RICE UNIVERSITY

Transparent operating system support for superpages

by

Juan E. Navarro

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Peter Druschel, Chairman
Professor
Computer Science

Alan Cox
Associate Professor
Computer Science

Edward Knightly
Associate Professor
Electrical and Computer Engineering and
Computer Science

Houston, Texas

April, 2004

To the memory of my friends Alvaro Campos and Javier Pinto.

Transparent operating system support for superpages

Juan E. Navarro

Abstract

This dissertation presents the design, implementation and evaluation of a physical memory management system that allows applications to transparently benefit from superpages. The benefit consists of fewer TLB misses and the consequent performance improvement, which is shown to be significant.

The size of main memory in workstations has been growing exponentially over the past decade. As a cause or consequence, the working set size of typical applications has been increasing at a similar rate. In contrast, the TLB size has remained small because it is usually fully associative and its access time must be kept low since it is in the critical path of every memory access. As a result, the relative TLB coverage — that is, the fraction of main memory that can be mapped without incurring TLB misses — has decreased by a factor of 100 in the last 10 years.

Because of this disparity, many modern applications incur a large number of TLB misses, degrading performance by as much as 30% to 60%, as opposed to the 4-5% degradation reported in the 80's or the 5-10% reported in the 90's.

To increase the TLB coverage without increasing the TLB size, most modern processors support memory pages of large sizes, called *superpages*. Since each superpage requires only one entry in the TLB to map a large region of memory, superpages can dramatically increase TLB coverage and consequently improve performance.

However, supporting superpages poses several challenges to the operating system, in terms of superpage allocation, promotion trade-offs, and fragmentation control. This dissertation analyzes these issues and presents a design of an effective superpage management

system. An evaluation of the design is conducted through a prototype implementation for the Alpha CPU, showing substantial and sustained performance benefits. The design is then validated and further refined through an implementation for the Itanium processor.

The main contribution of this work is that it offers a complete and practical solution for transparently providing superpages to applications. It is complete because it tackles all the issues and trade-offs in realizing the potential of superpages. It is practical because it can be implemented with localized changes to the memory management subsystem and it minimizes the negative impact that could be observed in pathological cases. It can therefore be readily integrated into any general-purpose operating system.

Acknowledgments

I would like to thank my advisor Peter Druschel for his guidance and support, but mostly for pushing me hard enough to get me through the finish line. I would also like to give thanks to Alan Cox for the time he spent advising me, and Ed Knightly for his insightful comments both regarding this thesis and that elusive windsurfing move, the carving jibe.

I owe much to my fellow student in this endeavor, Sitaram Iyer, who taught me a plethora of ugly and dirty hacks that made my life easier. I keep fond memories of all those allnighters, in spite of his tantrums due to the lack of sleep.

I would also like to thank Amit Saha who unselfishly managed to ship a 90-pound Itanium server to California so that I could finish my experiments, and all those who, before that, where available as meta-rebooters, that is, as rebooters of the not-very-reliable remote-reboot device I acquired in order not to have to ask people to reboot my hacked machine after leaving to California.

Thanks also to Dan Wallach for teaching me how to pronounce *contiguity* — an indispensable word to describe this work — and to the staff of the Department, always willing to help.

Last but far from least, I want to express my gratitude to three girls who do not have the faintest idea what a superpage is, but without whose support along all these years this work would not have been possible: Paula, Ada and Kathy.

Contents

Figures

Tables

Chapter 1

Introduction

Virtual memory was invented more than 40 years ago at the University of Manchester for the Atlas Computer [26]. The goal of its inventors was to simplify the task of the programmer by providing the abstraction of an address space much larger than physical memory. Virtual memory systems transparently decide what portions of the address space are kept in physical memory and what portions are kept on disk or *backing store*. The programmer is thus freed from the time-consuming task of explicitly managing the transfer of data back and forth between disk and physical memory for programs with memory requirements that exceed the physical memory of the machine. The goal of the Atlas Computer designers is reflected in the name that they used for this technique: automatic use of a backing store.

Virtual memory is based on a level of indirection that separates the notion of *address* (the identifiers that processes use to refer to memory cells) from *physical location* (the actual location of the cell in the memory banks) [19]. This indirection has proved powerful enough to solve a variety of memory management issues beyond the main goal of the Atlas team. It allows for both protection and controlled sharing of memory among different entities (processes and operating system), it enables optimizations such as demand paging, copy-on-write, zero-on-demand and dynamic linking, and it simplifies memory management by providing *artificial contiguity* [68]. Virtual memory can also be used by applications [1] to implement a variety of techniques, such as distributed shared memory [54, 67], garbage collection [17], and concurrent checkpointing [55]. Hennessy and Patterson point out that “the only computers today without virtual memory are a few supercomputers and older personal computers” [33]. Denning calls the pervasiveness of virtual memory “one of the engineering triumphs of the computer age” [20].

The indirection required by virtual memory is achieved through an address translation mechanism which must be performed on every memory access. To speedup this translation, processors cache virtual-to-physical-address mappings in the *translation lookaside buffer* or TLB, sometimes also called *translation buffer* or *address translation cache* [46, 52, 59]. *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 [18] can incur many TLB misses and suffer from a significant performance penalty. Recent research findings on the TLB performance of modern applications state that TLB misses are becoming increasingly performance critical [45]. 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 can be dramatically increased to cover the working set of most applications. Better TLB coverage results in performance improvements of over 30% in many cases, as demonstrated in Sections 5.2 and 6.4.

However, inappropriate use of large superpages can result in enlarged application footprints because of internal fragmentation, 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 [28, 85, 87].

This dissertation develops a general and transparent superpage management system. It balances various trade-offs 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, the 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. A complementary idle-loop defragmentation mechanism based on page migration is also proposed and evaluated.

A prototype of this system is implemented in FreeBSD for two dissimilar architectures, namely Alpha and IA-64, 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 five-fold. It extends a previous 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; 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; and it presents a detailed design and evaluation of superpage-oriented memory compaction as a defragmentation mechanism.

Chapter 2 provides some background and motivation for this work, and establishes the constraints and complexities of the problem. Chapter 3 examines related work not only on superpages, but also on the broader area of TLB overhead reduction. Chapter 4 describes

the design and implementation of a superpage system. Chapter 5 presents the results of an evaluation in the Alpha processor. Chapter 6 refines the design under the light of the results obtained for the IA-64 architecture. Chapter 7 proposes and evaluates memory compaction as a complementary mechanism for reducing fragmentation. Finally, Chapter 8 concludes this dissertation.

Chapter 2

Background

Main memory has grown exponentially in size over at least the last decade and, as cause or consequence, the memory requirements of applications have also increased [87]. Mauro and McDougall state that this increase is as much as 100% per year [57]. In contrast, TLB coverage has lagged behind.

Since the TLB is in the critical path of every memory access, a fundamental requirement is low access time [60]. In turn, low access time in a complex device that is usually multi-ported and fully-associative can only be achieved with a small number of entries. Hence, TLB size has remained relatively small, usually 128 or fewer entries, corresponding to a megabyte or less of TLB coverage. Figure 2.1 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.

In addition to a larger number of TLB misses due to lack of TLB coverage, technological trends have also led to an increase in the cost of each miss [86]. The main component of the TLB miss penalty is memory accesses to traverse the page table. The increasing gap between processor and memory speed [34] makes this component relatively more expensive. Aggravating this situation, 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 several accesses to the memory banks to find a translation for data that is already in the cache, increasing the miss penalty even more.

Section 5.3 shows that for many real applications, TLB misses degrade performance by

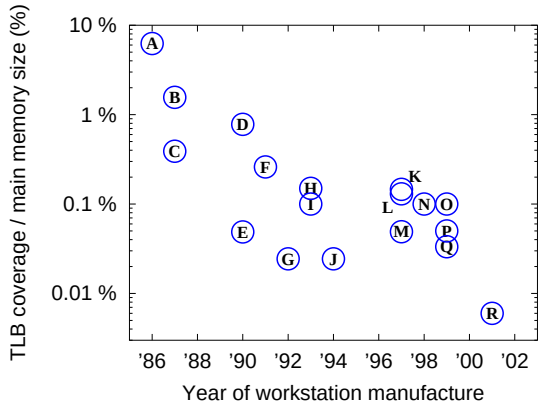


Figure 2.1 : TLB coverage as percentage of main memory for workstations, 1986-2001 (data collected from various websites). (A) Sun 3/50; (B) Sun 3/180; (C) Sun 3/280; (D) Personal Iris; (E) SPARCstation-5; (F) Iris Indigo; (G) SPARCstation-10; (H) Indy; (I) Indigo2; (J) SPARCstation-20; (K) Ultra-1; (L) Ultra-2; (M) O2; (N) Ultra-5; (O) Ultra-10; (P) Ultra-60; (Q) Ultra-450; (R) Octane2.

as much as 30% to 60%, contrasting to the 4% to 5% reported in the 1980's [16, 98] or the 5% to 10% reported in the 1990's [72, 93].

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 a premature onset of memory pressure. Also, the I/O demands become higher due to increased paging granularity. Tallury shows that an increase from 4KB to 64KB pages can make some applications to double their working set size [89].

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 chapter.

2.1 Hardware-imposed constraints

The design of TLB hardware in most processors establishes a series of constraints on superpages, akin to the constraints imposed on normal pages. Firstly, the superpage size is

always a power of 2 and 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 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, it must be *naturally aligned*, meaning that 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 attributes. The latter implies that all base pages that form a superpage must have the same read, write, and execute attributes. 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.

Upon a TLB miss for an address in a superpage, the processor loads the translation entry for the requested address, including the superpage size, from the page tables into the TLB. Since superpages are contiguous, and since the TLB entries contain the superpage size, this entry suffices to translate any subsequent address in the superpage. The page tables must therefore contain information about what superpage size must be used for each address mapped by a process. Many architectures use a page table structure with one entry per base page, which contains a page size field to indicate the superpage size.

Adding superpage support to an existing architecture is thus straightforward: a page size field must be added to each TLB entry, page tables must also be augmented with page size information, and the comparator in the TLB must be augmented to make use of the superpage size to determine how many of the most significant bits of an address identify a page. All mainstream processors today support superpages [43]: Alpha, IA-32, IA-64, MIPS, PA-RISC, PowerPC and UltraSPARC.

2.2 Issues and trade-offs

The task of managing superpages can be conceptually broken down into a series of steps, each governed by a different set of trade-offs. 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. 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.

2.2.1 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 in a system under load.

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 2.2 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.

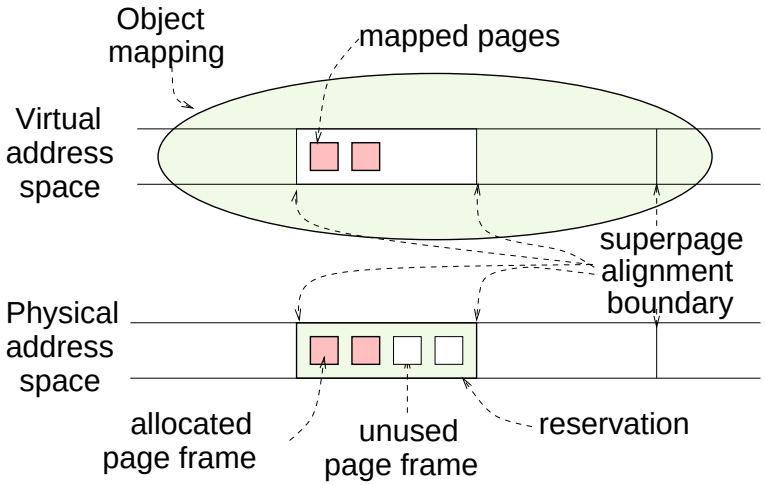


Figure 2.2 : Reservation-based allocation.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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 trade-offs; other researchers have also alluded to some of these issues [49, 60]. The next chapter describes previous approaches to the problem, and Chapter 4 describes how our design effectively tackles all these issues.

Chapter 3

Related work

There is a large body of literature on virtual memory. Smith's early bibliography on the topic comprises 333 publications [81]. However, the first works that study TLB performance appear in the early 1980's, and conclude that the TLB overhead for typical applications of that time is acceptable at 4 to 5% [16, 73, 98]. Only in the 1990's, TLB overhead starts to be a concern for researchers who observe that it can now reach 10% [72, 93]. More recent studies show that the TLB nowadays plays a critical role in performance [45, 62].

This chapter describes approaches that have been proposed to reduce the TLB overhead. The spectrum goes from hardware approaches that are fully transparent at the architecture level to 100% software mechanisms. Some techniques aim at reducing the number of TLB misses, others try to reduce the overhead of each miss, others attempt to reduce or hide the TLB access time, yet others propose to dispense with the TLB altogether. Superpage-based approaches belong to the first category and are described last.

Two other techniques that are related to superpages, namely page coloring and memory compaction, are also discussed.

3.1 Reducing TLB access time

Many modern processors have two-level TLBs, but their use goes as far back as 1984 with the MicroVAX [23]. The first level TLB or *micro-TLB* is faster thanks to a small number of entries, and is usually invisible outside the processor. The Itanium processor, for instance, has a 32-entry first-level TLB and a 96-entry second-level TLB for data.

Taylor et al. describe the MIPS R6000's *TLB slice* [91], which is a very small (four lines wide) and fast first-level TLB. The TLB slice is a direct-mapped translation cache

that maps a few of the least significant bits of the virtual page number to the few bits in the physical address that are required beyond the page offset to address a physically mapped cache.

The output of the TLB slice is concatenated with the page offset to form the physical index that is fed to a virtually tagged cache. Thus, the R6000 has a very uncommon physically-indexed and virtually-tagged cache. The small number of entries in the TLB slice — sixteen — is made effective by interposing a primary virtually-mapped cache between the processor and the TLB slice; hits in the primary cache do not require an address translation. A full-width translation is required on a secondary cache miss, which complicates the cache miss sequence. The R6000 reserves a portion of the secondary cache to be used as a second level TLB with 4096 full entries.

Virtually-addressed and physically-tagged first-level caches are used in many architectures to mask the TLB access time by initiating the cache access in parallel with the address translation [94]. Virtual address caches that are larger than the page size times the cache associativity introduce consistency issues due to *synonyms* [82]: when a page is mapped multiple times at virtual addresses that do not map to the same cache lines, multiple copies of the same data will be present at different locations in the cache. For this reason, unless complex logic is added to the hardware [30, 96], the OS must be involved in the management of virtual address caches by either taking corrective [40] or preventive actions [14, 53]

Another way to hide the TLB latency is by predicting the addresses of memory accesses to initiate a speculative cache load earlier in the processor pipeline [2].

Chiueh and Katz propose a mechanism to achieve with physically-addressed caches the benefits of parallel address translation that virtual caches provide [15]. The idea is to restrict the virtual-to-physical mappings in a way such that all the least significant bits that are required for a cache access are identical in corresponding virtual and physical addresses. This approach is similar to superpages in that the OS must collaborate to allocate frames according to hardware-imposed restrictions. An important difference is that while the restrictions are weaker than for superpages, the system will not operate correctly if

ignored. Chiueh and Katz's work focuses on evaluating the potential performance gains rather than on operating systems issues.

