USENIX Association

Proceedings of the 5th Symposium on Operating Systems Design and Implementation

Boston, Massachusetts, USA
December 9–11, 2002

Practical, transparent operating system support for superpages

Juan Navarro[‡] Sitaram Iyer[†] Peter Druschel[†] Alan Cox[†]

{jnavarro, ssiyer, druschel, alc}@cs.rice.edu

[†]*Rice University*

[‡]*Rice University and Universidad Católica de Chile*

Abstract

Most general-purpose processors provide support for memory pages of large sizes, called superpages*. Superpages enable each entry in the translation lookaside buffer (TLB) to map a large physical memory region into a virtual address space. This dramatically increases TLB coverage, reduces TLB misses, and promises performance improvements for many applications. However, supporting superpages poses several challenges to the operating system, in terms of superpage allocation and promotion tradeoffs, fragmentation control, etc. We analyze these issues, and propose the design of an effective superpage management system. We implement it in FreeBSD on the Alpha CPU, and evaluate it on real workloads and benchmarks. We obtain substantial performance benefits, often exceeding 30%; these benefits are sustained even under stressful workload scenarios.*

1 Introduction

Modern general-purpose processors provide virtual memory support, using page tables for address translation. Most processors cache virtual-to-physical-address mappings from the page tables in a translation lookaside buffer (TLB) [10]. *TLB coverage* is defined as the amount of memory accessible through these cached mappings, i.e., without incurring misses in the TLB. Over the last decade, TLB coverage has increased at a much lower pace than main memory size. For most general-purpose processors today, TLB coverage is a megabyte or less, thus representing a very small fraction of physical memory. Applications with larger working sets can incur many TLB misses and suffer from a significant performance penalty. To alleviate this problem, most modern general-purpose CPUs provide support for *superpages*.

A *superpage* is a memory page of larger size than an ordinary page (henceforth called a *base page*). They are usually available in multiple sizes, often up to several megabytes. Each superpage occupies only one entry in the TLB, so the TLB coverage dramatically increases to cover the working set of most applications. This results in performance improvements of over 30% in many cases, as we demonstrate in Section 6.2. Recent research findings on the TLB performance of modern applications state that TLB misses are becoming increasingly performance critical [9].

However, inappropriate use of large superpages can result in enlarged application footprints, leading to increased physical memory requirements and higher paging traffic. These I/O costs can easily outweigh any performance advantages obtained by avoiding TLB misses. Therefore the operating system needs to use a mixture of page sizes. The use of multiple page sizes leads to the problem of physical memory fragmentation, and decreases future opportunities for using large superpages. To ensure sustained performance, the operating system needs to control fragmentation, without penalizing system performance. The problem of effectively managing superpages thus becomes a complex, multi-dimensional optimization task. Most general-purpose operating systems either do not support superpages at all, or provide limited support [6, 19, 20].

This paper develops a general and transparent superpage management system. It balances various tradeoffs while allocating superpages, so as to achieve high and sustained performance for real workloads and negligible degradation in pathological situations. When a process allocates memory, our system reserves a larger contiguous region of physical memory in anticipation of subsequent allocations. Superpages are then created in increasing sizes as the process touches pages in this region. If the system later runs out of contiguous physical memory, it may preempt portions of unused contiguous regions from the processes to which they were originally assigned. If these regions are exhausted, then the system restores contiguity by biasing the page replacement scheme to evict contiguous inactive pages. This system is implemented in FreeBSD on the Alpha architecture, and is evaluated on real applications and benchmarks. It

is shown to yield substantial benefits when memory is plentiful and fragmentation is low. Furthermore, it sustains these benefits over the long term, by controlling the fragmentation arising from complex workload scenarios.

The contributions of this work are four-fold. It extends a previously proposed reservation-based approach to work with multiple, potentially very large superpage sizes, and demonstrates the benefits of doing so; it is, to our knowledge, the first to investigate the effect of fragmentation on superpages; it proposes a novel contiguity-aware page replacement algorithm to control fragmentation; and it tackles issues that have to date been overlooked but are required to make a solution practical, such as superpage demotion and eviction of dirty superpages.

Section 2 motivates the problem and establishes its constraints and complexities. Section 3 examines the related work on superpages. Section 4 and 5 describe our design and implementation, and Section 6 presents the results of an experimental evaluation. Finally, Section 7 concludes.

2 The superpage problem

This section discusses the motivation, hardware constraints, issues and tradeoffs in operating system support for superpages.

2.1 Motivation

Main memory has grown exponentially in size over at least the last decade and, as cause or consequence, the memory requirements of applications have proportionally increased [20]. In contrast, TLB coverage has lagged behind. The TLB is usually fully associative and its access time must be kept low, since it is in the critical path of every memory access [13]. Hence, TLB size has remained relatively small, usually 128 or fewer entries, corresponding to a megabyte or less of TLB coverage. Figure 1 depicts the TLB coverage achieved as a percentage of main memory size, for a number of Sun and SGI workstation models available between 1986 and 2001. Relative TLB coverage is seen to be decreasing by roughly a factor of 100 over ten years. As a consequence, many modern applications have working sets larger than the TLB coverage. Section 6.3 shows that for many real applications, TLB misses degrade performance by as much as 30% to 60%, contrasting to the 4% to 5% reported in the 1980's [2, 24] or the 5% to 10% reported in the 1990's [17, 23]. Another trend that has contributed to this performance degradation is that machines are now usually shipped with on-board, physically addressed caches that are larger than the TLB coverage. As a result, many TLB misses require access to the memory banks to find a translation for data that is already in the cache, making misses relatively more expensive.

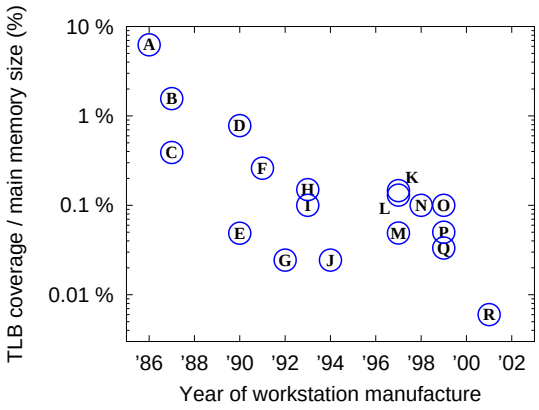

Figure 1: TLB coverage as percentage of main memory for workstations, 1986-2001 (data collected from various websites). (A) Sun 3/50; (B) Sun 3/180; (C) Sun 3/280; (D) Personal Iris; (E) SPARCstation-5; (F) Iris Indigo; (G) SPARCstation-10; (H) Indy; (I) Indigo2; (J) SPARCstation-20; (K) Ultra-1; (L) Ultra-2; (M) O2; (N) Ultra-5; (O) Ultra-10; (P) Ultra-60; (Q) Ultra-450; (R) Octane2.

We therefore seek a method of increasing TLB coverage without proportionally enlarging the TLB size. One option is to always use base pages of a larger size, say 64KB or 4MB. However, this approach would cause increased internal fragmentation due to partly used pages, and therefore induce premature onset of memory pressure [22]. Also, the I/O demands become higher due to increased paging granularity.

In contrast, the use of multiple page sizes enables an increase in TLB coverage while keeping internal fragmentation and disk traffic low. This technique, however, imposes several challenges upon the operating system designer, which are discussed in the rest of this section.

