To appear in "Proceedings of the 22nd Annual International Symposiumon Computer Architecture," 1995.

Reducing TLB and Memory Overhead Using Online Superpage Promotion

Theodore H. Romer Wayne H. Ohlrich Anna R. Karlin
Brian N. Bershad
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{romer,ohlrich,karlin,bershad}@cs.washington.edu

Abstract

Modern microprocessors contain small TLBs that maintain a cache of recently used translations. A TLB's *coverage* is the sum of the number of bytes mapped by each entry. Applications with working sets larger than the TLB coverage will perform poorly due to high TLB miss rates. Superpages have been proposed as a mechanism for increasing TLB coverage. A *superpage* is a virtual memory page with size and alignment that are a power of two multiple of the system's base page size. In this paper, we describe online policies for superpage management that monitor TLB miss traffic to decide when a superpage should be constructed. Our policies take into account both the benefit of a superpage promotion (potential for preventing future misses) and the cost (page copying). Although our approach increases the cost of each TLB miss, the net effect is to improve total execution time by eliminating a large number of misses without significantly increasing memory usage, thereby improving system performance.

1 Introduction

The performance of the Translation Lookaside Buffer (TLB) has become increasingly important to overall application performance. It is now common to find systems configured with several hundred megabytes or even gigabytes of primary storage in order to accommodate the demanding memory requirements of applications such as database, multimedia, parallel and scientific codes and voice recognition systems. Unfortunately, TLB coverage, which is the amount of virtual memory that can be directly accessed without incurring a TLB miss, has not scaled accordingly. Because modern systems typically incur a penalty of between 10 and 30 cycles per TLB miss [Kane & Heinrich 92, Dutton et al. 92], any application with a working set larger than the TLB's coverage can spend a significant fraction of its time waiting for TLB misses to be serviced [Chen et al. 92, Bala et al. 94, Talluri et al. 92].

This research was sponsored by the Office of Naval Research through a Research Initiation Award, and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award and a Presidential Faculty Fellowship. Karlin was supported by a grant from the National Science Foundation.

Superpages offer a way to increase TLB coverage without increasing the number of TLB entries. A superpage is made up of two or more base virtual pages that are contiguous in both virtual and physical memory. Hardware implementations of superpages typically restrict superpage size and alignment to a power of two of the base page size. This dual contiguity allows the pages to be represented by a single translation entry. Superpages have been used effectively to map large, static regions of memory such as the operating system's text or the window manager's frame buffer. They have not yet been used to support general applications, which have more dynamic memory requirements.

We believe the reason for this lack of support is straightforward: demonstrably good memory management policies for superpages have been elusive [Talluri et al. 92, Talluri & Hill 94, Khalidi et al. 93]. A cost/benefit analysis is required to determine if the overhead of constructing a superpage is outweighed by its benefit. Overhead arises because a superpage must map a physically contiguous range of memory, and the component pages of a candidate superpage may not be physically contiguous at the time of candidacy, requiring a copy. Benefit results because a single TLB entry can provide greater coverage.

In this paper, we address the problem of performing this cost/benefit analysis automatically at runtime. We describe online policies that take into account both past miss cost and superpage construction cost in order to determine which and when pages should be promoted into a larger superpage. Accounting takes place during the handling of TLB misses. While this approach increases the cost of each miss, we show through simulation that the total number of misses is substantially reduced, improving overall system performance. Our online superpage management policies reduce TLB overhead by as much as 99%, and overall execution time by as much as 48%, when compared to a system using small fixed-size pages. When compared to a system using large fixed-size pages, TLB overhead is approximately the same, but memory consumption is substantially less. We show that the performance of our online policies across a range of applications comes close to a nearly optimal offline allocation policy that uses future reference knowledge to allocate superpages. We also show that policies that do not consider previous reference patterns and promotion costs perform worse than those that do, in terms of TLB miss overhead, memory consumption, or both.

Our online policies require modest architectural support which can already be found in modern systems [Kane & Heinrich 92, Dig 92], namely a software-managed TLB with the ability to map entries of variable size. The policies can be implemented entirely within the machine-dependent layer of the virtual memory system, enabling all other components of the operating system to rely on a virtual memory interface based on small, fixed-size pages.

1.1 The rest of this paper

The rest of this paper is structured as follows. In Section 2 we discuss related work. In Section 3 we describe our experimental methodology. In Section 4 we motivate dynamic page sizes by showing that large superpages can improve application performance, but that no single statically allocated page size is appropriate for all applications. In Section 5 we describe the operating system and hardware mechanisms required to support our superpage policies. In Section 6 we describe the principles behind the design of our online superpage promotion policies. In Section 7 we present several promotion policies. In Section 8 we describe the performance impact of these policies on a range of common applications using trace-driven simulation. Finally, in Section 9 we present our conclusions.

2 Related work

Our online policies rely on a cost/benefit analysis that can be traced to theoretical work in competitive algorithms. Competitive algorithms make online decisions that result in performance within a constant factor of an optimal offline algorithm. Prior research in this area has influenced, for example, the design of synchronization [Karlin et al. 91], paging [Sleator & Tarjan 85], and cache management algorithms [Cao et al. 94].

Others [Chen et al. 92, Mogul 93, Khalidi et al. 93] have described the potential positive impact of a system that supports superpages, although they do not describe policies for promotion or demotion. Instead, they suggest that the programmer or compiler offer the operating system a hint about the appropriate page size for a particular range of memory.

Researchers at Wisconsin [Talluri & Hill 94] present a simple policy for page promotion in a system that supports two page sizes: 4 KB and 64 KB. They are primarily concerned with minimizing the software complexity of superpage management, and propose hardware that allows the operating system to declare invalid subpages within a 64 KB superpage. The Wisconsin approach relies on "page reservation," whereby pages are initially allocated from physical memory where they would ultimately lie when placed within a superpage. In this way, page promotion does not require any copying and can occur at no cost.

The approach described in this paper differs from Wisconsin's in three ways that enable greater TLB coverage and a promotion policy that is more flexible but does not require additional hardware. First, we permit the TLB to map superpages with sizes up to several megabytes. This allows applications with working sets that would exceed the capacity of a TLB with 64 KB entries to fit entirely within the TLB. Second, our policies do not rely on page reservation, which can limit the gain made possible by superpages. Specifically, reservation makes it difficult to construct superpages of varying size, since there is no way to decide *a priori* how many base pages to reserve at page allocation time. For small superpages, reservations may often hold until they are needed. As the superpage size grows, however, it is increasingly likely that a reservation will fail: that is, some base page within the reserved superpage is allocated for another purpose before a promotion occurs. The final difference is that the Wisconsin superpage policy only creates superpages if they are part of 64 KB page that is between 50% and 100% populated. Smaller superpages and large sparsely populated superpages that could eliminate TLB misses will not be created with this policy. To overcome this inflexibility, the Wisconsin study proposes subpage valid bits in each TLB entry to indicate that a particular subpage is not accessible through a referenced superpage. This new hardware allows partial superpages to be constructed with "holes" in the superpage where base pages are either not resident or not in a contiguous, aligned region of memory. In contrast, our strategy has the flexibility to create superpages with sizes tailored to the workload's memory reference pattern, and requires only that the TLB map pages with multiple sizes.

3 Methodology

We measured system behavior using trace-driven simulation of a collection of benchmarks, which are described in Table 1. We used five of the benchmarks (*compress, nasa7-5, nasa7-4, fft,* and *gcc*) to develop and tune the online policies. Once the policy parameters were established, we augmented this "training set" with the remaining five benchmarks (*coral, fpga, cecil, atom,* and *spice*), but did not make any further changes to the parameters. We used ATOM, a binary rewriting tool from DEC WRL [Srivastava & Eustace 94], to simulate the TLB behavior of the applications. The simulated system had two 32-entry fully-associative TLBs, one for instructions and one for data, and used an LRU replacement policy.

Our primary performance metric in this study is the TLB miss penalty per instruction, which is the total number of cycles spent in service of the TLB divided by the total number of user instructions executed. For convenience, we use the term "TLBMCPI" here (TLB memory cycles per instruction). On a single-issue machine with an infinite cache, overall CPI (cycles per instruction) would be 1+TLBMCPI. Our second performance metric is "Memory Usage Overhead," which is the percentage increase in memory consumed due to internal fragmentation compared to a system with fixed-size 4 KB pages.