In the same paper, Chiueh and Katz propose a transparent mechanism that bypasses the TLB access for all but the first of a sequence of accesses to the same page. This approach, called *lazy address translation* is only applicable to architectures in which memory accesses are always expressed by means of a base register and an offset; the processor keeps a hint associated with each base register for the last translation.

Discussion

Reducing TLB access time and using superpages are complementary techniques, since they aim at reducing different components of the total TLB overhead. They target, however, disjoint sets of applications. Applications with working sets larger than the TLB coverage are unlikely to benefit from smaller TLB access times, since the largest performance penalty is due to TLB misses.

3.2 Reducing cost of TLB misses

Huck and Hays present the *hashed page table*, a translation table that combines the flexibility of software TLB miss handling with the efficiency of hardware handling [37]. The hashed page table is essentially a cache of page table entries that is maintained by the OS and is accessed by a hardware state machine on a TLB miss. Only if the required entry is not present in the cache, the miss handling is transfered to the OS. This approach has been adopted by many architectures, including PA-RISC, Sparc64, Enhanced PowerPC and IA-64.

Bala et al. suggest that a cache of page table entries is also effective in reducing the TLB miss penalty on architectures with software-only miss handling. They also apply software prefetching techniques for TLB entries in order to reduce the number of kernel TLB misses [4].

Discussion

While these techniques have the same high-level goal as superpages, which is to reduce the TLB miss overhead, they can be considered complementary. In fact, architectures that use hash page tables also provide superpage support.

When applied simultaneously, there will be fewer TLB misses due to superpages, and each miss will cost less if the required entry is in the page table cache. Note, however, that the applications that benefit from a page table cache are a superset of those that benefit from superpages: applications with non-uniform memory mappings may not create superpages but can still take advantage of a page-table cache.

3.3 In-cache address translation: the no-TLB school

Virtual caches that are also virtually tagged only require to perform address translations on a cache miss or flush. However, permission checking still needs to be done upon every memory access. Wood et al. propose to move all the functionality of the TLB to the cache [98]. In particular, this approach requires reference, dirty and protection bits to be added to each cache line.

Jacob and Mudge present a similar no-TLB approach [42], but they go one step further by handling address translations in software; thus, every cache miss traps to the OS.

Discussion

Naturally, in an architecture with no TLB there will be no TLB miss overhead. Consequently, the techniques presented in this subsection are mutually exclusive with superpages.

While eliminating the TLB is attractive, in-cache address translation is not free from disadvantages. Not only does it make caches more complex and cache misses more expensive, but it also has the aforementioned drawbacks of virtual caches.

3.4 Reducing the number of TLB misses with superpages

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 thesis. 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.

3.4.1 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 [87].

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 3.4.3. Their design considers only one superpage size, and the effects of external memory fragmentation is not taken into account in this study.

They use this mechanism to evaluate different TLB organizations, and for a conventional superpage TLB that supports 64KB superpages they report TLB miss reductions from 0.7% to 99.9% on ten benchmarks, mostly from the SPEC 92 suite. The evaluation methodology is based on a trap-based simulator (an execution-driven simulation that only calls the simulator on TLB misses) that does not account for kernel references.

In contrast, superpages in both the HP-UX [85] and IRIX [28] 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 scheme with page size hints is that it is not transparent. Although it is not necessary to modify or recompile applications, 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.

Subramanian et al. use seven benchmarks to evaluate HP-UX's support for superpages [85]. Three of the test cases are synthetic benchmarks that simulate the memory reference behaviour of real applications, but the computation performed by the applications is not taken into account, making the speedup around 2.5 unrealistic for those specific cases. For the other applications, mostly from the the SPEC 95 integer benchmark set, they report improvements from 17% to 34%. For each benchmark they set the page size hint to one of the seven supported page sizes that range from 4KB to 16MB; the best page size (the size that maximizes performance without unduly increasing memory usage) is found to vary between 16KB and 4MB, depending on the benchmark.

Similarly, Ganapathy and Schimmel use three integer applications from SPEC 95 and two from the NAS parallel suite to evaluate IRIX's support for superpages [28]. Experimenting with page size hints of 16KB, 64KB, 256KB and 1MB, they obtain performance improvements in the range 10-20% and also show that the best size is application-dependent.

3.4.2 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. Approaches based on relocation can be entirely and transparently implemented in the hardware-dependent layer of the operating system, but then 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 on-line cost-benefit analysis to determine when the benefits of superpages outweigh the overhead of superpage promotion through relocation [71]. 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.

Using trace-driven simulation and a set of ten benchmarks mostly from the SPEC 92 suite, Romer et al. report results that go from a 0.8% of performance degradation to a 92% of performance improvement when all power of two page sizes from 4KB to 8MB are supported. Nevertheless, their simulations do not take into account the cache pollution that relocation produces, and they use a copying cost that is significantly smaller than that in real systems [24].

In the absence of memory contention, this approach has 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 against fragmentation.

3.4.3 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 [87]. By means of trap-based simulations they obtain TLB miss reductions from 3.7% to 99.8% on 10 benchmarks mostly from the SPEC 92 suite, with a partial-subblock TLB that can map 4KB base pages and 64KB superpages with holes.

Partial-subblock TLBs yield only moderately larger TLB coverage than the base system, and it is not clear how to extend them 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 [24].

Using execution-driven simulations, for a 128-entry TLB Fang et al. report performance degradation around 5% in two SPEC 95 benchmarks, and 2% to 101% improvements in other six benchmarks. They compare their approach to Romer et al.’s relocation mechanism, for which they obtained slowdowns of up to 17% in five of the benchmarks, and improvements ranging from 1% to 65% in the other three.

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

3.4.4 Discussion

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, and uses spare CPU cycles to compact memory. It also tackles the issues of demotion and eviction (described in Section 2.2) not addressed by previous work, and does not require special hardware support.

3.5 Page coloring

A technique that has nothing to do with TLBs but is related to superpages is *page coloring* [47, 56].

In physically-addressed caches, pages that have contiguous virtual addresses do not necessarily map to consecutive location in the cache. Since virtually contiguous pages are likely to be accessed together, it is desirable to prevent cache conflicts among them. By carefully selecting among free frames when mapping a page, the OS can prevent these conflicts. This technique is usually implemented by keeping the free frames, according to the least significant bits of their physical addresses, in bins of different *colors*. On a page allocation, a frame of the proper color is picked, depending on the color of the virtual neighbors of the new page.

Bershad et al. present performance-oriented page coloring schemes in which they dynamically relocate pages when too many cache misses are observed, both with special hardware support [7] and on standard hardware [70].

Discussion

In a way, superpages imply a very restricted form of page coloring, in which the color bins have cardinality one when they are not empty. Nevertheless, although page coloring generally improves performance, the main motivation is to reduce run time *variance*.

3.6 Memory compaction

Early systems that implemented virtual memory using a pure segmentation approach were exposed to external fragmentation [19]. Memory compaction is often cited — especially in textbooks [78, 84, 90] — as a mechanism to fight this problem, but very rarely applied in practice.

Haddon and Waite describe what seems to be the first memory compactor in history: a simple algorithm used on the English Electric KDF 9 computer to fully compact memory

when the allocator failed due to external fragmentation [31]. They assert that the program "takes 2.87 seconds to compact a 10000-word store in the worst case".

The designers of the Burroughs D-825 system [95] had planned to use compaction but later dropped the plan, somewhat surprised by the good performance of their best-fit allocation algorithm [77]. Knuth later found through simulations that when a first-fit allocation algorithm is unable to fulfill a request due to external fragmentation, memory is close to full utilization anyway [51].

Balkovich et al. develop a probabilistic model to determine the conditions under which it pays off to perform memory compaction (or repacking, as they call it) [5].

Since modern systems use paging or paged segmentation to implement virtual memory, memory compaction in operating systems has not been considered recently until the introduction of superpages. Note that paged virtual memory systems are not entirely free from external fragmentation, since some device drivers and kernel loadable modules still require to allocate contiguous physical memory chunks larger than a page. However, this allocations are usually performed during start-up, when fragmentation is low.

Superpages reintroduce the problem of external fragmentation in paged virtual memory. The scenario is different from the one in Knuth's simulations in that an allocated chunk is not necessarily deallocated at once: any subregion of the chunk can be deallocated by means of eviction or unmapping. This condition makes fragmentation a more serious problem, since the allocator can fail when memory is far from full utilization [62].

In the context of superpages, there is only one paper to date that proposes and describes a memory compaction mechanism, which is in use in the IRIX operating system [28]. Compaction is performed by a background task, the coalescing daemon, which tries to keep some unspecified contiguity metric above a watermark. To achieve its goal, the coalescing daemon scans superpage-sized memory chunks and vacates them if the number of occupied pages in the chunk is below a threshold. Thresholds for each size were "chosen by experimentation". Apparently, the coalescing daemon is not aware of existing superpages, and thus it might break them while migrating its constituent pages. If after several passes the

daemon has not been able to recover enough contiguity, then it becomes more aggressive by ignoring the thresholds and always vacating the scanned chunk. Neither the effectiveness of the coalescing daemon nor the impact it might have in foreground processes are reported in the paper.

Discussion

In this thesis we propose and evaluate a novel superpage-oriented memory compaction mechanism. It is different from IRIX's coalescing daemon in that (1) it is meant to run in the idle loop, thus minimizing the impact on other processes; (2) it is superpage- and reservation- aware in the sense that it relocates, if necessary, existing superpages and reservations as a block, without breaking them; and (3) it performs a cost/benefit analysis in order to maximize the profit of every cycle spent in compaction.

Chapter 4

Design

Our design adopts the reservation-based superpage management paradigm introduced in [87]. 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 Chapter 5, this combination of techniques is general enough to work efficiently for a range of realistic workloads, and thus 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 [65]. A multi-list reservation scheme is used to track partially used memory reservations, and to help in choosing reservations for preemption. A population map keeps track of memory allocations in each memory object. 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. The following sections elaborate on these concepts.

4.1 Reservation-based allocation

Most operating systems allocate physical memory upon 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 an opportunistic 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 An opportunistic policy for the preferred superpage size

This section describes the policy used to choose the desired superpage size during allocation. Since this decision is usually made early in the life of a process, 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, we use an opportunistic policy that 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 waste 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.

Note that, in the absence of memory fragmentation, this policy provides optimal TLB coverage for fixed-sized objects. For dynamically-sized objects, coverage is not optimal because the policy starts assigning small superpages to objects that eventually grow to a large size: an object of size N that could be covered by only one superpage will usually require $O(\log N)$ superpages.

If there are no contiguous extents of memory of the preferred size, then the reservation is made as large as possible. The consequence of this policy is that, when there is not enough contiguity to satisfy all the requests for the preferred size, then the largest superpages are likely to be created in the regions that are first touched by a process. This is not necessarily the place where they provide the most benefit, but the system lacks the information required to optimize the assignment of limited contiguity to regions of the address space. The system also does not try to reserve contiguity for future processes that may obtain larger benefits.

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 following key observation: useful reservations are often populated quickly, and reservations that have not experienced any recent allocations are less likely to be fully allocated in the near future.

We observed that this behaviour is common by examining memory traces of typical applications. Figures 4.1 to 4.4 show the memory traces of four programs, FFTW, GNU linker, SP and bzip2, where the first touch (i.e., allocation) to any given page is marked with a larger and darker dot than subsequent references. Refer to Section 5.2 for a description of the programs.

To make the amount of data produced by the tracer manageable, we took a statistical sample of the references, except for the first touch to each page. Also, we run FFTW, SP and bzip2 with smaller data sets than what we used for the evaluation in Chapter 5, and display only a small portion of the execution after the last first touch.

For FFTW in Figure 4.1, practically all of the memory is initialized in a sequential manner before the computation starts. The GNU linker in Figure 4.2 also allocates pages mostly in sequential order, but the allocation rate diminishes with time. SP and bzip2 in Figures 4.3 and 4.4 sequentially and quickly allocate about half of the memory; while the other half is not allocated sequentially, it is observed to be allocated quickly and with no "holes". This kind of behaviour supports our preemption policy: if a reservation has had any of its frames populated recently, then it is not a good candidate for preemption since it is likely that the other not-yet-populated frames get populated soon.

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

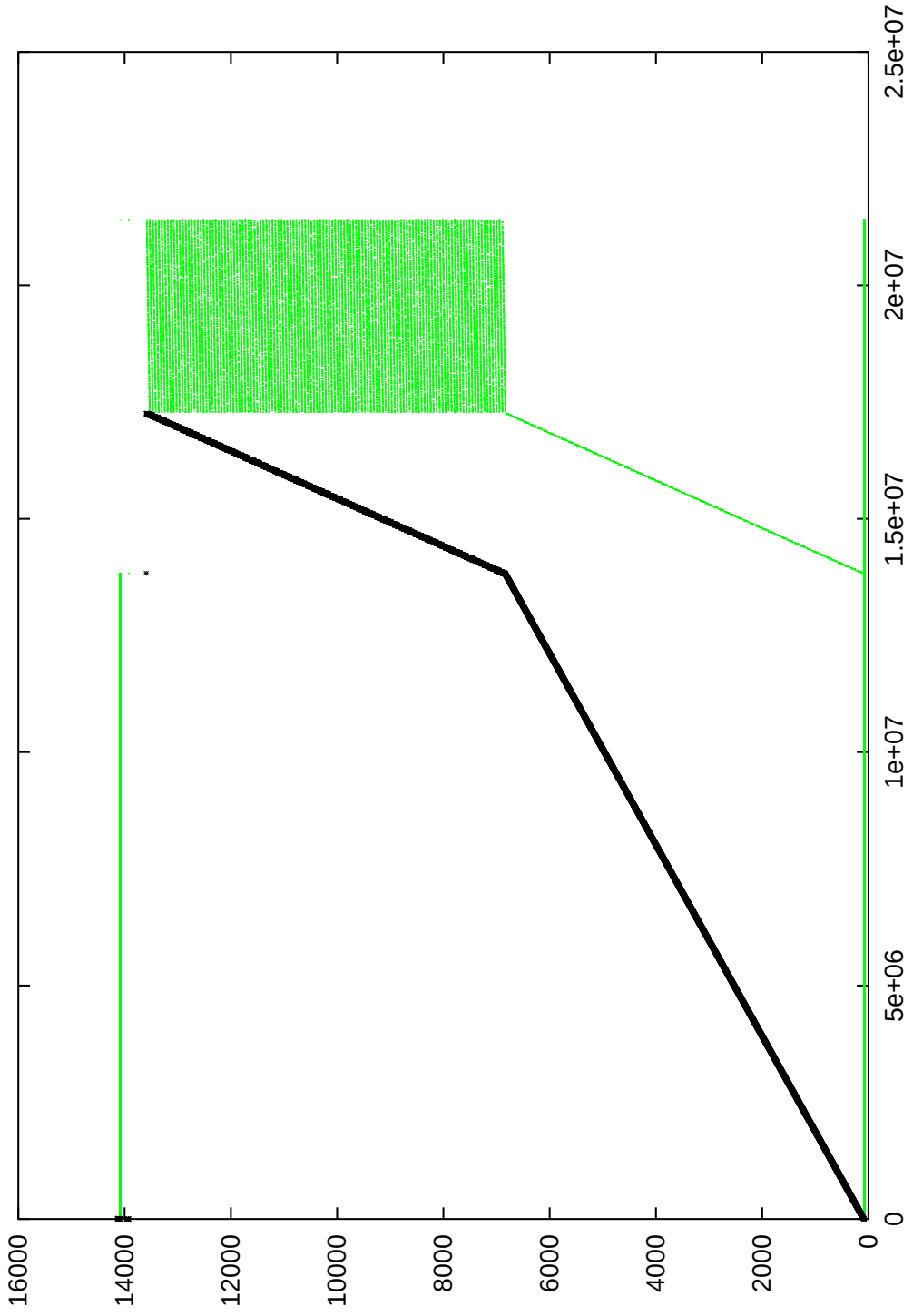


Figure 4.1 : Memory trace for FFTW. The vertical axis is the page id (for the data segment only), and the horizontal axis grows with time roughly at one tick each time that a different page is referenced. The first touch to any given page is marked with a larger and darker dot than subsequent references.

