognize that the pages are contiguous and can make a single DMA request.

This decision also means that the lowest page size we can ever use in the system will be the base page size. In practice, this is not such a severe restriction as the base page size is usually the smallest page size supported by the processor. In the next section, we see how a set of pfdats are treated as a large page even though the pfdats themselves do not carry any page size information.

6.2. Page Table Entry Format for Large Pages

The next important decision we need to make is regarding the format of the page table. The page table entries (PTEs) need to contain the page size information so that the TLB miss handler can load the right page size into the TLB entry.

We decided to retain the existing page table layout. At the page table level we have one PTE for each base page of the large page. Figure 2 shows how the PTEs would look for a 16K large page with a 4K base page size.

In this format, we use all the PTEs that span the large page range. The PTEs that map large pages look similar to those that map base pages. For the large page PTEs, the page frame numbers will be contiguous as they all belong to the large page. In addition, all PTEs have a field that contains the page size. The TLB miss handler needs to look at only one PTE to get all the information needed to drop in the TLB entry. As the information is present in each PTE, the handler traverses the page table just like it does for the base page size case. It finds the appropriate PTE and drops the contents into the TLB. As such, the large page TLB miss handler performance is comparable to the performance of the base page size TLB miss handler (described in section 6.3). This approach also helps while upgrading or downgrading the page size. Only the page size field in the PTE needs to be changed.

One important benefit to this format is that the subsystems that deal with PTEs need not be modified as the PTE format remains the same. There is an additional benefit to keeping the page size information only in the PTEs. It *allows different processes to map the same large page with different page sizes*. For example, suppose two processes memory map the same file into their address space. In this case they have to share the same set of physical pages. Let us also assume that one process maps the first 64K bytes of the file while the other maps the first 48K bytes. The first process may choose to map the file into a 64K page while the other process may choose to use three 16K pages. Both processes share the same physical pages but map them with different sizes. The page size choices one process makes do not affect others sharing the same file. Likewise, downgrading this virtual address range for the first process is completely transparent to the second process. This is possible because the page size information is only kept in the PTEs and hence private to a process.

This approach is not without its disadvantages. It does not reduce the size of the page tables required if one uses large pages. All the attributes and permissions in the set of PTEs that encompass the large page have to be kept consistent, although the number of cases where we need to maintain this consistency is limited.

6.3. Software TLB Miss Handler

On MIPS processors, a TLB miss generates an exception. The exception is handled by a TLB miss handler. It traverses the page table for the given virtual address and drops in the contents of the PTE into the TLB. The TLB miss handler has to be very efficient and is usually only a few instructions long. Otherwise the time spent handling TLB miss exceptions can be a significant fraction of the total runtime of the application. The format of the PTE directly affects the performance of the TLB miss handler. The more the PTE looks like a TLB entry, the faster the TLB miss handler can drop the entry into the TLB. As explained below, a TLB miss handler that supports multiple page sizes has more overhead compared to a single page size TLB miss handler because the MIPS processor has a separate page mask register [Mips94]. On a single page size system, the page mask register is set once at system initialization time to the fixed page size. On such systems, the TLB miss handler does not modify the page mask register. To support multiple page sizes, the TLB miss handler

has to set the page mask register in addition to other entries during each TLB miss. So the single page size TLB miss handler is much cheaper compared to the multiple page size TLB miss handler. This means that if we were to use the multiple page size TLB miss handler for all processes in the system, then processes that did not use large pages would have a performance impact. To avoid this problem we use a feature of IRIX that allows us to ***configure a TLB miss handler per-process***. Thus only processes that need large pages use the multiple page size TLB miss handler. All other processes use the fast single page size TLB miss handler. This supports one of our main goals: that processes not using the large page feature are not burdened with additional overhead.

6.4. Large Page allocator

This section discusses issues related to the allocation of large pages. It describes a mechanism called page migration to unfragment memory and also describes a kernel thread called the coalescing daemon that uses page migration to coalesce large pages.

6.4.1. Page allocation issues

Traditionally in single page size systems, pfdats of free pages are kept in linked lists called *free lists*. Allocating and freeing pages is done by simple removal and insertion of pfdats into the free lists. To be able to allocate pages of different sizes at runtime, the physical memory manager should manage memory in variable size chunks and handle external fragmentation. In busy systems, the pattern of allocating and freeing pages causes the free memory to be so fragmented that it is difficult to find a set of contiguous free pages that can be coalesced to form a large page. This problem of allocating chunks of different sizes has been very well studied in literature. The problem can be divided into two parts. One is to minimize fragmentation while allocating pages and the other is to provide mechanisms to unfragment memory.