2.2 Hardware-imposed constraints

The design of TLB hardware in most processors imposes a series of constraints on superpages. Firstly, the superpage size must be among a set of page sizes supported by the processor. For example, the Alpha processor provides 8KB base pages and 64KB, 512KB and 4MB superpages; the i386 processor family supports 4KB and 4MB pages, and the new Itanium CPU provides ten different page sizes from 4KB to 256MB.

Secondly, a superpage is required to be contiguous in physical and virtual address space. Thirdly, its starting address in the physical and virtual address space must be a multiple of its size; for example, a 64KB superpage must be aligned on a 64KB address boundary.

Finally, the TLB entry for a superpage provides only a single reference bit, dirty bit, and set of protection at-

tributes. The latter implies that all base pages that form a superpage must have the same protection attributes (read, write, execute). Also, due to the coarse granularity of reference and dirty bits, the operating system can determine whether some part of the superpage has been accessed or written to, but cannot distinguish between base pages in this regard.

2.3 Issues and tradeoffs

The task of managing superpages can be conceptually broken down into a series of steps, each governed by a different set of tradeoffs. The forthcoming analysis of these issues is independent of any particular processor architecture or operating system.

We assume that the virtual address space of each process consists of a set of virtual memory objects. A memory object occupies a contiguous region of the virtual address space and contains application-specific data, as shown in Figure 2. Examples of memory objects include memory mapped files, and the code, data, stack and heap segments of processes. Physical memory for these objects is allocated as and when their pages are first accessed.

Allocation: When a page in a memory object is first touched by the application, the OS allocates a physical page frame, and maps it into the application's address space. In principle, any available page frame can be used for this purpose, just as in a system without superpage support. However, should the OS later wish to create a superpage for the object, already allocated pages may require relocation (i.e., physical copying) to satisfy the contiguity and alignment constraints of superpages. The copying costs associated with this *relocation-based* allocation approach can be difficult to recover, especially on a busy system.

An alternative is *reservation-based* allocation. Here, the OS tries to allocate a page frame that is part of an available, contiguous range of page frames equal in size and alignment to the maximal desired superpage size, and tentatively reserves the entire set for use by the process. Subsequently, when the process first touches other pages that fall within the bounds of a reservation, the corresponding base page frames are allocated and mapped. Should the OS later decide to create a superpage for this object, the allocated page frames already satisfy the contiguity and alignment constraints. Figure 2 depicts this approach.

Reservation-based allocation requires the *a priori* choice of a superpage size to reserve, without foreknowledge of memory accesses to neighbouring pages. The OS may optimistically choose the desired superpage size as the largest supported size that is smaller or equal to the size of the memory object, but it may also bias this decision on the availability of contiguous physical memory. The OS must trade off the performance gains of using a large superpage against the option of retaining the contiguous region for later, possibly more critical use.

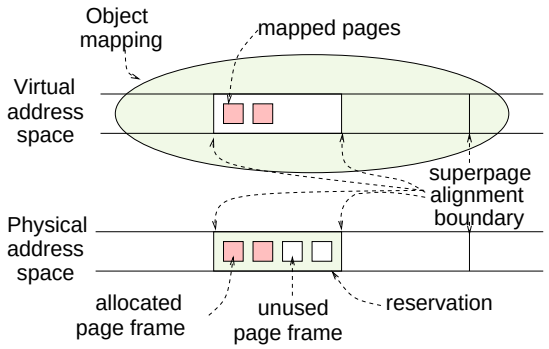

Figure 2: Reservation-based allocation.

Fragmentation control: When contiguous memory is plentiful, the OS succeeds in using superpages of the desired sizes, and achieves the maximum performance due to superpages. In practice, reservation-based allocation, use of different page sizes and file cache accesses have the combined effect of rapidly fragmenting available physical memory. To sustain the benefits of superpages, the OS may proactively release contiguous chunks of inactive memory from previous allocations, at the possible expense of having to perform disk I/O later. The OS may also preempt an existing, partially used reservation, given the possibility that the reservation may never become a superpage. The OS must therefore treat contiguity as a potentially contended resource, and trade off the impact of various contiguity restoration techniques against the benefits of using large superpages.

Promotion: Once a certain number of base pages within a potential superpage have been allocated, assuming that the set of pages satisfy the aforementioned constraints on size, contiguity, alignment and protection, the OS may decide to *promote* them into a superpage. This usually involves updating the page table entries for each of the constituent base pages of the superpage to reflect the new superpage size. Once the superpage has been created, a single TLB entry storing the translation for any address within the superpage suffices to map the entire superpage.

Promotion can also be performed incrementally. When a certain number of base pages have been allocated in a contiguous, aligned subset of a reservation, the OS may decide to promote the subset into a small superpage. These superpages may be progressively promoted

to larger superpages, up to the size of the original reservation.

In choosing when to promote a partially allocated reservation, the OS must trade off the benefits of early promotion in terms of reduced TLB misses against the increased memory consumption that results if not all constituent pages of the superpage are used.

Demotion: *Superpage demotion* is the process of marking page table entries to reduce the size of a superpage, either to base pages or to smaller superpages. Demotion is appropriate when a process is no longer actively using all portions of a superpage, and memory pressure calls for the eviction of the unused base pages. One problem is that the hardware only maintains a single reference bit for the superpage, making it difficult for the OS to efficiently detect which portions of a superpage are actively used.

Eviction: Eviction of superpages is similar to the eviction of base pages. When memory pressure demands it, an inactive superpage may be evicted from physical memory, causing all of its constituent base page frames to become available. When an evicted page is later faulted in, memory is allocated and a superpage may be created in the same way as described earlier.

One complication arises when a dirty superpage is paged out. Since the hardware maintains only a single dirty bit, the superpage may have to be flushed out in its entirety, even though some of its constituent base pages may be clean.

Managing superpages thus involves a complex set of tradeoffs; other researchers have also alluded to some of these issues [12, 13]. The next section describes previous approaches to the problem, and Section 4 describes how our design effectively tackles all these issues.

3 Related approaches

Many operating systems use superpages for kernel segments and frame buffers. This section discusses existing superpage solutions for *application memory*, which is the focus of this work. These approaches can be classified by how they manage the contiguity required for superpages: reservation-based schemes try to preserve contiguity; relocation-based approaches create contiguity; and hardware-based mechanisms reduce or eliminate the contiguity requirement for superpages.

3.1 Reservations

Reservation-based schemes make superpage-aware allocation decisions at page-fault time. On each allocation, they use some policy to decide the preferred size of the allocation and attempt to find a contiguous region of free physical memory of that size.

Talluri and Hill propose a reservation-based scheme, in which a region is reserved at page-fault time and promoted when the number of frames in use reaches a promotion threshold. Under memory pressure, reservations can be preempted to regain free space [20]. The main goal of Talluri and Hill's design is to provide a simple, best-effort mechanism tailored to the use of partial-subblock TLBs, which are described in Section 3.3.

In contrast, superpages in both the HP-UX [19] and IRIX [6] operating systems are eagerly created at page-fault time. When a page is faulted in, the system may allocate several contiguous frames to fault in surrounding pages and immediately promote them into a superpage, regardless of whether the surrounding pages are likely to be accessed. Although pages are never actually reserved, this eager promotion mechanism is equivalent to a reservation-based approach with a promotion threshold of one frame.

In IRIX and HP-UX, the preferred superpage size is based on memory availability at allocation time, and on a user-specified per-segment page size hint. This hint is associated with an application binary's text and data segments; IRIX also allows the hint to be specified at runtime.