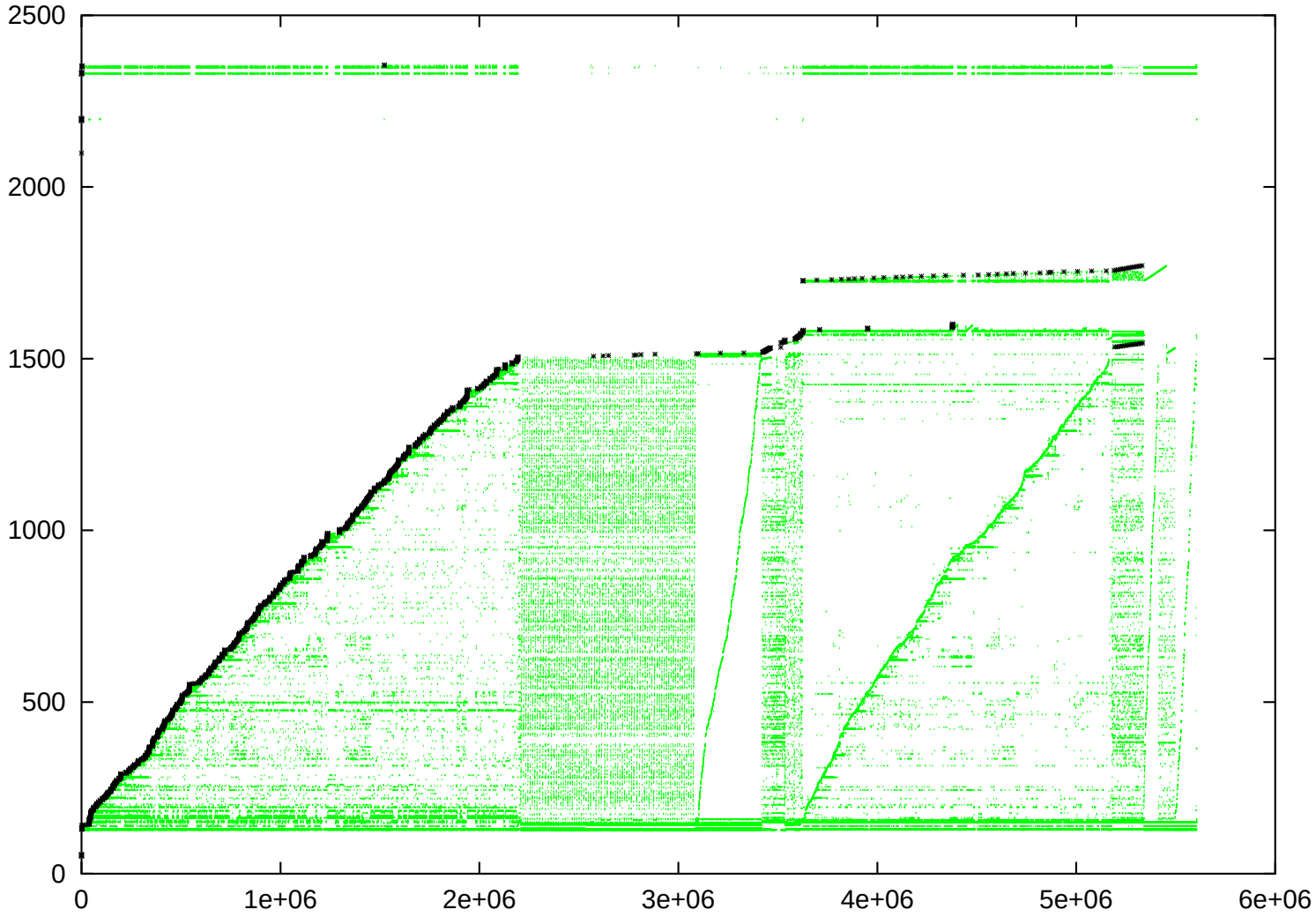


Figure 4.2 : Memory trace for the GNU linker.

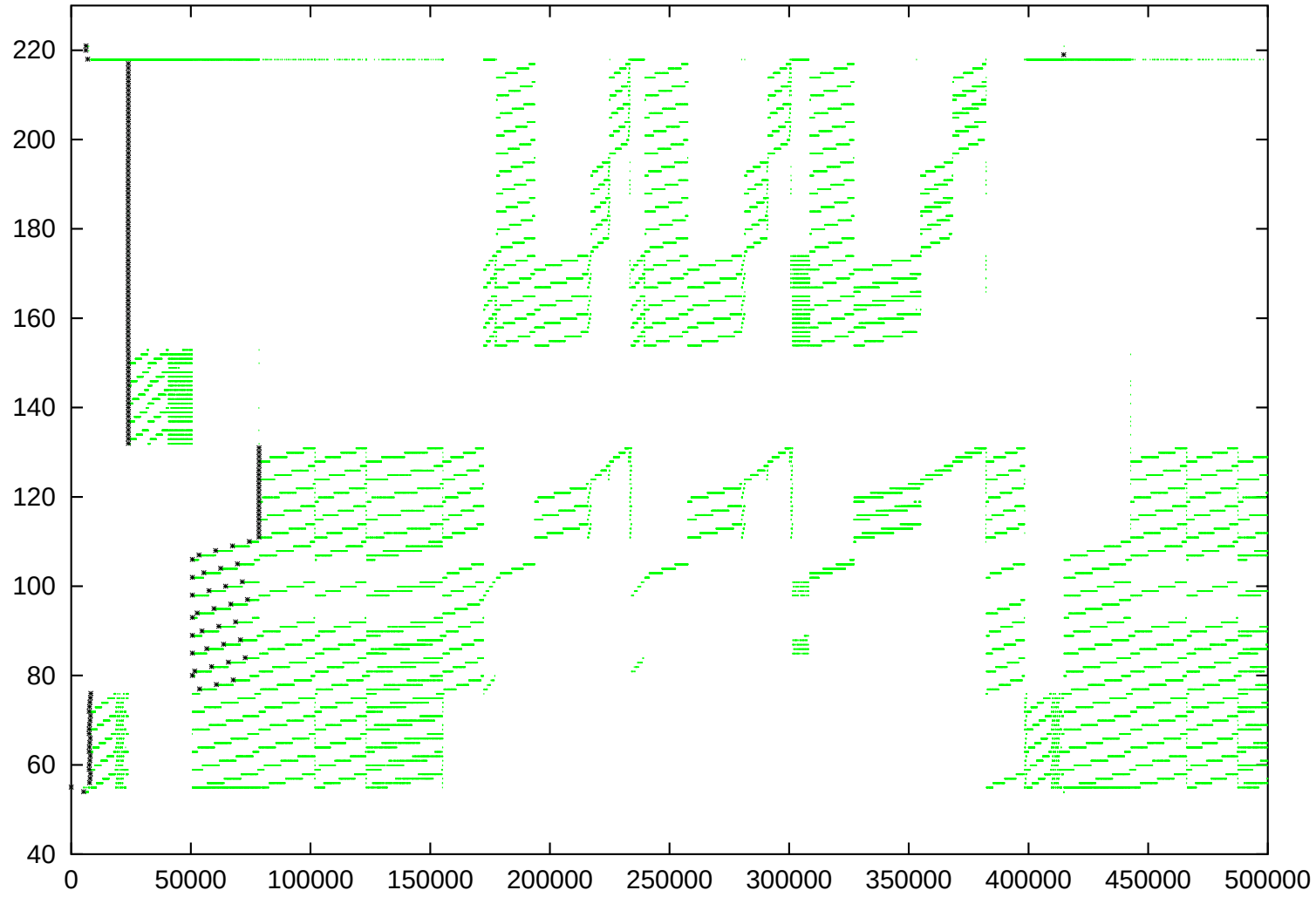


Figure 4.3 : Memory trace for SP.

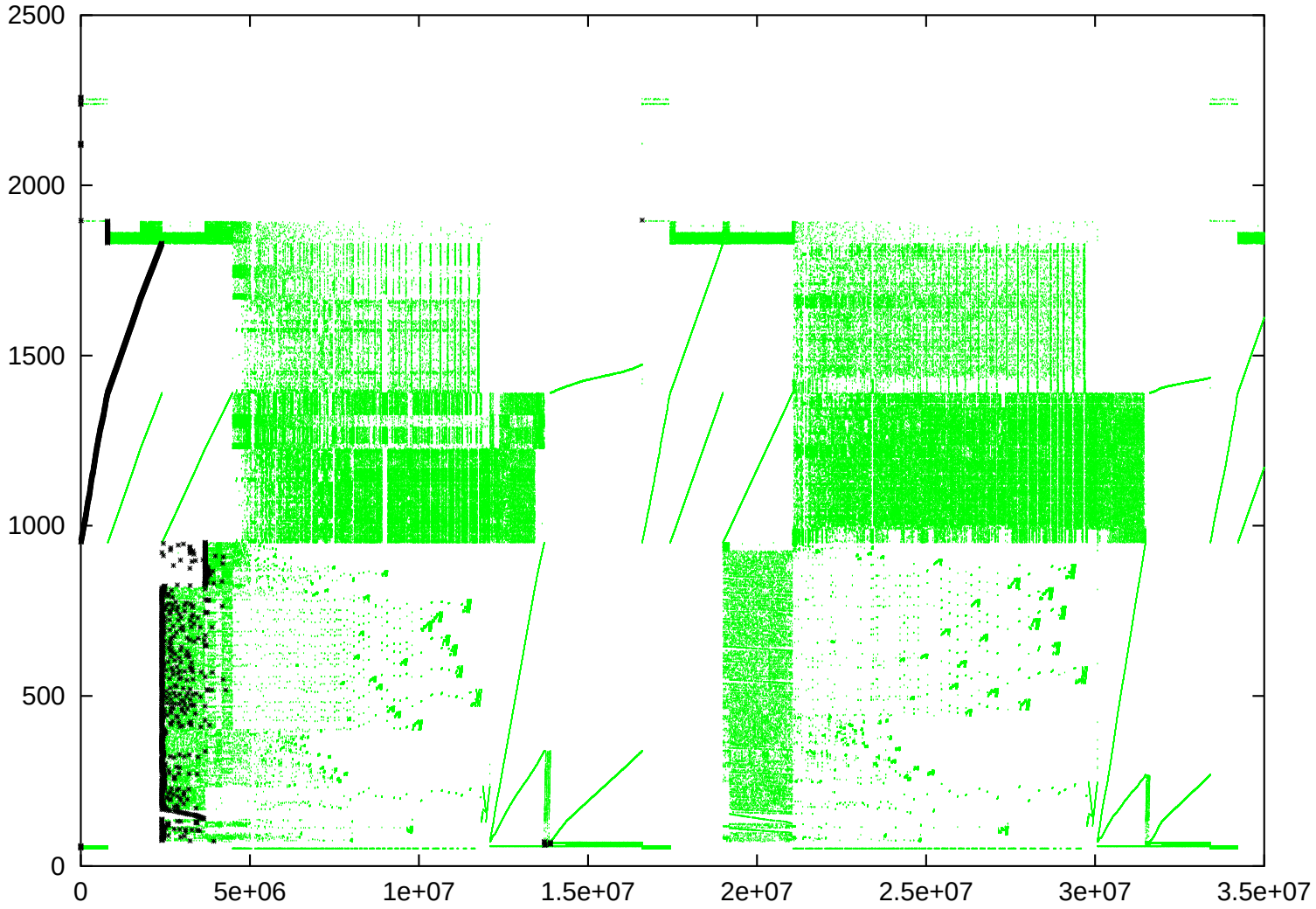


Figure 4.4 : Memory trace for bzip2.

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 4.10.1 discusses this in greater detail.

4.5 Incremental promotions

For promotions we also use an opportunistic policy that creates a superpage 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 5.7.

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.

This technique is considered safe because a 160-bit SHA-1 hash has a collision probability of about one in 2^{80} [25], which is much smaller than the probability of a hardware failure. This argument, however, sparked some debate [36]. In addition, preliminary microbenchmarks using SHA-1 reveal significant overhead, up to 15%, in disk-intensive applications. The pathological case of a large sequential read when the CPU is saturated incurs a worst-case degradation of 60%.

We did not explore this alternative any further, although we believe that these overheads can be reduced using a variety of optimizations. 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. In addition, 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.

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 into 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 the 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 resort.

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 4.5 shows one such tree.

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.

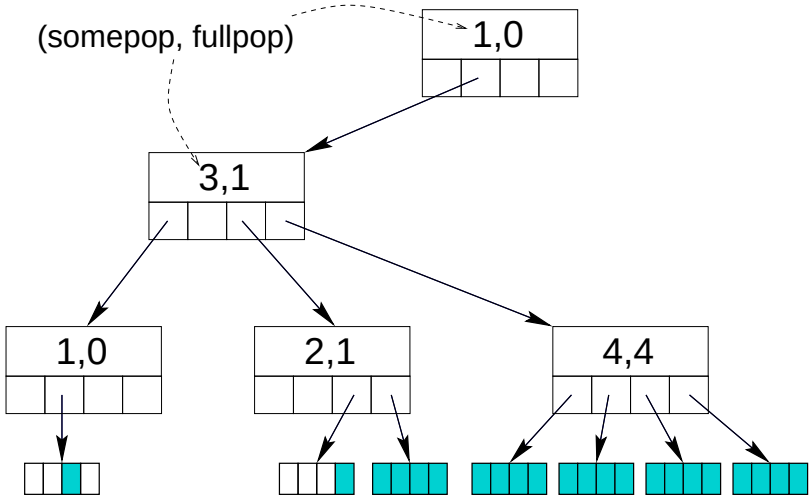


Figure 4.5 : A population map. At the base page level, the actual allocation of pages is shown.