Benchmark	Description
coral	The Wisconsin coral database performing a nested join [Ramakrishnan et al. 93].
compress	*compress* compressing a 977 KB input file.
nasa7-5	Gaussian elimination for a 3.8 MB matrix.
nasa7-4	Block tridiagonal matrix solver for a 131.8 KB matrix.
fpga	Logic bipartitioner targeted to FPGAs, using a 1626 KB input file [Hauck & Borriello 95].
cecil	*cecil* compiling a 21 KB input file [Chambers 93].
atom	*atom* instrumenting a 2.6 MB binary with our TLB simulator.
fft	Fast fourier transform of a 32 MB array.
spice	the *spice* circuit simulation benchmark on the 15 KB reference input set.
gcc	*gcc* compiling a 109 KB input file.

Table 1: *This table describes the applications considered in this study. compress, gcc, the two nasa7 programs, and spice are drawn from the SPEC92 benchmark suite. Coral is a benchmark used in the Wisconsin TLB study. Atom, cecil, and fpga are tools used in our local research environment.*

4 Motivation

Two aspects of performance are affected by page size: the number of TLB misses and memory utilization. Large pages can reduce the number of TLB misses, but may also waste memory due to internal fragmentation. Small pages can increase the number of misses, but use memory more efficiently since the average fragment size is smaller.

We reveal in this section and demonstrate in those following that variable-size superpages offer the best of both worlds.

A system with small fixed-size pages can cause applications with large working sets to spend a significant fraction of total execution time handling TLB misses. We ran each of the benchmarks on a DEC Alpha 3000/700, and measured the total execution time and the time spent handling TLB misses. These measurements are summarized in Table 2. The table shows that the applications we considered spend between 5% and 41% of their time in the TLB miss handler on the DEC Alpha 3000/700, which has fixed-size 8 KB pages. The majority of TLB misses were due to data references.

Benchmark	Exec. Time (s)	TLB Misses (100s)	$\frac{\text{TLB Time}}{\text{Exec. Time}}$ (%)
coral	51.6	1,242,456	41.4
compress	1.1	25,650	35.2
nasa7-5	9.0	205,845	34.7
nasa7-4	1.0	23,787	34.4
fpga	53.8	806,331	28.4
cecil	126.2	969,255	16.6
atom	9.8	94,304	16.1
fft	43.7	294,899	10.7
spice	210.5	1,212,271	9.4
gcc	1.5	4,389	5.2

Table 2: *Baseline performance. The TLB performance of the benchmark suite was measured on a DEC Alpha 3000/700 running DEC OSF/1 2.1. This system contained a 225 Mhz Alpha 21064 processor with a 32 entry DTLB and an 8 entry ITLB, a 2 MB offchip cache, and 160 MB of main memory. The execution time is the average of five runs on an unloaded system, after an initial run to warm the file buffer cache. The TLB miss statistics were collected using a TLB miss handler instrumented with the Alpha cycle counter.*

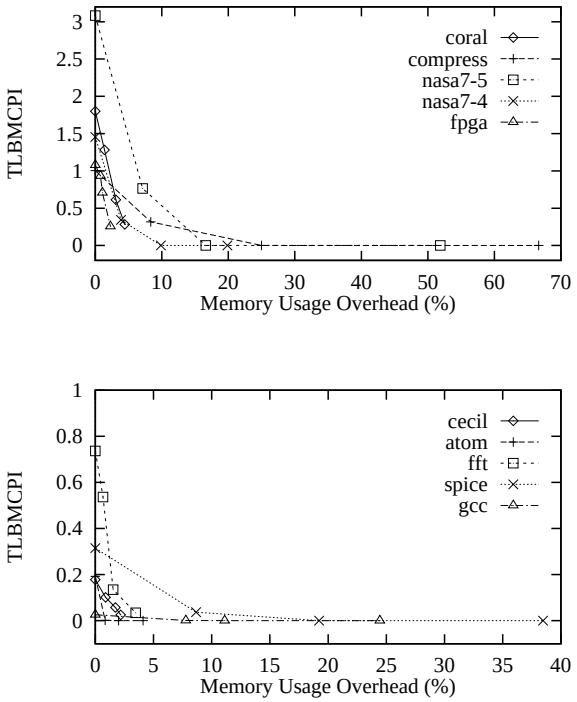


Figure 1: *TLB overhead as a function of memory consumption for a range of page sizes. This graph shows that increasing the page size can sometimes reduce TLBMCPI as well as increase memory consumption. For each program, each point on the line represents a different page size, ranging from 4 KB (the points with the highest TLB miss rate) to 256 KB (the points with the highest memory usage). Note that the two graphs use different scales.*

A larger page size can result in both lower TLBMCPI (due to increased TLB coverage) and higher memory consumption (due to increased fragmentation). We illustrate this tradeoff in Figure 3, which shows TLBMCPI as a function of memory overhead as the page size varies. While an ideal system would have both TLBMCPI and memory overhead of zero, this figure shows that there is no single page size that approaches this ideal point for any of the applications that have significant TLB overhead. With fixed-sized pages, performance is constrained to follow these curves. With an appropriate superpage policy, however, large pages can be constructed where memory is used densely and frequently, and small pages elsewhere. In the following sections, we describe mechanisms and policies that attain performance close to the origin on this tradeoff graph.

5 Mechanisms

Superpages require both operating system and hardware support. The operating system must support two fundamental operations: promotion and demotion. Page promotion occurs whenever two or more small pages are combined into a superpage. Page demotion occurs whenever a superpage is broken down into two or more smaller pages. Promotion of two smaller pages with contiguous and aligned virtual frame numbers V1 and V2, which are mapped to physical frame numbers P1 and P2, requires that zero, one, or both of pages P1 and P2 be copied to a new range of physical memory to ensure that they are contiguous. Furthermore, P1 and V1 must be aligned at addresses that are a multiple of the containing superpage's size.

In terms of hardware, superpages require that the TLB allow the simultaneous mapping of pages having more than one size. This requires that each TLB entry maintain additional bits that mask off the lower bits of the requested virtual page so that all base pages that are part of the same superpage are matched by the same TLB entry. This additional logic can be found in several contemporary processors [Kane & Heinrich 92, Blanck & Krueger 92, Dig 92].

There is a tension between superpages and other operating system mechanisms that rely on uniformly sized pages, such as the file system's buffer cache, copy-on-write virtual memory [Young et al. 87], and LRU-clock [Babaoğlu & Joy 81]. To resolve this tension, superpage management can be concealed entirely within the machine-dependent layer of the operating system's virtual memory system (MD-VM) [Rashid et al. 87]. When clients of the virtual memory system request operations on base pages that are component to superpages, the MD-VM layer can demote any superpages containing the target base pages so that the requested operation can be performed. In practice, most such demotions occur when the virtual memory system gathers reference information for its page replacement algorithm. If the replacement policy is a simulation of LRU using a two-handed clock, and the system is not paging, then the system only needs reference information about a small fraction of pages at any given time,

and demotions will occur infrequently. More esoteric functions that require fine-grained reference information [Appel & Li 91], such as write-trapping for a distributed shared memory [Carter et al. 91], will increase the frequency of page demotion. As a result, superpages may not be appropriate for applications that rely on such functions. However, recent results indicate that the use of the virtual memory system to detect application-level accesses can be less efficient than strategies that rely on the compiler [Hosking & Moss 93, Zekaukas et al. 94].

6 Policy design principles

In this section, we discuss the principles underlying the design of effective online superpage construction policies. The goal of our policies is to obtain TLB miss overhead, including copy costs, close to that of the optimal offline policy, which minimizes total TLB miss overhead. The overhead of the optimal offline policy establishes a lower bound on the performance of *any* policy. A good policy can achieve this goal by creating superpages early enough to bound the performance loss of not having a superpage, but late enough so that a promotion is unlikely to incur more overhead than it saves. Our policies do not explicitly attempt to optimize memory consumption. In practice, the online policies tend to create superpages in densely populated regions, so the increase in memory consumption is small.

Our approach for determining when and where to promote is straightforward. We monitor TLB miss traffic, and on each miss update reference information to reflect the number of TLB misses that would *not* have occurred had the set of already assigned superpages been larger. We then use this information to construct new superpages.

We assume that a superpage consists of a number of base pages that is divisible by a power of 2, is no smaller than the base page (4 KB), and can be as large as 8 MB. We also assume worst-case promotion overhead (no reservation), which requires that we move all of the data into the contiguous memory backing the newly created superpage. A single base page may be involved in a sequence of promotions as it gradually becomes component to larger and larger superpages.