The main drawback of IRIX and HP-UX's eager promotion is that it is not transparent. It requires experimentation to determine the optimum superpage size for the various segments of a given application. A suboptimal setting will result in lower performance, due to either insufficient TLB coverage if superpages are too small, or unnecessary paging and page population costs if superpages are too large.

3.2 Page relocation

Relocation-based schemes create superpages by physically copying allocated page frames to contiguous regions when they determine that superpages are likely to be beneficial. Relocation-based approaches can be entirely and transparently implemented in the hardware-dependent layer of the operating system, but they need to relocate most of the allocated base pages of a superpage prior to promotion, even when there are plenty of contiguous available regions.

Romer et al. propose a competitive algorithm that uses online cost-benefit analysis to determine when the benefits of superpages outweigh the overhead of superpage

promotion through relocation [16]. Their design requires a software-managed TLB, since it associates with each potential superpage a counter that must be updated by the TLB miss handler. In the absence of memory contention, this approach has a strictly lower performance than a reservation-based approach, because, in addition to the relocation costs, (1) there are more TLB misses, since relocation is performed as a reaction to an excessive number of TLB misses, and (2) TLB misses are more expensive — by a factor of four or more, according to Romer et al. — due to a more complex TLB miss handler. On the other hand, a relocation approach is more robust to fragmentation.

Reservations and page relocation can complement each other in a hybrid approach. One way would be to use relocation whenever reservations fail to provide enough contiguity and a large number of TLB misses is observed. Alternatively, page relocation can be performed as a background task to do *off-line memory compaction.* The goal is to merge fragmented chunks and gradually restore contiguity in the system. The IRIX *coalescing daemon* does this and is described in [6], but no evaluation is presented.

3.3 Hardware support

The contiguity requirement for superpages can be reduced or eliminated by means of additional hardware support.

Talluri and Hill study different TLB organizations. They advocate *partial-subblock* TLBs, which essentially contain superpage TLB entries that allow "holes" for missing base pages. They claim that with this approach most of the benefits from superpages can be obtained with minimal modifications to the operating system [20]. Partial-subblock TLBs yield only moderately larger TLB coverage than the base system, and it is not clear how to extend the partial-subblock TLBs to multiple superpage sizes.

Fang et al. describe a hardware-based mechanism that completely eliminates the contiguity requirement of superpages. They introduce an additional level of address translation in the memory controller, so that the operating system can promote non-adjacent physical pages into a superpage. This greatly simplifies the task of the operating system for supporting superpages [3].

To the best of our knowledge, neither partial-subblock TLBs nor address-remapping memory controllers are supported on commercial, general-purpose machines.

Our approach generalizes Talluri and Hill's reservation mechanism to multiple superpage sizes. To regain contiguity on fragmented physical memory without relocating pages, it biases the page replacement policy to select those pages that contribute the most to contiguity. It also tackles the issues of demotion and eviction (described in Section 2.3) not addressed by previous work, and does not require special hardware support.

4 Design

Our design adopts the reservation-based superpage management paradigm introduced in [20]. It extends the basic design along several dimensions, such as support for multiple superpage sizes, scalability to very large superpages, demotion of sparsely referenced superpages, effective preservation of contiguity without the need for compaction, and efficient disk I/O for partially modified superpages. As shown in Section 6, this combination of techniques is general enough to work efficiently for a range of realistic workloads, and is believed to be suitable for deployment in modern operating systems.

A high-level sketch of the design contains the following components. Available physical memory is classified into contiguous regions of different sizes, and is managed using a buddy allocator [14]. A multi-list reservation scheme is used to track partially used memory reservations, and to help in choosing reservations for preemption, as described in Section 4.8. A population map keeps track of memory allocations in each memory object, as described in Section 4.9. The system uses these data structures to implement allocation, preemption, promotion and demotion policies. Finally, it controls external memory fragmentation by performing page replacements in a contiguity-aware manner, as described in Section 4.4. The following subsections elaborate on these concepts.

4.1 Reservation-based allocation

Most operating systems allocate physical memory on application demand. When a virtual memory page is accessed by a program and no mapping exists in the page table, the OS's page fault handler is invoked. The handler attempts to locate the associated page in main memory; if it is not resident, an available page frame is allocated and the contents are either zero-filled or fetched from the paging device. Finally, the appropriate mapping is entered into the page table.

Instead of allocating physical memory one frame at a time, our system determines a preferred superpage size for the region encompassing the base page whose access caused the page fault. The choice of a size is made according to a policy described in Section 4.2. At page-fault time, the system obtains from the buddy allocator a set of contiguous page frames corresponding to the chosen superpage size. The frame with the same address

alignment as the faulted page is used to fault in the page, and a mapping is entered into the page table for this page only. The entire set of frames is tentatively *reserved* for potential future use as a superpage, and added to a reservation list. In the event of a page fault on a page for which a frame has already been reserved, a mapping is entered into the page table for the base page.

4.2 Preferred superpage size policy

Next, we describe the policy used to choose the desired superpage size during allocation. Since this decision is usually made early in a process's execution, when it is hard to predict its future behaviour, our policy looks only at attributes of the memory object to which the faulting page belongs. If the chosen size turns out to be too large, then the decision will be later overridden by preempting the initial reservation. However, if the chosen size is too small, then the decision cannot be reverted without relocating pages. For that reason, the policy tends to choose the maximum superpage size that can be effectively used in an object.

For memory objects that are fixed in size, such as code segments and memory-mapped files, the desired reservation size is the largest, aligned superpage that contains the faulting page, does not overlap with existing reservations or allocated pages, and does not reach beyond the end of the object.

Dynamically sized memory objects such as stacks and heaps can grow one page at a time. Under the policy for fixed size objects, they would not be able to use superpages, because each time the policy would set the preferred size to one base page. Thus a slightly different policy is required. As before, the desired size is the largest, aligned superpage that contains the faulting page and does not overlap with existing reservations or allocations. However, the restriction that the reservation must not reach beyond the end of the object is dropped to allow for growth. To avoid wastage of contiguity for small objects that may never grow large, the size of this superpage is limited to the current size of the object. This policy thus uses large reservations only for objects that have already reached a sufficiently large size.

4.3 Preempting reservations

When free physical memory becomes scarce or excessively fragmented, the system can preempt frames that are reserved but not yet used. When an allocation is requested and no extent of frames with the desired size is available, the system has to choose between (1) refusing the allocation and thus reserving a smaller extent than desired, or (2) preempting an existing reservation that has enough unallocated frames to yield an extent of the desired size.

Our policy is that, whenever possible, the system preempts existing reservations rather than refusing an allocation of the desired size. When more than one reservation can yield an extent of the desired size, the reservation is preempted whose most recent page allocation occurred least recently, among all candidate reservations. This policy is based on the observation that useful reservations are often populated quickly, and that reservations that have not experienced any recent allocations are less likely to be fully allocated in the near future.

4.4 Fragmentation control

Allocating physical memory in contiguous extents of multiple sizes leads to fragmentation of main memory. Over time, extents of large sizes may become increasingly scarce, thus preventing the effective use of superpages.

To control fragmentation, our buddy allocator performs coalescing of available memory regions whenever possible. However, coalescing by itself is only effective if the system periodically reaches a state where all or most of main memory is available. To control fragmentation under persistent memory pressure, the page replacement daemon is modified to perform contiguity-aware page replacement. Section 5.1 discusses this in greater detail.