Overlap avoidance: If the procedure above yields no reserved frame, then we try to make a reservation. The maximum size that does not overlap with existing 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.

4.10 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.

4.10.1 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 can thus be easily freed under memory pressure. Inactive pages are those that are mapped by some process, but 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 trade-off. Moreover, by judiciously selecting pages for replacement, it attempts to restore as much contiguity as possible by affecting as few pages as possible. Section 5.5 demonstrates the benefits of this design.

4.10.2 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.

4.10.3 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.

Chapter 5

Evaluation

This chapter 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.

5.1 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 [79].

5.2 Workloads

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

CINT2000: SPEC CPU2000 integer benchmark suite [35].

CFP2000: SPEC CPU2000 floating-point benchmark suite [35].

Web: The thttpd web server [66] version 2.20c 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 [39], version 5.3.9.

Povray: Ray tracing of a simple image using povray 3.1g.

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 [3], version 2.3. The input size corresponds to the "workstation class" in NAS's nomenclature.

FFTW: The Fastest Fourier Transform in the West [27], version 3.0, with a 200x200x200 matrix as input.

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

5.3 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 5.1 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.

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

Benchmark	Superpage usage 8 KB	64 KB	512 KB	4 MB	TLB miss reduction	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		99.33	1.036
parser	168	5	14	8	99.92	1.078
eon	297	6			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		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		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		95.65	1.360
Tree	207	6	14	1	97.14	1.503
SP	151	103	15		99.55	1.193
FFTW	163	13	7	60	99.59	1.549
Matrix	198	12	5	3	99.47	7.546

Table 5.1 : Speedups and superpage usage when memory is plentiful and unfragmented.

without the zeroed-page feature. We obtained an average penalty of 0.9%, and a maximum of 1.7%. These penalties represent an upper bound, since the buddy allocator, even though it ignores the hint, still has a chance of returning a zeroed-out page when requested.

A side effect of using superpages is that it subsumes page coloring [47], 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.

5.4 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 5.2 and 5.3 respectively present the speedup and TLB miss reduction for the benchmarks, excluding those that have the same speedup (within 5%) in all four cases.

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.

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 5.2 : Speedups with different superpage sizes.

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 5.3 : TLB miss reduction percentage with different superpage sizes.

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.

5.5 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 4.10.1. 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.

5.5.1 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 5.1 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%.

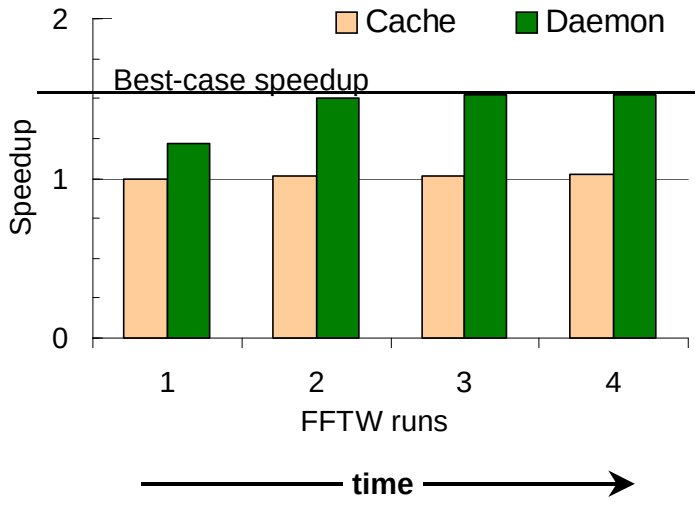


Figure 5.1 : Two techniques for fragmentation control.

The web server closes its files on exit, and our page daemon treats this file memory as inactive, as described in Section 4.10.1. 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 5.2 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.

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 con-

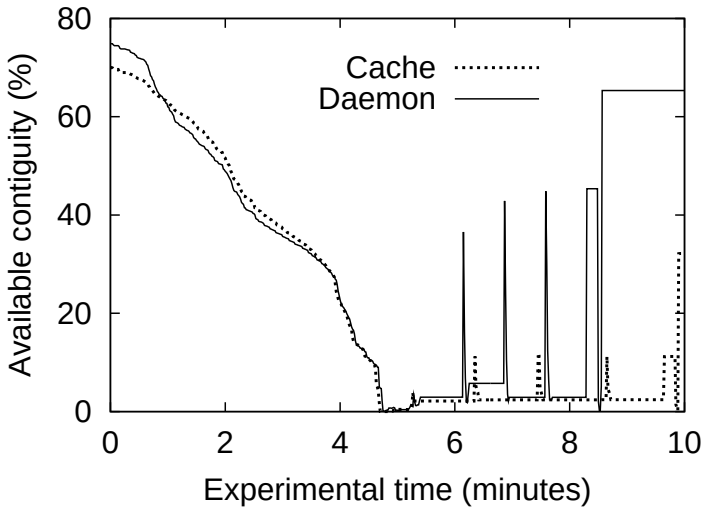


Figure 5.2 : Contiguity as a function of time.

trast, the daemon scheme is 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.

5.5.2 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, continuous 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.

5.6 Adversary applications

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

5.6.1 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 5.1, 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.

5.6.2 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%.

5.6.3 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 eight smaller chunks. One remains reserved with the page that made the original reservation; another is taken for the page being allocated, and six are returned to the free list. We measured a performance degradation of 1.1% for this process.

5.6.4 Overhead in practice

Finally, we measure the total overhead of our implementation in real scenarios. We use the same benchmarks of Section 5.2, 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%, which shows that our system only imposes negligible overhead in practice.

5.7 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.

5.8 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 usually 8 or less in 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 5.1.

5.8.1 Promotions and demotions

Often, under no memory pressure, pages are incrementally promoted early in the life of a process, 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 5.6.1, 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.

5.8.2 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 support very large superpages in a more scalable way. *Clustered page tables* proposed by Talluri et al. [88] represent one step in this direction.

Chapter 6

Validating the design in the IA-64 platform

The prototype system was ported to FreeBSD 5.0 for IA-64, the architecture to which the Itanium family of processors belong [61]. With this port to a more modern architecture than the Alpha, we expected to gain new insights into our approach. An interesting feature of IA-64 is its support for 10 pages sizes, from 4KB to 256MB, which allows us to put more stress on our system in order to better evaluate its overhead and scalability.

In addition, while IA-64 requires the presence of a software TLB handler (like the Alpha architecture), most of the misses are actually handled in hardware. Therefore, it is interesting to know whether this potentially more efficient handling translates to smaller benefits from using superpages.

In this chapter we describe the port of our system to the IA-64 architecture, then analyze the experiments that motivated a change in our design, and present results for our benchmarks with an implementation of the refined design.

6.1 Platform

For the experiments, we used an HP workstation i2000 with the following characteristics:

- Intel Itanium processor at 733 MHz;
- ten page sizes: 4KB, 8KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB and 256MB (FreeBSD for IA-64 uses 8KB as the base page size);
- 2 GB RAM, 4-way associative separate 16KB data and 16KB instruction L1 caches, 6-way associative 96KB unified L2 cache, 4-way associative 2MB unified L3 cache;

- fully associative, two-level data TLB with 32 entries in the first level and 96 in the second, and 64-entry one-level instruction TLB, also fully associative.

To handle TLB misses, Itanium processors use a hardware finite state machine to lookup the required translation entry in a hash table that resides in main memory and acts as a cache of the page table. Only if the entry is not found in this cache, a software handler must take over. Thus, most of the misses are handled in hardware with no pollution of the instruction cache and TLB, and no pipeline flushing.

This page table software cache is known as VHPT for Virtual Hash Page Table in Intel's nomenclature. Although the OS is free to use any page table organization to service VHPT misses, the VHPT is structured in a way that encourages the implementation of a hashed page table. The architecture also provides instructions to compute hashes and tags.

The VHPT is essentially an array in which each entry is a TLB entry augmented with a tag and enough space to store a pointer. On a TLB miss, the hardware state machine uses the virtual address to compute a hash index and a tag. It then retrieves the entry given by the hash index, and compares the tag in the entry with the computed tag. If they match, then the entry is used to fill the TLB; otherwise, a trap to the OS is triggered. FreeBSD follows the architectural guidelines, and thus uses the extra space in the VHPT entry to store a pointer to a collision chain, and keeps all the page table entries with colliding hash values in this chain. The size of the VHPT is configurable, and FreeBSD sets it so that there are as many VHPT entries as page frames in the physical memory of the machine.

Sparc64-III processors have a similar page table cache with a different name: the Translation Memory Buffer or TMB [83].

6.2 New insights

The design that we described for the Alpha architecture is based on a radix tree for population maps and a hash table for looking up reservations. Using large superpages and several superpage sizes exposed some problems of this design. The following subsections describe these problems.

6.2.1 Reserved frame lookup

As shown in section 4.9, reserved frame lookup for a given virtual address is performed by rounding the virtual address down to a multiple of the largest superpage size (e.g., 256MB), determining the page index within the object for the rounded address, and feeding the result along with the object identifier into a hash table to find the root of the population map's radix tree. From the root, the tree is traversed to locate the reserved page frame, if there is one.

Note that regardless of the size of the object, the radix tree has a depth equal to the number of superpage sizes supported, which is overkill for small objects.

6.2.2 Shared objects

The hash table approach based on rounded virtual addresses for looking up reserved frames produces also the following problem. Say that two processes map an object at virtual addresses a and b such that $b - a$ is not a multiple of the largest superpage size. Then the two processes will not see each others' reservations. As a consequence, these processes may produce overlapping reservations and unnecessarily prevent each other from building superpages.

To make the case more concrete, let us say that the base page size is 8KB, and that superpages of sizes 4 and 16 base pages are supported. Assume that process P_1 maps an object at virtual address 0, while process P_2 maps the same object at virtual address 64KB, that is, 8 base pages beyond the origin. Further suppose that process P_1 has touched the first 15 pages of the object, which are populated into a 16-page reservation.

If at this point P_2 touches the 16^{th} page, then the system will use page index 8 to lookup a population map in the hash table, but the existing reservation was stored in the table with page index 0. Hence, the first reservation will not be found and a new reservation that partially overlaps with the first one will be created, as shown in Figure 6.1.

The overlapping reservations unnecessarily waste contiguity. In addition, between the two processes only three 4-page superpages will be created for the first 16 pages, in cir-

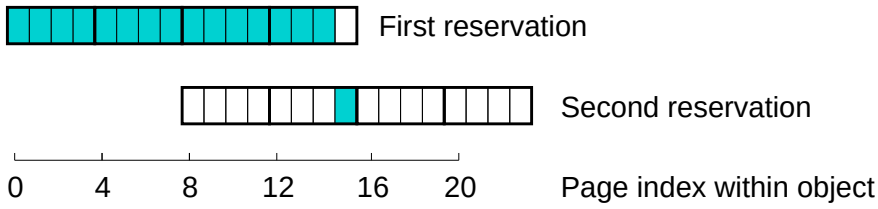


Figure 6.1 : Overlapping reservations.

cumstances where it should be possible to create one 16-page superpage for one process and two 4-page superpages for the other.

Note that the approach to multiple mappings described in Section 4.10.3 — in which the OS picks virtual addresses that are compatible with superpage allocation when a process maps an object — does not completely avoid this issue. In particular, this problem may still arise when applications request to map an object at a fixed address, or when they map a file starting at some offset.

6.2.3 Overhead for an adversary case

We used the adversary application described in section 5.6 to gauge the overhead of our system when many superpage sizes are supported. Recall that such an application simply allocates memory, accesses one byte in each page, and deallocates the memory. We set the system to support all page sizes that the Itanium provides, but limited in the low end by FreeBSD's page size and in the high end by the maximum size that provides enough TLB coverage for the entire 2 GB of physical memory of the machine. Therefore, the system supported eight sizes: 8 KB base pages and superpages of sizes 16KB, 64KB, 256KB, 1MB, 4MB, 16MB and 64MB.

This adversary application shows a slowdown of 32.9% when run with our superpage support system, composed of 20.6% of overhead due to page table traversal and 12.3% of data structure bookkeeping. Recall that most page table entries in this experiment are traversed twice per superpage size due to incremental promotions and once per superpage size due to incremental demotions.

This rather high overhead for an adversary case begs the question of whether real applications will suffer a noticeable slowdown due to the use of superpages.

6.2.4 Overhead in practice

We repeated the experiment of Section 5.6.4, where we measure the overhead paid by our benchmarks. Again, we factor out the benefit of superpages by not creating any actual superpage, but preserve the overhead of promotions and demotions by writing the superpage size field into an unused portion of the page table entry.

We found that the overhead is generally low, below 2%, with the exception of FFTW and Image, for which the overhead reaches 4.1% and 2.1% respectively. However, in both cases the overhead is more than compensated by the benefit that superpages provide to these applications.

6.3 Refining the design

We found that most of the problems mentioned above can be solved with a redesign of the population map.

6.3.1 Reservations: lookup and overlap avoidance

FreeBSD 5.x uses a splay tree [80] as a container for the pages in a memory object, while previous versions use a hash table for this effect. As a consequence, the doubly-linked list that links the pages in an object is now kept in page index order, and finding the predecessor and successor of any given page is trivially efficient.

In section 4.9 we described the four distinct purposes of the population map. The first two are reserved frame lookup, and reservation overlap avoidance. By leveraging the splay tree, these two goals can be achieved more efficiently in the following manner.

Each page (or rather, the data structure that describes the page) contains a pointer to a data structure that describes the reservation that the page belongs to. The reservation data structure includes the starting physical address and size of the reservation. On a page fault,

we use the splay tree and the linked list to find the pages that are going to be the predecessor and successor of the faulting page. From these pages we obtain two candidate reservations that may contain a frame reserved for the faulting page (in general, the number of candidates can be zero, one, or two; for instance, both the predecessor and the successor might be pointing to the same reservation, or there might not be a predecessor or a successor). If the faulting page does not fall in the range covered by these reservations, then we know that we must create a new reservation, and we can also determine the maximum size for this reservation that prevents overlapping with the existing ones. The hash table used in the original design to locate the root of a population map is not needed.

6.3.2 Streamlined population maps

With the change described above, the role of the population map is limited to assisting in preemptions and promotion decisions. We streamlined the design of the population map to perform only these two functions, and optimized it for sequential allocations, which is the most common case according to our observations.

We still use a radix tree for the population map, but instead of keeping track of the population status of the superpage-sized portions of a reservation (fully populated, partially populated, or empty), we use a scheme that does not need to develop the tree beyond the root node when a reservation is populated sequentially.

The root (and each node) maintains a pair of indices that describe a fully populated range of frames. Only if the population pattern in the reservation becomes non-contiguous we need to make the tree one level deeper, and so on recursively. The recursion is stopped at the level corresponding to the smallest superpage, since below that level we only need the number of frames populated, but we do not need to keep track of which frames are actually populated.