6.1 Policy components

Each TLB miss is charged to each of the potential superpages that would have prevented the miss if it had already been created. As the number of misses charged to a potential superpage P grows, by locality of reference we expect that promoting to P will eliminate future TLB misses. Therefore, as soon as P's miss charge exceeds a certain threshold, P is constructed. In the rest of this section we discuss how to maintain the counters that reflect the history of past TLB misses, and how to select a promotion threshold based on these counters.

Maintaining miss charges

We charge each TLB miss to any superpage in the system that could have prevented the miss. A superpage can prevent a miss to a page p in one of two ways. First, the superpage may cover both p and a page already in the TLB. In this case the miss would have been prevented if the superpage had already been constructed, because the translation for the page already in the TLB would have covered p. Second, a superpage may increase the capacity of the TLB, so that a prior translation for p would not have been evicted from the TLB, avoiding the miss. We can tally preventable misses by maintaining two counters that keep track of the two different miss charges to a potential superpage P: *prefetch*(P) and *capacity*(P). The counters are updated on a TLB miss to a page p as follows:

- *Prefetch* charges are updated on each miss by incrementing *prefetch*(P) for each potential superpage P that contains the currently referenced page p and one or more TLB entries.
- *Capacity* charges are computed by examining the TLB miss stream at the time of the reference to p. *Capacity*(P) is incremented for each potential superpage P that coalesces enough TLB entries to have kept p from having been evicted from the TLB in the past. (The TLB miss stream is maintained in a LRU stack data structure that reflects the set of referenced pages in order of most recently referenced to least recently referenced. That is, if the LRU stack at some time is $(p_1, p_2, \ldots, p_n)$, then p_i is the ith most recently referenced page.) [1]

We can illustrate the use of these counters with an example, depicted in Figure 2. Consider a TLB with three entries. Immediately after servicing the stream of virtual page references 8,1,7,6,5,0 the TLB contains translations for pages 0, 5 and 6. Suppose the next reference is to virtual page 1, resulting in a miss. If superpage $\{0, 1\}$ had been created from the start, this miss would not have occurred: on the prior reference to page 0, a translation for $\{0, 1\}$ would have entered the TLB, and the subsequent reference to page 1 would have hit. Therefore, we increment *prefetch*($\{0, 1\}$). Alternatively, if superpage $\{4, 5, 6, 7\}$ had been created from the start, this miss would not have occurred. This can be seen by observing the entire LRU stack prior to the reference to page 1. In order of most recently to least recently referenced, the LRU stack contains 0,5,6,7,1,8. If $\{4, 5, 6, 7\}$ had been initially promoted, the LRU stack would reduce to 0,$\{4, 5, 6, 7\}$,1,8. Page 1 is now one of the top 3 entries in the stack and, under LRU replacement, its translation would be in the TLB. Since the promotion of $\{4, 5, 6, 7\}$ would have prevented virtual page 1 from being evicted from the TLB, we increment *capacity*($\{4, 5, 6, 7\}$).

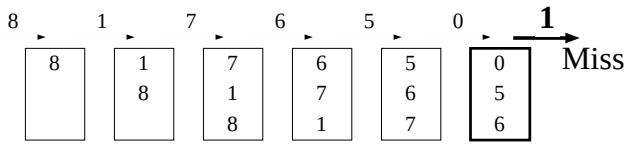
The two counters differ substantially in the information they reflect. The promotion of $\{0, 1\}$ in the previous example effectively causes a prefetch of the translation for virtual page 1, since the first reference to page 0 causes a translation for $\{0, 1\}$ to enter the TLB. On the other hand, the promotion of $\{4, 5, 6, 7\}$ effectively increases the capacity of the TLB by reducing a set of entries in the LRU stack to a single entry, thereby preventing the prior eviction of some other entry.

We show in the Section 8 that in practice the prefetch counter is more important than the capacity counter in identifying the best potential superpage. Fortunately, the prefetch counter is the less expensive to maintain of the two. A simple way to compute prefetch charges is to scan the TLB on a miss to page p and check if some potential superpage contains both p and a current TLB entry. A more efficient algorithm for maintaining prefetch charges is given in Appendix A. On the other hand, determining capacity charges requires scanning both the TLB and the LRU stack of pages referenced prior to the last reference of p to determine if there is a potential superpage that coalesces enough entries to have prevented p's eviction.

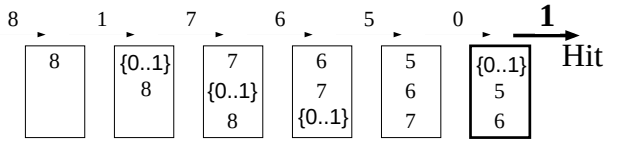
Choosing the threshold

The prefetch and capacity charges combine to give a total miss charge for each potential superpage P. Miss charges on a superpage indicate

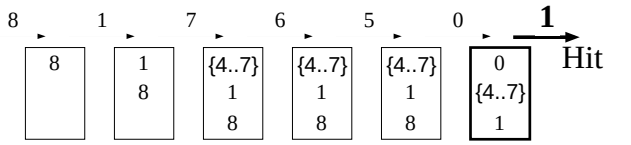
[1] We describe the LRU stack as a totally ordered list to simplify the exposition. Since information about the LRU stack is updated on a TLB miss, we are only able to record the *set* of pages that have translations in the TLB in addition to the precise LRU stack of pages that do not have translations in the TLB. This information is sufficient, though, to accurately update capacity charges.



(a) *A sequence of TLB references with no superpages. In this scenario, the final reference to page 1 results in a TLB miss.*



(b) *The same sequence of TLB references with the superpage {0,1}. In this scenario, the reference to page 0 effectively* prefetches *the TLB entry for page 1, so that the final reference is a TLB hit.*



(c) *The same sequence of TLB references with the superpage {4,5,6,7}. In this scenario, the existence of this large superpage effectively increases the* capacity *of the TLB, so that the final reference to page 1 results in a TLB hit.*

Figure 2: *Example of prefetch and capacity charges. Each part of the figure shows the changing contents of the TLB when presented with the reference stream of virtual pages 8,1,7,6,5,0,1. In part (a), the final reference to page 1 results in a TLB miss. Part (b) shows why superpage {0,1} incurs a prefetch charge: had the superpage {0,1} existed, the final reference would have resulted in a TLB hit. Similarly, part (c) shows why superpage {4,5,6,7} incurs a capacity charge: had the superpage {4,5,6,7} existed, the final reference would have resulted in a TLB hit.*

that prior promotion to that page would have prevented TLB misses. In the presence of locality, a promotion will also prevent future misses. Therefore, when the miss charges to a potential superpage exceed a certain threshold, we promote the superpage. Our goal in choosing this threshold is to ensure that we promote a superpage early enough to avoid future misses, but late enough to ensure that the cost of promotion does not dominate TLB overhead.

Since an online promotion policy does not have information about future references, in the worst case it may promote pages that are never subsequently referenced. Nonetheless, we can still bound the overhead of an online policy. In the remainder of this section we show how to select a promotion threshold that limits the overhead of an online policy relative to the overhead of the optimal offline policy.

We begin by drawing an analogy with the ski-rental problem.[2] Consider a novice skier who is faced with the choice of buying skis at a cost of $100, or renting skis for some period of time, at a cost of $10 a day. The "optimal offline policy" is to buy skis on the first day if it were known that more than 10 days would be spent skiing. Otherwise, the skier is financially better off renting every time. Although the skier must make this decision online, without knowledge of the number of times she will go skiing in the future, she would nevertheless prefer to know that she will not make a decision "much worse" than the optimal offline decision. In this case, the right thing for her to do is to begin by renting skis. If she makes it to her eleventh excursion, she should then buy skis. This policy guarantees that she will spend no more than twice the optimal offline's expense no matter how many excursions she takes. (In the worst case, the optimal offline policy would have spent $100, whereas the skier would have spent $200.)

We can use the ski-rental problem to understand the problem facing an online superpage construction policy. An optimal offline superpage construction policy has the luxury of observing all future references prior to making a promotion. Suppose that there is one candidate N KB superpage P consisting of two $N/2$ KB pages, and that in the run of an application, m TLB misses will become hits if P is constructed at time 0. The optimal offline policy, knowing m, will perform the promotion at time 0 if m is at least the ratio of P's promotion cost (cycles lost due to copying) and the cost of a TLB miss. We call this ratio $R = \text{Promotion Cost}/\text{TLB Miss Cost}$. The optimal offline policy will never promote to P if $m < R$. An online policy that promotes when the miss charges to a page reach R is guaranteed to deliver performance no worse than twice the optimal offline with respect to this single superpage. If $m \leq R$, then both the online and offline policies pay for m TLB misses. If $m > R$, the offline policy pays for a promotion at time 0, while the online policy pays for R TLB misses *and* the promotion, for a total cost of two promotions.