4.5 Incremental promotions

A superpage is created as soon as any superpage-sized and aligned extent within a reservation gets fully populated. Promotion, therefore, is incremental: if, for instance, pages of a memory object are faulted in sequentially, a promotion occurs to the smallest superpage size as soon as the population count corresponds to that size. Then, when the population count reaches the next larger superpage size, another promotion occurs to the next size, and so on.

It is possible to promote to the next size when the population count reaches a certain fraction of that size. However, before performing the promotion the system needs to populate the entire region, which could artificially inflate the memory footprint of applications. We promote only regions that are fully populated by the application, since we observe that most applications populate their address space densely and relatively early in their execution.

4.6 Speculative demotions

Demotion occurs as a side-effect of page replacement. When the page daemon selects a base page for eviction that is part of a superpage, the eviction causes a demotion of that superpage. This demotion is also incremental, since it is not necessary to demote a large superpage all the way to base pages just because one of its constituent base pages is evicted. Instead, the superpage is first demoted to the next smaller superpage size, then the process is applied recursively for the smaller superpage that encompasses the victim page, and so on. Demotion is also necessary whenever the protection attributes are changed on part of a superpage. This is required because the hardware provides only a single set of protection bits for each superpage.

The system may also periodically demote active superpages *speculatively* in order to determine if the superpage is still being actively used in its entirety. Recall that the hardware only provides a single reference bit with each superpage. Therefore, the operating system has no way to distinguish a superpage in which all the constituent base pages are being accessed, from one in which only a subset of the base pages are. In the latter case, it would be desirable to demote the superpage under memory pressure, such that the unused base pages can be discovered and evicted.

To address this problem, when the page daemon resets the reference bit of a superpage's base page, and if there is memory pressure, then it recursively demotes the superpage that contains the chosen base page, with a certain probability p. In our current implementation, p is 1. Incremental repromotions occur when all the base pages of a demoted superpages are being referenced.

4.7 Paging out dirty superpages

When a dirty superpage needs to be written to disk, the operating system does not possess dirty bit information for individual base pages. It must therefore consider all the constituent base pages dirty, and write out the superpage in its entirety, even though only a few of its base pages may have actually been modified. For large, partially dirty superpages, the performance degradation due to this superfluous I/O can considerably exceed any benefits from superpages.

To prevent this problem, we demote clean superpages whenever a process attempts to write into them, and repromote later if all the base pages are dirtied. This choice is evaluated in Section 6.7.

Inferring dirty base pages using hash digests: As an alternative, we considered a technique that retains the benefits of superpages even when they are partially dirty, while avoiding superfluous I/O. When a clean memory page is read from disk, a cryptographic hash digest of its contents is computed and recorded. If a partially dirty set of base pages is promoted to a superpage, or if a clean superpage becomes dirty, then all its constituent base pages are considered dirty. However, when the page is flushed out, the hash of each base page is recomputed and compared to determine if it was actually modified and must be written to disk.

A 160-bit SHA-1 hash has a collision probability of about one in 2^{80} [4], which is much smaller than the probability of a hardware failure. Hence this technique can be considered safe. However, preliminary microbenchmarks using SHA-1 reveal significant overhead, up to 15%, on disk-intensive applications. The pathological case of a large sequential read when the CPU is saturated incurs a worst-case degradation of 60%. Therefore, we did not use this technique in our implementation.

However, these overheads can be reduced using a variety of optimizations. First, the hash computation can be postponed until there is a partially dirty superpage, so that fully-clean or fully-dirty superpages and unpromoted base pages need not be hashed. Second, the hashing cost can be eliminated from the critical path by performing it entirely from the idle loop, since the CPU may frequently be idle for disk-intensive workloads. An evaluation of these optimizations is the subject of future work.

4.8 Multi-list reservation scheme

Reservation lists keep track of reserved page frame extents that are not fully populated. There is one reservation list for each page size supported by the hardware, except for the largest superpage size. Each reservation appears in the list corresponding to the size of the largest free extent that can be obtained if the reservation is preempted. Because a reservation has at least one of its frames allocated, the largest extents it can yield if preempted are one page size smaller than its own size. For instance, on an implementation for the Alpha processor, which supports 4MB, 512KB, 64KB and 8KB pages, the 64KB reservation list may contain reservations of size 512KB and 4MB.

Reservations in each list are kept sorted by the time of their most recent page frame allocations. When the system decides to preempt a reservation of a given size, it chooses the reservation at the head of the list for that size. This satisfies our policy of preempting the extent whose most recent allocation occurred least recently among all reservations in that list.

Preempting a chosen reservation occurs as follows. Rather than breaking the reservation into base pages, it is broken to smaller extents. Unpopulated extents are

transferred to the buddy allocator and partially populated ones are reinserted into the appropriate lists. For example, when preempting a 512KB reservation taken from head of the 64KB list, the reservation is broken into eight 64KB extents. The ones with no allocations are freed and the ones that are partially populated are inserted at the head of the 8KB reservation list. Fully populated extents are not reinserted into the reservation lists.

When the system needs a contiguous region of free memory, it can obtain it from the buddy allocator or by preempting a reservation. The mechanism is best described with an example. Still in the context of the Alpha CPU, suppose that an application faults in a given page for which there is no reserved frame. Further assume that the preferred superpage size for the faulting page is 64KB. Then the system first asks the buddy allocator for a 64KB extent. If that fails, it preempts the first reservation in the 64KB reservation list, which should yield at least one 64KB extent. If the 64KB list is empty, the system will try the 512KB list. If that list is also empty, then the system has to resort to base pages: the buddy allocator is tried first, and then the 8KB reservation list as the last resource.

4.9 Population map

Population maps keep track of allocated base pages within each memory object. They serve four distinct purposes: (1) on each page fault, they enable the OS to map the virtual address to a page frame that may already be reserved for this address; (2) while allocating contiguous regions in physical address space, they enable the OS to detect and avoid overlapping regions; (3) they assist in making page promotion decisions; and (4) while preempting a reservation, they help in identifying unallocated regions.

A population map needs to support efficient lookups, since it is queried on every page fault. We use a radix tree in which each level corresponds to a page size. The root corresponds to the maximum superpage size supported by the hardware, each subsequent level corresponds to the next smaller superpage size, and the leaves correspond to the base pages. If the virtual pages represented by a node have a reserved extent of frames, then the node has a pointer to the reservation and the reservation has a back pointer to the node.

Each non-leaf node keeps a count of the number of superpage-sized virtual regions at the next lower level that have a population of at least one (the `somepop` counter), and that are fully populated (the `fullpop` counter), respectively. This count ranges from 0 through R, where R is the ratio between consecutive superpage sizes (8 on the Alpha processor). The tree is lazily updated as the object's pages are populated. The absence of a child node is equivalent to having a child with both counters zero. Since counters refer to superpage-sized regions, upward propagation of the counters occurs only when `somepop` transitions between 0 and 1, or when `fullpop` transitions between $R-1$ and R. Figure 3 shows one such tree.

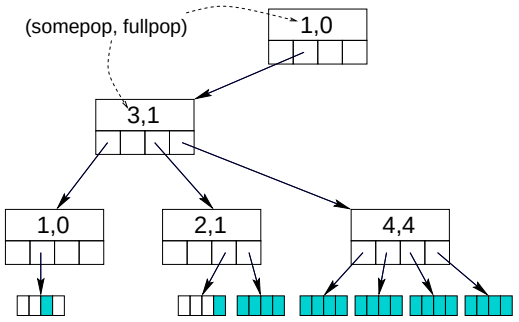

Figure 3: A population map. At the base page level, the actual allocation of pages is shown.