In addition, at each node we keep track of the largest superpage-sized and -aligned unused chunk in the subtree below the node. This facilitates the maintenance of the reservation lists described in Section 4.8.

Figure 6.2(a) depicts a population map for a reservation of 64 base pages, in which frames 17 to 26 have been populated; only the root node is needed. If frame 53 is then populated, the map will look as shown in Figure 6.2(b). If frame 58 is populated next, one more level is needed as shown in Figure 6.2(c). This example assumes that superpages of size 4, 16 and 64 base pages are supported.

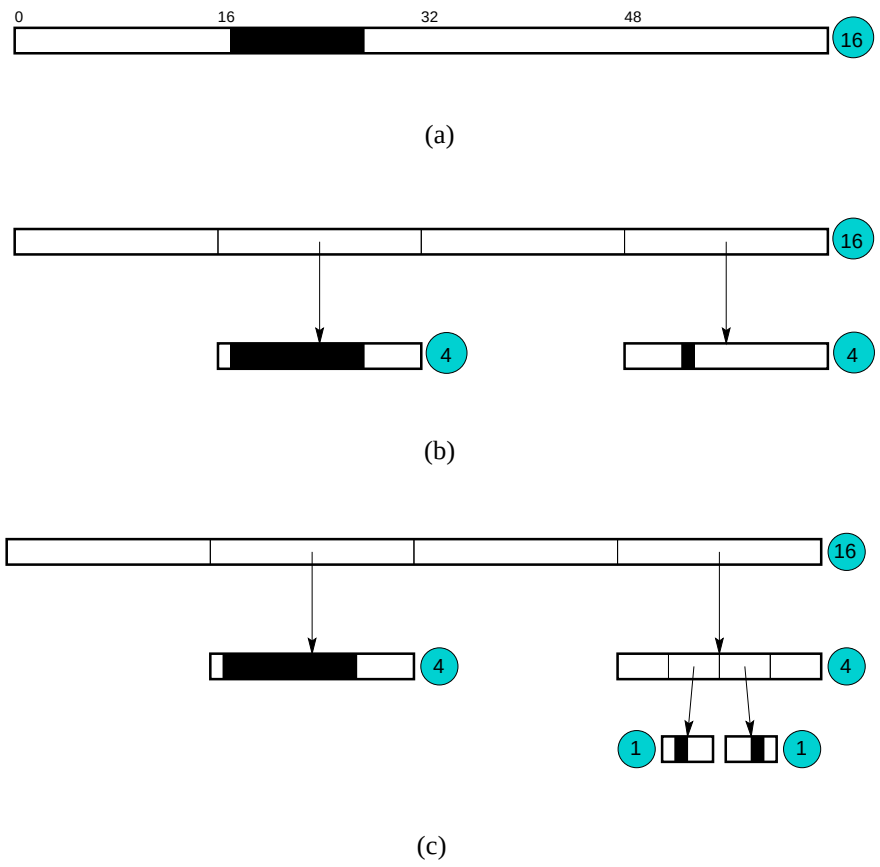


Figure 6.2 : Stages of a population map. White rectangles represent nodes of the radix tree. Black rectangles represent ranges of populated pages. Gray circles next to a node indicate the size of the largest unused chunk appropriate for superpages at or below the node. (a) The map after frames 17-26 have been populated; (b) after frame 53 is also populated; (c) after frame 58 is also populated.

6.3.3 Reservation preemption

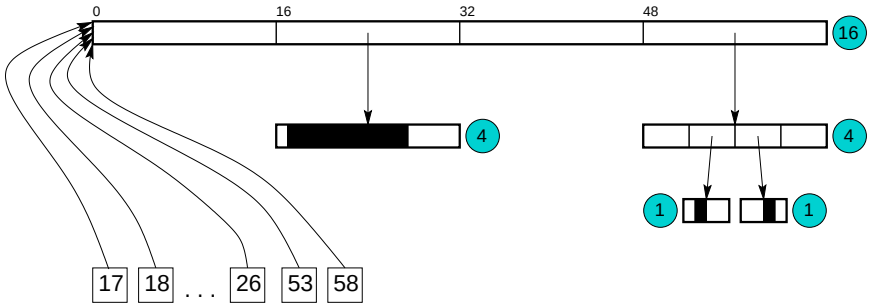
When a reservation is broken into smaller ones, we do not want to go through potentially thousands of pages in the original reservation to adjust their pointers to point to the new, smaller reservations. To avoid this, we keep the original reservation data structure, mark it as broken, create data structures for the smaller reservations, and add pointers from the original reservation to the smaller ones, in a radix-tree-like manner.

These pointers are used to lazily adjust the reservation pointers of the pages in the following manner. Whenever a page's reservation is needed, the reservation that the page currently points to is examined. If it is broken, then we descend down the radix tree using the pointer that corresponds to the position of the page within the reservation, until we find a non-broken reservation. At that point, we can adjust the page pointer to point to the new reservation. A reference counting scheme is used in order to know when it is possible to free a broken reservation's data structure.

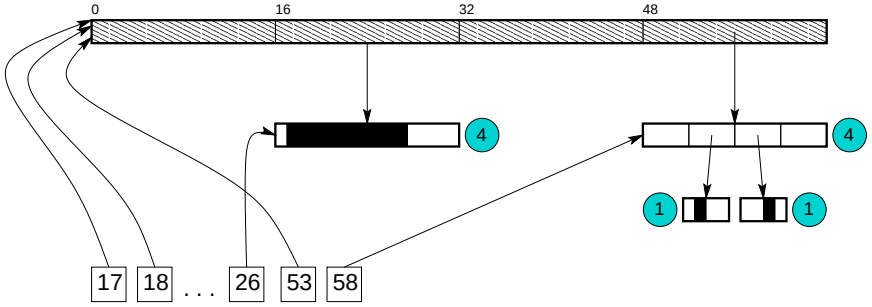
Reservation preemptions can happen because of two different reasons, and the preemption process varies slightly depending on the reason: a preemption can be triggered either by memory pressure or by the reactivation of a cache page that winds up as part of a reservation, as described in Section 4.10.1.

Figure 6.3(a) shows the same population map as Figure 6.2(c), with the addition of the page data structures, which all point initially to the root node. If in this situation the reservation is preempted to regain memory, then the goal is to return to the free list the largest available chunks, and keep the rest as smaller reservations. The system will therefore free the first and third quarter of the original reservation, and mark the reservation as broken. All pages remain pointing to the root node that represents the original reservation, because these pointers are lazily updated as previously said. Figure 6.3(b) shows the resulting data structures after the reservation pointers for pages 26 and 58 have been updated.

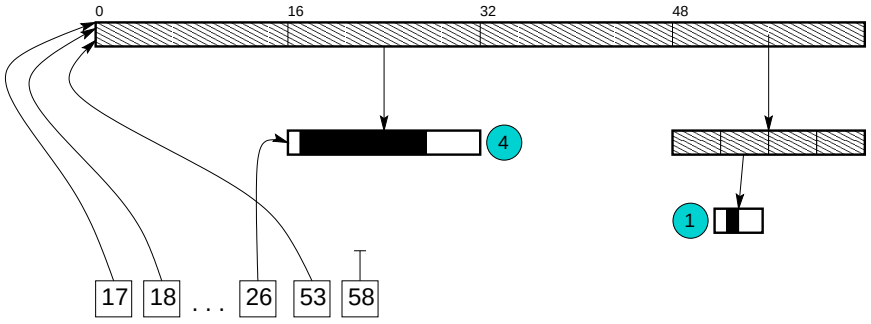
On the other hand, if the preemption is due to the reactivation of a cache page, say page 58, then the system must recursively break all the reservations in the path to that page. The



(a)



(b)



(c)

Figure 6.3 : Reservation preemption. (a) The original reservation. (b) After preemption due to memory pressure. (c) After preemption due to reactivation of page 58. Nodes representing broken reservations are shaded.

result is depicted by Figure 6.3(c), again assuming that reservation pointers for pages 26 and 58 have been updated.

6.3.4 Shared objects

The new design also solves the problem with shared objects mentioned in Section 6.2.2. Overlapping reservations due to mappings of incompatible alignment cannot happen anymore, because the system will always find the predecessor and successor of a new page, and from there the respective reservations, regardless of the intended alignment for those reservations.

6.4 Results for IA-64

In this section we present the results obtained for our benchmarks with the refined implementation for IA-64.

6.4.1 Best-case benefits

We used the same workloads from sections 5.2, with the exception of the SPEC CPU2000 floating-point benchmarks and Povray, which we were unable to compile under this platform.

The methodology is the same as the one described in Section 5.3, that is, benchmarks are run under conditions of memory and contiguity abundance, so that allocation for reservations always succeed and reservations are never preempted. The speedups are computed against the unmodified system using the mean elapsed runtime of several runs after an initial warm-up run. Table 6.1 shows the results.

Overall, the results are similar to those obtained for the Alpha architecture. Almost all applications experience speedups. Out of the 20 benchmarks, 16 experience improvements above 5%, and 6 experience over 25%. Comparing with Table 5.1, some applications, such as twolf and Image, show a significantly better speedup for the Itanium than the Alpha, while the inverse is true for some others, such as bzip2, FFTW and Matrix. However, the

Benchmark	Superpage usage 8 KB	16 KB	64 KB	256 KB	1 MB	4 MB	16 MB	64 MB	Speedup
CINT2000									**1.127**
gzip	157	24	14	10	9	10	8		1.000
vpr	219	27	15	12	8	6	1		1.367
gcc	561	21	44	20	7	7	3	1	1.012
mcf	177	13	8	6	4	6	6	1	1.707
crafty	218	20	12	4					1.083
parser	215	18	1	5	3	4	1		1.081
eon	311	28	12	1					1.000
perl	361	28	16	10	6	6	7		1.031
gap	326	22	9	3	3	3	3	2	1.068
vortex	312	17	5	5	3	10	2		1.125
bzip2	148	25	16	11	13	14	7		1.082
twolf	242	18	7	10	2				1.137
Web	10716	3041	1145	214	50	1			1.094
Image	260	33	18	16	3	3	1		1.379
Linker	196	18	11	13	12	7	2		1.401
C4	143	11	4	3	4				1.279
Tree	130	15	6	8					1.312
SP	210	60	50	35	2				1.063
FFTW	192	42	7	4	3	4	6	2	1.134
Matrix	119	13	6	6	4	6			5.148

Table 6.1 : Speedups and peak superpage usage for IA-64 when memory and contiguity are abundant.

Matrix benchmark for the Alpha consisted of highly efficient code produced by Compaq's compiler, while for the Itanium we used the more modest gcc compiler. If gcc is used for both platforms, then the speedups for Matrix tend to equalize: 5.490 for the Alpha against 5.148 for the Itanium.

Better speedups for a processor that performs hardware TLB miss handling might seem unexpected, but there are two reasons that in combination can explain it. Firstly, without superpages, Itanium's data TLB covers a smaller portion of benchmarks' footprint than Alpha's data TLB, not only because it is smaller, but also because benchmarks have a larger footprint on the Itanium. For instance twolf's footprint is 6.2 MB for the Alpha and 7.1 MB for the Itanium, although the base page size is the same for both architectures. And secondly, Itanium supports larger pages. These two factors imply that for any given benchmark, it is likely that superpages eliminate a larger absolute number of TLB misses in the Itanium.

On top of these two reasons, the disparity of processor and memory speed make memory accesses required to fill the TLB the largest component of TLB miss cost, regardless of whether this access is performed by a hardware state machine or by a software handler.

6.5 Overhead and adversary case revisited

With the aforementioned changes, the bookkeeping overhead for the adversary case is reduced to a mere 2%. Nevertheless, since we have optimized the population map for sequential allocation, it is fair for our adversary to perform non-sequential but still full allocation of the memory region. We therefore modified the adversary to first touch the even pages in the allocated memory region, and then the odd pages. This pattern forces the system to fully develop the population map's radix tree, and in fact increases the adversary's bookkeeping overhead to 4.9%, which is still much better than the 12.3% with the old population map.

The page table overhead remains close to 20% when all seven sizes are supported, since these changes do not affect it.

For the other benchmarks, the overhead is now always below 1%, with the usual two

exceptions: FFTW and Image. For these two applications, the total overhead is now 1.1%, which is much lower than the previous 2.1% and 4.1% with the original design.

6.6 Reducing page table overhead

This section analyzes mechanisms to reduce the 20% worst-case page table overhead. The overhead is due to the way superpages are supported in the page table, combined with the use of multiple superpage sizes through incremental promotions and demotions.

6.6.1 Alternative page table organizations: existing approaches

IA-64 and Alpha processors support superpages in the page table by means of PTE replication, as discussed in Section 5.1. Even though one entry would be enough to describe the address translation for the entire superpage, there is still one entry per base page in the page table. Each entry replicates the same information: superpage size, protection attributes, and the most significant bits of the physical address. The motivation for PTE replication is quick translation lookup, because the page size is not an input to the lookup procedure. Alternative organizations are likely to increase TLB miss costs.

One option would be to have parallel page tables, one per superpage size. On a TLB miss, an entry needs to be looked up in every page table. Sparc64-III processors [83] provide a parallel page table for 64KB superpages; larger superpages are supported through conventional PTE replication.

Clustered page tables [88] offer partial relief to the overhead involved in PTE replication. Clustered page tables are similar to IA-64's virtual hash page tables, with the difference that each entry contains the translation information for a group of virtually contiguous base pages. The number of base pages represented in each entry is called the *clustering factor* and is necessarily small. Superpage sizes beyond the clustering factor are handled through replication of clustered entries.

A very clean, non-replicated page table organization for superpages is that of IA-32 processors. It consists of a two-level radix tree where a 4MB superpage is created by storing

the superpage information in the common parent of the 1024 base pages that compose the superpage. A naive extension of this approach to 64-bit address spaces with a more flexible collection of superpage sizes than factor-of-1024, would make the tree prohibitively deep. It also complicates management and addressing of the nodes, since they would be much smaller than a physical page. However, a more sophisticated scheme based on this approach can overcome these problems, as shown next.

6.6.2 Alternative page table organizations: a proposal

This section suggests an efficient page table organization that does not impose a linear overhead for superpage promotions and demotions. The idea is to use an approach similar to PTE replication, except that replication is avoided by always referring to the first entry in the group of entries that would have been otherwise replicated. The other entries are ignored, and hence need not be replicated.

Consider a leaf page in a radix-tree page table, like IA-32's, but with 128 entries in each page, and assume a factor of four between consecutive page sizes. What we need is extra information that tells us how to interpret the data in this page. For this particular example, that extra information can be encoded in only 17 bits, which can be stored in the parent node of the page. Those 17 bits represent a 3-level radix tree of bits. The first one indicates whether the entire page of page-table entries represents a superpage. If so, we only need to retrieve the first entry. Otherwise, we consider the page as four groups of 32 entries each, and the next four bits tell us whether each of these groups corresponds to a 32-page superpage or we should further recur.