In practice, the online policy of waiting until the miss charges exceed R is overly conservative. Because of locality in page reference patterns, most potential superpages either never accumulate miss charges exceeding a fraction of R, or else accumulate miss charges greatly exceeding R. Therefore, we can afford to use a promotion threshold substantially less than R. In the next section, we present data that support this hypothesis.

Resetting miss charges on promotion

The miss charge of a potential superpage should reflect the number of misses that promotion would have eliminated that were not *already* eliminated by previously promoted pages. So in addition to updating the counters on each TLB miss, the counters must be updated when promoting to a page P, to reflect the new, perhaps diminished benefit of subsequent promotions. To adjust the prefetch charges when P is promoted, *prefetch*(P') is decremented by *prefetch*(P) for all superpages P' containing P, since whenever *prefetch*(P) is incremented, *prefetch*(P') is also incremented. Adjusting the capacity charges when P is promoted is more difficult: for every superpage Q, *capacity*(Q) must be decremented for each miss that incremented both *capacity*(Q) and *capacity*(P).

7 Promotion policies

In this section we describe several promotion polices. Three of these policies – *OFFLINE*, *ONLINE*, and *APPROX-ONLINE* – are based on the principles described in the previous section. That is, they construct a superpage when enough miss charges have accumulated to offset the cost of promotion. We also discuss two policies, *ASAP* and *ASAP-4-64*, that do not consider the cost and benefit of superpage construction.

[2] This analogy was originally suggested by Larry Rudolph in explanation of the results described in [Karlin et al. 88].

7.1 *OFFLINE* – an approximation to the optimal offline policy

The optimal offline policy takes as input the complete sequence of page references made by an application. It outputs a set of superpages that minimizes the total cost of executing the application, considering the TLB miss overhead and promotion cost. We require that the offline policy pay a worst-case promotion cost for copying the component base pages. Otherwise, the offline policy would start with a single large superpage that covered the application's entire working set, and incur no TLB overhead and no promotion cost. The problem of computing the optimal offline solution is NP-complete by reduction from Set-Cover. Therefore, we use the following greedy approximation which we will call *OFFLINE*.

Like the optimal offline policy, *OFFLINE* reduces future TLB misses by creating appropriate superpages. Unlike the optimal solution, however, it greedily selects the superpages to promote. The value of a potential superpage is the ratio of TLB miss cycles eliminated to the promotion cost in cycles. Initially, *OFFLINE* analyzes the TLB miss stream and promotes the superpages with the highest ratio of misses eliminated to promotion cost. *OFFLINE* then continues promoting incrementally, examining a revised TLB miss stream based on the current set of superpages, until it can find no superpages with benefit exceeding the promotion cost.

Clearly, *OFFLINE* is infeasible in a real system, as it relies on information about future reference patterns of the application. Nevertheless, its performance offers an approximate lower bound on the performance of any algorithm that starts with small base pages and pays worst-case copy cost for promotions.

7.2 *ONLINE* – the basic online algorithm

The algorithm *ONLINE* is based on the prefetch and capacity counters described Section 6. Once the miss counters for a superpage reach a certain threshold, *ONLINE* promotes the superpage. To determine an appropriate threshold, we analyzed the distribution of final miss charges for all potential superpages that were *not* constructed by *OFFLINE*. For these unpromoted superpages, 99% of the capacity charges were less than $0.625 \times R$, and 99% of the prefetch charges were less than $0.125 \times R$, where R is the ratio of promotion cost to TLB miss cost. An online policy that uses promotion thresholds above these values is unlikely to promote unnecessarily, whereas one that uses thresholds below may. Therefore, *ONLINE* promotes to a superpage P if *capacity*(P) exceeds $0.625 \times R$, or *prefetch*(P) exceeds $0.125 \times R$. When a promotion occurs, the prefetch charges are decremented for each superpage containing the promoted superpage, as described in Section 6. We do not decrement the capacity charges precisely on a promotion, as properly resetting them requires keeping track of the capacity counters to which each miss contributed. Because this involves considerable bookkeeping overhead, we instead simply reset capacity charges to zero on a promotion. Although this is overly conservative, we found that our results were relatively insensitive to how sharply we reduced capacity charges following a promotion. Despite the simplification, *ONLINE* has high overhead, both in terms of time and space, since it maintains detailed information about the miss charges for each superpage, *and* requires an LRU stack of TLB references.

7.3 *APPROX-ONLINE* – an approximate online algorithm

The algorithm *APPROX-ONLINE* is the same as *ONLINE* except that it only maintains prefetch charges. As a result, *APPROX-ONLINE* does not incur the overhead of maintaining the capacity counters. When *prefetch*(P) reaches $0.125 \times R$, the superpage P is created and the *prefetch* counters for larger superpages containing P are decremented by the threshold value. In Section 8 we will see that *APPROX-ONLINE* is nearly as effective as *ONLINE* in eliminating TLB misses, although it has much lower bookkeeping overhead.

7.4 *ASAP* – the as-soon-as-possible algorithm

ASAP promotes a superpage as soon as all of its component base pages have been referenced, without considering the frequency of reference to those pages. Although it requires minimal bookkeeping, it has two drawbacks. First, the algorithm may copy too frequently, since it creates superpages without concern for promotion cost. Even if pages are rarely referenced, the algorithm will still pay the cost to create a superpage. Second, *ASAP* may fail to create beneficial superpages if any one of the component base pages is not referenced. We also simulate a variant of *ASAP*, called *ASAP-4-64*, that allows only two page sizes: 4 KB and 64 KB. Once at least half of the pages in a potential superpage of size 64 KB have been referenced, *ASAP-4-64* directly promotes those pages.

We include *ASAP* because it is simple and offers a plausible alternative to the more sophisticated policies, and *ASAP-4-64* because it is similar to a policy used in other studies [Talluri et al. 92]. We do not consider the impact of page reservation, so *ASAP* and *ASAP-4-64* incur copy costs. Accurately assessing the impact of reservation is difficult, since the effectiveness of reservation depends on the level of contention for memory, and this in turn depends on the behavior of the entire system, not just a single application. In a system in which memory is plentiful, reservations always succeed, and *ASAP* and *ASAP-4-64* incur no copy cost. In such a system they should have nearly the same TLBMCPI as a system with fixed-size 64 KB pages.

8 Simulation results

In this section we describe the performance of the policies presented in the previous section. We also consider the performance of systems with fixed-size pages, which we refer to as *FIXED 4 KB*, *FIXED 16 KB*, and *FIXED 64 KB*. We first discuss the space and time overhead of our bookkeeping strategies. We then present the impact on application performance by analyzing the results of trace-driven simulations.

8.1 Space and time considerations

The online policies introduce both space and time overhead. Space overhead is incurred by the counters associated with each potential superpage. These counters only need to be maintained for each 8 MB segment of virtual memory that is actually referenced, since we do not create superpages larger than 8 MB. Within an 8 MB segment, the number of potential superpages equals the number of allocated virtual base pages, since each superpage of size 2^i is composed of two lesser superpages of size 2^{i-1}. Our algorithm for maintaining the prefetch counters incurs additional space overhead for auxiliary data structures. As described in Appendix A, this results in an average of 3.125 counters per base page for *APPROX-ONLINE*. Altogether, even at one four-byte word per counter, the maximum space overhead

is 33 KB/8 MB (0.4%) for *ONLINE* and 25 KB/8 MB (0.3%) for *APPROX-ONLINE* using 4 KB base pages. Although not insignificant, the space required for the counters is less than the space wasted by internal fragmentation when larger fixed-size pages are used.

Every TLB miss also introduces time overhead. We account for the overhead of a TLB miss for the various policies as follows. All of the policies incur a baseline TLB miss overhead of 30 cycles. (This is consistent with TLB miss costs on current systems. For example, we measured a minimum TLB miss penalty on a DEC Alpha 3000/700 of 31 cycles.) In addition, the dynamic policies incur overhead to maintain bookkeeping information. *ONLINE* incurs the highest overhead, because on each TLB miss it must update both the prefetch and capacity counters. As described in Appendix A, updating the prefetch counters costs 100 cycles per TLB miss, and updating the capacity counters costs about 2470 cycles per TLB miss. Thus the total overhead per TLB miss for *ONLINE* is $30 + 100 + 2470 = 2600$ cycles per TLB miss. *APPROX-ONLINE* uses the prefetch counters alone, so the overhead per TLB miss is $30 + 100 = 130$ cycles per TLB miss. These cycle counts assume that the bookkeeping code does not incur any overhead due to cache misses. *ASAP* and *ASAP-4-64* are not charged any additional overhead, since the necessary bookkeeping can be performed as pages are mapped. Likewise, *OFFLINE* does not incur overhead beyond the baseline TLB miss cost, since it can perform all of its accounting computation offline. These overheads are summarized in Table 3.