A hash table is used to locate population maps. For each population map, there is an entry associating a *memory_object, page_index* tuple with the map, where *page_index* is the offset of the starting page of the map within the object. The population map is used as follows:

Reserved frame lookup: On a page fault, the virtual address of the faulting page is rounded down to a multiple of the largest page size, converted to the corresponding *memory_object, page_index* tuple, and hashed to determine the root of the population map. From the root, the tree is traversed to locate the reserved page frame, if there is one.

Overlap avoidance: If the above procedure yields no reserved frame, then we attempt to make a reservation. The maximum size that does not overlap with previous reservations or allocations is given by the first node in the path from the root whose `somepop` counter is zero.

Promotion decisions: After a page fault is serviced, a promotion is attempted at the first node on the path from the root to the faulting page that is fully populated and has an associated reservation. The promotion attempt succeeds only if the faulting process has the pages mapped with uniform protection attributes and dirty bits.

Preemption assistance: When a reservation is preempted it is broken into smaller chunks that need to be freed or reinserted in the reservation lists, depending on their allocation status, as described in Section 4.8. The allocation status corresponds to the population counts in the superpage map node to which the reservation refers.

5 Implementation notes

This section describes some implementation specific issues of our design. While the discussion of our solution is necessarily OS-specific, the issues are general.

5.1 Contiguity-aware page daemon

FreeBSD's page daemon keeps three lists of pages, each in approximate LRU (A-LRU) order: active, inactive and cache. Pages in the cache list are clean and unmapped and hence can be easily freed under memory pressure. Inactive pages are those mapped into the address space of some process, and have not been referenced for a long time. Active pages are those that have been accessed recently, but may or may not have their reference bit set. Under memory pressure, the daemon moves clean inactive pages to the cache, pages out dirty inactive pages, and also deactivates some unreferenced pages from the active list. We made the following changes to factor contiguity restoration into the page replacement policy.

(1) We consider cache pages as available for reservations. The buddy allocator keeps them coalesced with the free pages, increasing the available contiguity of the system. These coalesced regions are placed at the tail of their respective lists, so that subsequent allocations tend to respect the A-LRU order.

The contents of a cache page are retained as long as possible, whether it is in a buddy list or in a reservation. If a cache page is referenced, then it is removed from the buddy list or the reservation; in the latter case, the reservation is preempted. The cache page is reactivated and its contents are reused.

(2) The page daemon is activated not only on memory pressure, but also when available contiguity falls low. In our implementation, the criterion for low contiguity is the failure to allocate a contiguous region of the preferred size. The goal of the daemon is to restore the contiguity that would have been necessary to service the requests that failed since the last time the daemon was woken. The daemon then traverses the inactive list and moves to the cache only those pages that contribute to this goal. If it reaches the end of the list before fulfilling its goal, then it goes to sleep again.

(3) Since the chances of restoring contiguity are higher if there are more inactive pages to choose from, all clean pages backed by a file are moved to the inactive list as soon as the file is closed by all processes. This differs from the current behaviour of FreeBSD, where a page does not change its status on file closing or process termination, and active pages from closed files may never be deactivated if there is no memory pressure. In terms of overall performance, our system thus finds it worthwhile to favor the likelihood of recovering the contiguity from these file-backed pages, than to keep them for a longer time for the chance that the file is accessed again.

Controlling fragmentation comes at a price. The more aggressively the system recovers contiguity, the greater is the possibility and the extent of a performance penalty induced by the modified page daemon, due to its deviation from A-LRU. Our modified page daemon aims at balancing this tradeoff. Moreover, by judiciously selecting pages for replacement, it attempts to restore as much contiguity as possible by affecting as few pages as possible. Section 6.5 demonstrates the benefits of this design.

5.2 Wired page clustering

Memory pages that are used by FreeBSD for its internal data structures are *wired*, that is, marked as non-pageable since they cannot be evicted. At system boot time these pages are clustered together in physical memory, but as the kernel allocates memory while other processes are running, they tend to get scattered. Our system with 512MB of main memory is found to rapidly reach a point where most 4MB chunks of physical memory contain at least one wired page. At this point, contiguity for large pages becomes irrecoverable.

To avoid this fragmentation problem, we identify pages that are about to be wired for the kernel's internal use. We cluster them in pools of contiguous physical memory, so that they do not fragment memory any more than necessary.

5.3 Multiple mappings

Two processes can map a file into different virtual addresses. If the addresses differ by, say, one base page, then it is impossible to build superpages for that file in the page tables of both processes. At most one of the processes can have alignment that matches the physical address of the pages constituting the file; only this process is capable of using superpages.

Our solution to this problem leverages the fact that applications most often do not specify an address when mapping a file. This gives the kernel the flexibility to assign a virtual address for the mapping in each process. Our system then chooses addresses that are compatible with superpage allocation. When mapping a file, the system uses a virtual address that aligns to the largest superpage that is smaller than the size of the mapping, thus retaining the ability to create superpages in each process.

6 Evaluation

This section reports results of experiments that exercise the system on several classes of benchmarks and real applications. We evaluate the best-case benefits of superpages in situations when system memory is plentiful. Then, we demonstrate the effectiveness of our design, by showing how these benefits are sustained despite different kinds of stress on the system. Results show the efficiency of our design by measuring its overhead in several pathological cases, and justify the design choices in the previous section using appropriate measurements.

6.1 Platform

We implemented our design in the FreeBSD-4.3 kernel as a loadable module, along with hooks in the operating system to call module functions at specific points. These points are page faults, page allocation and deallocation, the page daemon, and at the physical layer of the VM system (to demote when changing protections and to keep track of dirty/modified bits of superpages). We were also able to seamlessly integrate this module into the kernel. The implementation comprises of around 3500 lines of C code.

We used a Compaq XP-1000 machine with the following characteristics:

- Alpha 21264 processor at 500 MHz;
- four page sizes: 8KB base pages, 64KB, 512KB and 4MB superpages;
- fully associative TLB with 128 entries for data and 128 for instructions;
- software page tables, with firmware-based TLB loader;
- 512MB RAM;
- 64KB data and 64KB instruction L1 caches, virtually indexed and 2-way associative;
- 4MB unified, direct-mapped external L2 cache.

The Alpha firmware implements superpages by means of *page table entry (PTE) replication*. The page table stores an entry for every base page, whether or not it is part of a superpage. Each PTE contains the translation information for a base page, along with a page size field. In this PTE replication scheme, the promotion of a 4MB region involves the setting of the page size field of *each of the 512 page table entries* that map the region [18].

6.2 Workloads

We used the following benchmarks and applications to evaluate our system.

CINT2000: SPEC CPU2000 integer benchmark suite [7].

CFP2000: SPEC CPU2000 floating-point benchmark suite [7].

Web: The thttpd web server [15] servicing 50000 requests selected from an access log of the CS departmental web server at Rice University. The working set size of this trace is 238MB, while its data set is 3.6GB.

Image: 90-degree rotation of a 800x600-pixel image using the popular open-source ImageMagick tools [8].

Povray: Ray tracing of a simple image.

Linker: Link of the FreeBSD kernel with the GNU linker.

C4: An alpha-beta search solver for a 12-ply position of the connect-4 game, also known as the fhourstones benchmark.

Tree: A synthetic benchmark that captures the behaviour of processes that use dynamic allocation for a large number of small objects, leading to poor locality of reference. The benchmark consists of four operations performed randomly on a 50000-node red-black tree: 50% of the operations are lookups, 24% insertions, 24% deletions, and 2% traversals. Nodes on the tree contain a pointer to a 128-byte record. On insertions a new record is allocated and initialized; on lookups and traversals, half of the record is read.