We can repeat this scheme at each level of the tree. The depth of the tree can be greatly reduced by combining a hash table and multiple smaller trees. Each small tree would describe a chunk of address space of the size of the largest supported superpage. The hash table would be used to locate the root of the tree. Since each level describe three superpage sizes, with three levels we can support nine sizes. A tenth level can be supported at the hash table entry.

Note that this approach can also be seen as a tree of radix 4, where multiple levels of the tree (a node along with its children, grandchildren and great-grandchildren) are placed together into one physical page. Also, only one generation is accessed per page, thanks to the extra information stored in the parent entry, and to the fixed layout of the descendants in the page.

In summary, this approach requires one hash table lookup and at most three additional memory accesses per TLB miss to support up to ten page sizes without PTE replication. We did not explore this alternative in favor of solutions that work efficiently with existing hardware.

6.6.3 Fewer superpage sizes: static approach

In Section 5.4 it was shown that supporting multiple page sizes is necessary to achieve maximum performance. The question now is whether it is necessary to support *all* the page sizes that the platform provides, especially considering that fewer sizes would reduce the page table overhead of incremental promotions.

Such overhead is proportional to the number of sizes supported. Therefore, if the goal is to reduce it by any significant amount, then the number of sizes must be reduced also significantly. To evaluate this possibility, we decided to experiment with a reduction of superpages sizes to roughly one half, that is, to three or four.

Clearly, the only reasonable choice within this constraint is to support every other size, that is, sizes that are separated by a factor of 16. In effect, it was shown before that some applications only benefit from small sizes, so it is not a good idea to support only large sizes. On the other hand, if only three or four small sizes are supported, then applications that require a large TLB coverage will suffer. An instance of FFTW, for example, with the largest input we could run in our platform (a cubic matrix of 390 elements per side) pays a performance penalty of 47% if 64MB superpages are not provided.

Hence, the choices would be to support either 16KB, 256KB, 4MB and 64MB superpages, or 64KB, 1MB and 16MB. We picked the latter to experiment with. Table 6.2 details

the results, where the second column shows the speedup when all sizes are supported, the third column shows the speedup when only 64KB, 1MB and 16MB are supported, and the last column shows the percentage of difference between the two (negative numbers mean that the benchmark runs faster with all sizes).

Benchmark	All sizes	64KB + 1MB + 16MB	% Difference
CINT2000			
gzip	1.000	1.000	0.00
vpr	1.367	1.376	0.66
gcc	1.012	1.017	0.49
mcf	1.707	1.711	0.23
crafty	1.083	1.076	-0.65
parser	1.081	1.081	0.00
eon	1.000	1.006	0.60
perl	1.031	1.034	0.29
gap	1.068	1.067	-0.09
vortex	1.125	1.123	-0.18
bzip2	1.082	1.083	0.09
twolf	1.137	1.055	-7.21
Web	1.094	1.051	-3.93
Image	1.379	1.379	0.00
Linker	1.401	1.400	-0.07
C4	1.279	1.287	0.63
Tree	1.312	1.331	1.45
SP	1.063	1.058	-0.47
FFTW	1.134	1.138	0.35
Matrix	5.148	2.486	-51.71

Table 6.2 : Speedups for all seven sizes compared to speedups for three fixed sizes.

Expectedly, although not shown in the table, the page table overhead got reduced to about one half. However, the benchmarks show almost no benefit from this reduction: there is only one case, Tree with 1.45%, that shows an improvement above 1%. On the other hand, the penalty paid for the lack of support for some sizes is much larger than the small benefit from a lower overhead: twolf runs 7.21% slower, Web runs 3.93% slower, and Matrix takes more than twice the time to finish.

6.6.4 Fewer superpage sizes: dynamic approach

We can reduce the overhead of incremental promotions and demotions without the disadvantages of the previous three-sizes-fit-all approach, by adapting what three sizes to provide according to the circumstances.

As demonstrated in the previous section, different processes have different needs. However, choosing a set of three sizes on a per-process basis is still unnecessarily restrictive, since a process may have different needs for its different memory segments. Moreover, memory conditions should also be considered: if we have a budget to provide three page sizes, it would be wasteful to aim at very large sizes in conditions of low contiguity.

A very simple and effective approach that takes all these factors into account is to provide three sizes on a per-reservation basis, and pick those sizes according to the size of the reservation. This works well because (1) the reservation size depends on our preferred superpage size policy of Section 4.2, which tends to choose the maximum superpage size that can be effectively used in a memory object, and (2) the actual reservation size is further restricted by availability.

We found that what works best for our benchmarks is to support the sizes that corresponds to the size of the reservation, plus one size smaller, plus the size between the latter and 16KB, the smallest one. For instance, for a 16MB reservation we use 16MB, 4MB and 256KB; for anything smaller than 1MB we use all sizes up to the size of the reservation.

We further reduced the page table overhead by forcing a full demotion instead of an incremental one whenever the system removes all pages in an object. This happens when an object is unmapped or the entire address space is torn down. Without this optimization a superpage is gradually demoted without any benefit, since none of the pages will be accessed anymore.

With these changes the overhead of the adversary case was reduced to an acceptable 13%, composed of 8.1% of page table overhead and an unchanged 4.9% of bookkeeping overhead. This improvement for the worst case comes with virtually no impact for the other benchmarks, as shown in Table 6.3. The majority of benchmarks show a slowdown rather

than a speedup, but the absolute difference with the case when all superpages are supported is under 1%, with the only exception of linker, whith -1.14%.

Benchmark	All sizes	Dynamic three	% Difference
CINT2000			
gzip	1.000	1.000	0.00
vpr	1.367	1.376	0.01
gcc	1.012	1.017	0.49
mcf	1.707	1.705	-0.12
crafty	1.083	1.080	-0.28
parser	1.081	1.081	0.00
eon	1.000	0.999	-0.10
perl	1.031	1.028	-0.29
gap	1.068	1.064	-0.37
vortex	1.125	1.122	-0.27
bzip2	1.082	1.072	-0.92
twolf	1.137	1.136	-0.09
Web	1.094	1.085	-0.82
Image	1.379	1.379	0.00
Linker	1.401	1.378	-1.14
C4	1.279	1.281	0.16
Tree	1.312	1.322	0.76
SP	1.063	1.058	-0.47
FFTW	1.134	1.131	-0.26
Matrix	5.148	5.097	-0.99

Table 6.3 : Speedups for all seven sizes compared to speedups for three sizes dynamically chosen.

Chapter 7

Memory compaction in the idle loop

In an idle or partially idle system, spare CPU cycles can be used to regain contiguity by compacting memory. Memory compaction consists of relocating pages to reorganize memory in a way such that free pages are physically contiguous and suitable for superpage use.

Spare cycles are available in servers due to uneven load, and in workstations due to inactivity periods. By using these cycles, memory compaction can complement the contiguity recovery mechanism based on page eviction described in Section 5.5. A proactive compaction approach can help applications run faster before the reactive eviction mechanism triggers.

This chapter proposes and evaluates a memory compaction algorithm for the idle loop that balances the relocation impact against the performance potential of superpages.

7.1 A memory compaction algorithm

From a high level standpoint, the goal of a compaction algorithm is straightforward: to maximize contiguity in the system at the minimum cost. The question is, how do we measure contiguity and what exactly are the costs involved in relocation?

The fact that compaction is done in the idle loop, i.e., only when there is nothing else to do, imposes several requirements. The compactor can be interrupted at any time and for arbitrarily long periods; by the time it resumes, memory layout can be completely different. We therefore want to do the job in a way that provides incremental benefits. A corollary of this requirement is that the algorithm must not spend too much time thinking what to do. Furthermore, page allocations should not only be allowed while the compactor runs, but they also should not undo the job of the compactor, i.e., they should not occur in the middle

of a chunk the compactor is trying to make available. In essence, this last point means that there must be some synchronization between the compactor and the allocator.

Finally, current superpages and reservations must be brought into the equation, since they must be relocated together as a block. A sketch of an algorithm that fulfills these requirements follows:

1. Within a set of candidate chunks to vacate, pick the one with the best cost/benefit ratio.

2. Do not allow the page allocator to pick frames from this chunk, unless it has no alternative. If the latter happens, start over.

3. Vacate the selected chunk by moving occupied pages to free frames outside the chunk. Relocate reservations and superpages as a block, and carefully pick the destination so as to not fragment memory. Also, be greedy to obtain incremental benefits: start relocating those pages that provide the most contribution to contiguity at the smallest cost.

4. Repeat.

The following sections discuss how to compute the cost and benefit of vacating a given chunk, which is required by the first and third steps above.

7.1.1 The cost

To compute the cost of vacating a chunk, we use a simple metric that relates directly to the cost in cycles, which is the number of occupied pages in the chunk, that is, the number of pages that must be relocated.

Note that, although spare cycles are available, we want to use them effectively, since every cycle spent in migrating a page cannot be used for migrating other pages or for other idle loop tasks such as zeroing out free pages.

When a page is relocated, the page tables of each process that has the page mapped need to be updated with the new physical address of the page. This remapping is not necessarily uniform, because shared pages have a higher cost given that they have more mappings, while cache pages have no remapping cost. However, since it is much smaller than the copying cost, it is not considered in our metric.

7.1.2 The benefit

The benefit of vacating a given chunk is, in simple terms, the contiguity provided by the chunk minus the contiguity required to evacuate it. The complexity in this definition is hidden in the concept of contiguity. Instead of measuring contiguity in absolute terms, we propose to consider the relative value of that contiguity in relation to what is currently available, motivated by the following example.

Consider two 4MB candidate chunks with the same evacuation cost, but one containing only base pages and the other containing base pages plus one 1MB superpage; the latter therefore needs a 1MB free chunk to evacuate the superpage. If free memory in the system consists of a very large number of 1MB and smaller chunks, then it makes little difference to pick one or the other. On the other hand, if free memory consists of one 1MB chunk and many smaller ones, then we should prefer the first chunk to spare the only 1MB chunk available. More to the extreme, if there is no 1MB chunk available, then we simply cannot pick the second candidate, because there is no place where to move the existing superpage without breaking it. Breaking a superpage or reservation with the purpose of obtaining contiguity for a future reservation, at the additional relocation expenses, is not sensible since the system has no way of predicting that a future reservation will provide more benefits than an existing one.

Our metric for contiguity is based on the TLB coverage that a new process can obtain with the existing free chunks of memory. Define $\mathcal{T}$ as the number of entries in the TLB, for now, and define $\mathcal{K}_i$ as the size of the i^{th} largest free chunk. Then, the coverage $\mathcal{C}$ provided by the free chunks — which we will hereinafter call the *buddy coverage*, given that free

pages are managed in a buddy system — is the sum of the sizes of the largest $\mathcal{T}$ chunks:

$$\mathcal{C} = \sum_{i=0}^{\mathcal{T}} \mathcal{K}_i$$

assuming, for now, that there are at least $\mathcal{T}$ free chunks.

This way of defining contiguity means that we consider as valuable sizes only those that are at least as large as the largest $\mathcal{T}$ chunks currently in the free list. As an example, assume $\mathcal{T} = 8$ and suppose that the free list is composed of one 4MB chunk, four 1MB chunks, and many 64KB chunks, i.e., in KB,

$$\mathcal{K}_1, \ldots, \mathcal{K}_8 = 4096, 1024, 1024, 1024, 1024, 64, 64, 64$$

In this scenario, if we consider to vacate a 4MB chunk containing four 1MB reservations, then we would obtain a buddy coverage increase of 192KB. We will obtain a 4MB chunk at the expense of the four 1MB ones required to hold the reservations. The net effect is thus that three 64KB chunks will get into the largest 8:

$$\mathcal{K}_1, \ldots, \mathcal{K}_8 = 4096, 4096, 64, 64, 64, 64, 64, 64$$

On the other hand, if we vacate a 4MB chunk that only contains 64KB reservations, then the coverage increase would be 4032KB ($4\text{MB} - 64\text{KB}$), reflecting the lack of value of 64KB chunks since there are plenty of them:

$$\mathcal{K}_1, \ldots, \mathcal{K}_8 = 4096, 4096, 1024, 1024, 1024, 1024, 64, 64$$

Basing the computation of the benefit on this metric will therefore make the algorithm tend to choose large chunks (chunks that are at least as large as $\mathcal{K}_\mathcal{T}$), and chunks that do not contain relatively large reservations inside. Also, since the benefit is divided by the number of populated pages, it allows us to choose between a fully-occupied large chunk and a lightly occupied small one.

7.1.3 What TLB size to target

The Itanium processor has three TLBs with different sizes: the first-level data TLB has 32 entries, the second-level data TLB has 96 entries, and the instruction TLB has 128 entries. There is, therefore, several possible values for $\mathcal{T}$. Multiprocessor machines multiply the number of choices for $\mathcal{T}$, since one could give good arguments to base it in the number of entries per processor, and good arguments to base it in the total number of TLB entries in the machine. This section describes how this decision is made uncritical with a small change in the algorithm.

Note that a special case occurs when there are fewer than $\mathcal{T}$ free chunks in the system, because in this case any further compaction will not change the buddy coverage. Similarly, once $\mathcal{T}$ chunks of the largest supported size have been recovered, no further compaction will increase the buddy coverage. This is an indication that the system is in a state where the potential benefits of compaction are likely to be marginal. In particular, a new process for which there is enough memory is guaranteed to get enough TLB coverage without any further compaction.

Since the buddy coverage is equal to the amount of free memory in the system, the compactor could declare its job done and stop until memory conditions change. Instead, we set a new goal for the compactor, by redefining $\mathcal{T}$ and hence changing the computation of coverage: whenever there are more than $\mathcal{T}$ free chunks of the largest size or fewer than $\mathcal{T}$ chunks in total, we change $\mathcal{T}$ by doubling or halving it as many times as necessary.

In essence, this means that the system picks a value for $\mathcal{T}$ depending on the configuration of the free lists. The advantage is that now only a non-crucial initial value for $\mathcal{T}$ needs to be provided. The disadvantage is that the compactor may compact beyond what is necessary. In our implementation the compactor runs at a strictly lower priority than the only other idle loop task, which is the zeroing out of pages. Therefore this disadvantage does not exist, since the cycles spent by the compactor cannot be spent in anything else. A different implementation may choose to run the compactor at the lowest priority within the

idle loop only if the actual value for $\mathcal{T}$ differs significantly from the initial one.

7.2 Selecting a chunk to vacate

Every superpage-sized and naturally aligned chunk of physical memory is a potential candidate for evacuation. For every candidate, we need to evaluate its cost and benefit. However, we can discard chunks that are larger than the total amount of free memory, since they cannot be fully evacuated.