We have designed an algorithm that keeps the overhead of the allocation and freeing procedures to a minimum and leaves the work of unfragmenting memory to a background kernel thread. This allows better control of the physical memory manager. For example, if an administrator chooses not to configure large pages, the background thread will not run and there will not be any additional overhead compared to a system with one base page size. Alternatively, the administrator can configure large pages and make the background thread aggressively unfragment memory.

To minimize fragmentation, the manager keeps free pages of different sizes on different free lists, one for each size. The allocation algorithm first tries to allocate pages from the free list for the requested page size. If a free page cannot be found, the algorithm tries to split a page of the next higher size.

The manager also uses a *bitmap* to keep track of free pages. For every base sized page in the system, there is a bit in the bitmap. The bit is set if the page is free and is cleared if the page is not free. This helps in coalescing a set of adjacent pages to form a large page as we can determine if they are free by scanning the bitmap for a sequence of bits that are set. When a page is freed, the bitmap is quickly scanned to see if a large page can be formed and if so pages corresponding to the bits in the bitmaps are removed from their free lists and the first pfdat of the newly formed large page is inserted into the large page free list. Note that the manager must also ensure that the processor alignment restrictions are met (the physical address of the large page should be aligned to the page size boundary) when coalescing the pages.

This algorithm uses high watermarks to limit the coalescing activity. The watermarks provide a high degree of control over the allocator and can be changed by the system administrator, even while the system is running. The high watermarks provide upper limits on the number of free pages of a given page size. Coalescing activity stops once the high watermarks have been reached.

6.4.2. Page Migration

As mentioned earlier, on a long running system a mechanism to *unfragment* memory is needed, since pages can be randomly allocated to the kernel, the file system buffer cache and to user processes. It is very difficult in these cases to find a set of adjacent free pages to form a large page even though there is a lot of free memory left in the system. To solve this problem we use a mechanism called ***page migration***. Page migration transfers the identity and contents of a physical page to another and can be used to create enough adjacent free pages to form a large page.

For example, in Figure 3, assume a 16K chunk of memory (A) has four adjacent 4K pages, three of which are free and one (A4) is allocated to a process. By transferring the contents of the busy page A4 to a free page B1 from another chunk B, we can free chunk A completely and thus use it to form a 16K large page.

The page migration algorithm that replaces page A4 with page B1 works as follows.

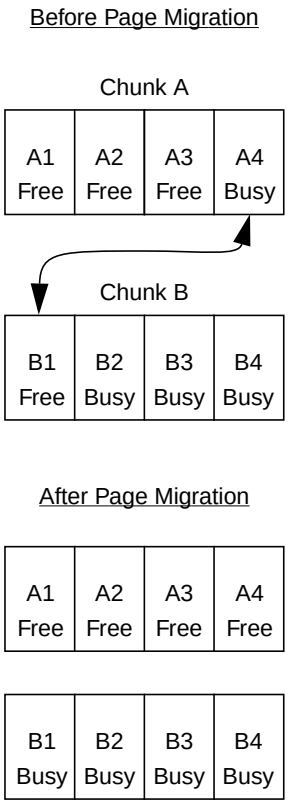


Figure 3: Page Migration Example.

a) It checks to see if the page A4 can be replaced. Pages that are held by the kernel for DMA or other purposes cannot be replaced. Pages that are locked in memory using the `mlock()` system call are also not replaced.

b) The next step is to get rid of any file system buffer references to page A4.

c) The page tables entries that map the page A4 are modified to point to page B1. The PTE valid bits are turned off and a TLB flush is done to force the processes that map page A4 to take a page fault. The pfdat of page B1 is marked to indicate that its data is in transit. This forces the processes to wait for page B1 to be ready. PTEs that map page A4 are found using its *reverse map* structure for page A4. The reverse map structure is part of the pfdat and contains a list of PTEs that map that page. At the end, the reverse map structure itself is transferred to page B1.

d) Page A4 is removed from the page cache and page B1 is inserted in its place.

e) Page A4's contents are copied to page B1. Page B1 is marked ready and the processes (if any) that were sleeping waiting for page B1 to be ready are woken up.