SP: The sequential version of a scalar pentadiagonal uncoupled equation system solver, from the NAS Parallel Benchmark suite [1]. The input size corresponds to the "workstation class" in NAS's nomenclature.

FFTW: The Fastest Fourier Transform in the West [5] with a 200x200x200 matrix as input.

Matrix: A non-blocked matrix transposition of a 1000x1000 matrix.

6.3 Best-case benefits due to superpages

This first set of experiments shows that several classes of real workloads yield large benefits with superpages when free memory is plentiful and non-fragmented. Table 1 presents these best-case speedups obtained when the benchmarks are given the contiguous memory regions they need, so that every attempt to allocate regions of the preferred superpage size (as defined in Section 4.2) succeeds, and reservations are never preempted.

The speedups are computed against the unmodified system using the mean elapsed runtime of three runs after an initial warm-up run. For both the CINT2000 and CFP2000 entries in the table, the speedups reflect, respectively, the improvement in SPECint2000 and SPECfp2000 (defined by SPEC as the geometric mean of the normalized throughput ratios).

The table also presents the superpage requirements of each of these applications (as a snapshot measured at peak memory usage), and the percentage data TLB miss reduction achieved with superpages. In most cases the

data TLB misses are virtually eliminated by superpages, as indicated by a miss reduction close to 100%. The contribution of instruction TLB misses to the total number of misses was found to be negligible in all of the benchmarks.

Benchmark	Superpage usage 8 KB	64 KB	512 KB	4 MB	Miss reduc (%)	Speedup
CINT2000						**1.112**
gzip	204	22	21	42	80.00	1.007
vpr	253	29	27	9	99.96	1.383
gcc	1209	1	17	35	70.79	1.013
mcf	206	7	10	46	99.97	1.676
crafty	147	13	2	0	99.33	1.036
parser	168	5	14	8	99.92	1.078
eon	297	6	0	0	0.00	1.000
perl	340	9	17	34	96.53	1.019
gap	267	8	7	47	99.49	1.017
vortex	280	4	15	17	99.75	1.112
bzip2	196	21	30	42	99.90	1.140
twolf	238	13	7	0	99.87	1.032
CFP2000						**1.110**
wupw	219	14	6	43	96.77	1.009
swim	226	16	11	46	98.97	1.034
mgrid	282	15	5	13	98.39	1.000
applu	1927	1647	90	5	93.53	1.020
mesa	246	13	8	1	99.14	0.985
galgel	957	172	68	2	99.80	1.289
art	163	4	7	0	99.55	1.122
equake	236	2	19	9	97.56	1.015
facerec	376	8	13	2	98.65	1.062
ammp	237	7	21	7	98.53	1.080
lucas	314	4	36	31	99.90	1.280
fma3d	500	17	27	22	96.77	1.000
sixtr	793	81	29	1	87.50	1.043
apsi	333	5	5	47	99.98	1.827
Web	30623	5	143	1	16.67	1.019
Image	163	1	17	7	75.00	1.228
Povray	136	6	17	14	97.44	1.042
Linker	6317	12	29	7	85.71	1.326
C4	76	2	9	0	95.65	1.360
Tree	207	6	14	1	97.14	1.503
SP	151	103	15	0	99.55	1.193
FFTW	160	5	7	60	99.59	1.549
Matrix	198	12	5	3	99.47	7.546

Table 1: Speedups and superpage requirements when plenty of memory is available.

Nearly all the workloads in the table display benefits due to superpages; some of these are substantial. Out of our 35 benchmarks, 18 show improvements over 5% (speedup of 1.05), and 10 show over 25%. The only application that slows down is mesa, which degrades by a negligible fraction. Matrix, with a speedup of 7.5, is close to the maximum potential benefits that can possibly be gained with superpages, because of its access pattern that produces one TLB miss for every two memory accesses.

Several commonplace desktop applications like Linker (gnuld), gcc, and bzip2 observe significant performance improvements. If sufficient contiguous memory is available, then these applications stand to benefit from a superpage management system. In contrast, Web gains little, because the system cannot create enough superpages in spite of its large 315MB footprint. This is because Web accesses a large number of small files, and the system does not attempt to build superpages that span multiple memory objects. Extrapolating from the results, a system without such limitation (which is technically feasible, but likely at a high cost in complexity) would bring Web's speedup closer to a more attractive 15%, if it achieved a miss reduction close to 100%.

Some applications create a significant number of large superpages. FFTW, in particular, stands out with 60 superpages of size 4MB. The next section shows that FFTW makes good use of large superpages, as there is almost no speedup if 4MB pages are not supported.

Mesa shows a small performance degradation of 1.5%. This was determined to be not due to the overhead of our implementation, but because our allocator does not differentiate zeroed-out pages from other free pages. When the OS allocates a page that needs to be subsequently zeroed out, it requests the memory allocator to preferentially allocate an already zeroed-out page if possible. Our implementation of the buddy allocator ignores this hint; we estimated the cost of this omission by comparing base system performance with and without the zeroed-page feature. We obtained an average penalty of 0.9%, and a maximum of 1.7%.

A side effect of using superpages is that it subsumes page coloring [11], a technique that FreeBSD and other operating systems use to reduce cache conflicts in physically-addressed and especially in direct-mapped caches. By carefully selecting among free frames when mapping a page, the OS keeps virtual-to-physical mappings in a way such that pages that are consecutive in virtual space map to consecutive locations in the cache. Since with superpages virtually contiguous pages map to physically contiguous frames, they automatically map to consecutive locations in a physically-mapped cache. Our speedup results factor out the effect of page-coloring, because the benchmarks were run with enough free memory for the unmodified system to always succeed in its page coloring attempts. Thus, both the unmodified and the modified system effectively benefit from page coloring.

6.4 Benefits from multiple superpage sizes

We repeated the above experiments, but changed the system to support only one superpage size, for each of 64KB, 512KB and 4MB, and compared the resulting performance against our multi-size implementation. Tables 2 and 3 respectively present the speedup and TLB miss reduction for the benchmarks, excluding those that have the same speedup (within 5%) in all four cases.

Benchmark	64KB	512KB	4MB	All
CINT2000	1.05	1.09	1.05	1.11
vpr	1.28	1.38	1.13	1.38
mcf	1.24	1.31	1.22	1.68
vortex	1.01	1.07	1.08	1.11
bzip2	1.14	1.12	1.08	1.14
CFP2000	1.02	1.08	1.06	1.12
galgel	1.28	1.28	1.01	1.29
lucas	1.04	1.28	1.24	1.28
apsi	1.04	1.79	1.83	1.83
Image	1.19	1.19	1.16	1.23
Linker	1.16	1.26	1.19	1.32
C4	1.30	1.34	0.98	1.36
SP	1.19	1.17	0.98	1.19
FFTW	1.01	1.00	1.55	1.55
Matrix	3.83	7.17	6.86	7.54

Table 2: Speedups with different superpage sizes.

The results show that the best superpage size depends on the application. For instance, it is 64KB for SP, 512KB for vpr, and 4MB for FFTW. The reason is that while some applications only benefit from large superpages, others are too small to fully populate large superpages. To use large superpages with small applications, the population threshold for promotion could be lowered, as suggested in Section 4.5. However, the OS would have to populate regions that are only partially mapped by the application. This would enlarge the application footprint, and also slightly change the OS semantics, since some invalid accesses would not be caught.