To compute the cost of vacating a chunk we simply count the number of occupied pages in the chunk. However, we consider that chunks that have wired pages have infinite cost, since wired pages cannot be moved. To compute the benefit, we calculate the current buddy coverage and subtract it from the coverage resulting from the hypothetical evacuation of the candidate chunk. We take into account the allocation needs for evacuating the candidate, and the fact that these allocations might result in the breakage of free chunks larger than requested in the absence of free chunks of the proper size.

Conceptually, the algorithm to select a chunk to vacate is the following. Let F be the total amount of free memory, rounded down to a superpage size, i.e., F is the largest chunk size that should be targeted for evacuation. Then,

```
coverage0 = current buddy coverage
for all superpage sizes S from the smallest to F
  for all chunks C of size S
    if no wired pages in the chunk
      cost = number of occupied pages
      simulate evacuation of chunk
      coverage1 = buddy coverage in simulation
      benefit = coverage1 – coverage0
      if cost/benefit is best so far
        best_chunk = C
```

In practice we trade accuracy for time and examine just a portion of physical memory each time. After computing the cost and benefit of every candidate chunk in the selected memory region, we pick the one with the lowest cost-to-benefit ratio. Once this chunk is evacuated, the cycle is repeated in a different portion of physical memory.

Also, for performance reasons, in our implementation of the algorithm we visit each page (or rather, the structure that contains its information) only once. We traverse the array of page structures counting the number of occupied pages. Every time we hit a superpage boundary, we update the cost for all superpage-sized chunks that end in that boundary, and compute their benefits.

Note that this algorithm may move more pages than necessary to achieve maximum compaction. It can even move the same page more than once. For instance, in the context of a hypothetical system with superpages of size 2, 4 and 8 base pages, consider what happens if $\mathcal{T} = 2$ and memory has the state depicted in Figure 7.1.

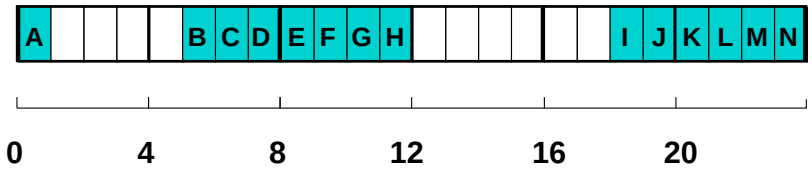


Figure 7.1 : A possible status of memory. Shaded blocks represent occupied pages.

In this situation, the buddy coverage is 6 pages, and the best cost/benefit migration operation is to move page A to frame 4, which would increase the coverage to 8 pages. After that operation is complete, the next move would be to obtain an 8-page superpage by migrating either the first or the second group of four pages to the adjacent 8-page chunk, yielding a new coverage of 10 pages. Note that, depending on how the algorithm resolves ties, the next page to move might be A, the very same page that was last moved.

There are two justifications for this seemingly suboptimal behaviour. First, avoiding it involves complex trade-offs. When migrating a page, the destination frame must be chosen not only according to how much contiguity is lost by using that frame, but also according

to how likely it is that the frame belongs to a chunk that may be chosen for evacuation in the near future. This in turn affects the computation of the benefits of evacuating a chunk, which would increase the "think time" of the algorithm. Longer times to make decisions go against one of the requirements stated at the beginning of this chapter.

Second, in some situations it might be *better* to move the same page twice. Consider again the example in Figure 7.1, and note that if we move page A directly to frame 12, then the buddy coverage would actually be *reduced* while the copy is taking place. This reduction does not occur if we move page A first to frame 4 and then to frame 12.

7.3 Vacating the selected chunk

The selected chunk is vacated in greedy order: it is divided into subregions of the next smaller superpage size and these subregions are vacated in ascending order of their cost/benefit values, recursively applying the same evacuation procedure to each. If a page relocation fails, (for instance, because the source page became wired for I/O), the evacuation is aborted and a new iteration to select another candidate is started.

Special care must be taken when moving reservations: only after the last page is copied to its new location, the population map to which the reservation belongs is updated with the new physical address, and the vacated space is returned to the buddy allocator.

To synchronize the compactor with the allocator, we use a pair of global variables to indicate the starting address and length of the chunk being evacuated. The buddy allocator refrains from allocating memory within that range, unless (1) it has no alternative, or (2) the only alternative is to break a free chunk of the same or larger size as the one being vacated. In either case, the compactor aborts its current task and restarts the algorithm from the beginning.

When a page is relocated, the page table entries that refer to the page must be either eagerly updated with the new physical address of the page, or invalidated to force lazy updates through page faults. We prefer eager updating because the page table entries must be modified anyway and hence we save the page fault costs of invalidation, essentially

for free.

Finally, since the FreeBSD kernel is not preemptible, we have to avoid holding the CPU for too long during the evacuation of a large chunk. We therefore check at various points whether there is any process ready to run, and if so yield the CPU. Specifically, we introduce a yield point after a number of pages have been scanned (500 in our implementation) and also after each page is copied; page relocation is thus atomic on a page basis.

7.4 Evaluation

The compactor is evaluated in the same platform as the one described in the previous chapter, in Section 6.1, i.e., an Itanium workstation with 2GB of RAM.

7.4.1 Scanning

In an idle system with plenty of free memory, the compactor quickly reaches a steady state in which it keeps scanning physical memory unsuccessfully trying to find pages to move that would increase contiguity in the system. This state is reached because as memory gets compacted, the compactor starts targeting larger chunks to evacuate, since small ones do not contribute to any increase in the buddy coverage. However, there is a point at which no large chunk can be evacuated because all potential candidates contain wired pages. After an unsuccessful scan cycle the compactor could sleep until memory status changes, but in our implementation it keeps scanning.

In such a steady state, provided there are no other idle-loop tasks, the compactor completes about 390 full memory scans per second in our 2GB platform. While scanning rate is irrelevant in steady state, faster rates means less time spent scanning whenever there is compaction work to do. The scanning rate can be increased with two simple optimizations.

First, the compactor can “jump” over free blocks, since there is no need to scan every base page; the way the buddy allocator is implemented allows it to determine the presence and size of a free block on hitting the first page of the block.

Second, when hitting a wired page, the compactor can ignore an entire block of pages

and continue scanning after that block. The size of such block is given by the minimum block size the compactor is interested in, which is precomputed at the beginning of each run. From the definition of buddy coverage, that minimum size corresponds to the size of the $\mathcal{T}^{\text{th}}$ largest free chunk, because the buddy coverage will not change if chunks smaller than that are made available.

With these optimizations the scanning rate increases to 3470 full scans per second for the same scenario as above. As memory gets closer to full utilization, the rate gets slower and approaches the original value, reaching a minimum of 420 scans per second, because the compactor finds fewer and smaller free chunks to skip, and it also ignores only small regions surrounding a wired page because the minimum candidate size is smaller in such conditions.

7.4.2 Effectiveness

To test the effectiveness of the compactor, we start from a state of maximum memory fragmentation where every other physical page is free. To achieve this state, we force the system to use all the available memory for large reservations, then populate every other page in those reservations, and finally break all the reservations. For this purpose only, we expose the reservation preemption function through a system call. This way, a simple user process that maps a 1.9GB memory region and touches every other page in the region before using this system call will bring memory to a fully fragmented state, where there are 116576 single-page chunks in the free lists.

At a CPU utilization close to 99%, since there are no other active tasks in the system, the compactor takes 4.5 seconds to reach steady state. In that time it migrates 85027 pages (at a rate of 147MB per second) to make twelve 64MB chunks, ten 16MB chunks, two 4MB chunks, and a few smaller ones. The same level of compaction could have been achieved by moving only 61550 pages. The extra 23487 pages is therefore the cost of the algorithm's greed and rush.

Next, we repeat the experiment but with memory fragmented in a random manner: the

1.9GB address space of the process is populated by alternating allocated chunks and unused chunks. These chunks have a random length, measured in pages, between 1 and 2^n, where n is uniformly distributed between 1 and 9. The graph in Figure 7.2 shows how the buddy coverage increases as a function of the number of migrated pages. As expected, the curve has a convex shape, demonstrating the fact that the algorithm gives preference to migrating the pages that provide the largest coverage increase. The curve has some local concavities, due to the fact that the algorithm is not able to examine all possible candidates in each cycle, and possibly also due to small changes in memory status while the algorithm runs.

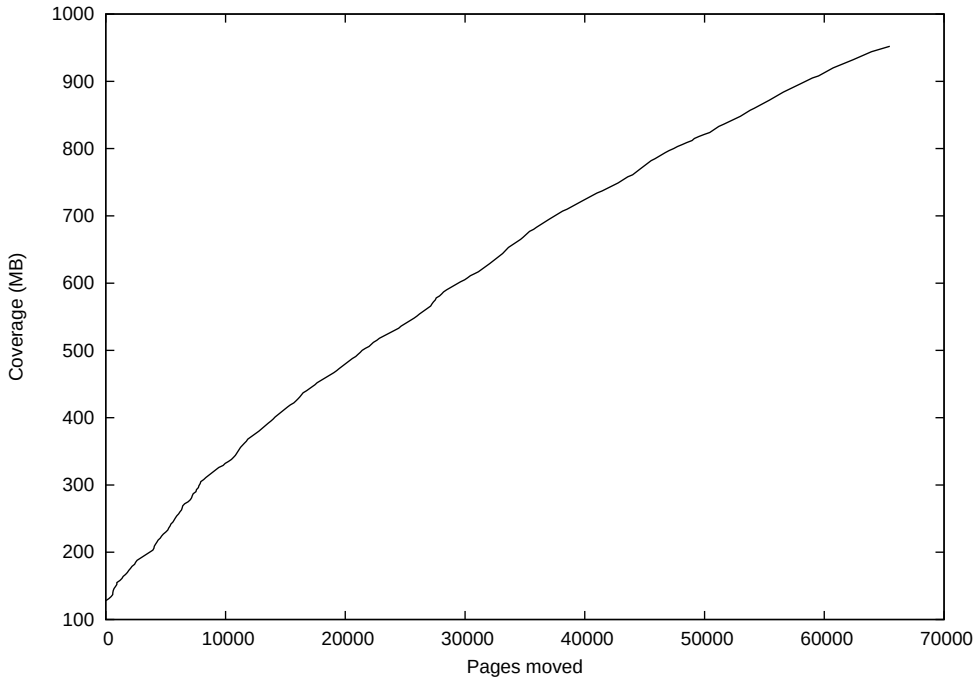


Figure 7.2 : Buddy coverage as a function of number of page migrations.

7.4.3 Impact

To measure the impact of the compactor, we again start from a state of full memory fragmentation, and run a kernel build, excluding the final linking phase. If the cache is cold, the blocking time due to I/O leaves about 4% of CPU to the idle loop, in what can be con-

sidered a fairly common scenario. The compactor gets most of the cycles of the idle loop, since the page zeroing task takes only one fifth of these cycles. In the roughly 90 seconds that the build takes, the compactor is able to move 47768 pages to reach a buddy coverage of 722MB. In contrast, without compaction, the buddy coverage remains minimal at 1MB throughout the experiment.

On the other hand, the kernel build takes longer when the compactor is making use of the idle cycles, but only by 2.2%. This is due to the cache pollution effect of page migration, and to the fact that the FreeBSD kernel is not preemptible. As a consequence of the latter, when the build process becomes ready — in spite of its higher priority — it still has to wait for the compactor to reach a yield point before acquiring the CPU.

7.4.4 Overall

We now repeat the experiment of Section 5.5.1, where we run FFTW four times in sequence after the web server benchmark. This time we use the compactor in place of the contiguity-aware page daemon as a fragmentation control mechanism.

To make the fragmenting effect of the web server noticeable, and also to run this experiment under the same conditions as the one in Chapter 5, we fooled the system into believing that the total physical memory was only 512MB.

When the web server terminates, the amount of free memory in the system is only 21MB, fragmented to the extent that it provides a buddy coverage of 17MB. Since FFTW starts immediately, the compactor gets to relocate only 12 pages using the few idle cycles that are available while FFTW's binary is read form disk. Once FFTW consumes most of the free memory, the page eviction mechanism is used to keep a pool of free pages. This pool provides very little contiguity because pages are not evicted in physical order, but in LRU order. As a consequence, FFTW runs with no speedup because it gets insufficient TLB coverage. With superpages no larger than 1MB, its TLB coverage is only 35MB, representing 14% of its 247MB footprint. With no available cycles to perform relocation, subsequent runs of FFTW are no different, because they get the same physical memory

pages.

Figure 7.3 shows how speedup changes when a 2-second pause is inserted before each FFTW run, mimicking an interactive workstation session. While the compactor is able to make most of the free 21MB contiguous, providing three 4MB superpages, it does not significantly change the coverage that the first FFTW run obtains. However, once this instance of FFTW terminates and returns its memory to the system, there are 266MB of free memory for the compactor to make contiguous. As a consequence, the next FFTW runs obtain enough contiguity to build twelve 16MB superpages, enough to reach full speedup.

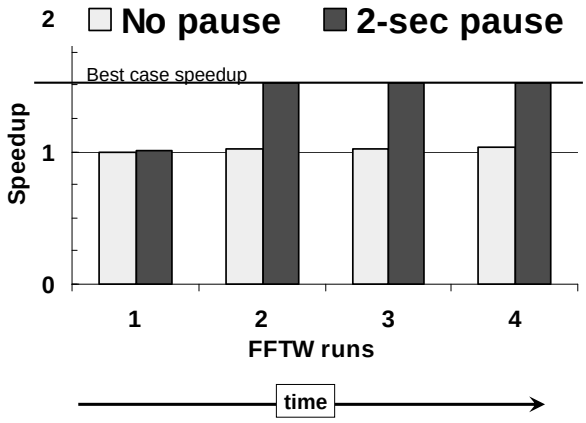


Figure 7.3 : Compaction in a busy versus partially busy system.

7.5 Discussion

This chapter showed that memory compaction in the idle loop is a viable and low-impact mechanism to restore contiguity in a system with fragmented memory.

Page replacement biasing, as described in Section 4.10.1, is an alternative fragmentation control mechanism. Although we did not run these two mechanisms together, we claim that they can be used simultaneously as complementary approaches. Compaction requires two elements that are not always present: available CPU cycles and free memory. Page replacement biasing, in contrast, performs well both in busy systems, because it requires a

minimal amount of CPU just to decide what pages to evict, and in low memory situations, because it regains contiguity regardless of the amount free memory. On the other hand, it is a reactive mechanism that adjusts its work according to the perceived need for contiguity, which may take longer and not necessarily provide enough contiguity to match the requirements of the application that runs next.

The maximum amount of contiguity that the eviction-based mechanism can recover is limited by the distribution of wired and active pages in physical memory. For a compaction-based mechanism like the one described in this chapter, it is limited by wired pages and the amount of free memory. In the best case it will make all of free memory contiguous; if there is a small amount of free memory, it will be of little help for a contiguity-hungry application. However, it is conceivable to drop this limitation if the compactor moves pages around trying to make the replacement order match physical order. In an ideal and extreme case, assuming LRU page replacement, the compactor would place all free memory at the lowest addresses, followed by the least recently used pages, and so on, with the most recently used pages in the highest addresses. In this case the job of the page daemon would be nothing more than to move upward the physical address mark that divides free memory from used memory, increasing both free memory and contiguity at the same time.