In practice a set of pages are migrated together to minimize the number of global TLB flushes.

6.4.3. Coalescing Daemon

The mechanisms to unfragment memory is implemented by a background thread called the ***coalesce daemon***. The daemon scans memory chunks of different sizes. It goes through several passes each with a different *level of aggressiveness*. There are three levels. In the first level or the *weak* level, it does not do page migration. The daemon cycles through all the pages looking for free contiguous pages and tries to coalesce them into a large page. In the second level or the *mild* level, it limits the number of page migrations per chunk. The daemon examines each chunk and computes the number of page migrations needed to make that chunk into a large page. The number of page migrations should be equal to the number of clear bits in the bitmap for that chunk. If the number is below a *threshold* the daemon proceeds to migrate that page. There is a threshold for every page size. The thresholds have been chosen by experimentation. The third or the *strong* level uses page migration very aggressively. For every chunk, the daemon tries to migrate all the pages that are not free. For most of the passes the daemon uses the first two levels. If after several passes the daemon is still not able to meet the high watermark it uses the strong level.

If a process wants to allocate a large page but the request cannot be satisfied immediately, it can choose to wait (by setting a special policy as described in the next section). If a process is waiting for a large page, the coalesce daemon uses the strong level of aggressiveness. This allows the daemon to quickly respond to the process's large page needs.

6.4.4. Minimizing Fragmentation

There are limitations to the page migration technique. Pages that have been allocated to the kernel or have been locked in memory cannot be migrated. The reason is that pages allocated to the kernel are generally not mapped via page tables and the TLB. For example, IRIX kernel memory is directly mapped through the K0SEG segment of the virtual address space of the R10000. Such pages cannot be migrated since many kernel subsystems have direct pointers to such pages. Pages that have been locked for DMA or locked using the `mlock(2)` system call cannot be migrated either,

since the callers assume that the underlying physical page does not change. Consequently the coalescing daemon performs well if most chunks contain pages that can be migrated. This is the case with many supercomputing applications which have large data segments whose pages can be migrated easily. In such systems the percentage of memory taken up by the kernel is very low and hence most of the memory is migratable. This is not true for desktop workstations running graphics applications. They usually have a limited amount of physical memory and a significant percentage of it is used by the kernel or locked for doing DMA. For these machines, we have extended the physical memory allocator to keep track of migratable pages. The physical memory manager is notified at the time of page allocation whether the page will be migratable. The manager keeps track of the number of migratable pages per chunk. If the manager wants to allocate a page for the kernel, it chooses a page from the chunk which has the least number of migratable pages. On the other hand, if the manager wants to allocate a migratable page it chooses a page from the chunk with the most number of migratable pages. This algorithm has the advantage of maximizing the number of chunks whose pages can be migrated although it adds overhead to the page allocating and freeing algorithms by keeping track of a list of chunks sorted by the number of migratable pages per chunk. This algorithm is used in low end workstations, where the advantage of being able to allocate large pages under conditions of severe fragmentation outweigh the disadvantage of the overhead incurred while allocating and freeing pages.

6.5. Policies Governing Page Sizes

IRIX provides a facility by which application can choose specific VM policies to govern each part of its address space. It provides system calls [IRIX96] to create policies and attach them to a virtual address range of a given process. The `pm_create(2)` system call allows an application or a library to create a policy module descriptor. The system call takes several arguments which specify the policy as well as some parameters needed by the policy. The `pm_attach(2)` system call allows an application to attach a policy module descriptor to a virtual address range.

The application can choose among a variety of policies provided by the kernel. Policies fall into several categories. The *page allocation* policies specify how a physical page should be allocated for a given virtual address. For example on a NUMA system, the application can specify that pages for a given virtual address range should be allocated from a node passed as a parameter to the policy. Another policy decides whether allocating a page of the right cache color takes precedence over allocating a page from the closest node. The *migration policies* allow the application to specify whether the pages in a given virtual address range can be migrated to another NUMA node if there are too many remote references to that page. It can also be used to set the threshold that triggers the migration.