Policy	TLB Miss Cost
FIXED (4 KB, 16 KB, 64 KB)	30
OFFLINE	30
ONLINE	2600
APPROX-ONLINE	130
ASAP	30
ASAP-4-64	30

Table 3: *Cost of a single TLB miss for each policy, computed as baseline TLB miss cost, plus any bookkeeping cost. The calculation of the bookkeeping cost for ONLINE and APPROX-ONLINE is described in Appendix A. The value for ONLINE reflects the high overhead of maintaining capacity counters. ASAP and ASAP-4-64 can perform the necessary bookkeeping as pages are mapped, while OFFLINE can perform its bookkeeping offline, so these three policies have the same overhead as FIXED.*

All of the promotion policies incur time overhead when creating a new superpage. We charge a copy cost of 3,000 cycles per 1 KB copied during a promotion. This cost reflects both the instructions to perform the copy and stall cycles due to cache misses that may occur during the copy. Combining the copy cost with the 30 cycle TLB miss cost yields a value of R (the ratio of the copy cost to the baseline TLB miss cost) of 100 per 1 KB of a superpage. For example, an 8 KB superpage has $R = 800$. In Section 6.1 we showed that an online policy that promotes a superpage when the miss charges reach a threshold of R has cost at most twice that of the optimal offline cost for that superpage. That analysis ignored the additional overhead the online policies incur on each TLB miss. In particular, the ratio between promotion cost and TLB miss cost is less than R for *ONLINE* and *APPROX-ONLINE*. Intuitively, it might seem that these policies should use a different promotion threshold in order to best bound their overhead compared to the optimal offline algorithm. In Appendix B we show that this is in fact not the case: despite the increased cost of each TLB miss, using a threshold of R still minimizes the ratio between the cost of the online and offline algorithms.

Benchmark	*FIXED 4 KB*	*FIXED 16 KB*	*FIXED 64 KB*	*OFFLINE*	*ASAP*	*ASAP-4-64*	*ONLINE*	*APPROX-ONLINE*
coral	1,205,951	858,980	410,424	405	1,193,847	412,253	12,517	16,107
compress	29,198	9,201	0	1	2	1	85	117
nasa7-5	407,140	100,880	5	3	6	8	249	249
nasa7-4	30,044	7,084	3	38	8	18	316	352
fpga	744,183	643,967	487,609	251	734,965	489,516	8,827	10,446
cecil	482,885	274,927	156,059	42,006	433,048	158,025	28,731	35,497
atom	49,511	428	70	930	1,164	128	864	1,041
fft	317,447	231,161	58,162	10	83	58,199	4,112	4,112
spice	1,852,305	215,924	355	3	1,002,570	380	605	823
gcc	1,207	48	0	237	21	2	187	232

Table 4: *The number of TLB misses, in 100s, that occurred using each policy.*

8.2 TLB Misses and TLBMCPI

Table 4 shows the impact of the policies described in Section 7 on TLB miss counts. All of the promotion policies are capable of eliminating the majority of TLB misses for many of the programs. However, *ASAP* performs poorly for programs with fragmented memory access patterns, such as *coral, fpga, cecil,* and *spice*, since it never promotes a superpage that is not fully populated. For *fpga* and *coral*, any policy that imposes too small a maximum page size does not result in a large reduction in the miss count. The table also shows that *ONLINE* and *APPROX-ONLINE* sometimes cause slightly more TLB misses than the more anxious *ASAP* policies because the online policies defer promotion until there is evidence that it will be beneficial. The TLB miss counts counts show that *ASAP* sometimes promotes pages that *OFFLINE* does not, indicating that *ASAP* creates some superpages that do not eliminate enough misses to recover the promotion cost.

Figure 3 shows the impact that each of the policies described in the previous section has on TLBMCPI. The figure illustrates that:

- *OFFLINE* reduces TLBMCPI substantially when compared to fixed-size 4 KB or 16 KB pages.
- Although neither *ONLINE* nor *APPROX-ONLINE* performs as well as *OFFLINE*, for most of the workloads their performance is significantly better than a policy that uses small fixed-size pages.
- For *APPROX-ONLINE* the largest component of TLBMCPI is page promotion cost, which reflects the overhead of physically coalescing pages. This indicates that both baseline TLB miss cost and bookkeeping overhead can be negligible in a system that eliminates most TLB misses.
- *ASAP-4-64* achieves an overall TLBMCPI that is close to *APPROX-ONLINE* except for applications with extremely large working sets such as *coral, fpga, cecil* and *fft*. For these applications, *ASAP-4-64*'s imposition of a maximum superpage size of 64 KB results in performance worse than that of *APPROX-ONLINE*.
- Some workloads, such as *gcc*, do not benefit from larger pages. This is unsurprising, since *gcc* has a low TLB miss rate even with 4 KB pages. In such cases, no policy performs better than one that uses fixed-size 4 KB page. It is straightforward to modify the online policies so that they remain inactive until the TLB miss rate exceeds some maximally acceptable TLBMCPI.

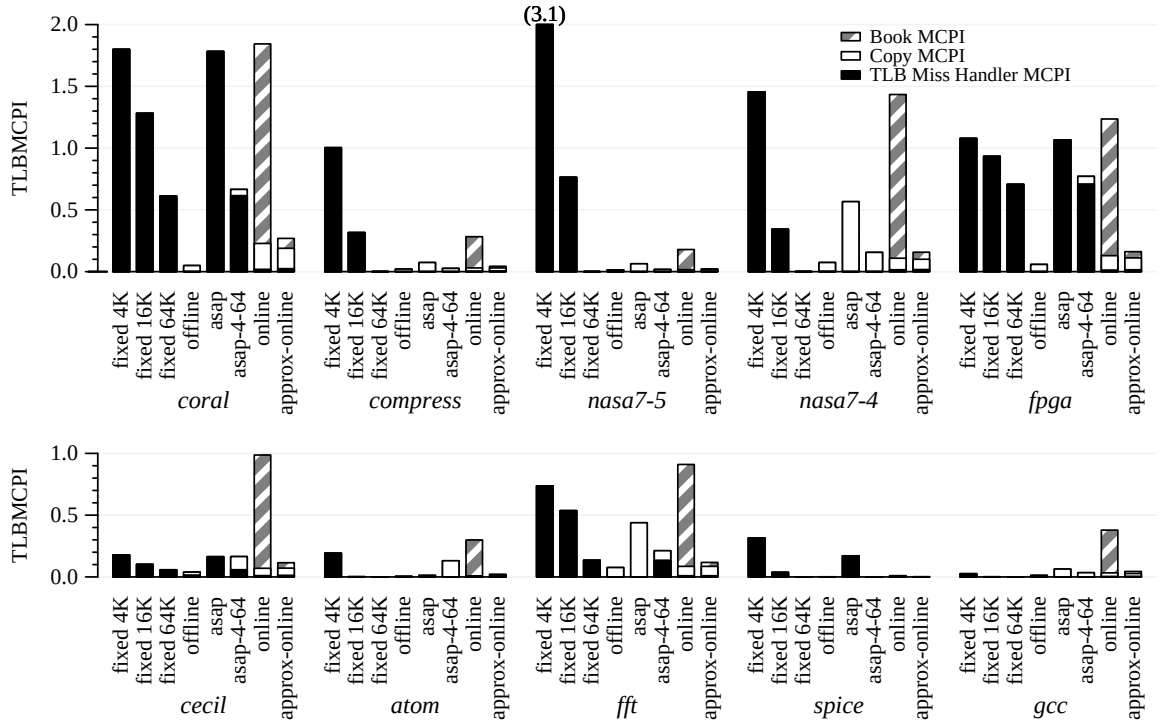


Figure 3: *This figure shows the overhead due to TLB management for policies using fixed and variable-sized pages. The fixed-size page policies incur a fixed overhead on each TLB miss (TLB Miss Handler MCPI). The variable-size page policies also incur promotion overhead when superpages are created (Copy MCPI), and bookkeeping overhead on each TLB miss (Book MCPI).*

ONLINE does not always reduce TLBMCPI compared to *APPROX-ONLINE*, even if we disregard the additional bookkeeping overhead incurred by *ONLINE*. This indicates that the prefetch counters are generally a more important predictor of the future benefit of promotion than the capacity counters. While the prefetch counters reflect an application's current working set (pages in the TLB and the just-referenced page), the capacity counters can reflect pages that are no longer in the TLB and may no longer even be part of the program's working set. We conclude that the considerable space and time overhead for maintaining the capacity counters is not justified, and that *APPROX-ONLINE* is a much more desirable policy than *ONLINE*, for reasons of both cost and accuracy.