Another approach to fragmentation control also based in page migration would be to use spare CPU cycles to relocate existing pages and build superpages to increase the TLB coverage for currently running processes. We did not consider this approach since it is much more likely to have free cycles *between* process runs than *during* a process run, and hence it is better to compact memory in preparation for future runs than to fix the memory layout of currently running processes.

Memory compaction will become more attractive with the new generation of memory controllers that allow memory movement without the intervention of the processor, and hence without cache pollution [41]. Note that our cost/benefit algorithm does not need to be changed in this new scenario, since the main cost of moving pages will still be the opportunity cost.

Chapter 8

Concluding remarks

In this final chapter we show how this thesis compares with previous work on superpages, describe areas that were not explored but still look promising, and then present our conclusions.

8.1 Comparison with previous work

Previous work relating to superpages was described in detail in Chapter 3. It includes superpage support available in some commercial operating systems [28, 85], which lacks the transparency that our solution offers.

Here we revisit the two previous efforts that lie in the area of transparent support for superpages without unconventional hardware support. They explore solutions for specific points in the problem space, omitting matters such as those derived from the coarse dirty/reference information in superpages. This thesis offers an integral solution that tackles all the issues.

Talluri and Hill were the first to propose the use of reservations as a mechanism to deal with the allocation problem [87]. Their study is, however, limited to the support of only one superpage size. We proved in Sections 5.4 and 6.6.3 that, in order to fully realize the potential of superpages, multiple superpage sizes are necessary, which our system supports in a scalable way. Talluri and Hill also do not address fragmentation control, which is fundamental to provide sustainable benefits.

Page relocation is used by Romer et al. [71] in a different way than the memory compactor described in the previous chapter. They relocate pages to make them contiguous and create a superpage whenever a cost/benefit analysis determines that the benefits are likely

to outweigh the copying costs. Apart from the fact that TLB misses need to be monitored, making them more expensive, the problem with the proposed approach is that the relocation costs must be paid even if there is plenty of contiguity available.

On the other hand, since the system creates superpages only when and where needed, it conserves contiguity better than our solution. Fragmentation, however, cannot be avoided, but the authors do not extend the design to address the case when a contiguous memory region is not available as a destination for the pages to be relocated.

8.2 Unexplored alternatives

The design space for a complete solution to the problem of transparently supporting superpages proved to be vast. Here we describe some promising areas that we did not explore.

For the allocation problem an interesting option is to use what we call *implicit reservations*. In that approach, when a frame is needed for a page that has no implicit reservation, the page is placed in a region of physical memory that has enough free frames to create a superpage of the desired size. However, no reservation is explicitly made; the sole presence of the page constitutes an implicit reservation on the neighboring frames since unrelated pages will prefer to be assigned to other, less crowded regions of physical memory.

This model can be refined to consider a region crowded based not just on the free or occupied status of its frames, but also on the recency of the last access to occupied frames. One advantage of this approach is that, in the same way that the implicit reservations become smaller on memory pressure, they can become larger if pressure decreases. Another is that there is no hard limit on what is available for allocating a contiguous region: regions with active pages are not unavailable, but just relatively less available than regions with inactive pages. For that reason, this approach subsumes the contiguity restoration mechanism based on page replacement biasing. This mechanism, however, has to deal with many decisions that involve complex trade-offs, and seems difficult to implement without a steep bookkeeping overhead.

For promotions, our opportunistic policy that incrementally creates superpages may

be subject to a potentially high page-table overhead in the worst case. We tackle this problem by supporting a reduced number of sizes on a per-reservation basis, as described in Section 6.6.4. An alternative is to delay promotions a little so that when a reservation gets fully populated quickly it will become a superpage at once without intermediate steps. An adversary can still force the overhead of incremental promotions, but it would be a much more convoluted case, especially if the delay is measured in virtual time. Apart from introducing another parameter that must be tuned, the disadvantage is a potentially smaller speedup in some cases, since superpages are not created as quickly as they could be.

Another technique that can be used to improve the effectiveness of superpages when contiguity is scarce is to rely on past history to predict the future, and use appropriate amounts of contiguity to create superpages only where it is likely that they provide benefits. For instance, our experience shows that most processes have a minor overhead in instruction TLB misses. This fact could be used to simply not waste contiguity in memory regions that hold code. Or, instead of treating all processes equally, we could use code-segment superpages only for those processes that are running a binary that has shown high instruction TLB miss overhead in the past, provided it is feasible to monitor TLB misses. One issue that this approach must consider is that process behaviour, particularly data TLB miss overhead, is likely to depend on the input size and characteristics.

Another area that has room for further research is what to do with extra processor cycles in the idle loop, after the compactor has done its part. In the previous chapter we already hinted at the idea of reducing the impact of the contiguity-aware page daemon, which is caused mainly by the deviation of the page replacement algorithm from the original approximate LRU. By moving pages around in a way such that the pages to be chosen next according to the original replacement algorithm are physically contiguous, this deviation can be minimized. Similarly, inactive pages that wind up as part of a reservation can be moved elsewhere to avoid breaking the reservation if such pages are accessed again. Finally, the zeroing out of free pages can be extended to consider pages that are part of reservations.

8.3 Conclusions

Superpages are physical pages of large sizes, which may be used to increase TLB coverage and thus improve application performance because of fewer TLB misses. However, supporting superpages poses several challenges to the operating system, in terms of superpage allocation, promotion trade-offs, and fragmentation control. This dissertation analyzes these issues and presents a complete, transparent, and effective solution to the problem of superpage management in operating systems. It describes a practical design and demonstrates that it can be integrated into an existing general-purpose operating system.

The system balances various trade-offs by using preemptible reservation-based allocation for superpages, an opportunistic policy to determine reservation sizes, and incremental promotion and speculative demotion techniques. The system is evaluated on a range of real workloads and benchmarks, on two different platforms, observing performance benefits of 30% to 60% in several cases. The overall improvement for the CPU2000 integer benchmarks is 11.2% in an Alpha platform and 12.7% in an Itanium-based machine. The overheads are small, and the system is robust even in pathological cases.

Support for multiple page sizes, from small to very large ones is important because superpage size requirements are different not only across processes, but sometimes also within processes. The system supports multiple sizes in a way that scales to very large superpages. However, typical page-table designs are not friendly towards supporting many sizes at once. To keep page-table overhead low, the system dynamically picks a limited number of sizes to support. The way in which these sizes are picked match the actual requirements very well, and hence the worst-case page-table overhead is reduced with no noticeable impact in the speedup achieved in the benchmarks.

Contiguity restoration is indispensable to sustain these benefits under complex workload conditions and memory pressure. Two low-impact complementary techniques that effectively keep memory fragmentation bounded were proposed and evaluated. One is based in a contiguity-aware page replacement algorithm, and the other is a greedy superpage-oriented compaction algorithm.

Bibliography

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107. ACM Press, 1991.

[2] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining data cache access with fast address calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 369–380. ACM Press, 1995.

[3] 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.

[4] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 243–253, 1994.

[5] E. Balkovich, W. Chiu, L. Presser, and R. Wood. Dynamic memory repacking. *Communications of the ACM*, 17(3):133–138, 1974.

[6] O. Ben-Yitzhak, I. Goft, E. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In *Proceedings of the Third International Symposium on Memory Management*, pages 100–105, Berlin, Germany, June 2002. ACM Press.

[7] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 158–170, San Jose, CA, 1994.

[8] D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Shienbrood, J. V. Sciver, and P. Wang. OSF/1 virtual memory improvements. In USENIX, editor, *Proceedings of the USENIX Mach Symposium*, pages 87–103, Berkeley, CA, 1991. USENIX.

[9] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: a software approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, New York, NY, 1989. ACM Press.

[10] A. Braunstein, M. Riley, and J. Wilkes. Improving the efficiency of UNlX file buffer caches. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 71–82, Dec. 1989.

[11] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor computer servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, 1994.

[12] C. Chao, M. Mackey, and B. Sears. Mach on a virtually addressed cache architecture. In *Mach Workshop Conference Proceedings, October 4–5, 1990. Burlington, VT*, Berkeley, CA, 1990. USENIX.

[13] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 114–123, Gold Coast, Australia, May 1992.

[14] R. Cheng. Virtual address cache in UNIX. In *Proceedings of the Summer 1987 USENIX Conference: Phoenix, AZ*, pages 217–224, Berkeley, CA, Summer 1987. USENIX.

[15] T. cker Chiueh and R. H. Katz. Eliminating the address translation bottleneck for physical address cache. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–148, 1992.

[16] 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.

[17] E. Cooper, S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 43–52. ACM Press, 1992.

[18] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[19] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[20] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 28(1):213–216, 1996.

[21] C. Dougan, P. Mackeras, and V. Yodaiken. Optimizing the idle task and other MMU tricks. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 229–237, 1999.

[22] R. P. Draves. Page replacement and reference bit emulation in Mach. In *Proceedings of the USENIX Mach Symposium*, pages 201–212, Berkeley, CA, USA, 1991. USENIX.

[23] B. et al. A 32 bit microprocessor with on-chip virtual memory management. In *Proceedings of the IEEE International Solid State Conference*, pages 178–179, 1984.

[24] 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.

[25] 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.

[26] J. Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, 1961.

[27] 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, 1998.

[28] 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, 1998.

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

[30] J. R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 72–81. IEEE Computer Society Press, 1987.

[31] B. Haddon and W. Waite. A compaction procedure for variable length storage elements. *The Computer Journal*, 10(2):162–165, Aug. 1967.

[32] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. Technical Report CS-TR-91-1394, Stanford University,

Department of Computer Science, Oct. 1991.

[33] J. L. Hennessy and D. A. Patterson. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, second edition, 1996.

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, third edition, 2002.

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

[36] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Berkeley, CA, May 2003. USENIX.

[37] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–51, San Diego, CA, May 1993. IEEE Computer Society Press.

[38] L. Iftode, J. P. Singh, and K. Li. Understanding the performance of shared virtual memory from an applications perspective. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 122–133. ACM Press, May 1996.

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

[40] J. Inouye, R. Konuru, J. Walpole, and B. Sears. The effects of virtually addressed caches on virtual memory design and performance. *ACM SIGOPS Operating Systems Review*, 26(4):14–29, 1992.

[41] Intel 80321 I/O processor DMA and AAU library APIs and testb ench. Technical Report 273921-001, Intel Corp, 2001.

[42] B. Jacob and T. Mudge. Software-managed address translation. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 156–167, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[43] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, July/Aug. 1998.

[44] T. Kagimasa, K. Takahashi, T. Mori, and S. Yoshizumi. Adaptive storage management for very large virtual/real storage systems. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 372–379, New York, NY, June 1991. ACM Press.

[45] 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.

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

[47] 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.

[48] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 131–139, Jerusalem, Israel, 1989. IEEE Computer Society Press.

[49] 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, 1993.

[50] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, pages 264–274. ACM Press, 2000.

[51] D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, Massachusetts, 1968.

[52] F. F. Lee. Study of 'look aside' memory. *IEEE Transactions on Computers*, 18(11):1062–1064, 1960.

[53] R. B. Lee. Precision architecture. *Computer*, 22(1):78–91, Jan. 1989.

[54] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[55] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88. ACM Press, 1990.

[56] W. L. Lynch, B. K. Bray, and M. J. Flynn. The effect of page allocation on caches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 222–225. IEEE Computer Society Press, 1992.

[57] J. Mauro and R. McDougall. *Solaris Internals*. Prentice-Hall, Upper Saddle River, NJ, 2001.

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

[59] M. Milenkovic. Microprocessor memory management units. *IEEE Micro*, 10(2):70–85, Apr. 1990.

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

[61] D. Mosberger and S. Eranian. *IA-64 Linux kernel: design and implementation*. Hewlett-Packard professional books. Prentice-Hall PTR, Upper Saddle River, NJ, 2002.

[62] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, Dec. 2002.

[63] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *Operating Systems Review*, 21(5):3–4, 1987.

[64] Y. Park, R. Scott, and S. Sechrest. Virtual memory versus file interfaces for large, memory-intensive scientific applications. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, Nov. 1996. IEEE.

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

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

[67] J. Protić, M. Tomašević, and V. Milutinović. *Distributed shared memory: concepts and systems*. IEEE Press, Aug. 1997.

[68] B. Randell and C. J. Kuehner. Dynamic storage allocation systems. *Communications of the ACM*, 11(5):297–306, 1968.

[69] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Bar, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor operating system. *IEEE Transactions on Computers*, C-37:896–908, 1988.

[70] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In USENIX, editor, *Proceedings*

of the First USENIX Symposium on Operating Systems Design and Implementation, Monterey, CA, pages 255–266, Berkeley, CA, Nov. 1994. USENIX.

[71] 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.

[72] 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, 1995.

[73] M. Satyanarayanan and D. P. Bhandarkar. Design trade-offs in VAX-11 translation buffer organization. *Computer*, 14(12):103–111, Dec. 1981.

[74] M. Satyanarayanan and D. P. Bhandarkar. Design trade-offs in VAX-11 translation buffer organization. *Computer*, 14(12):103–111, Dec. 1981.

[75] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, Feb. 1994.

[76] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, 2000. ACM SIGARCH, IEEE.

[77] J. E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8):433–440, 1975.

[78] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison Wesley, 4. edition, 1994.

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

[80] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[81] A. J. Smith. Bibliography on paging and related topics. *Operating Systems Review*, 12(4):39–49, Oct. 1978.

[82] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[83] *SPARC64-III User's Guide*. HAL Computer Systems, Campbell, CA, 1998.

[84] W. Stallings. *Operating Systems*. Prentice Hall, Englewood Cliff, NJ, 1997.

[85] 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, 1998.

[86] M. Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. PhD thesis, University of Wisconsin, Madison, 1995.

[87] 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.

[88] 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, 1995.

[89] 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.

[90] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliff, NJ, 1992.

[91] G. Taylor, P. Davies, and M. Farmwald. The TLB slice—a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 355–363. ACM Press, 1990.

[92] P. J. Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, June 1990.

[93] 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.

[94] U. Vahalia. *UNIX Internals*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[95] B. Wald. Utilization of a multiprocessor in command control. *Proceedings of the IEEE*, 54(12):1885–1888, 1966.

[96] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148. ACM Press, 1989.

[97] A. Wiggins, S. Winwood, H. Tuch, and G. Heiser. Legba: Fast hardware support for fine-grained protection. In *Proceedings of the 8th Australia-Pacific Computer Systems Architecture Conference (ACSAC'2003)*, Aizu-Wakamatsu City, Japan, Sept. 2003.

[98] 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.

[99] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the im-

plementation of a Multiprocessor Operating System. In *Proceedings of the 11th Symposium on Operating System Principles*, 1987.

[100] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, 2001.