Page size hints for a given virtual range can be specified via a policy parameter. `pm_create()` takes the page size as an argument. Thus the application or a runtime library can specify which page size to use for a given virtual address range. The kernel tries to allocate a page of the specified page size as described in the next section. The page size parameter works in conjunction with other page allocation policies. For example, the page allocation policy can specify that a page has to be allocated from a specific NUMA node and that it should be a large page. There are currently two large page specific policies:

a) On NUMA systems, if large pages are not available on an application's home node, the kernel can either borrow large pages from adjacent nodes or use lower sized pages on the home node. By default the kernel borrows a large page from an adjacent node but the application can indicate that locality is more important using this policy.

b) If a large page of the requested size is not available, the application can wait for the coalesce daemon to run and coalesce a large page. Sometimes the wait time is not acceptable and in that case the application can choose to use a lower page size (even the base page size). The application can specify if it wants to wait for a large page and also the time period for which it wants to wait before using a lower page size. The coalescing daemon runs aggressively when a process is waiting for large pages.

These policies will be refined and new ones will be added as we learn more about applications and how they behave with large pages. In the future, we want to be able to automatically detect TLB miss rates using the performance counters provided by the processor and upgrade a virtual address range of a process to a specific page size. We decided not to do on-line page size upgrade initially as it was not clear the performance benefits would outweigh the cost of collecting the TLB miss data and the page size upgrade. The kernel always uses the base page size unless the application or a runtime library overrides the default.

Currently the page size hints are provided by the application or special runtime libraries like the parallelizing Fortran compiler runtime, MPI runtime, etc. These runtime libraries usually have a better understanding of the application's behavior and can monitor the performance of the application.

6.5.1. Tools to Specify Page Size Hints Without Modifying Binaries

Applications need not be modified or recompiled to be able to use large pages. IRIX has a tool called `dplace(1)` [IRIX96] that can be used to specify policies for a given application without modifying the application. The tool inserts a dynamic library that gets invoked before the application starts. The library reads configuration files or environment variables and uses them to set up the page size and NUMA placement policies for a given virtual address range. For example the following command,

```
dplace -data_pagesize 64k ./a.out
```

sets the policies for the address space of a.out such that its data section uses 64K page size.

IRIX provides a valuable tool called `perfex(1)` [Marco96] that allows a user to measure the number of TLB misses incurred by a process. It gets the data from the performance counters built into the R10000 processor.

Another tool called `dprof(1)` analyzes memory reference patterns of a program and can be used to determine which parts of an application's address space encounters the most TLB misses and hence will benefit from large pages.

The parallelizing fortran compiler runtime also takes page size hints via environment variables and can be used to set up page size hints for applications that have been compiled with that library.

6.6. Page Faulting

The page fault handler is invoked whenever a reference is made to a virtual address for which the PTE is not marked as valid. The page fault handler allocates and initializes a page and sets up the PTE for the virtual address that generated the TLB miss. Initializing the page can involve finding the page in the page cache for the memory object, reading in the data from a disk file or a swap device, or just zeroing the page. Page faults usually happen when a process accesses the virtual address for the first time as the page tables will not yet be initialized. The fault handler's functionality must be extended to handle large pages. In particular, the handler should be able to allocate and initialize large pages, ensure that the processor restrictions are not violated and set up the PTEs.

Adding these extensions to the handler is simplified by our choice of data structures for the pfdats and the PTEs. The extensions are minimal and the majority of the fault handler remains the same as the base page size fault handler. The large page fault handler is written as a layer on top of the standard page fault handler for base pages. The fault handler is described below. Let us assume that the virtual address is `va`.

a) The first step is to consult the policy module (described above) to determine the page size that should be mapped. The virtual address is then aligned to this page size. Let us call the aligned virtual address `ava`.

b) The next step is to verify that the virtual address range `[ava, ava + page_size)` has the same protections and other attributes. As discussed earlier, since there is only one TLB entry per large page, the protections and the attributes have to be the same for the entire virtual address range mapped by the TLB entry. It also verifies that no pages of a different size are already present in the range `[ava, ava + page_size)`.

c) The handler verifies that both sub-entries in a TLB entry that maps this address range have the same page size. This satisfies the MIPS R10000 processor restriction.

d) The handler then allocates a large page. If a large page cannot be allocated and the process has chosen to wait (specified via a policy) for a large page, it waits for a large page. If a large page of the required page size can be obtained it retries the algorithm. If after a timeout period it cannot get a large page or the process chose not to wait, the handler retries the algorithm with a lower page size.

e) The handler enters a loop and faults each base page in the large page one at a time. Thus for a 64K page, the handler has to loop 16 times. The fault algorithm is identical to the base page fault algorithm. This is where the page is initialized, read from a file, swapped in from disk, etc. We can do this because the large page pfdats and PTEs look identical to those of base page pfdats and PTEs.