Benchmark	*FIXED 16 KB*	*FIXED 64 KB*	*OFFLINE*	*ASAP*	*ASAP-4-64*	*ONLINE*	*APPROX-ONLINE*
coral	10.0	22.9	33.8	0.3	21.9	12.3	29.5
compress	23.7	34.6	33.9	32.0	33.7	28.5	33.1
nasa7-5	36.2	48.1	47.9	47.1	47.8	46.4	47.8
nasa7-4	28.7	37.6	35.7	22.9	33.5	15.0	33.5
fpga	2.8	7.1	19.4	0.2	5.8	5.9	17.4
cecil	2.5	3.9	4.5	0.5	0.4	-13.7	2.0
atom	7.3	7.4	7.2	6.9	2.4	0.6	6.6
fft	2.7	8.0	8.8	4.0	7.0	2.3	8.2
spice	1.1	1.3	1.3	0.6	1.3	1.3	1.3
gcc	1.1	1.1	0.6	-1.6	-0.4	-8.8	-0.8

Table 5: *This table shows the estimated percentage improvement in end-to-end execution time relative to a system with fixed-size 4 KB pages. Negative entries reflect a slowdown.*

8.3 Execution time

We can use the measurements of a live system presented in Table 2 to estimate the end-to-end performance of each policy. We add the simulated TLB overhead for each program and policy to the measured *non*-TLB execution time on the DEC Alpha, which is independent of the Alpha's TLB size. We then compute the percentage improvement of each policy relative to a system with fixed-size 4 KB pages. The results of this computation are shown in Table 5. The table shows that for these workloads, the reduction in TLBMCPI due to *APPROX-ONLINE* translates into a reduction in program execution time of as much as 48%.

8.4 Memory usage

The promotion policies can increase the memory usage relative to a system using 4 KB pages because a superpage may not be fully populated with referenced data. As described in Section 4, large fixed-size pages can also increase memory consumption because of internal fragmentation. Since the online policies delay promotion until warranted by TLB miss patterns, these policies should waste less memory than a system with large fixed-size pages. Figure 4 confirms this hypothesis, showing that the online policies never increase memory usage by more than 4%, and only in one case by more than 2%. In contrast, *ASAP 4-64* increases memory usage by 5% or more for four of the benchmarks, and as much as 12% for one.

Table 6 shows the distribution of final page sizes created with *APPROX-ONLINE*, revealing that this policy limits its memory consumption by using a wide range of page sizes.

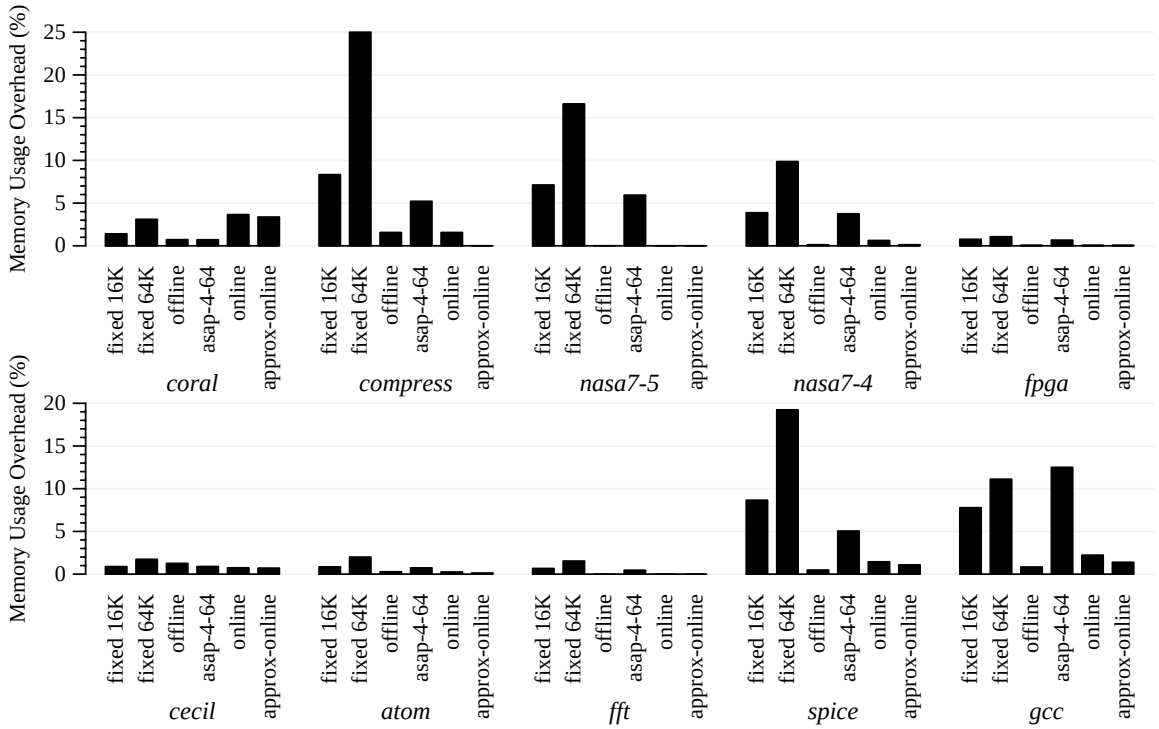


Figure 4: *Memory overhead. This figure shows the increase in memory usage for each policy relative to a system with fixed-size 4 KB pages. The dynamic superpage policies typically incur a memory usage overhead of less than 2%, compared to overheads of 10-25% for 64 KB fixed-size pages. ASAP uses no more memory than FIXED 4 KB, and is not shown. ASAP-4-64 promotes as soon as a 64 KB superpage is at least half full, so its memory usage can be greater than ASAP.*

Benchmark	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	1 MB	2 MB	4 MB	8 MB	Total 4 KB Pages
coral	105	9	11	4	8	3	4	1	11	8	1	0	8,743
compress	22	1	6	10	2	1	0	0	0	0	0	0	192
nasa7-5	126	0	0	0	1	0	1	1	1	0	0	0	590
nasa7-4	406	0	1	13	10	4	0	0	0	0	0	0	802
fpga	378	2	0	5	1	1	3	12	13	2	4	0	10,646
cecil	44,526	154	61	45	40	19	14	9	4	8	8	5	72,286
atom	8,134	17	11	6	3	2	0	0	0	0	0	0	8,372
fft	97	0	0	0	0	2	1	1	1	1	1	3	8,289
spice	117	2	4	4	8	9	4	0	0	0	0	0	841
gcc	109	8	18	15	3	0	0	0	0	0	0	0	365

Table 6: *The final page distributions for APPROX-ONLINE. This table shows the number of pages of a given size at the end of the run of each program.*

8.5 Summary

In Section 4 we showed that large fixed-size pages can reduce TLBMCPI at the expense of increased internal fragmentation. Figure 5 revisits this tradeoff between TLBMCPI and memory usage in the context of superpages. The figure shows that *APPROX-ONLINE* attains the low TLBMCPI of large fixed-size pages, while keeping memory consumption close to that of small pages. While the other promotion policies also improve upon the performance of fixed-size pages, *APPROX-ONLINE* outperforms them on every benchmark in terms of TLBMCPI, memory overhead, or both.

APPROX-ONLINE achieves TLBMCPI within 0.10 of *OFFLINE* for nine of the ten programs, even though *OFFLINE* has full information about the future reference stream, and has no accounting overhead. By paying a small bookkeeping cost on each TLB miss, *APPROX-ONLINE* identifies and constructs those superpages that result in a net reduction in TLB overhead, while avoiding the internal fragmentation of large fixed-size pages.

9 Conclusions

Superpages permit a system's TLB to map a substantially larger portion of memory than is possible with small fixed-size pages, and to use memory more effectively than when pages are large. We have described a methodology for detecting when and where a superpage should be constructed based on TLB miss behavior gathered at runtime. A simple online policy, *APPROX-ONLINE*, that considers both past TLB miss behavior and promotion cost delivers a TLBMCPI nearly as good as a policy using fixed-size pages, consumes less memory, and outperforms policies that obliviously promote.