The tables also demonstrate that allowing the system to choose between multiple page sizes yields higher performance, because the system dynamically selects the best size for every region of memory. An extreme case is mcf, for which the percentage speedup when the system gets to choose among several sizes more than doubles the speedup with any single size.

Some apparent anomalies, like different speedups with the same TLB miss reduction (e.g., Linker) are likely due to the coarse granularity of the Alpha processor's TLB miss counter (512K misses). For short-running benchmarks, 512K misses corresponds to a two-digit percentage of the total number of misses.

Benchmark	64KB	512KB	4MB	All
CINT2000				
vpr	82.49	98.66	45.16	99.96
mcf	55.21	84.18	53.22	99.97
vortex	46.38	92.76	80.86	99.75
bzip2	99.80	99.09	49.54	99.90
CFP2000				
galgel	98.51	98.71	0.00	99.80
lucas	12.79	96.98	87.61	99.90
apsi	9.69	98.70	99.98	99.98
Image	50.00	50.00	50.00	75.00
Linker	57.14	85.71	57.14	85.71
C4	95.65	95.65	0.00	95.65
SP	99.11	93.75	0.00	99.55
FFTW	7.41	7.41	99.59	99.59
Matrix	90.43	99.47	99.47	99.47

Table 3: TLB miss reduction percentage with different superpage sizes.

6.5 Sustained benefits in the long term

The performance benefits of superpages can be substantial, provided contiguous regions of physical memory are available. However, conventional systems can be subject to memory fragmentation even under moderately complex workloads. For example, we ran instances of grep, emacs, netscape and a kernel compilation on a freshly booted system; within about 15 minutes, we observed severe fragmentation. The system had completely exhausted all contiguous memory regions larger than 64KB that were candidates for larger superpages, even though as much as 360MB of the 512MB were free.

Our system seeks to preserve the performance of superpages over time, so it actively restores contiguity using techniques described in Sections 4.4 and 5.1. To evaluate these methods, we first fragment the system memory by running a web server and feeding it with requests from the same access log as before. The file-backed memory pages accessed by the web server persist in memory and reduce available contiguity to a minimum. Moreover, the access pattern of the web server results in an interleaved distribution of active, inactive and cache pages, which increases fragmentation.

We present two experiments using this web server.

Sequential execution: After the requests from the trace have been serviced, we run the FFTW benchmark four times in sequence. The goal is to see how quickly the system recovers just enough contiguous memory to build superpages and perform efficiently.

Figure 4 compares the performance of two contiguity restoration techniques. The *cache* scheme treats all cached pages as available, and coalesces them into the

buddy allocator. The graph depicts no appreciable performance improvements of FFTW over the base system. We observed that the system is unable to provide even a single 4MB superpage for FFTW. This is because memory is available (47MB in the first run and 290MB in the others), but is fragmented due to active, inactive and wired pages.

The other scheme, called *daemon*, is our implementation of contiguity-aware page replacement and wired page clustering. The first time FFTW runs after the web server, the page daemon is activated due to contiguity shortage, and is able to recover 20 out of the requested 60 contiguous regions of 4MB size. Subsequent runs get a progressively larger number of 4MB superpages, viz. 35, 38 and 40. Thus, FFTW performance reaches near-optimum within two runs, i.e., a speedup of 55%.

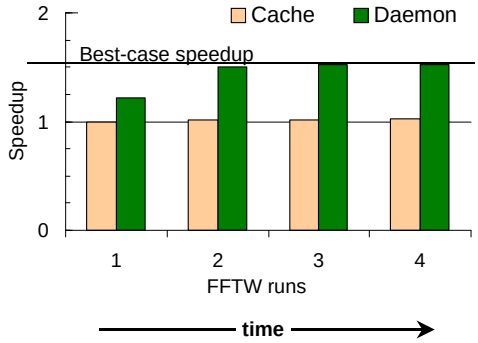

Figure 4: Two techniques for fragmentation control.

The web server closes its files on exit, and our page daemon treats this file memory as inactive, as described in Section 5.1. We now measure the impact of this effect in conjunction with the page daemon's drive to restore contiguity, on the web server's subsequent performance. We run the web server again after FFTW, and replay the same trace. We observe only a 1.6% performance degradation over the base system, indicating that the penalty on the web server performance is small.

We further analyze this experiment by monitoring the available contiguity in the system over time. We define an empirical *contiguity metric* as follows. We assign 1, 2 or 3 points to each base page that belongs to a 64KB, 512KB, or 4MB memory region respectively, assuming that the region is contiguous, aligned and fully available. We compute the sum of these per-page points, and normalize it to the corresponding value if every page in the system were to be free. Figure 5 shows a plot of this contiguity metric against experimental time. Note that this metric is unfavorable to the daemon scheme since it does not consider as available the extra contiguity that can be regained by moving inactive pages to the cache.

At the start of the experiment, neither scheme has all of the system's 512MB available; in particular, the cache scheme has lost 5% more contiguity due to unclustered wired pages. For about five minutes, the web server consumes memory and decreases available contiguity to zero. Thereafter, the cache scheme recovers only 8.8% of the system's contiguity, which can be seen in the graph as short, transitory bursts between FFTW executions. In contrast, the daemon scheme recovers as much as 42.4% of the contiguity, which is consumed by FFTW while it executes, and released each time it exits. The FFTW executions thus finish earlier, at 8.5 minutes for the daemon scheme, compared to 9.8 minutes for the cache scheme.

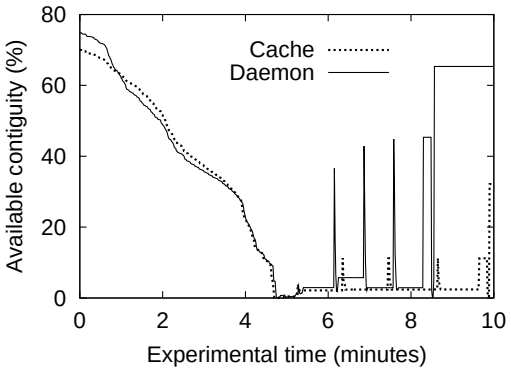

Figure 5: Contiguity as a function of time.

To estimate the maximum contiguity that can be potentially gained back after the FFTW runs complete, we run a synthetic application that uses enough anonymous memory to maximize the number of free pages in the system when it exits. At this point, the amount of contiguity lost is 54% in the cache scheme, mostly due to scattered wired pages. In contrast, the daemon scheme in unable to recover 13% of the original contiguity. The reason is that the few active and inactive pages that remain at the end of the experiment are scattered in physical memory over as many as 54 4MB chunks. Since the experiment starts on a freshly booted system, active and inactive pages were physically close at that time, occupying only 22 such chunks. Part of the lost 13% is due to inactive pages that are not counted in the contiguity metric, but can be recovered by the page daemon. Therefore, the real loss in the long term for the daemon scheme is bounded only by the number of active pages.

Concurrent execution: The next experiment runs the web server *concurrently* with a contiguity-seeking application. The goal is to measure the effect of the page replacement policy on the web server during a single, con-

tinuous run. We isolate the effect of the page replacement policy by disabling superpage promotions in this experiment.

We warm up the web server footprint by playing 100,000 requests from the trace, and then measure the time taken to service the next 100,000 requests. We wish to avoid interference of the CPU-intensive FFTW application with the web server, so we substitute it with a dummy application that only exercises the need for contiguity. This application maps, touches and unmaps 1MB of memory, five times a second, and forces the page daemon to recover contiguity rather than just memory.