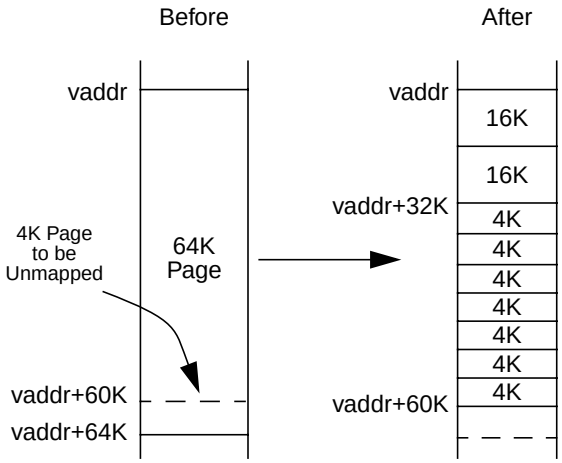


Figure 4: Downgrading a 64K Page.

f) Once the pages have been faulted, the page size field is set in the PTEs and they are made valid. Note that a large page is mapped by many adjacent PTEs. The handler also drops in a TLB entry for `va`.

6.7. Downgrade Mechanism

The motivation behind downgrading comes from system calls whose behavior is highly tied to the base page size. UNIX system calls like `mmap()`, `munmap()` and `mprotect()` work on virtual address ranges at the granularity of the base page size. If we map a large page to a virtual address range and the application makes a system call to change the protections for part of the address range, we can either return a failure to the system call or downgrade the large page to a lower size small enough to enforce the protection criteria. Downgrading allows the operating system to not change the system API and ABI, while internally using different page sizes. This is a huge advantage as it allows applications to run unmodified.

The downgrade algorithm allows us to have no restrictions on which parts of the address space can be mapped with large pages. For example, there is no restriction that one region or segment of the address space should be of the same page size. Pages of different sizes can be stacked one on top of the other.

Figure 4 shows an example where a large page is downgraded automatically by the kernel to satisfy an `munmap()` system call. The virtual address range `[vaddr, vaddr+64K)` has been mapped with a 64K page. Suppose the application wants to unmap the last 4K bytes of that address range (as indicated in Figure 4) using the `munmap()` system call. In this case, we have to downgrade the 64K page into smaller page sizes. The downgrade algorithm also has to obey the R10000 processor restriction that page sizes of the pages at the even and odd address boundaries must be the same. For example, since `vaddr` is guaranteed to be aligned at a 64K boundary, `vaddr` is also aligned at an even 16K boundary. The downgrade algorithm downgrades the page size of the PTEs mapping the range `[vaddr, vaddr+32K)` to 16K. The address range `[vaddr+32K, vaddr+64K-16K)` cannot be mapped by two 16K pages as the range is not big enough. Nor can a single 16K page be used in the range `[vaddr+32K, vaddr+32K+16K)` because of the hardware restriction that even and odd pages must be the same size. So the algorithm downgrades the range to the next lower page size (4K) which happens to the base page size.

The downgrade algorithm is quite simple. It first clears the valid bit of all the PTEs that map the large page and then flushes the TLB. The TLB flush only flushes those TLB entries which map the given virtual address range of the process. The invalidation is necessary to make the update of the page size atomic. Once the TLB has been flushed, the page size fields in all the PTEs are updated to reflect the new page size. The new page size is the next lower page size to which the address range can be downgraded without violating the MIPS processor restrictions.

In practice, downgrades rarely happen for large supercomputing and database applications, as they do not spend much time mapping and unmapping their address spaces.

6.8. Upgrade Mechanism

An upgrading mechanism is useful to dynamically increase the page size of an application based on feedback from other parts of the operating system. Processors like the MIPS R10000, provide counters that track the TLB misses incurred by a process. This counter can provide feedback to the VM manager to upgrade the page size. As said earlier, we have decided not to do on-the-fly page upgrade based on feedback from the process's TLB miss profile. Currently an application or the runtime can advise the operating system to upgrade the page size for a given virtual address range that corresponds to a key data structure using a new command to the `madvise()` system call.