Acknowledgements

Jeff Dean, Dave Grove, Wayne Wong, and the anonymous referees provided helpful comments on earlier drafts of this paper. Alan Eustace provided invaluable assistance in the form of expertise on Atom and access to CPU cycles at DEC WRL.

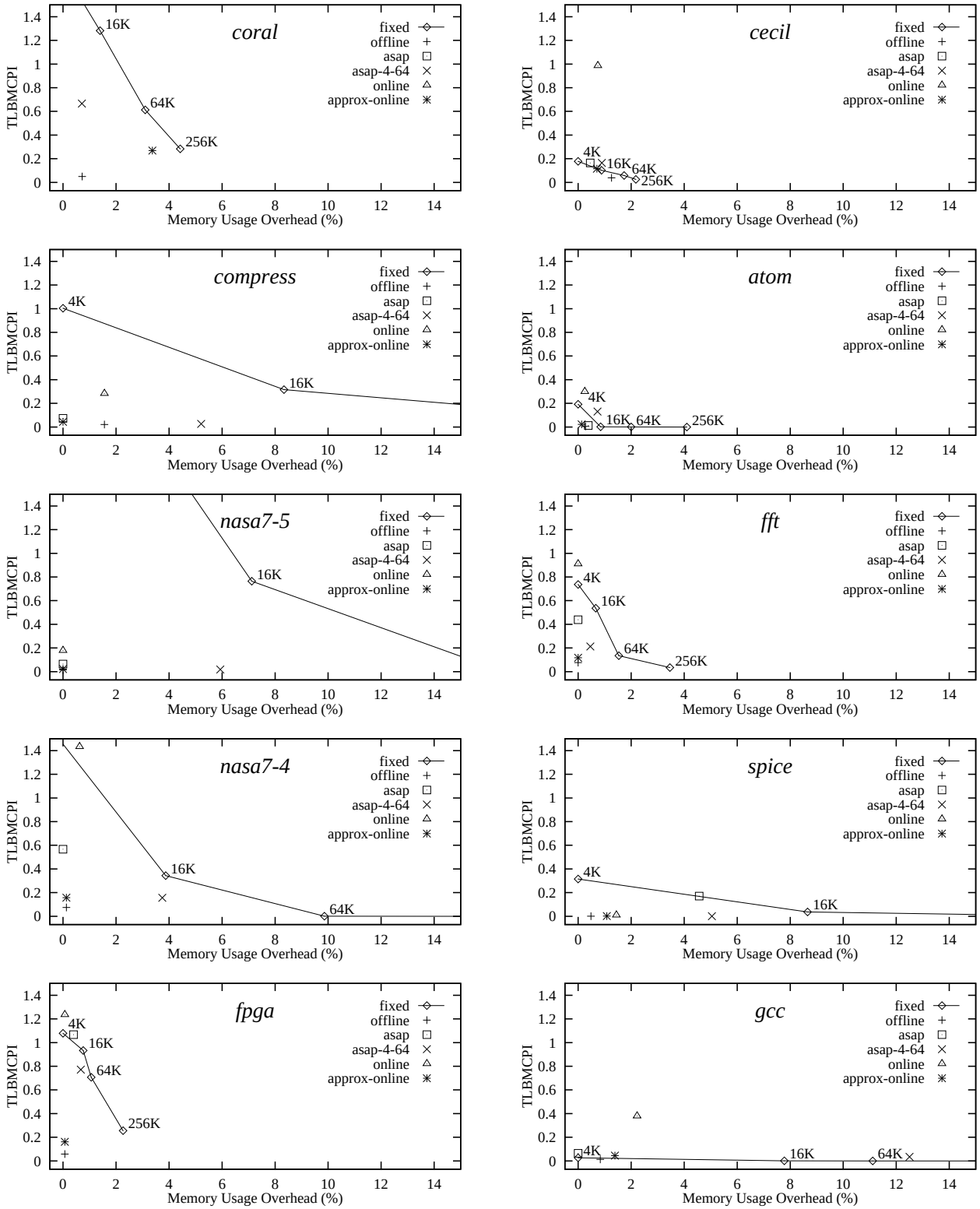


Figure 5: *Tradeoff between memory usage and TLBMCPI. The dynamic policies (ONLINE, APPROX-ONLINE, and OFFLINE) can achieve the benefits of small pages, in terms of memory usage, and those of large pages, in terms of TLBMCPI. The origin represents the ideal point in the tradeoff between memory usage and TLBMCPI. The graphs show that the dynamic policies come close to that ideal point. For comparison we consider the performance of fixed-size pages as large as 256 KB. In some cases the FIXED policies have such high TLBMCPI or memory usage that they do not fit on these graphs.*

References

[Appel & Li 91] Appel, W. and Li, K. 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, April 1991.

[Babaoğlu & Joy 81] Babaoğlu, Özalp. and Joy, W. Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 78–86, December 1981.

[Bala et al. 94] Bala, K., Kaashoek, F., and Weihl, W. Software Prefetching and Caching for Translation Buffers. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation*, pages 243–254, November 1994.

[Blanck & Krueger 92] Blanck, G. and Krueger, S. The SuperSPARC Microprocessor. In *COMPCON*, pages 136–141, February 1992.

[Cao et al. 94] Cao, P., Felten, E., and Li, K. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation*, pages 165–177, November 1994.

[Carter et al. 91] Carter, J., Bennett, J., and Zwaenepoel, W. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Chambers 93] Chambers, C. The Cecil Language: Specification and Rationale. Technical Report 93-03-05, University of Washington, March 1993.

[Chen et al. 92] Chen, J. B., Borg, A., and Jouppi, N. P. A Simulation-based Study of TLB Performance. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 114–123, May 1992.

[Dig 92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor, Hardware Reference Manual*, 1992. Order Number: EC-N0079-72.

[Dutton et al. 92] Dutton, T., Eiref, D., Kurth, H., Reisert, J., and Stewart, R. The Design of the DEC 3000 AXP Systems, Two High-Performance Workstations. *Digital Technical Journal*, 4(4):66–81, 1992. Special Issue.

[Hauck & Borriello 95] Hauck, S. and Borriello, G. An Evaluation of Bipartitioning Techniques. Submitted for publication to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1995.

[Hosking & Moss 93] Hosking, A. L. and Moss, J. E. B. Protection Traps and Alternatives for Memory Management of an Object Oriented Language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119, December 1993.

[Kane & Heinrich 92] Kane, G. and Heinrich, J. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Karlin et al. 88] Karlin, A., Manasse, M., Rudolph, L., and Sleator, D. Competitive Snoopy Caching. *Algorithmica*, 3(1):70–119, 1988.

[Karlin et al. 91] Karlin, A. R., Li, K., Manasse, M., and Owicki, S. Empirical Studies of Competitive Spinning for Shared Memory Multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[Khalidi et al. 93] Khalidi, Y. A., Talluri, M., Nelson, M., and Williams, D. Virtual Memory Support for Multiple Page Sizes. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 104–109, October 1993.

[Mogul 93] Mogul, J. Big Memories on the Desktop. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 110–115, October 1993.

[Ramakrishnan et al. 93] Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P. Implementation of the CORAL Deductive Database System. In *Proceedings of ACM SIGMOD Internation Conference on Management of Data*, 1993.

[Rashid et al. 87] Rashid, R., Avadis Tevanian, J., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, April 1987.

[Sleator & Tarjan 85] Sleator, D. D. and Tarjan, R. E. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28:202–208, February 1985.

[Srivastava & Eustace 94] Srivastava, A. and Eustace, A. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*. ACM, 1994.

[Talluri & Hill 94] Talluri, M. and Hill, M. D. 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*, pages 171–182, October 1994.

[Talluri et al. 92] Talluri, M., Kong, S., Hill, M. D., and Patterson, D. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 415–424, May 1992.

[Young et al. 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[Zekaukas et al. 94] Zekaukas, M., Sawdon, W., and Bershad, B. Software Write Detection for Distributed Shared Memory. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.

A Maintaining prefetch and capacity counters

The *ONLINE* and *APPROX-ONLINE* page promotion policies presented in Section 7 rely on one or both of the prefetch and capacity counters described in Section 6. We now present algorithms for maintaining these counters.