The web server keeps its active files open while it is running, so our page daemon cannot indiscriminately treat this memory as inactive. The web server's active memory pages get scattered, and only a limited amount of contiguity can be restored without compacting memory. Over the course of the experiment, the dummy application needs about 3000 contiguous chunks of 512KB size. The original page daemon only satisfied 3.3% of these requests, whereas our contiguity-aware page daemon fulfills 29.9% of the requests. This shows how the change in the replacement policy succeeds in restoring significantly more contiguity than before, with negligible overhead and essentially no performance penalty.

The overhead of the contiguity restoration operations of the page daemon is found to be only 0.8%, and the web server suffers an additional 3% of performance degradation, as a consequence of the deviation of the page replacement policy from A-LRU.

6.6 Adversary applications

This section exercises the system on three synthetic pathological workloads, and concludes with a measurement of realistic overhead.

Incremental promotion overhead: We synthesized an adversary application that makes the system pay all the costs of incremental promotion without gaining any benefit. It allocates memory, accesses one byte in each page, and deallocates the memory, which renders the TLB useless since every translation is used only once. This adversary shows a slowdown of 8.9% with our implementation, but as much as 7.2% of this overhead is due to the following hardware-specific reason. PTE replication, as described in Section 6.1, forces each page table entry to be traversed six times: once per each of the three incremental promotions, and once per each of the three incremental demotions. The remaining 1.7% of the overhead is mainly due to maintenance of the population maps.

Sequential access overhead: Accessing pages sequentially as in our adversary is actually a common behaviour, but usually every byte of each page is accessed, which dilutes the overhead. We tested the `cmp` utility, which compares two files by mapping them in memory, using two identical 100MB files as input, and observed a negligible performance degradation of less than 0.1%.

Preemption overhead: To measure the overhead of preempting reservations, we set up a situation where there is only 4MB of memory available and contiguous, and run a process that touches memory with a 4MB stride. In this situation, there is a pattern of one reservation preemption every seven allocations. Every preemption splits a reservation into 8 smaller chunks. One remains reserved with the page that made the original reservation; another is taken for the page being allocated, and 6 are returned to the free list. We measured a performance degradation of 1.1% for this process.

Overhead in practice: Finally, we measure the total overhead of our implementation in real scenarios. We use the same benchmarks of Section 6.2, perform all the contiguous memory allocation and fragmentation management as before, but factor out the benefit of superpages by simply not promoting them. We preserve the promotion overhead by writing the new superpage size into some unused portion of the page table entries. We observe performance degradations of up to 2%, with an average of about 1%. This shows how our system only imposes negligible overhead in practice, so the pathological situations described above are rarely observed.

6.7 Dirty superpages

To evaluate our decision of demoting clean superpages upon writing, as discussed in Section 4.7, we coded a program that maps a 100MB file, reads every page thus triggering superpage promotion, then writes into every 512^{th} page, flushes the file and exits. We compared the running time of the process both with and without demoting on writing. As expected, since the I/O volume is 512 times larger, the performance penalty of not demoting is huge: a factor of more than 20.

Our design decision may deny the benefits of superpages to processes that do not write to all of the base pages of a potential superpage. However, according to our policy, we choose to pay that price in order to keep the degradation in pathological cases low.

6.8 Scalability

If the historical tendencies of decreasing relative TLB coverage and increasing working set sizes continue, then

to keep TLB miss overhead low, support for superpages much larger than 4MB will be needed in the future. Some processors like the Itanium and the Sparc64-III provide 128MB and larger superpages, and our superpage system is designed to scale to such sizes. However, architectural peculiarities may pose some obstacles.

Most operations in our implementation are either $O(1)$; or $O(S)$, where S is the number of distinct superpage sizes; or in the case of preempting a reservation, $O(S*R)$, where R is the ratio between consecutive sizes, which is never more than 8 on modern processors. The exceptions are four routines with running time linear in the size (in base pages) of the superpage that they operate on. One is the page daemon that scans pages; since it runs as a background process, it is not in the critical path of memory accesses. The other three routines are promotion, demotion, and dirty/reference bit emulation. They operate on each page table entry in the superpage, and owe their unscalability to the hardware-defined PTE replication scheme described in Section 6.1.

Promotions and demotions: Often, under no memory pressure, pages are incrementally promoted early in a process's life and only demoted at program exit. In such case, the cost amortized over all pages used by the process is $O(S)$, which is negligible in all of our benchmarks. The only exception to this is the adversary experiment of Section 6.6, which pays a 7.2% overhead due to incremental promotions and demotions. However, when there is memory pressure, demotions and repromotions may happen several times in a process's life (as described in Sections 4.6 and 4.7). The cost of such operations may become significant for very large superpages, given the linear cost of PTE replication.

Dirty/reference bit emulation: In many processors, including the Alpha, dirty and reference bits must be emulated by the operating system. This emulation is done by protecting the page so that the first write or reference triggers a software trap. The trap handler registers in the OS structures that the page is dirty or referenced, and resets the page protection. For large superpages, setting and resetting protection can be expensive if PTE replication is required, as it must be done for every base page.

These problems motivate the need for more superpage-friendly page table structures, whether they are defined by the hardware or the OS, in order to scalably support very large superpages. *Clustered page tables* proposed by Talluri et al. [21] represent one step in this direction.

7 Conclusions

This paper provides a transparent and effective solution to the problem of superpage management in operating systems. Superpages are physical pages of large size, which may be used to increase TLB coverage, reduce TLB misses, and thus improve application performance. We describe a practical design and demonstrate that it can be integrated into an existing general-purpose operating system. We evaluate the system on a range of real workloads and benchmarks, observe performance benefits of 30% to 60% in several cases, and show that the system is robust even in pathological cases. These benefits are sustained under complex workload conditions and memory pressure, and the overheads are small.

Acknowledgments

We wish to thank our shepherd Greg Ganger and the anonymous referees for their helpful comments. This work was supported in part by NSF grant CCR-98110603, Texas ATP grant 003604-0150-1999, and equipment donations from Compaq WRL and HP Labs. Juan Navarro was supported in part by a USENIX Student Research Grant.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

[2] D. W. Clark and J. S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions Computer Systems*, 3(1):31–62, Feb. 1985.

[3] Z. Fang, L. Zhang, J. Carter, S. McKee, and W. Hsieh. Reevaluating online superpage promotion with hardware support. In *Proceedings of the 7th International IEEE Symposium on High Performance Computer Architecture*, Monterrey, Mexico, Jan. 2001.

[4] FIPS 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), National Institute of Standards and Technology, US Department of Commerce, Washington D.C., Apr. 1995.

[5] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics,*

Speech, and Signal Processing, volume 3, Seattle, WA, May 1998.

[6] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

[7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[8] Imagemagick. http://www.imagemagick.org.

[9] G. B. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Marina del Rey, CA, June 2002.

[10] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Upper Saddle River, NJ, 1992.

[11] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, Apr. 1992.

[12] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. Virtual memory support for multiple page sizes. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993.

[13] J. C. Mogul. Big memories on the desktop. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993.

[14] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[15] J. Poskanzer. thttpd – tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd/.

[16] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita, Italy, June 1995.

[17] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[18] R. L. Sites and R. T. Witek. *Alpha Architecture Reference Manual*. Digital Press, Boston, MA, 1998.

[19] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

[20] 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*, San Jose, CA, Oct. 1994.

[21] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[22] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[23] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, 12(3):175–205, Aug. 1994.

[24] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. Taylor, R. H. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, 1986. ACM.