The system call is particularly useful for upgrading text pages. Consider the case where an executable is invoked and the base page size is used. The text file cor-

responding to the executable is faulted into base pages and the pages become part of the file's page cache. If the same executable is now run a second time with large page size hints, it is quite unlikely that the kernel will be able to use large pages as the pages for the text are already in the page cache and hence it has to reuse them. In this situation the application can invoke the `madvise()` system call to upgrade its existing text pages to the large page size. The algorithm to upgrade a page from its old size to a new size is described below.

a) The first step clears the valid bit for the PTEs that span the large page and performs a TLB flush.

b) Next a large page of the requested size is allocated.

c) We use the page migration algorithm (described in section 6.4.2) to replace the old pages in the virtual address range with the large page.

d) The PTE size fields are updated to reflect the new size and they are validated.

6.9. Page Replacement Algorithm Extensions

Most page replacement algorithms (like the BSD clock algorithm [BSD89]) usually use reference bits in the PTEs to keep track of recently referenced pages for a given process. Pages which are not recently referenced are usually paged out to disk. The memory reference patterns are tracked for the entire large page. The default paging policy for large pages is simple. If part of a large page is chosen to be swapped out we downgrade the large page. We try to minimize downgrades and improve performance by swapping out entire large pages.

7 Performance

Figures 5 and 6 show the performance improvements obtained when using large pages on some standard benchmarks. The benchmarks were performed on an SGI Origin 2000 running IRIX 6.4. The base page size for that system is 16K. The benchmarks were run with 16K, 64K, 256K and 1M page sizes. *turb3d*, *vortex* and *tomcatv* are from the Spec 95 suite [Spec95]. *Fftpde* and *appsp* are from the NAS parallel suite. The benchmarks were not modified. All of them were single threaded runs. The same binaries were used for all the runs. Figure 5 shows the percentage reduction in the number of TLB misses and Figure 6 shows the percentage performance improvement at different page sizes

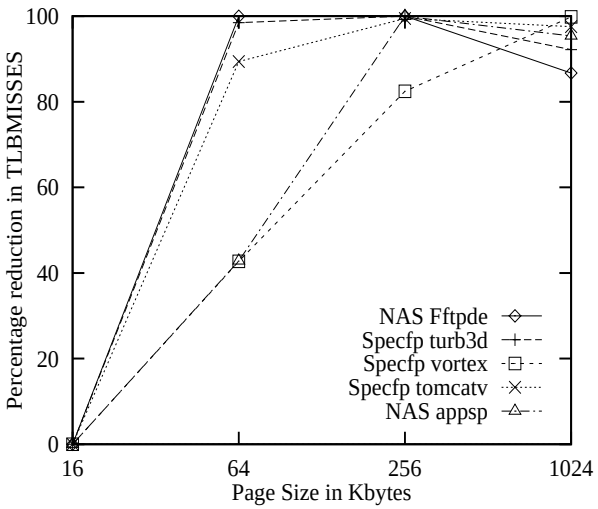


Figure 5: Percentage Reduction in TLB Misses with Large Pages.

with respect to the base page size (16K). The page size hints were given using `dplace(1)`. From Figure 6 we can see that some applications get 10 to 20% improvement with large pages. The improvements will be even greater for larger problem sizes.

The *TLB miss overhead* (defined as the ratio of the time spent handling TLB misses to the total runtime of the application) is a significant part of the runtime for *vortex*, *turb3d* and *appsp*. The overhead is minor for *tomcatv.* We can deduce this by looking at Figures 5

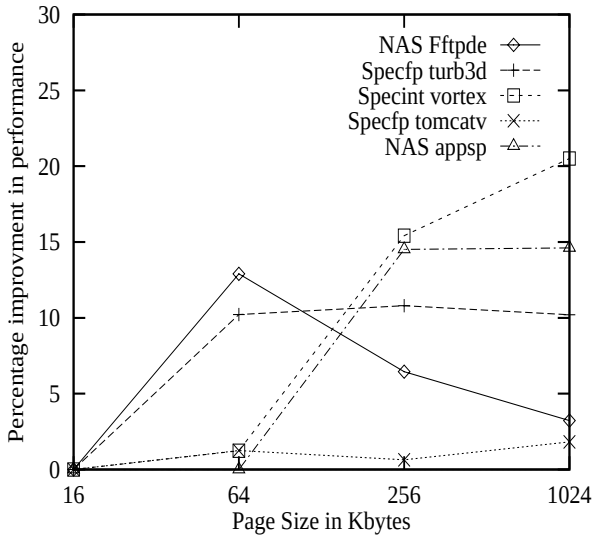


Figure 6: Percentage Improvement in Performance with Large Pages.