Prefetch counters

Recall that the prefetch counters are updated as follows. On a TLB miss to a page p, *prefetch*(P) is incremented for each superpage P that covers both p and some page already in the TLB. To avoid the overhead of scanning the contents of the TLB on each miss, we use an additional counter for each superpage P, called *tlbcount*(P), that indicates whether or not P or one of its component pages is currently in the TLB. With this information, we can quickly update the prefetch charges on a miss.

tlbcount(P) takes on one of four values, and satisfies the following invariants:

- If P is part of a larger superpage that has already been promoted, *tlbcount*(P) $= -1$.
- If P and none of its component pages are in the TLB, *tlbcount*(P) $= 0$.
- If a translation for P is in the TLB, *tlbcount*(P) $= 1$.
- If a translation for a subpage of P is in the TLB and only one of *tlbcount*(P_1) and *tlbcount*(P_2) is positive, where P_1 and P_2 are the two component subpages of P, *tlbcount*(P) $= 1$.
- If a translation for a subpage of P is in the TLB and both *tlbcount*(P_1) and *tlbcount*(P_2) are positive, where P_1 and P_2 are the two component subpages of P, *tlbcount*(P) $= 2$.

On a miss to p, the invariants are maintained and the prefetch counters are updated as described below. Let $p = p_0, p_1, \ldots p_m$ be the set of superpages containing p, in increasing order of size. (The size of p_m is the maximum superpage size.) For each p_i with *tlbcount*(p_i) $= 0$, and the smallest i with *tlbcount*(p_i) > 0, *tlbcount*(i) is incremented. For all i with *tlbcount*(p_i) > 0, *prefetch*(i) is incremented. Suppose that the translation for q is replaced in order to bring p into the TLB. Let $q = q_0, q_1, \ldots, q_r$ be the superpages containing q in increasing order of size, such that q_r is the smallest one with *tlbcount*(q_r) $= 2$. Then for each q_i, $0 \leq i \leq r$, *tlbcount*(q_i) is decremented.

Finally, if a page P is promoted, *tlbcount*(P') is set to -1 for each P' component to P. (Note that *tlbcount*(P) > 0 already.) If a page P is demoted, *tlbcount*(P) is reset to 0.

While the algorithm as described so far avoids the overhead of scanning the TLB on every miss, it still updates a counter for pages as large as the maximum page size. Since these large superpages are promoted only rarely, we can reduce the bookkeeping overhead of the algorithm by maintaining precise prefetch charges for large pages only when they are candidates for promotion.

For example, in our simulations, *APPROX-ONLINE*'s promotion threshold for a 128 KB superpage is 1600 ($0.125 \times R$, where R is the ratio of promotion cost to TLB miss cost for a 128 KB page). Thus it is impossible that a 128 KB superpage will be promoted until at least 1600 TLB misses have occurred. Rather than performing 1600 individual updates to superpages of sizes 128 KB through 8 MB (the maximum superpage size), we can defer the

updates until we experience 1600 TLB misses, and then perform all of the updates at once, sharply reducing the average overhead per TLB miss.

Implementing an algorithm based on this idea of "batching" updates to the prefetch counters requires choosing the degree to which we maintain accurate information. If we maintain precise prefetch counters for pages that are too large, then the overhead of each TLB miss remains too high. On the other hand, if we maintain precise information only for very small pages, then the batched updates will have to be performed too frequently.

We implemented an algorithm that uses the *tlbcount* counters and batched updates to reduce the overhead per TLB miss. We found that propagating updates eagerly to superpages of size 64 KB or less and deferring updates to larger superpages resulted in the best overall performance. Our implementation requires an additional counter, *deferred*(P), that reflects the updates to prefetch counters that have not yet been propagated to superpages containing P. We maintain *deferred*(P) for all superpages of size 64 KB and larger. We profiled the execution of this algorithm with the TLB miss stream of our benchmark suite, and found that on average it executed 100 instructions per TLB miss. The algorithm as implemented also consumes space: one *prefetch* counter for every potential superpage, one *tlbcount* counter for every potential superpage and every base page, and one *deferred* counter for every superpage of size 64 KB or larger. Altogether this results in an average of 3.125 counters per base page. Since *APPROX-ONLINE* uses the prefetch counter alone, it incurs a total bookkeeping overhead per TLB miss of 100 instructions. *ONLINE* incurs additional overhead in order to maintain the capacity counters, described below.

Capacity counters

ONLINE maintains capacity charges as well as prefetch charges. Here we describe a simple algorithm in order to establish a rough estimate of the cost of maintaining the capacity counters.

Recall that the capacity count is incremented for each superpage that would have coalesced enough TLB entries to prevent the current miss to page p. We can update the capacity count as follows. The number of TLB entries that must be coalesced depends on the depth of the previous reference to p in the LRU stack, which we will refer to as *lru-depth*. We compute the number of TLB entries covered by each superpage Q. The pseudo-code below computes this number in the variable *coverage*(Q), and increments *capacity*(Q) for every potential superpage Q that covers at least *lru-depth* TLB entries.

lru-depth: the depth of the page that missed in the LRU stack
coverage(Q): number of TLB entries covered by superpage Q, initially 0.

```
for each page P in the TLB {
    for each superpage Q containing P {
        increment coverage(Q);
        if ( coverage(Q) == lru-depth ) {
            increment capacity(Q);
            check for promotion;
        }
    }
}
```

In a system with 32 TLB entries and page sizes ranging from 4 KB to 8 MB (*i.e.* each base page is contained in 11 superpages), the body of the loop will be executed at least 32*11 = 352 times. If we assume that the loop body requires 7 instructions (load, increment, store, subtract, conditional branch, update index, conditional branch), then updating the capacity counters will require $352 * 7 = 2464$ instructions per TLB miss. Since the *ONLINE* maintains both prefetch and capacity counters, its total bookkeeping overhead is at least $100 + 2464 = 2564$ instructions per TLB miss.

B Effect of TLB miss overhead on promotion threshold

In Section 6.1 we showed that we can bound the overhead of an online promotion policy by using a promotion threshold of R, the ratio of promotion cost to baseline TLB miss cost. We now show that even when the additional bookkeeping overhead is included in the analysis, the online policy should still use a promotion threshold of R to minimize its overhead compared to the optimal offline policy.

Recall that in Section 6.1, we considered the case that there is one candidate N KB superpage P consisting of two $N/2$ KB pages, and that in the run of an application, m TLB misses can be prevented by constructing P at time 0. If we let C denote P's promotion cost, and b denote the baseline TLB miss cost, then the ratio of promotion cost to the TLB miss cost is $R = C/b$. We observed that the optimal offline policy, knowing m, will perform the promotion at time 0 if m is at least R, and will never promote to P if $m < R$. We also showed that an online policy that promotes to P when P's miss charges reach R incurs a cost of at most twice the optimal offline cost.

We now consider how to choose $thresh$, the promotion threshold for the online algorithm, assuming that the online algorithm incurs an additional overhead of o time units on each TLB miss. The worst-case ratio between online cost and offline cost occurs when the promoted page would not prevent any future misses. For any given value of *thresh*, this worst-case ratio occurs when m, the number of misses prevented if P is constructed initially, is exactly *thresh*. For this value of m, the ratio between online and offline cost is

$$\frac{(b+o)\,\mathit{thresh} + C}{\min(b(\mathit{thresh}), C)}$$

This ratio is equal to

$$\max\left(1 + \frac{o}{b} + \frac{R}{\mathit{thresh}}, \frac{\mathit{thresh}}{R} + \frac{o \cdot \mathit{thresh}}{C} + 1\right)$$

When *thresh* is set to $R = C/b$, this worst case ratio between online and offline cost has a value of $2 + o/b$, since

$$1 + \frac{o}{b} + \frac{R}{\mathit{thresh}} = 1 + \frac{o}{b} + 1 = 2 + \frac{o}{b}$$

and

$$\frac{\mathit{thresh}}{R} + \frac{o \cdot \mathit{thresh}}{C} + 1 = 1 + \frac{o}{b} + 1 = 2 + \frac{o}{b}$$

This is the ratio's minimum, since for any *thresh* $> R$,

$$\frac{\mathit{thresh}}{R} + \frac{o \cdot \mathit{thresh}}{C} + 1 > 1 + \frac{o}{b} + 1 = 2 + \frac{o}{b}$$

and for any *thresh* $< R$,

$$1 + \frac{o}{b} + \frac{R}{\mathit{thresh}} > 1 + \frac{o}{b} + 1 = 2 + \frac{o}{b}$$

Hence, the promotion threshold for P that minimizes the online to offline performance ratio is R. With this threshold, the online policy incurs at most $2 + o/b$ times the cost incurred by offline policy with respect to P.