and 6. We can see that for *turb3d* and *vortex*, the performance improves significantly with reduction in TLB misses. For example, vortex gives a 20% improvement in performance when using 1MB pages. On the other hand although *tomcatv*'s TLB misses were significantly reduced with 64K pages (Figure 5), the performance improvement (Figure 6) is negligible for the same page size.

Once the TLB miss overhead has been reduced by about 98%, further increases in page sizes provide diminishing returns. For example, *turb3d* does not show much improvement beyond 64K pages. Some times at higher page sizes, the operating system cannot map large pages for the entire address space range due to alignment restrictions. So it may be forced to use base pages. This causes the performance to drop due to higher TLB misses. We can see this behavior with *fftpde* when using 1MB pages.

Figure 7 shows how large pages perform under contention. The experiment measures the performance of the *fftpde* NAS benchmark using 64K pages on a single CPU Origin 200 system with a variety of threads running in the background. Three kinds of background threads were used to simulate different TLB and cache usages. In one case the background thread is a simple spinner program. The program is an extremely small tight loop that is completely CPU bound and fits into the CPU's cache and TLB and should not cause too many TLB misses. In the second case, the background thread is another invocation of the *fftpde* program using 16K pages. This should cause considerable thrashing of the TLB and the cache. The third case also uses *fftpde* program as the background thread but this time with 64K pages. This should reduce the TLB thrashing as the working set can fit into fewer TLB entries. The performance improvement was measured with an increasing number of background threads. As we can see, the performance degrades with contention but the drop is not so steep if the background thread is a spinner or a *fftpde* program with 64K pages. When the background thread is an *fftpde* program with 16K pages, more TLB entries are replaced by the background threads and hence the performance degradation is more. The benefits pro-

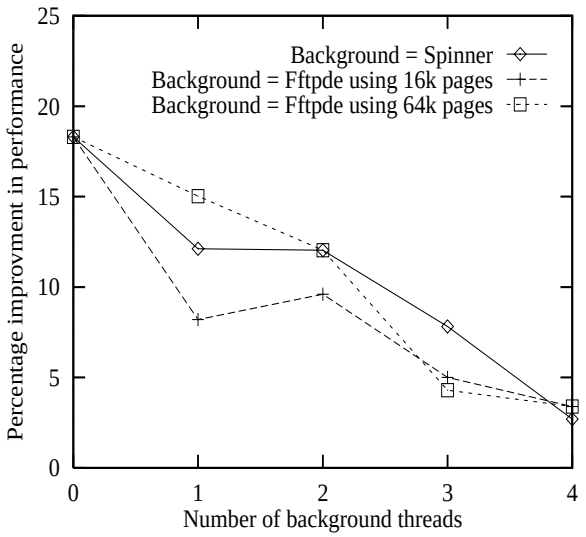
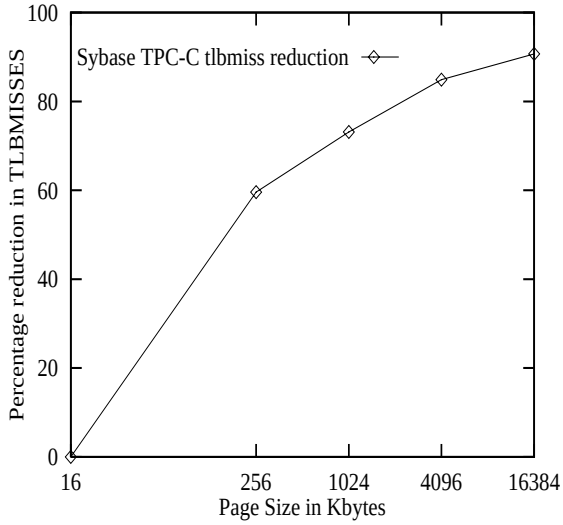


Figure 7: Large Page Size Performance Improvement under Contention.



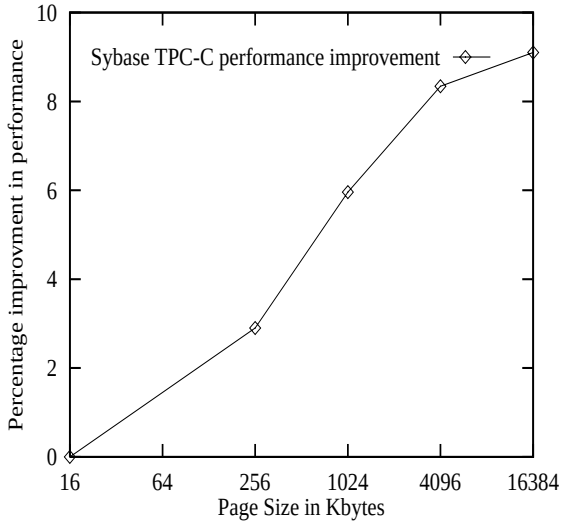


Figure 8: Large Page Size Performance Improvement for a Commercial Benchmark.

vided by large pages decrease with increase in number of background threads as fewer TLB entries are available and more TLB entries have to be replaced.

Figure 8 shows an example of how large pages can benefit a real business application. It shows the percentage reduction in TLB misses and the percentage improvement in performance at various page sizes for the TPC-C benchmark run on a two processor Origin 200QC using the Sybase Adaptive Server Enterprise database program. The program has very poor locality of reference and very large working set and consequently suffers from a high TLB miss overhead. As seen from the graph, we need very large pages (16M) to almost completely eliminate the TLB miss overhead.

8 Future Directions

One of our plans is to investigate the performance benefits of providing on-line page size upgrade based on feedback from the R10000 TLB miss counters. We also have opportunities to improve the performance of the large page allocator by minimizing page fragmentation. Some new policies may be added based on feedback from users who have used large pages on commercial applications.

9 Acknowledgments

We would like to thank Paul Mielke, Luis Stevens and Bhanu Subramanya for the many discussions we had during the course of this project. We also thank William Earl, Brad Smith and Scott Long for providing insight on the mechanisms to minimize fragmentation on workstations. We thank Marco Zagha for providing us with performance data for a commercial database application.

10 References

[BSD89] Samuel Leffler, Kirk McKusick, Michael Karels, John Quarterman, The Design and Implementation of the 4.3BSD Unix Operating System, *Addison-Wesley, ISBN 0-201-06196-1*, 1989.

[Cox94] Berny Goodheart, James Cox. The Magic Garden Explained: The Internals of Unix System V Release 4, *Prentice Hall, ISBN: 0132075563*, 1994.

[Denn70] Peter J. Denning. Virtual Memory, *Computing Surveys, 2(3):153-189*, September 1970.

[IRIX96] IRIX 6.4 man pages for `dplace(1)`, `dprof(1)`, `pm(3)`, `madvise(2)`, *Silicon Graphics*, 1996.

[Khal93] Yousef Khalidi, Madhusudhan Talluri, Michael N. Nelson, Dock Williams. Virtual memory support for multiple page sizes. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 104-109, October 1993.

[Marco96] Marco Zagha, Brond Larson, Steve Turner and Marty Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of Supercomputing '96, Nov '96.*

[Mips94] Joe Heinrich. MIPS R10000 Microprocessor User's Manual, *MIPS Technologies, Inc.*, 1994.

[PA-RISC] Ruby Lee and Jerry Huck, 64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture. On-line documentation. *http://www.hp.com/wsg/strategies/pa2go3/pa2go3.html.*

[Romer95] Theodore H.Romer, Wayne H.Ohlrich, Anna R.Karlin and Brian N. Bershad. Reducing TLB and Memory overhead Using On-line Superpage Promotion. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[Schim94] Curt Schimmel, UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers, *Addison-Wesley, ISBN 0-201-63338-8*, 1994.

[Spec95] SPEC news letter. On-line documentation at *http://www.spec.org/osg/news/articles/news9509/cpu95descr.html*, September, 1995.

[Sparc97] Ultrasparc II data sheet. On-line documentation at *http://www.sun.com/microelectronics/datasheets/stp1031/.*

[Tall94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171-182, October, 1994.

[Tall95] Madhusudhan Talluri, Mark D. Hill and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proc. of Symposium of Operating System Principles (SOSP)*, Dec 1995.